Flexibles Matrix-Vektor-Format

Bachelorarbeit im Fach Informatik

vorgelegt von **Silvan Marti**
geb. 09.12.1993 in Glarus

angefertigt am

Department Informatik
Lehrstuhl für Informatik 10
Systemsimulation
Friedrich-Alexander-Universität Erlangen-Nürnberg
(Prof. Dr. U. Rüde)

Betreuung:
Priv.-Doz. Dr. Nicolas Neuß
Prof. Dr. Ulrich Rüde

Beginn der Arbeit: 28.04.2020
Abgabe der Arbeit: 28.09.2020

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 10 (Simulationssysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 28.09.2020

Silvan Marti

Inhaltsverzeichnis

Kurzzusammenfassung

Das in Common Lisp implementierte Finite-Elemente-Framework FEMLISP, stellt viele Funktionen zum Lösen von partiellen Differentialgleichungen zur Verfügung. Unter anderem unterstützt es den Umgang mit großen dünnbesetzten Matrizen. Diese Funktionen sind sehr flexibel auf verschiedene Probleme anwendbar, jedoch lässt ihre Performanz zu wünschen übrig.

In dieser Arbeit untersuchen wir den Grund für diese langsame Berechnung. Wir testen neue Methoden, um die Matrix-Vektor-Multiplikation in FEMLISP zu optimieren. Außerdem entwickeln wir alternative Speicherformate und Multiplikations-Funktionen. In experimentellen Tests werden die neuen Ansätze mit der aktuellen Implementierung verglichen. Ergebnisse aus Performanztests zeigen, dass günstigere Lösungen gefunden werden können.

Zum Schluss zeigen wir einen Weg auf, wie man FEMLISP erfolgreich weiterentwickeln kann.

Abstract

The finite element framework FEMLISP is a universal tool to solve partial differential equations. So far, it offers a wide array of feeatures to cope with problems of many different shapes. It also supports functions to process very large sparse matrices. At the moment this implementation is very flexible. However, it lacks a convincing computing performance.

In this thesis, the causes of this underperformance are beeing investigated. Moreover, we discover new approaches to enhance the matrix-vector computing in FEMLISP. We develop alternative sparse matrix formats and new multiplication functions and performance is experimentally studied. Results show, that very well performing solutions can be found.

Finally, we draw an outlook for further development of FEMLISP.

1 Einleitung

1.1 Überblick

In dieser Arbeit untersuchen wir verschiedene Speicherstrukturen, um dünnbesetzte Matrizen zu repräsentieren. Insbesondere fokussieren wir uns auf Flexibilität und Performanz potentieller Lösungen. Die Formate sind Teil von Femlisp, eines universellen Frameworks implementiert in Common Lisp (CL). Femlisp löst partielle Differentialgleichungen (PDGLs) mit Hilfe der Finite-Elemente-Methode (FEM).

1.2 Motivation

Im Augenblick verwendet Femlisp eine Speicherstruktur für DMs, bei denen ein mit Hilfe der Finite-Elemente-Diskretisierung (FED) implementierter Matrix-Graph für die geometrischen Elemente des Gitters auf Blockmatrizen verweist. Genauer sind die Zeilen der Matrix A als eine Hashtabelle (HT) implementiert, die geometrische Zellen (Vertices, Kanten, Flächen, Volumina) des Finite-Elemente (FE)-Gitters auf die Zeilen der Matrix abbildet. Diese Zeilen sind wiederum HTs, die geometrische Einheiten auf den zugehörigen Matrix-Block abbilden.

Dieses Format ist relativ flexibel, weil es lokale Veränderungen der Gitterstruktur lokal in der Matrix nachbilden kann. Die Verwendung der HTs erfordert außerdem keine Nummerierung des Gitters, ist jedoch in einigen Fällen ineffizient, insbesondere wenn -wie im Falle skalarer Gleichungen und FED niedriger Ordnungen- nur sehr kleine Blockmatrizen auftreten.

1.3 Aufgabenstellung

In dieser Bachelorarbeit soll nun erstens dieses Format von der Performanz her kritisch untersucht werden und verschiedene Alternativen implementiert und getestet werden.

Insbesondere:

1. **Implementierung der Zeilentabellen mit Hilfe von Listen**

   Anstelle der bisherigen HTs werden neu Listen als Zeilentabelle implementiert und die Performanz untersucht. Es ist zu erwarten, dass diese Änderung

bei den in üblichen 2D- und 3D-Gittern G auftretenden$Bandbreiten \leq 30$ leichte Verbesserungen gegenüber den im Augenblick verwendeten HTs liefert.

2. **Einführung einer Nummerierung der Gitterelemente**

   Eine Nummerierung der Gitterelemente wird implementiert und die HT-Spaltentabellen werden durch geordnete Vektoren ersetzt. Diese Implementierung erlaubt einen

   $$O(log(n) + log(k)), n = dim(A), k \leq 2 * dim(G) + 1$$

   Zugriff auf beliebige Matrix-Einträge, allerdings ist eine Änderung der Gitterstruktur nicht mehr mit geringem Aufwand möglich.

3. **Verwendung einer Compressed Column Storage (CCS) oder Compressed Row Storage (CRS)-Datenstruktur**

Alle Alternativen sollen dabei so in Femlisp implementiert werden, dass zur Laufzeit entschieden werden kann, welche der Varianten verwendet wird.

1.4 Struktur dieser Arbeit

Der Rest dieser Arbeit ist wie folgt strukturiert:

- **Kapitel 2. Grundlagen**, gibt eine kurze Einführung in Lisp, FEM und Femlisp.
- **Kapitel 3. Implementierung**, erläutert einige Aspekte der Implementierung.
- **Kapitel 4. Performanz**, präsentiert die aus Performaztests gewonnenen Daten.
- **Kapitel 5. Fazit**, zieht eine Schlussfolgerung zu dieser Arbeit.

2 Grundlagen

Partielle Differentialgleichung entstehen bei der mathematischen Modellierung vieler ingenieur- oder naturwissenschaftlicher Phänomene. Als prominentes Beispiel kann man die Strömung von linear-viskosen Newtonschen Fluiden, beschrieben durch die Navier-Stokes-Gleichungen nennen. Tatsächlich ist es für die meisten dieser Probleme unmöglich oder sehr aufwendig, eine exakte Lösung dieser Gleichungen z.B. durch die Anwendung von Laplace und Fouriertransformationen zu finden. Aus diesem Grund werden numerische Approximationsverfahren zu Hilfe genommen. Eines der beliebtesten Verfahren ist die FEM (siehe z.B. das Buch von Süli [**5**]).

