Optimierung des Datenzugriffs

Daniel Schicker
Friedrich-Schiller-Universität Jena
schicker.daniel@uni-jena.de

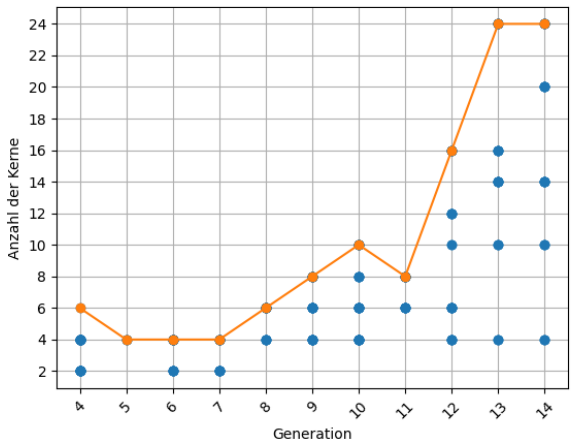
ZUSAMMENFASSUNG

Die Art und Weise, wie auf Daten zugegriffen wird, kann einen erheblichen Einfluss auf die Performance eines Algorithmus haben. Insbesondere wenn große Datenmengen häufig zwischen Hauptspeicher und dem Prozessor ausgetauscht werden, spricht man häufig von einem Engpass des Speicherinterfaces. Durch effiziente Datenzugriffe und optimale Nutzung aller Cache Ebenen lässt sich nicht nur die Leistung signifikant steigern. Selbst bei großen Datenmengen, die die Größe der Cache-Ebenen überschreiten, wird gewährleistet, dass alle Cache Ebenen weiterhin effizient genutzt werden (siehe Blocking). In diesem Paper werden Methoden wie Loop Unrolling, Loop Fusion, Blocking analysiert. Um die Effektivität der Methoden zu demonstrieren, werden diese gebenchmarkt und mit den unoptimierten Implementierungen verglichen.

ACM Reference Format:
Daniel Schicker. 2024. Optimierung des Datenzugriffs. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

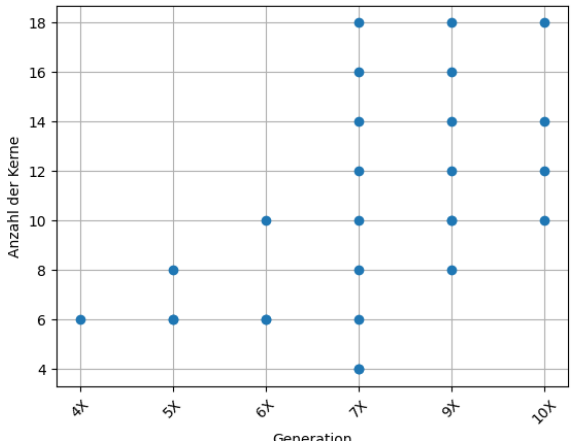
1 EINLEITUNG

Abbildung 1: Anzahl der Kerne pro Prozessor-Generation von Intel (Datenquelle: [?])



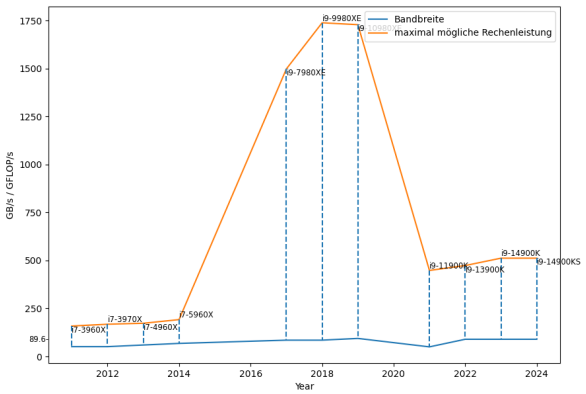
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Abbildung 2: Anzahl der Kerne der X-series pro Prozessor-Generation von Intel (Datenquelle: [?])



In Abbildung 1 und Abbildung 2 ist zu erkennen, dass seit der Einführung der 4th Generation (Haswell) im Jahr 2013 [**?**] sich die Anzahl der Kerne bei Intel, für Desktop Computer, von 6 auf 24 Kernen erhöht hat. Es ist wichtig zu beachten, dass es zwar Prozessoren mit einer noch höheren Anzahl von Kernen gibt, diese jedoch in der Regel für Server bestimmt sind. In der Abbildung **??** ist jedoch zu erkennen, dass die Gigahertz-Werte stagnieren. Wenn

Abbildung 3: Vergleich der Entwicklung der Speicherbandbreite und max. Rechenleistung (Datenquellen: [?])



man nun die theoretische maximale Rechenleistung, definiert als:
P_{max} = Anzahl der Kerne × Turbo Taktfrequenz ×
Flops pro Taktzyklus × Anzahl der Instruktionssets,
und der Entwicklung der Speicherbandbreite gegenüberstellt, wird in Abbildung 3 deutlich, dass die Zunahme der Speicherbandbreite nicht im gleichen Maße wie die Rechenleistung ansteigt. Im Detail hat sich die Bandbreite von dem i7-3960X von 51.2 $\frac{\text{GByte}}{\text{s}}$ im Jahr 2013 auf 89.6 $\frac{\text{GByte}}{\text{s}}$ bei dem i9-14900KS im Jahr 2024 erhöht. Parallel dazu hat sich die theoretische maximale Rechenleistung im gleichen Zeitraum von 158.4 $\frac{\text{Flops}}{\text{s}}$ auf 512 $\frac{\text{Flops}}{\text{s}}$ gestiegen. In Abbildung 3 ist ein Flop definiert als eine Gleitkommazahl-Operation mit einer FP32-Zahl. Die Werte für die drei Prozessoren i9-7980XE, i9-9980XE und i9-10980XE scheinen inkonsistent mit dem erwarteten Entwicklungstrend zu sein. Der Grund dafür ist, dass diese drei Prozessoren über zwei 512-Bit-Instruktionssets verfügen. Dies erhöht das Ergebnis der maximalen Rechenleistung erheblich im Vergleich zu den anderen Prozessoren, die lediglich über ein AVX2 256-Bit-Instruktionsset verfügen. Diese Diskrepanz zwischen der gesteigerten Rechenleistung und der vergleichsweise langsamer wachsenden Speicherbandbreite verdeutlicht die Notwendigkeit einer Optimierung von Datenzugriffen. Ein effizienter Einsatz des Caches ist dabei unerlässlich, um die Leistung zu maximieren. Des Weiteren arbeitet der Intel® AMX, eine spezielle Hardware-Komponente in Intel® Xeon® Scalable Prozessorkernen, daran, Matrix-Mathematik-basierte Deep-Learning-Workloads zu optimieren und zu beschleunigen. Es erreicht dies, indem es mehr Daten pro Kern speichert und größere Matrizen in einer einzigen Operation berechnet, ohne die Notwendigkeit, Workloads auf einen separaten Beschleuniger auszulagern [**?**].

2 DIE BERECHNUNG DER PERFORMANCEGRENZEN

In den nachfolgenden Unterabschnitten werden die Formeln vorgestellt, die zur Bewertung von loop-basiertem Code und zur Berechnung der Performancegrenzen herangezogen werden. Es ist jedoch wichtig zu berücksichtigen, dass diese Formeln lediglich eine Annäherung darstellen und nur unter bestimmten Bedingungen gültig sind. Beispielsweise wird vorausgesetzt, dass alle Ressourcen vollständig ausgeschöpft werden die die CPU zu bieten hat. Zudem basieren die Formeln auf Aspekten des idealen Cache-Modells, wie einem unendlich schnellen Cache und der Nichtexistenz von Latenzzeiten.

2.1 Maschinenbalance

Die Maschinenbalance B_m ist das Verhältnis aus der maximalen Bandbreite und der theoretischen Rechenleistung $B_m = \frac{B_{max}}{P_{max}}$. Sie beschreibt, wie viele Daten pro Flop übertragen werden können. Zur Veranschaulichung betrachten wir die B_m für den i7-9700K Prozessor, der 2018 erschienen ist und eine maximale Bandbreite von 41.6 $\frac{\text{GByte}}{\text{s}}$ sowie eine theoretische Rechenleistung von 8 Kerne × 4.9 GHz × 16 $\frac{\text{Flops}}{\text{Taktzyklus}}$ = 627,2 $\frac{\text{Flops}}{\text{s}}$ besitzt. Die Bandbreite muss noch in Fließkommazahlen umgerechnet werden, die pro Sekunde geladen werden können. Also $\frac{41.6\,\text{GB/s}}{8\,\text{Byte}} = 5.2\ \frac{\text{Fließkommazahlen}}{\text{s}}$. Daraus ergibt sich, dass $B_m = \frac{5.2\,\text{Fließkommazahlen/s}}{627.2\,\text{Flops/s}} \approx 0.008\ \frac{\text{Fließkommazahlen}}{\text{Flop}}$ geladen werden können. Mit anderen Worten, bis eine Fließkommazahl geladen ist, müssen 125 Flops auf einem Wert durchgeführt werden, bis der nächste Wert geladen ist. Die Maschinenbalance bei den neuesten Prozessoren wie dem i9-14900KS liegt bei ≈ 0.0058, und somit wären 172 Flops pro geladene Fließkommazahl erforderlich. Man könnte daher schlussfolgern, dass das Rechnen quasi kostenlos ist und das Laden der Werte den limitierende Faktor darstellt.