2.1 Finite Elemente Methode

Bei diesem Verfahren wird das zu berechnende Gebiet durch ein Gitter G unterteilt und die Gleichung an den Gitterknoten $v \in G$ numerisch approximiert. Diese nun diskrete Unterteilung kann als Matrix dargestellt werden. Da die Genauigkeit der Approximation mit der Anzahl der Elemente zunimmt, können je nach Problem sehr große Matrizen entstehen. Um ein solches Problem zu simulieren, definieren wir eine beispielhafte Matrix:

$$L_{ij} = \begin{cases} 2\dim(G) & \text{wenn } i = j \\ -1 & \text{wenn } i \neq j \text{ und } v_i \text{ ist benachbart zu } v_j \\ 0 & \text{sonst.} \end{cases} \tag{2.1}$$

Beispiel 2.1.1. Ein berandetes zweidimensionales Gebiet wird in eine Menge von $n = 9$ Knoten diskretisiert. Das dabei entstehende System aus Knoten kann durch

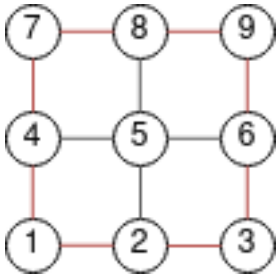

Abbildung 2.1: Beispiel einer einfachen Diskretisierung eines Gebiets

folgende Matrix repräsentiert werden:

$$L^{9x9} = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix}$$

Man beachte nun, dass die Matrix sehr viele 0-Einträge enthält. Genauer ist die Zahl der Nichtnull-Einträge in einer typischen Zeile gegeben durch

$$Q = 2d + 1\,. \tag{2.2}$$

Eine Matrix, wie im oben gezeigten Beispiel, als vollbesetzte Matrix zu repräsentieren wäre ineffizient. Es werden viele unnötige Informationen gespeichert und später berechnet (eine Multiplikation mit 0 gibt 0). Um effektiv mit solchen Matrizen umzugehen, benötigt man deshalb spezielle Datenstrukturen.

2.2 FEMLISP

FEMLISP ist ein CL-Framework, welches unter anderem spezielle Datenstrukturen zum Umgang mit dünnbesetzten Matrizen zur Verfügung stellt. Weiter bietet es eine große Vielfalt an weiteren Funktionalitäten, welche auf der offiziellen Website von FEMLISP [**3**] näher beschrieben wird. Zur Zeit unterstützt FEMLISP die folgenden Matrix Speicherformate:

Definition 2.2.1. Hash-Tabelle-Hash-Tabelle (HTHT)-Format, besteht aus einer HT für die Zeilen der Matrix, welche wiederum aus HTs bestehen, welche die Zeileneinträge enthalten. Der Vorteil des Formats ist die direkte Darstellung des Operators durch einen Graphen ohne das Gitter nummerieren zu müssen. Die Matrix kann beliebige Einträge, auch Blockmatrizen, enthalten.

Definition 2.2.2. Compressed Matrix (CM)-Format, ist ein CCS oder CRS-Format. Es besteht aus drei Feldern: Wert, Zeilen-Indices und Spalten-Zeiger. Es ist ein simples Format und benötigt weniger Speicher als das HTHT-Format, benötigt allerdings eine Nummerierung des Gitters. Auch hier sind beliebige Einträge möglich.

Definition 2.2.3. Full Matrix (FM)-Format, welches vollbesetzte Matrizen, gespeichert in Feldern darstellt.

Leider sind diese Formate je nach Anwendung nicht sehr effizient. Femlisp arbeitet normalerweise so, dass die Einträge der HTHT-Matrix FM-Blöcke sind. Da generische Funktionen beim Zugriff auf diese Blöcke aufgerufen werden, ist dies nicht sehr effizient. Im Fall, dass die Matrix-Einträge 1x1-Matrizen sind, kommt dieser Nachteil besonders zum Ausdruck. Wir machen einen Test mit einer simplen Multiplikation:

$$z = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \tag{2.3}$$

wie wir im Folgenden sehen:

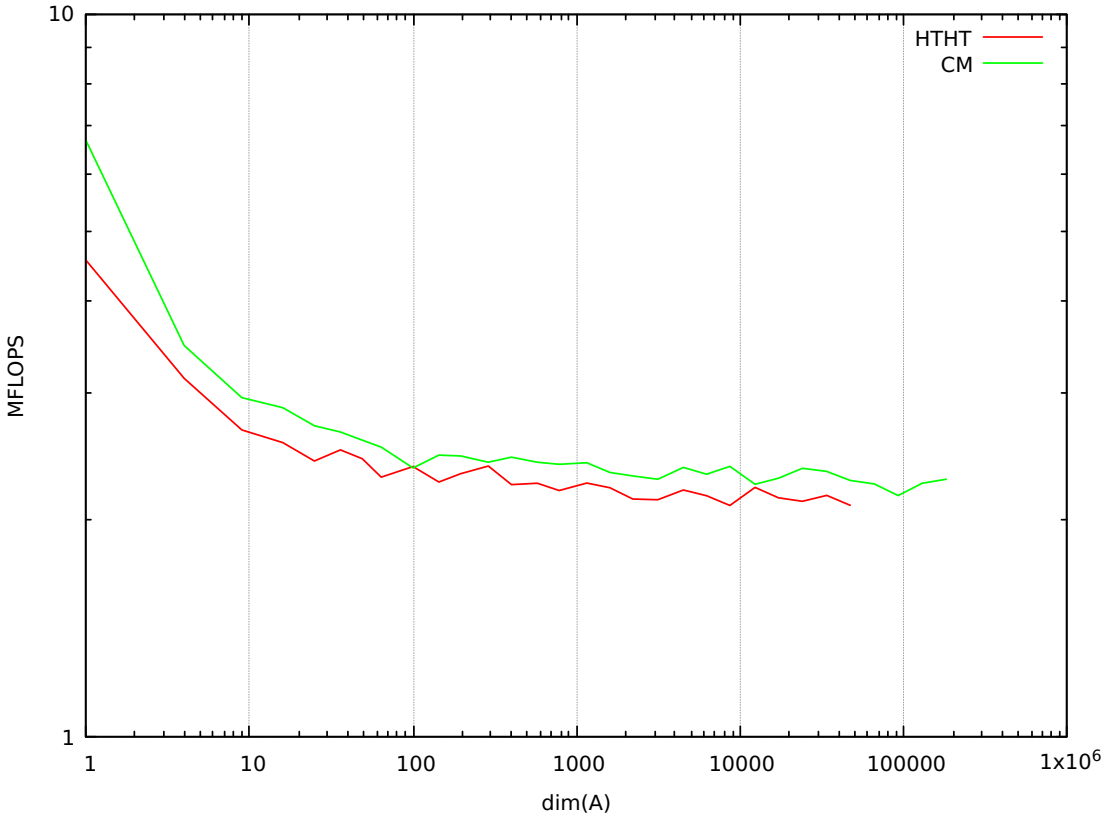

Abbildung 2.2: Mittelwert bei der Multiplikation mit 1x1 Matrix-Einträge der unoptimierten Formate CM und HTHT.

Wie in Abbildung 2.2 ersichtlich ist, sind beide Formate bei der Berechnung von simplen Gleitkommazahlen (1x1 Matrizen) nicht sonderlich leistungsfähig. Diese tiefe Performanz ist vor allem auch eine Folge von vielen Aufrufen von generischen Funktionen, die wir im nächsten Abschnitt erklären.

Mit heutzutage üblichen Prozessorentaktraten $> 1GHz$ und mehreren Kernen, sollte man ein Ergebnis jenseits von 1000 Mega Floating Point Operations per Second (MFLOPS) erwarten. Wir werden uns in den folgenden Kapiteln damit auseinandersetzen, wie diese Operation optimiert werden kann.