2.2 Codebalance

Die Codebalance $B_c = \frac{\text{Datenverkehr}}{\text{Flops}}$ ist das Verhältnis der zu ladenden und speichernden Fließkommazahlen und der Anzahl der Flops in einer loop Iteration. Hierbei zählt man aber nur die load und store Operationen, welche wirklich über den langsamen Datenpfad verläuft, daher wird l_i nicht mitgezählt, weil es sich im Register befindet. In Abbildung 4 ist ein solcher loop dargestellt, welcher 4 Flops ausführt und 3 Elemente aus den drei Arrays X, Y und Z lädt und eine speicher Operation in Z durchführt. Die Codebalance ist also $B_c = \frac{3+1}{4} = 1$.

2.3 Berechnung der zu erwartenden Performance

Um die maximal erreichbare Performance eines loop-basierten Codes zu berechnen, bestimmt man den Anteil der maximalen CPU-Performance, die tatsächlich erreicht werden kann. Dieser Anteil wird als „lightspeed"l bezeichnet und ist definiert als $l = \min(1, \frac{B_m}{B_c})$. Da diese Formel auf den ersten Blick nicht intuitiv ist, kann ein Beispiel zur Veranschaulichung dienen. Betrachten wir erneut den Code aus Abbildung 4. Die Maschinenbalance B_m beträgt 0.5 $\frac{\text{Fließkommazahlen}}{\text{Flop}}$, während die Codebalance B_c 1 beträgt, da vier Fließkommazahlen geladen und gespeichert werden müssen und vier Flops ausgeführt werden. Da das Bereitstellen und Abspeichern von Fließkommazahlen nicht so schnell ist wie das Rechnen (was durch $B_m = 0.5$ ausgedrückt wird), kann man in der Zeit, in der man vier Flops ausführen würde, nur halb so viele, also $4 \times 0.5 = 2$ Fließkommazahlen laden und/oder abspeichern. Daher ist die maximale Performance, die erreicht werden kann, $l = \min(1, \frac{0.5}{1}) = 0.5$.
Oder anders betrachtet, es könnte einen Code geben, der doppelt so viele Flops ausführt wie er Fließkommazahlen lädt und speichert. Dies würde zu einer Codebalance von 0.5 führen. In einem solchen Szenario wäre die Anzahl der zu ladenden und speichernden Fließkommazahlen genau gleich der theoretisch möglichen Anzahl an Fließkommazahlen, die geladen und gespeichert werden könnten. Somit würde der maximal zu erreichende Anteil an P_{max} bei $l = \min(1, \frac{0.5}{0.5}) = 1$ liegen. Wenn dieser Wert nah an 1 liegt ist man nicht Speichergebunden. Die maximal erreichbare Performance ist somit $P = l \times P_{max}$ oder $P = \min(P_{max}, \frac{b_{max}}{B_c})$.

3 LOOP FUSION

Die erste Methode, die zur Optimierung des Zugriffsverhaltens vorgestellt wird und bei der anhand des präsentierten Modells eine Verbesserung erkennbar ist, ist Loop Fusion. Es ist wichtig zu verstehen, dass der Cache ein kleiner, schneller Speicher ist, der die Least Recent Used (LRU) Policy verwendet, um zu entscheiden, welche Daten im Cache verbleiben. Dies bedeutet, dass