2.3 Lisp

FEMLISP ist in CL implementiert, weshalb wir hier eine kurze Einführung in die Eigenschaften der Programmiersprache LISP und im speziellen in den Dialekt CL geben. Dies ist ein kurzer Einblick, welcher von Neuß [**1**] und Seibel [**4**] genauer beschrieben wurde. Lisp verwendet die Präfix-Notation der Form:

Beispiel 2.3.1.

```
(<operator> <operand1> <operand2> ...)
```

Diese Notation wird, bis auf einige Spezialfälle, immer gleich angewendet. Die Operanden werden der Reihe nach durch einen Read Evaluate Print Loop (REPL) bearbeitet und dem Operator übergeben. Das berühmte "Hello World!"gestaltet sich in LISP deshalb besonders simpel:

Beispiel 2.3.2.

```
CL-USER> "Hello World!"
"Hello World!"
```

CL unterstützt mehrere interessante Features, die man bei anderen älteren und auch vielen neueren Programmiersprachen nicht finden wird. So sind unter anderem eine automatische Speicherverwaltung, Interaktivität und mehrere höhere Datenstrukturen wie HTs und Listen Teil des Sprachstandards. CL kann interpretiert oder compiliert ausgeführt werden. Die Programmierung und Entwicklung mit Lisp fühlt sich aus diesem Grund sehr dynamisch und flexibel an. CL ist zudem sehr ausdrucksstark, so dass mit sehr wenigen Zeilen Code komplexe Abläufe beschrieben werden können.

All diese Eigenschaften machen Lisp zu einem sehr starken Werkzeug, jedoch lauern auch Gefahren.

2.3.1 Datenstrukturen

Wir geben einen kurzen Überblick über die verwendeten Datenstrukturen:

Definition 2.3.1. Common Lisp, Notation der Datenstrukturen

- **Cons-Zellen**

  ```
  (x0 . x1)
  ```

- **Listen**

  ```
  (x0 x1 ...)
  ```

- **Felder**

  ```
  #(x0 x1 ...)
  ```

Cons-Zellen

Cons-Zellen sind die einfachste, in Lisp implementierte Datenstruktur. Im Grunde genommen ist es ein Feld mit der Länge zwei. Wir verwenden zukünftig immer die folgende Notation für eine Cons-Zelle.

Beispiel 2.3.3.

```
CL-USER> (cons 10 14)
(10 . 14)
```

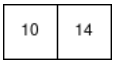

Abbildung 2.3: Einzelne Cons-Zelle

Listen

Listen werden in CL aus Cons-Zellen zusammengebaut. Anstatt einem zweiten Wert wird ein Zeiger auf eine andere Cons-Zelle gespeichert. Die folgenden Ausdrücke sind deshalb identisch:

Beispiel 2.3.4.

```
CL-USER> (cons 10 (cons 14 (cons 6 nil)))
(10 14 6)
CL-USER> (list 10 14 6)
(10 14 6)
```

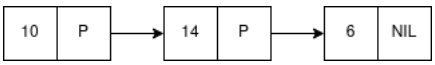

Abbildung 2.4: Listen aus zusammengesetzten Cons-Zellen und deren Allokation im Speicher

Eine Eigenschaft von Listen ist für uns von besonderer Bedeutung. Elemente können beliebig hinzugefügt oder entfernt werden, ohne den Speicher für den Rest der Liste neu allokieren zu müssen. Ein Nachteil der Liste is allerdings, dass der Speicherbedarf m_l mit der Anzahl der Elemente n gegeben ist durch:

$$m_l(n) = 2n \tag{2.4}$$

Dies führt bei langen Listen zu einem erhöhten Speicherbedarf gegenüber einem Array.

Arrays

CL unterstützt auch Arrays, oder Deutsch "Felder", welche an einem Stück allokiert werden. Auf die einzelnen Elemente kann mit einem Zeiger direkt zugegriffen werden, was sie besonders performant macht. Will man die Länge verändern, muss man allerdings ein neues Feld erzeugen und die Elemente verschieben. Aufgrund dieses Nachteils unterstützt CL auch Felder mit veränderbarer Länge. Diese sind so implementiert, dass auf dem Speicher tatsächlich ein größeres Feld allokiert wird als angefordert.

Beispiel 2.3.5.

```
CL-USER> (make-array 3 :initial-contents '(10 14 6))
#(10 14 6)
```

Ein großer Vorteil von Feldern gegenüber Listen ist der kleinere Speicherbedarf m_a bei vielen Elementen n. Dann belegt ein Feld aufgrund des Wegfalls der Zeiger nur rund die Hälfte des Speichers einer Liste. Allerdings kommt ein Feld mit einem overhead o, weshalb bei wenigen Elementen Listen effizienter sein können.

$$m_a(n) = o + n \tag{2.5}$$

Wenn man Abbildung 2.4 mit Abbildung 2.5 vergleicht, kann man dies deutlich erkennen.

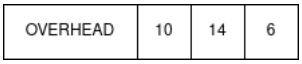

Abbildung 2.5: Veranschaulichung eines Feldes auf dem Speicher

Hash-Tabellen

Eine Hash-Funktion evaluiert einen Schlüssel, der einen Zeiger auf den entsprechenden Bucket bzw. die Speicherstelle erzeugt (siehe Abbildung 2.6. Sie ist die flexibelste, allerdings auch die teuerste Struktur. Der genaue Speicherbedarf kann nicht mehr ganz einfach ermittelt werden.

2.3.2 Generische Funktionen

Common Lisp verfügt über ein Common Lisp Object System (CLOS) genanntes Objektsystem. Anders als in anderen objektorientierten Programmiersprachen gehören Methoden nicht zu Klassen, sondern zu einer Generic Function (GF) genannten Funktion. Eine GF verfügt über keine Implementierung, sondern sie bildet

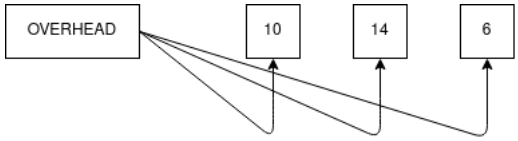

Abbildung 2.6: Veranschaulichung einer Hash-Tabelle

nur den Rahmen für zu ihr dazugehörige Methoden. Diese Methoden enthalten spezielle Implementierungen der GF und definieren, was passiert, wenn die GF mit bestimmten Datentypen als Argumente aufgerufen wird. Diese Herangehensweise ist sehr vielfältig einsetzbar, allerdings besteht das Problem, dass ein Aufruf einer GF eher teuer ist. Denn bei jedem Funktionsaufruf müssen die Datentypen der Argumente geprüft und die dazugehörige Methode während der Laufzeit gefunden werden. Ein Inlining ist normalerweise nicht möglich, da CL es erlaubt, Methoden zur Laufzeit hinzuzufügen oder zu entfernen.

2.4 Speicherschnittstelle

Als Abschluss dieses Kapitels machen wir einen kurzen Exkurs und setzten uns mit dem Aufbau eines Mikroprozessors auseinander. Die Speicherschnittstelle stellt bei Mikroprozessoren einen bekannten Flaschenhals dar, wie dies Hager und Wellein in [**2**] zeigen.