die am längsten nicht genutzten Daten aus dem Cache entfernt werden, wenn eine neue Cache Line geladen werden muss und kein Platz mehr im Cache vorhanden ist. Dies beschreibt die zeitliche Lokalität. Wenn eine Cache Line geladen wird und die umliegenden Daten ebenfalls in den Cache geladen werden, spricht man von räumlicher Lokalität, da man davon ausgeht, dass diese Daten in naher Zukunft ebenfalls genutzt werden. Typischerweise beträgt die Größe einer Cache-Zeile 64 Bytes, kann aber durch die Befehle `getconf LEVEL1_DCACHE_LINESIZE`, `getconf LEVEL2_CACHE_LINESIZE`, `getconf LEVEL3_CACHE_LINESIZE` leicht bestimmt werden. Um die Größe des Caches zu ermitteln, kann man den Befehl `lstopo` verwenden, sofern installiert. Betrachtet man Abbildung 5, ist leicht zu erkennen, dass zunächst in einer for-Schleife die Elemente aus dem Array A mit dem Skalar 2.0 multipliziert und in das Array Z gespeichert werden. Anschließend wird eine weitere for-Schleife ausgeführt, die das Gleiche tut, nur dass hier das Ergebnis in Y gespeichert wird. Wenn die Arrays zu groß sind und am Ende der ersten for-Schleife die Cache Line, die die ersten Elemente aus A enthält verdrängt wurde, müssen diese für die zweite for-Schleife erneut in den Cache geladen werden. Daten, die in den Cache geladen und aus Platzgründen wieder verdrängt wurden, aber später wieder benötigt und erneut in den Cache geladen werden, nennt man `Capacity Miss`. Das erstmalige Laden von Daten über langsame Datenpfade wird als `Cold Miss` bezeichnet. Die Code-Balance B_c der beiden for-Schleifen beträgt $\frac{2}{1} = 2$. Wenn man jedoch die beiden for-Schleifen zusammenführt (siehe Abbildung 6), spart man sich einen langsamen Load, da der Wert von A[i] in der Zeile zuvor in den Cache geladen wurde, wodurch sich die Code-Balance auf $\frac{3}{2} = 1.5$ reduziert. Um das Modell zu überprüfen, wurden beide Varianten gebenchmarkt. Die Ergebnisse sind in Abbildung 7 dargestellt. Man erkennt, dass die Loop-Fusion-Variante schneller ist als die nicht fusionierte Variante.

4 DER STREAM BENCHMARK

Der STREAM Benchmark besteht aus vier Kernels, die die Bandbreite des Speicherinterfaces meist in GB/s messen sollen. Die vier Kernels sind Copy, Scale, Add und Triad wobei in der Tabelle auch die Codebalance nochmal aufgeführt ist.

Kernelname	Operation	Code-Balance
Copy	A[i] = B[i]	-
Scale	A[i] = q*B[i]	2/1
Add	A[i] = B[i] + C[i]	2/1
Triad	A[i] = B[i] + q*C[i]	3/2

Tabelle 1: STREAM Benchmark Kernels und ihre Code-Balance

Für die Ausführung des STREAM-Benchmarks ist es wichtig, dass die Datenmengen ausreichend groß sind. AMD empfiehlt, dass die Arrays mindestens viermal so groß sein sollten wie die Summe aller Last-Level-Caches. Im vorherigen Abschnitt wurde der Scale-Kernel genutzt, um die Effektivität der Loop-Fusion zu demonstrieren. Dabei wurde die Zeit jedoch in Nanosekunden anstelle von GB/s gemessen. Da diese synthetischen Kernels die Hardware direkt testen, sind die erreichten Bandbreitenwerte aussagekräftigere Vergleichswerte als die, die man beispielsweise auf den Webseiten von AMD oder Intel findet, um die Leistung von Code und Hardware einzuschätzen. Abbildung 8 stellt die Ergebnisse des Triad-Kernels auf einem i7-9700K dar. Es ist erkennbar, dass die Bandbreite mit jedem Cache-Level abnimmt, wobei der Rückgang bei dem L3-Cache besonders stark ist. Derselbe Benchmark wurde auch auf dem ARA-Cluster auf einem der FSU Jena durchgeführt. Die Ergebnisse, die in Abbildung 9 dargestellt sind, verdeutlichen nochmals, wie die Bandbreite von Cache-Level zu Cache-Level abnimmt. Sind Ergebnisse eines passenden Tests vorhanden sollte man also die vorhin vorgestellte Formel zur Berechnung der maximalen erreichbaren Performance zu $P = \min(P_{max}, \frac{b_{Stream}}{B_c})$ abändern.

5 DIE OPTIMIERUNG DES SPEICHERZUGRIFFSMUSTERS

In Computern werden Daten linear im Speicher abgelegt, das heißt, sie werden in einer sequenziellen Reihenfolge gespeichert, wobei zum Beispiel ein Byte eine eindeutige Adresse hat, die auf den genauen Speicherort verweist. Diese lineare Anordnung ermöglicht es, effizient auf Daten zuzugreifen, indem man die Adresse des gewünschten Datenstücks kennt oder berechnet. Greift man auf ein Element zu, dann wird die zugehörige Cache Line geladen, wenn die Cache Line sich noch nicht im Cache befindet. Im besten Fall werden anschließend alle Elemente der Cache Line abgerufen dann die Cache Line verdrängt und nie wieder benötigt. Im schlechtesten Fall wird die Cache Line verdrängt bevor alle Elemente abgerufen wurden oder sogar nur ein Element genutzt, wodurch der Zweck des Caches verfehlt wird. In Abbildung 10 ist ein Beispiel für ein 2D Array in Row-Major-Order dargestellt. Würde man in dieser Matrix Spaltenweise auf die Elemente zugreifen, dann würde jeder Zugriff das Laden einer neuen Cache Line erfordern, weil man restlichen Elemente der Zeile überspringt. Die Schrittweite von einem Zugriff nennt man Stride. Überspringt man beispielsweise n Elemente, dann dann ist das ein Stride von n Elementen. Um die Auswirkungen des Speicherzugriffsmusters zu demonstrieren, sind hier (klicke hier) zwei C++ Varianten einer Matrix Matrix Multiplikation dargestellt. Die Elemente der Matrizen sind in Row Major abgespeichert und die Matrizen sind quadratisch und gleich groß.

Auflistung 1: Zeilenorientierte Matrix Matrix Multiplikation

```
1  void gemm_mkn(double* X, double* Y, double* Z, std::size_t n){
2
3  for (std::size_t l_m = 0; l_m < n; l_m++)
4  {
5      for (std::size_t l_k = 0; l_k < n; l_k++)
6      {
7          for (std::size_t l_n = 0; l_n < n; l_n++)
8          {
9             Z[l_m * n + l_n] += X[l_m * n + l_k] * Y[l_k * n + l_n];
10         }
11     }
12 }
13 }
14
```

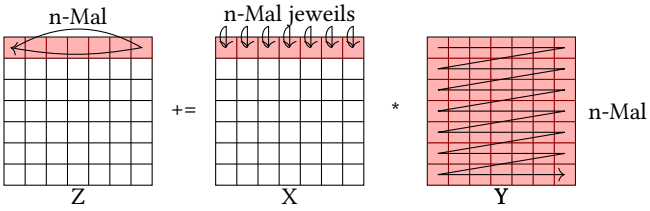
Die Elemente von Z, X und Y werden für die Matrix Matrix Multiplikation mit einem Stride-N-Muster abgerufen.
Die Elemente von Z, X und Y werden für die Matrix Matrix Multiplikation mit einem Stride-N-Muster abgerufen.

Auflistung 2: Zeilenorientierte Matrix Matrix Multiplikation

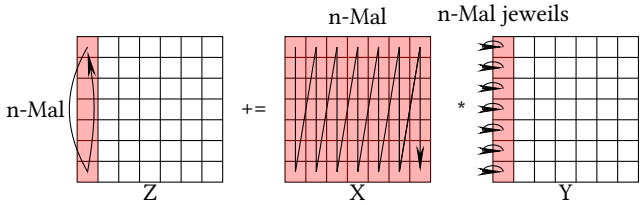
```
void gemm_nkm(double* X, double* Y, double* Z, std::size_t n){

for (std::size_t l_n = 0; l_n < n; l_n++)
{
    for (std::size_t l_k = 0; l_k < n; l_k++)
    {
        for (std::size_t l_m = 0; l_m < n; l_m++)
        {
            Z[l_m * n+ l_n] += X[l_m * n + l_k] * Y[l_k * n + l_n];
        }
    }
}
}
```

Um zu entscheiden welcher der beiden Implementierungen effizienter ist, ohne sie zu benchmarken kann man sich jeweils die Zugriffsmuster ansehen. In der ersten Implementierung wird auf die Elemente von Z mit einem Stride-1 zugegriffen, dabei wird die selbe Zeile n-Mal durchlaufen bis die nächste Zeile bearbeitet wird. Auf das gleiche Element von X wird n-Mal zugegriffen, bevor mit einem Stride-1 auf das nächste Element zugegriffen wird. Dabei bleibt man bei X auch in der selben Zeile, also in der selben Cache Line. Bei Y wird sich zeilenweise durch die Matrix gearbeitet und dies n-Mal wiederholt. Die Abbildung 5 zeigt nochmal das Verhalten des Algorithmus. Es ist wichtig zu beachten, dass eine Cache Line deutlich kleiner ist als eine Zeile/Spalte und dass dies lediglich ein Modell darstellt, keine direkte Abbildung einer Matrix-Matrix-Multiplikation.