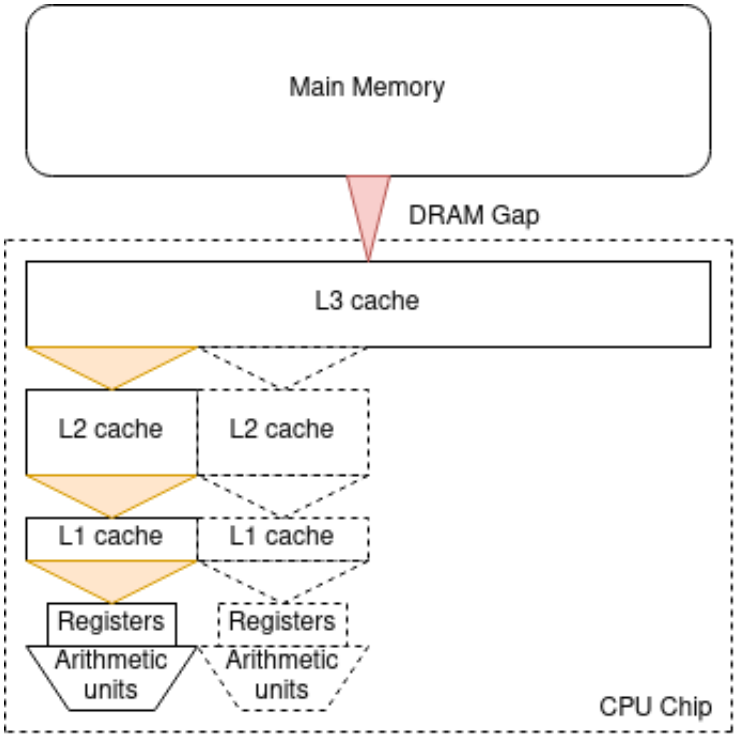

Abbildung 2.7: Veranschaulichung der Speicheranbindung eines Mehrkernprozessors. Die für alle Messungen verwendende Intel CPU, i7-6700HQ der Skylake Generation entspricht diesem Schema. Der problematische DRAM-Gap ist rot markiert.

Bei Prozessen, die größere Mengen an Speicher allokieren, reicht ab einer bestimmten Schwelle der schnelle, prozessorinterne Cache-Speicher nicht mehr aus. Die überschüssigen Daten werden ab diesem Zeitpunkt auf den Arbeitsspeicher verschoben und passieren die Prozessor-Arbeitsspeicherschnittstelle. Diese Schnittstelle besitzt eine verhältnismäßig schmale Bandbreite, welche die Kapazität begrenzt. Dieser Engpass ist auch als „DRAM Gap“ bekannt. Wird dieser kritische Punkt überschritten, ist nicht mehr die Rechenleistung des Prozessors der limitierende Faktor, sondern die Übertragungsrate der Speicherschnittstelle.

3 Implementierung

Ein bekanntes Zitat Paul Grahams aus On Lisp [**6**] besagt:

> It is easy to write fast programs in Lisp. Unfortunately, it is very easy to write slow ones.

Wir merken uns diese Worte, wenn wir anschließend die Implementierung der Matrix-Multiplikation genauer unter die Lupe nehmen.

3.1 Multiplikation

3.1.1 Compiler Optimierungen

Eine Besonderheit von CL, ist die nicht nötige Datentypendeklarierung. Dies macht die Programme kürzer und lesbarer, macht es aber auch dem Compiler schwer, effizienten Code zu erzeugen. Glücklicherweise kann man in CL Datentypen auch manuell deklarieren und dadurch dem Compiler Optimierungen ermöglichen. Wir machen ein kleines Beispiel und wählen eine einfache Funktion, die aus einer Matrix alle Einträge aufsummiert.

Beispiel 3.1.1. Innere Schleifen der Iteration über eine Matrix des HTHT-Formats ohne Optimierungen.

```
(let ((result 0.0))
  (maphash (lambda (i row)
    (maphash (lambda (j value)
      (incf result value))
    row))
  (row-table matrix)))
```

Beispiel 3.1.2. Innere Schleifen der Iteration über eine Matrix des HTHT-Formats mit Optimierungen. Die Datentypen wurden manuell deklariert und der Compiler optimiert die Berechnungsgeschwindigkeit.

```
(let ((result 0.0))
  (declare
    (optimize (safety 0) (space 0) (speed 3)
              (type double-float result)
      (maphash (lambda (i row)
        (declare (ignorable i))
        (maphash (lambda (j value)
          (declare (ignorable j)
                   (type double-float value))
          (incf result value))
        row))
      (row-table matrix)))))
```

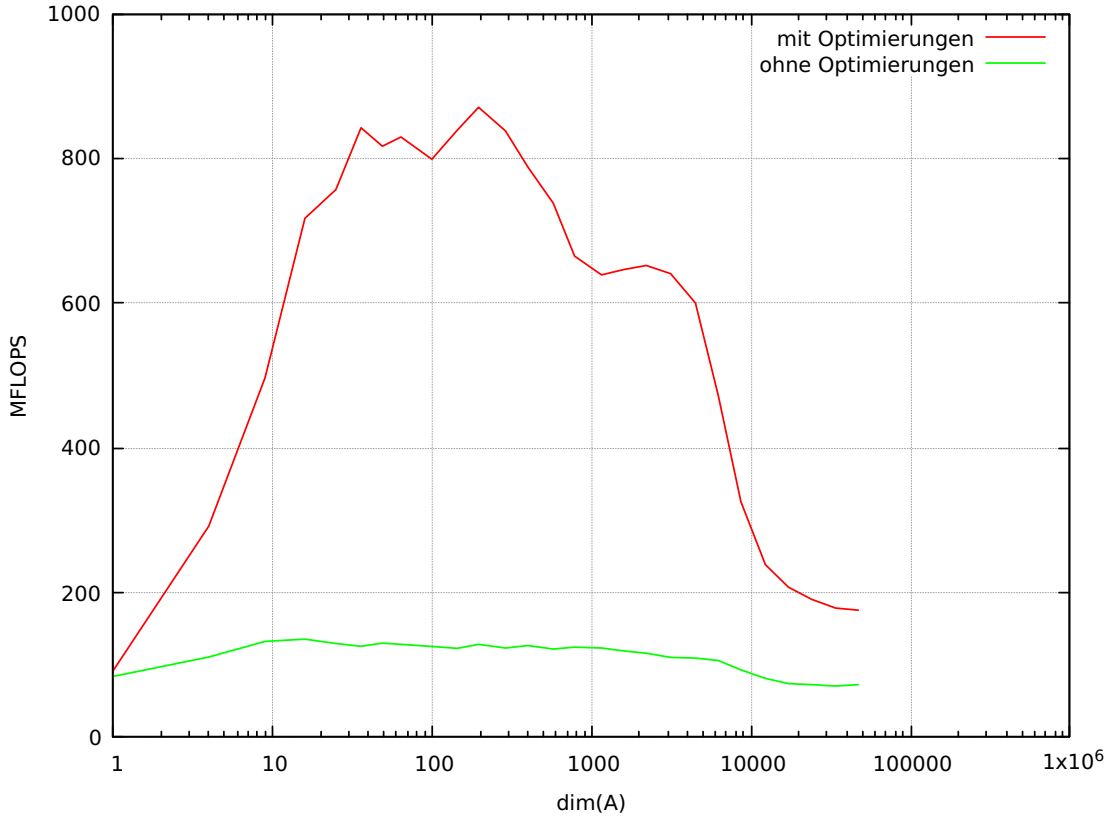

Abbildung 3.1: Vergleich der Performanz einer einfachen Funktion mit und ohne Compileroptimierungen

Wie in Abbildung 3.1 erkennbar ist, ermöglichen die Deklarationen enorme Performanzgewinne. Infolge dieses Umstandes, wurden bei allen weiteren Implementierungen von spezialisierten Multiplikations-Funktionen die Datentypen genau deklariert. Ein starker Performanzabfall ab einem bestimmten Schwellenwert ist bei der optimierten Funktion zu erkennen, dies lässt sich mit dem im letzten Kapitel erwähnten „DRAM Gap“-Effekt erklären und ist nicht Folge der Optimierung.

Wichtig ist hier anzumerken, dass Optimierungen nur vorgenommen werden können, wenn der Typ jeder einzelne Variable dem Compiler genau bekannt ist. Zwar ist gerade SBCL eine Lisp-Implementation, welche den Typ von Variablen des öfteren deduzieren kann (sogenannte “Typ-Inferenz”), leider aber bei weitem nicht in allen Fällen. Wenn es dem Compiler nicht möglich ist, den Datentyp einer Variable zu deduzieren, muss er manuell deklariert werden. Nimmt man in diesem Fall keine Deklarierung vor, fällt die Performanz auf ein Niveau, welches ähnlich dem einer nicht optimierten Funktion ist. Werden generische Funktionen innerhalb der Optimierung verwendet, ist Optimierung ebenfalls kaum möglich. Für uns bedeutet

dies, dass eine spezialisierte und optimierte Multiplikations-Funktion ausschließlich mit primitiven Datentypen implementiert werden darf.

3.1.2 Spezialisierung der General Multiplication Method (GEMM)

Wie zuvor erwähnt wurde, bietet FEMLISP momentan eine generische General Matrix Multiplication Method (GEMM), die mit vielen verschiedenen Datentypen von Matrizen und Vektoren rechnen kann. Da man diese Funktion aus den oben genannten Gründen nicht optimieren kann, verfolgen wir einen anderen Ansatz, und implementieren eine einfache optimierte Multiplikations-Funktion aus primitiven Datentypen. Diese Funktion kann fürs erste nur mit Skalaren rechnen. Weiter wird angenommen, dass eine Nummerierung des Gitters vorliegt (so dass mit normalen Vektoren für die rechte Seite und die Lösung gearbeitet werden kann). So möchten wir die theoretisch mögliche Spitzenleistung ermitteln, bevor wir die Funktionalität erweitern.

3.2 Speicherstrukturen

Neben der Multiplikations-Funktion gibt es noch eine andere Stellschraube, an der wir drehen können. Wie wir oben bereits angedeutet haben, führen wir neue Formate ein. Dabei nutzen wir das, im letzten Beispiel gewonnene Wissen, über die Speicherallokation, die eine ganz zentrale Rolle bei der Länge der Berechnungszeit darstellt. Wenn wir mit größeren Matrizen rechnen, ist eine effizientere Speichernutzung von Vorteil, da der Cache weniger schnell erschöpft ist. Ist dies der Fall, wird der „DRAM Gap“ erst ab einer höheren Dimension deutlich.

3.2.1 Zeilenrepräsentation

Als erstes führen wir neue Matrix-Formate für die Zeilenrepräsentation ein, um die vielversprechendste Implementierung für weitere Verbesserungen zu finden:

Definition 3.2.1. Hash-Tabelle-Cons-List (HTCL)-Format, die Zeilen sind in einer HT mit folgendem Inhalt:

```
((j0 . v0) (j1 . v1) ...)
```

Definition 3.2.2. Hash-Tabelle-Array-Array (HTAA)-Format, Zeilen-HT mit folgendem Inhalt:

```
#(#(j0 j1 ...) #(v0 v1 ...))
```

Definition 3.2.3. Hash-Tabelle-Array-Cons (HTAC)-Format, Zeilen-HT mit folgendem Inhalt:

```
#((j0 . v0) (j1 . v1) ...)
```

Definition 3.2.4. Hash-Tabelle-Cons-Array (HTCA)-Format, Zeilen-Feld mit folgendem Inhalt:

```
(#(j0 j1 ...) . #(v0 v1 ...))
```

3.2.2 Matrixrepräsentation

Als nächstes werden wir die beste Zeilenrepräsentation in einem umhüllenden Array speichern:

Definition 3.2.5. Array-Array (AA)-Format, Matrix Array mit der Zeilenrepräsentation des HTAC-Formats:

```
#(#((j00 . v00) (j01 . v01) ...)
  #((j10 . v10) (j11 . v11) ...)
  ...)
```

4 Performanz

Wir wählen die Funktion General Matrix Multiplication Method als Gradmesser und untersuchen die Performanz. Als erstes untersuchen wir den Fall, dass die Matrix nur aus skalaren Einträgen besteht. Wir vergleichen die bisherigen generischen Varianten mit den im letzten Kapitel vorgestellten, spezialisierten und optimierten Varianten.

4.1 Performanz im skalaren Fall

Als erstes betrachten wir nur den skalaren Fall, wo alle Matrix-Einträge Gleitkommazahlen sind. Anhand dieses Falles, versuchen wir das bestehende System zu optimieren.