Abbildung 5: Visualisierung der ersten Implementierung

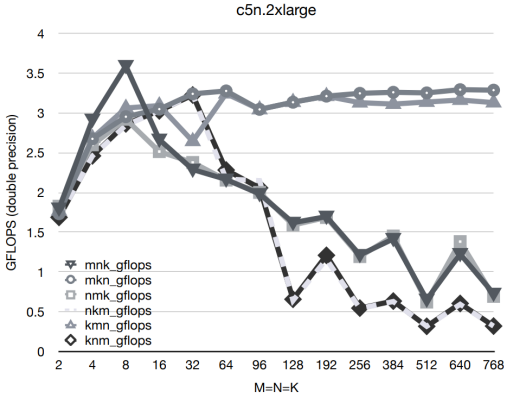


In der zweiten Implementierung 5 wird immer die Zeile übersprungen, wenn ein neues Element geladen wird. Dies führt dazu, dass die zweite Implementierung bei jedem neuen Element eine neue Cache Line laden muss. In Abbildung 5 ist das Verhalten der zweiten Implementierung dargestellt. Es ist wichtig zu beachten, dass eine Cache Line deutlich kleiner ist als eine Zeile/Spalte und dass dies lediglich ein Modell darstellt, keine direkte Abbildung einer Matrix-Matrix-Multiplikation. **Abbildung 12: Visualisierung der zweiten Implementierung**



Bei ausreichend großen Matrizen kommt es zu ständigen Capacity Misses. Die Anzahl dieser kann einfach approximiert werden, indem man den einfachen Fall betrachtet, dass jeweils eine Zeile oder Spalte für Z, X und Y im Cache gehalten werden kann. Es gibt $\frac{n \times n}{L}$ Cache Lines. Das erstmalige Laden jeder Cache Line für Z ist ein Cold Miss, aber die restlichen Zugriffe auf die gleiche Cache Line sind Capacity Misses. Das bedeutet, es gibt $n * n * n - \frac{n \times n}{L}$ Capacity Misses bei Z, wobei L die Anzahl der Elemente in einer Cache Line ist. Bei X ist es ein ähnlicher Fall, da auch jedes Element nach einmaliger Nutzung in naher Zukunft verdrängt wird und später wieder benötigt wird. Es gibt also ebenfalls $n * n * n - \frac{n \times n}{L}$ Capacity Misses bei X. Bei Y wird jedes Element n-Mal genutzt, bevor die Cache Line verdrängt wird. Daher kann man nur für jedes Element einen Capacity Miss zählen, mit Ausnahme des ersten Elemente jeder Cache Line. Anders ausgedrückt, es gibt $L - 1$ Capacity Misses pro Cache Line. Da es $\frac{n \times n}{L}$ Cache Lines gibt, sind insgesamt $\frac{n \times n}{L} \times (L - 1)$ Capacity Misses bei Y zu verzeichnen. Insgesamt ergeben sich also $2 * n * n * n - \frac{n \times n}{L} + \frac{n \times n}{L} \times (L - 1)$ Capacity Misses. Bei der ersten Implementierung hingegen gibt es keine Capacity Misses bei Z, da man stets eine Zeile für Z im Cache halten kann, ebenso wie bei X. Bei Y durchläuft man die Matrix zwar auch in Row-Major-Order, jedoch wird die erste Zeile nach der Bearbeitung der letzten verdrängt sein. In der ersten Implementierung iteriert man n-Mal durch die Matrix Y. Der erste Durchlauf führt ausschließlich zu Cold Misses, während in den folgenden Durchläufen lediglich das erste Element jeder Cache Line einen Capacity Miss verursacht. Daher ergibt sich eine Gesamtzahl von $(n - 1) \times \frac{n \times n}{L}$ Capacity Misses. Da die Anzahl der Capacity Misses bei der zweiten Implementierung deutlich höher ist, wird erwartet, dass die erste Implementierung, insbesondere bei großen Werten von n, eine bessere Performance aufweist. In den Abbildungen 4 ist die Performance der beiden Implementierungen und die 4 restlichen Möglichkeiten der Matrix-Matrix Multiplikation dargestellt.

Abbildung 4: Bandbreitenergebnisse der verschiedenen Implementierungen der Matrix-Matrix Multiplikation

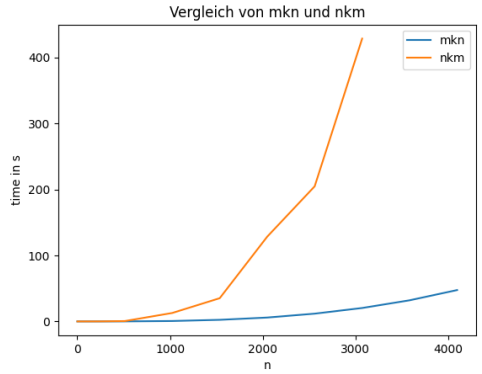


(Quelle : [?])

Die zeitlichen Werte für die beiden vorgestellten Implementierungen 1 und 5, die auf dem i7-9700K Prozessor erzielt wurden, sind in Abbildung 5 dargestellt.