4.1.1 Einführung der Optimierungen für vorhandene Formate

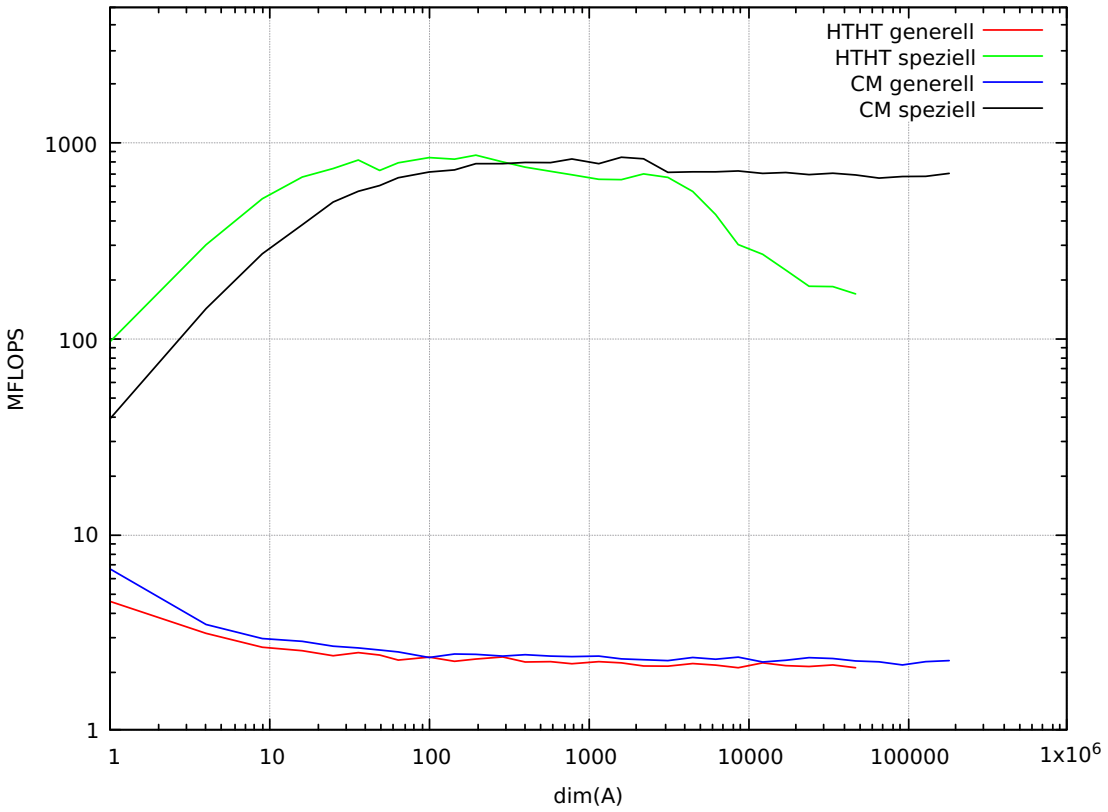

Abbildung 4.1: Vergleich der bisherigen General Matrix Multiplication Method mit der neuen, optimierten und spezialisierten Implementierung. Die Dimension des Gitters ist 2.

Wie in Abbildung 4.1 ersichtlich ist, führt die starke Vereinfachung und Spezialisierung auf Skalare zu einer massiven Erhöhung der MFLOPS Zahl und somit zu einer Verminderung der Berechnungszeit. Dies ist vor allem darauf zurückzuführen, dass teure generische Funktionsaufrufe wegfallen. Vor allem das optimierte CM-Format überzeugt mit einer konstant hohen Performanz. Das HTHT-Formats kann zwar bei kleinen Dimensionen mithalten, der starke Performanzabfall nach $\dim(A) \approx 5 * 10^3$ lässt jedoch einen erhöhten Speicherbedarf vermuten. Dies kann man auch daran erkennen, dass der Speicher bei $\dim(A) \approx 5 * 10^4$ für HTHT im Vergleich zu $\dim(A) \approx 3 * 10^5$ CM überläuft.

4.1.2 Performanz der neuen Speicherstrukturen in der Zeilenrepräsentation

Nun werfen wir einen Blick auf die neu definierten Formate:

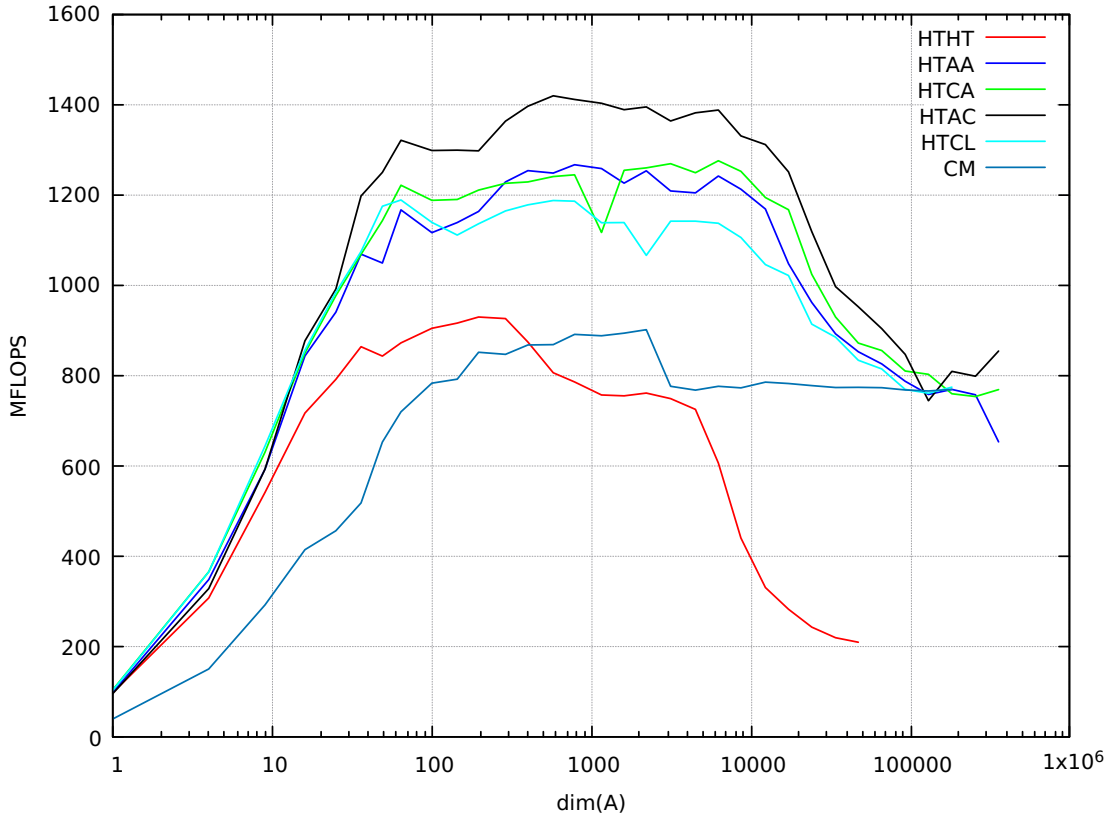

Abbildung 4.2: Vergleich verschiedener Format-Varianten mit dem optimierten HTHT- und CM-Format. Die Dimension des Gitters ist 2.

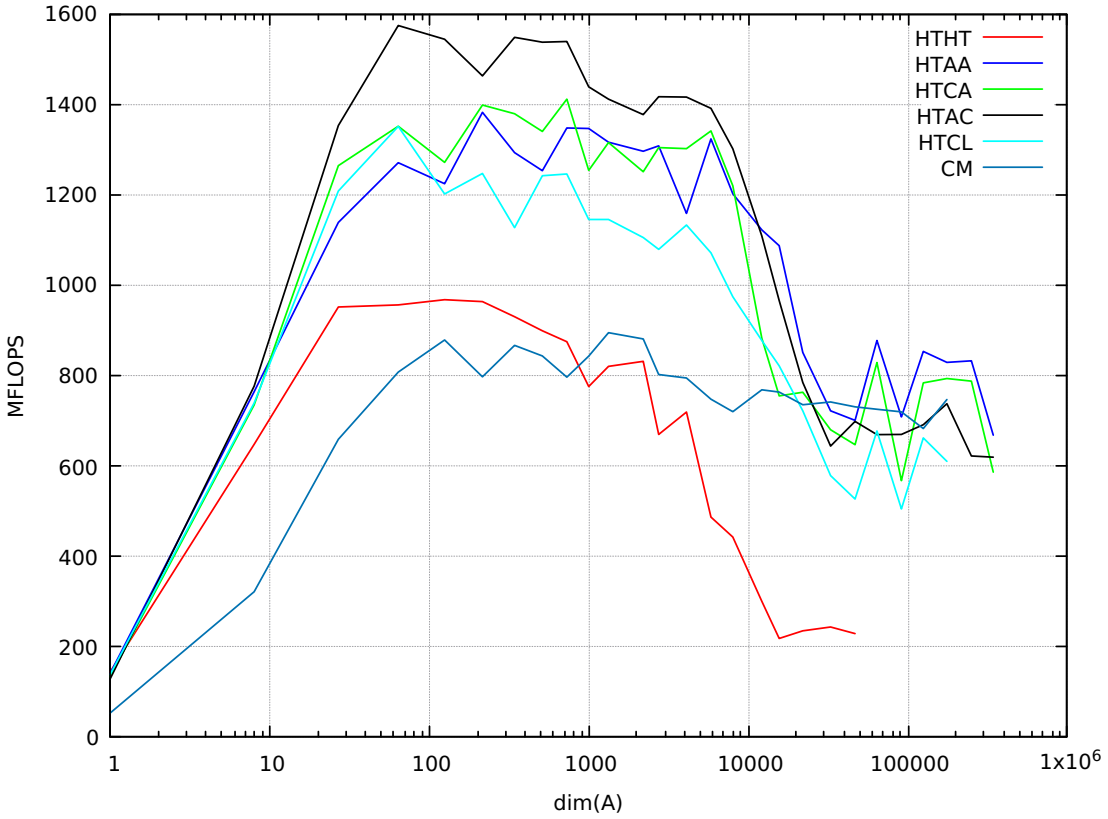

Abbildung 4.3: Gleiches wie in Abbildung 4.2. Die Dimension des Gitters ist 3.

Wie man anhand der theoretischen Grundlagen bereits vermuten würde, sind HT-basierte Formate grundsätzlich langsamer als listenbasierte Implementierungen, welche wiederum langsamer als feldbasierte sind. Dies kann man in Abbildung 4.2 und 4.3 klar erkennen. Etwas enttäuschend ist hier die Performanz des CM-Formats, welches eigentlich schneller als die anderen sein könnte.

4.1.3 Parallelisierung

Eine weitere Option um die Berechnung zu optimieren ist die Parallelisierung. Dies wurde auch hier versucht, leider hat es zu keinem befriedigenden Ergebnis geführt. Die Erklärung liegt sehr wahrscheinlich darin, dass bei unserem Test der Matrix-Vektor Multiplikation die Speicherzugriffe der limitierende Faktor sind und nicht fehlende Rechenkapazität.

4.1.4 Beobachtungen aus Tests mit Skalaren

In Anbetracht der obigen Tests, kann man zwei Dinge erkennen:

1. **Compileroptimierungen ermöglichen enorme Performanzgewinne** Im skalaren Spezialfall erreichen Typdeklarationen und die Kompilierung typspezifischen Codes eine Performanz, welche mehrere hundert mal höher liegt, als nicht optimierte Varianten.

2. **Kein Format dominiert jenseits des „DRAM Gap“** Die Performanz aller Formate gleicht sich bei hohen Dimensionen an. Man kann keines erkennen, welches klare Vorteile bieten würde. Auch hat Parallelisierung zu keiner Verbesserung geführt. Wie oben bereits erwähnt wurde, ist sehr wahrscheinlich die Hardware der limitierende Faktor.

Da das HTAC-Format am besten abgeschnitten hat, verwenden wir diese Zeilenrepräsentation, um das AA-Format, siehe letztes Kapitel, zu komplettieren.

4.2 Performanz mit Blockmatrizen

Zum Schluss wurde die spezialisierte Multiplikations-Funktion modifiziert, damit sie mit Blockmatrix-Einträgen rechnen kann. Als Datenstruktur wurde das neu gebildete AA-Format verwendet. Wir Verglichen es mit der bisherigen Implementierung und kommen zu folgendem Resultat:

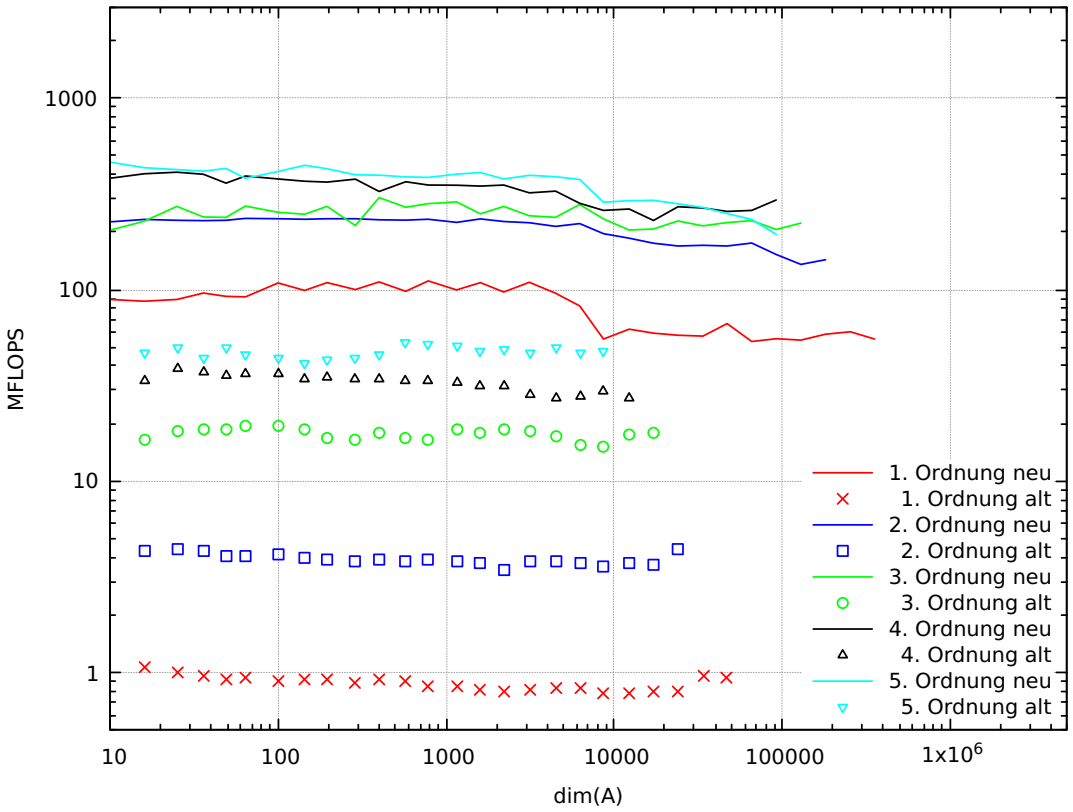

Abbildung 4.4: Vergleich von generischer Multiplikation mit HTHT-Format „alt“ und spezialisierter Multiplikation mit AA-Format „neu“. Ordnung bedeutet die Dimension der Blockmatrix-Einträge. Die Dimension des Gitters ist 2.