In beiden Abbildungen ist zu erkennen das die Analyse der Zugriffsmuster und die Berechnung der Capacity Misses einen guten

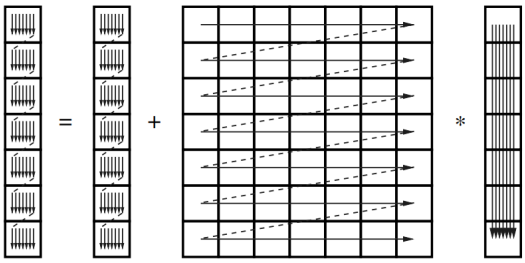
Abbildung 5: Zeitliche Ergebnisse der verschiedenen Implementierungen der Matrix-Matrix Multiplikation



Hinweis darauf geben können welcher Code effizienter arbeitet und das das Zugriffsmuster starken Einfluss auf die Leistung hat.

6 UNROLL AND JAM

Abbildung 6: Zugriffsmuster von einer Matrix Vektor Multiplikation [?]



Auflistung 3: Zeilenorientierte Matrix Matrix Multiplikation

```
1     void MVM(double* A, double* B, double* C, std::size_t n){
2
3         for (std::size_t l_i = 0; l_i < n; l_i++)
4         {
5             for (std::size_t l_j = 0; l_j < n; l_j++)
6             {
7                 C[l_i] += A[l_i * n + l_j] * B[l_j];
8             }
9         }
10    }
11
```

Gegeben ist eine unoptimierte Implementierung einer Matrix-Vektor-Multiplikation, die in Abbildung 6 dargestellt ist und analysieren den zugehörigen Code in Auflistung 4. Jedes Element von C wird n-Mal wiederholt verwendet, bevor das nächste Element mit einem Stride-1 geladen wird. Array A wid ein mal durchlaufen aber für jede äußere Schleifeniteration wird einmal durch das Array B iteriert. Um die Anzahl der Ladevorgänge für B zu reduzieren, kann man die Technik Unroll and Jam anwenden. Dabei wird die äußere Schleife nicht mehr um eins, sondern um m inkrementiert.

In der inneren for-Schleife wird dann der Code m-mal repliziert. Dies bedeutet, dass für jede Iteration der äußeren Schleife m Berechnungen gleichzeitig durchgeführt werden. Das Ergebnis

Auflistung 4: Implementierung der Matrix-Vektor-Multiplikation mit Unroll and Jam

```
1   void MVM(double* A, double* B, double* C, std::size_t n, std::size_t l_m){
2
3   for (std::size_t l_i = 0; l_i < n;l_i = l_i + l_m)
4   {
5       for (std::size_t l_j = 0; l_j < n; l_j++)
6       {
7           C[l_i] = C[l_i] + A[l_i * n + l_j] * B[l_j]
8           C[l_i] = C[l_i+1] + A[(l_i+1) * n + l_j] * B[l_j]
9           //! m times ...
10          C[l_i + l_m-1] = C[i+m-1] + A[(l_i+m-1) * n + l_j] * B[l_j]
11      }
12  }
13
14  }
15
```

Wir nehmen der Einfachheit halber an, dass n%m = 0 und dass m = 0 nicht zu groß gewählt wird. All dies setzt jedoch voraus, dass der Registerdruck nicht zu groß ist, also dass die CPU genügend Register hat, um alle benötigten Operanden in dem nun größeren Schleifenkörper zu halten. Wenn dies nicht der Fall ist, muss der Compiler Registerdaten in den Cache auslagern, was die Berechnung verlangsamt.Hier können Compiler-Protokolle helfen, eine solche Situation zu identifizieren [?]. In Auflistung 4 wird nun jedes Element von B m-mal verwendet in der inneren Schleife, bevor das nächste Element geladen wird. Anders gesagt reduziert sich die Anzahl der Ladevorgänge für B von $n \times n$ auf $\frac{n \times n}{m}$.

7 DAS TRANSPONIEREN EINER MATRIX

Auflistung 5: unoptimierte Version: Transponieren einer Matrix

```
void transpose(double* A, double* B, std::size_t n)
{
  for (std::size_t l_y = 0; l_y < n; l_y++)
  {
      for (std::size_t l_x = 0; l_x < n; l_x++)
      {
          B[l_x * n + l_y] = A[l_y * n + l_x];
      }
  }
}
```

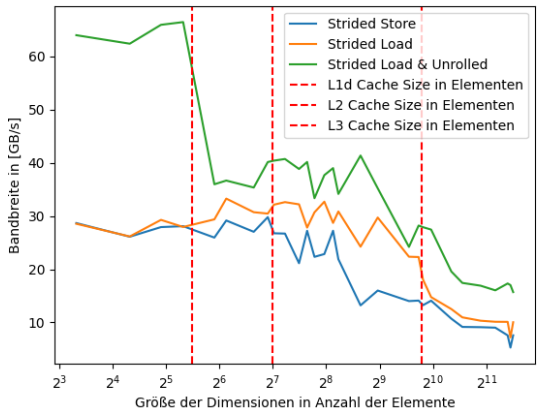
Die intuitive Lösung zum Transponieren einer beispielsweise quadratischen Matrix ist in Auflistung 5 dargestellt. (Die Elemente sind in Row-Major-Order abgespeichert)

Diese Implementierung durchläuft Matrix A zeilenweise und fügt Elemente spaltenweise in Matrix B ein, was zu strided Store-Operationen führt. Diese sind in der Regel teurer als strided Load-Operationen, da bei einem Cache-Miss die Cache-Zeile geladen und nach einer Änderung zurück in den langsameren Speicher geschrieben werden muss, statt sie einfach zu verwerfen.

Auflistung 6: strided load Version: Transponieren einer Matrix

```
1  void transpose(double* A, double* B, std::size_t n)
2  {
3      for (std::size_t l_y = 0; l_y < n; l_y++)
4      {
5          for (std::size_t l_x = 0; l_x < n; l_x++)
6          {
7              B[l_y * n + l_x] = A[l_x * n + l_y];
8          }
9      }
10 }
11
```

Abbildung 7: Bandbreitenergebnisse



Die erste Optimierung besteht darin, in Zeile 7 die Load-Operation strided zu wählen siehe Auflistung 6. Die Elemente von A werden nun spaltenweise geladen und in B zeilenweise gespeichert. In der Abbildung 7 ist der Vergleich der Implementierungen auf dem i7-9700K dargestellt und es ist zu erkennen, dass die strided Load Implementierung im Allgemeinen eine höhere Bandbreite aufweist. Die effektive Bandbreite bei der Verarbeitung großer Datenmengen, die den L3-Cache überschreiten, beträgt etwa 10 GB/s. Ein Hauptproblem bei der Arbeit mit großen Dimensionen besteht darin, dass der Cache ständig überschrieben wird und Cachezeilen verdrängt werden, obwohl sie in naher Zukunft erneut benötigt werden könnten.
Bei großen Dimensionen führt das Stride-n Zugriffsmuster bei A häufig zu einem Cache-Miss, was das Laden einer neuen Cachezeile erfordert. Diese wird dann jedoch nur für ein einziges Element verwendet, was die Bandbreite bei großen Dimensionen erheblich reduziert.
Eine mögliche Lösung zur Verbesserung der Cache-Nutzung besteht offensichtlich darin, mehr als ein Element pro Laden einer Cachezeile bei A zu verwenden. Dies kann durch die zuvor vorgestellte Unroll-and-Jam-Technik erreicht werden. Dabei wird bei jeder inneren Operation m Elemente gleichzeitig bei A aus meist nur einer Cache Line genutzt und in B gespeichert. Der Preis dafür ist aber dass B ständig mindestens m Cache Lines hält. In Auflistung **??** ist eine Implementierung des Transponierens einer Matrix mit

Auflistung 7: Unroll und jam Version von: Transponierens einer Matrix

```
1  for (size_t l_j = 0; l_j < array_size; l_j = l_j + 5)
2  {
3      for (size_t l_i = 0; l_i < array_size; l_i++)
4      {
5          array_B[l_j * array_size + l_i] = array_A[l_i * array_size + l_j];
6          array_B[(l_j + 1) * array_size + l_i] = array_A[l_i  * array_size + l_j + 1];
7          array_B[(l_j + 2) * array_size + l_i] = array_A[l_i  * array_size + l_j + 2];
8          array_B[(l_j + 3) * array_size + l_i] = array_A[l_i  * array_size + l_j + 3];
9          array_B[(l_j + 4) * array_size + l_i] = array_A[l_i  * array_size + l_j + 4];
10     }
11 }
```

Unroll and Jam dargestellt, wobei hier m = 5 gewählt wurde. In Abbildung **??** kann man erkennen, dass diese Implementierung eine höhere Bandbreite aufweist also die strided Load Implementierung. Vor allem wenn die Matrizen zusammen größer sind als der last level Cache ist die Unroll and Jam Implementierung circa 50 Prozent besser. Um genau zu sein hat sich die Bandbreite von 10.0312 auf 15.6997 GB/s erhöht.