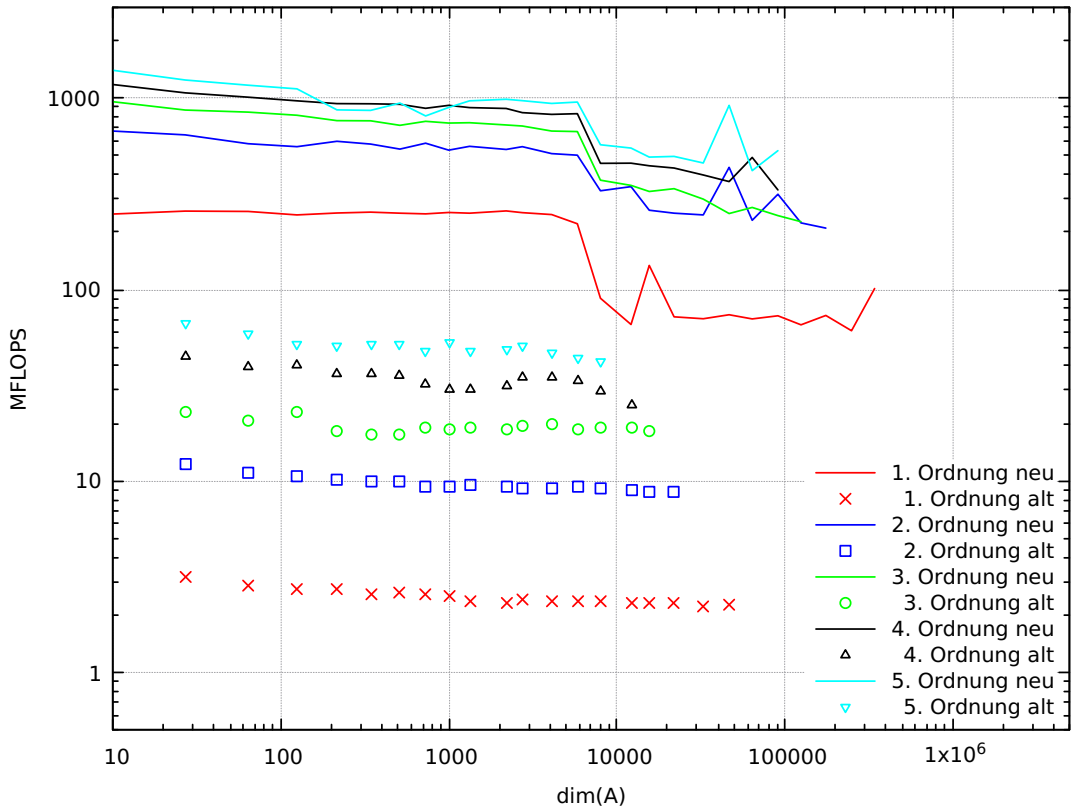

Abbildung 4.5: Gleiches wie in Abbildung 4.4. Die Dimension des Gitters ist allerdings 3.

Wie in Abbildung 4.4 ersichtlich ist, können wir durch eine geschickte Wahl des Formats und ausschließlichem Verwenden von primitiven Datentypen eine Beschleunigung der Berechnung in jedem Fall erreichen. Man beachte auch hier den Abfall von „1. Ordnung neu“ bei rund $\dim(A) \approx 7 * 10^3$, dies lässt wieder den „DRAM Gap“-Effekt vermuten. Es ist ein Zeichen, dass wir uns der Performanzlimite der Hardware nähern, also nicht mehr unsere Implementierung die Performanz begrenzt.

5 Fazit

Wir wollen nun nochmals Stellung zu den am Anfang gestellten Zielen nehmen:

5.1 Rückblick

1. **Implementierung der Zeilentabellen mit Hilfe von Listen**
   Es wurden mehrere neue Formate auf Performanz geprüft, unter anderem auch Listen, welche im skalaren Fall besser als die bestehende Implementierung abgeschnitten haben. Wenn die bestehende GEMM verwendet wird, werden jedoch keine Performanzverbesserungen erzielt. Aus diesem Grund haben wir geschlossen, dass der größte limitierende Faktor der bestehenden Implementierung die Verwendung von GFs ist.

2. **Einführung einer Nummerierung der Gitterelemente**
   Bei unseren Tests sind wir davon ausgegangen, dass eine Nummerierung der Gitterelemente aus primitiven Datentypen vorliegt. Im letzten Test war ersichtlich, dass eine solche Implementierung auf Basis des AA-Formats erheblich schneller als die bisherige Implementierung ist. Allerdings sind die von uns gezeigten Performanzvorteile nur zu realisieren, wenn die genannte Nummerierung vorliegt. Diese Nummerierung der Gitterelemente hat sich aber doch als ein erheblicher Eingriff in die Implementierung von FEMLISP herausgestellt und ist daher im Augenblick noch in Arbeit.

3. **Verwendung einer CCS oder CRS-Datenstruktur**
   Das CM-Format, welches eine solche Datenstruktur verwendet, hat sich zwar als schneller als die bisherige Implementierung erwiesen, die neu entworfenen Formate wie AA waren jedoch überzeugender. Möglicherweise lässt sich dies noch verbessern. Jedoch braucht auch dieses Format eine Nummerierung der Gitterelemente. Der Aufwand zur Implementierung in FEMLISP ist deshalb gleich dem des AA-Formats.

5.2 Ausblick

Aufgrund der gewonnenen Erkenntnisse können wir für die weitere Entwicklung von FEMLISP eine Schlussfolgerung ziehen:

Es ist möglich, die Matrix-Vektor-Multiplikation erheblich zu beschleunigen, der erforderliche Aufwand an Änderungen ist allerdings erheblich.

Abbildungsverzeichnis

Abkürzungen

HT	Hashtabelle
CM	Compressed Matrix
HTHT	Hash-Tabelle-Hash-Tabelle
HTAA	Hash-Tabelle-Array-Array
HTAC	Hash-Tabelle-Array-Cons
HTCA	Hash-Tabelle-Cons-Array
HTCL	Hash-Tabelle-Cons-List
AA	Array-Array
CL	Common Lisp
MFLOPS	Mega Floating Point Operations per Second
FE	Finite-Elemente
FEM	Finite-Elemente-Methode
PDGL	Partielle Differentialgleichung
FED	Finite-Elemente-Diskretisierung
CCS	Compressed Column Storage
CRS	Compressed Row Storage
REPL	Read Evaluate Print Loop
GEMM	General Matrix Multiplication Method
FM	Full Matrix
GF	Generic Function
CLOS	Common Lisp Object System

Literatur

[1] N. Neuß, *Schnelle numberische Approximation von effektiven Parametern mit einer interaktiven Finite-Elemente Umgebung*. Habilitationsschrift, Ruprecht-Karls-Universität Heidelberg, Heidelberg, 2003.

[2] G. Hager und G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. Champman and Hall/CRC , ISBN: 978-1-4398-1192-4, 2011.

[3] FEMLISP Homepage, *FEMLISP - a Common Lisp Framework for Finite Element Methods*. Abgerufen am: 27.09.2020.

[4] P. Seibel, *Practical Common LISP*. Apress, ISBN: 978-1-5905-9239-7, 2005.

[5] E. Süli, *Lecture Notes on Finite Element Methods for Partial DifferentialEquations*. University of Oxford, Oxford, 2020.

[6] P. Graham, *On Lisp*. Prentice Hall, ISBN: 978-0-1303-0552-7, 1993.