

# Data Access

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

## ABSTRACT

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 langsamen Speichermedien und dem Prozessor ausgetauscht werden, spricht man häufig von einem Engpass des Speicherinterfaces. Durch effiziente Datenzugriffe und optimale Nutzung des Caches kann nicht nur die Performance signifikant gesteigert werden, sondern es wird auch sichergestellt, dass die Leistungsfähigkeit selbst bei der Verarbeitung großer Datenmengen stabil bleibt. In diesem Paper werden Methoden wie Loop Unrolling, Loop Fusion, Blocking analysiert. Um die Effektivität mancher Methoden zu demonstrieren, werden diese gebenchmarkt und mit den Standardmethoden verglichen.

## ACM Reference Format:

Daniel Schicker. 2024. Data Access. In . ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 EINLEITUNG

Im letzten Jahrzehnt hat sich die Rechenleistung von Prozessoren erheblich gesteigert. In Abbildung 1 und 1.1 ist zu erkennen, dass seit der Einführung der 4th Generation im Jahr 2013 [1] sich die Anzahl der Kerne bei Intel, die für den allgemeinen Verbrauchermarkt verfügbar sind, 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 nicht für den Standard-Endverbraucher bestimmt sind. Ebenso zeigt sich in den Abbildungen 2 und 2.1 ein Anstieg der maximalen Taktfrequenz über die Jahre. Wenn man nun die theoretische maximale Rechenleistung, definiert als:

$P_{\max} = \text{Anzahl der Kerne} \times \text{Turbo Taktfrequenz} \times \text{Flops pro Taktzyklus}$ ,  
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  $51.2 \frac{\text{GByte}}{\text{s}}$  im Jahr 2013 auf  $89.6 \frac{\text{GByte}}{\text{s}}$  im Jahr 2024 erhöht. Parallel dazu ist die Performance im gleichen Zeitraum von  $187.2 \frac{\text{Flops}}{\text{s}}$  auf  $1945.6 \frac{\text{Flops}}{\text{s}}$  gestiegen. 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.

## 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 Kernen  $\times 4.9 \text{ GHz} \times 16 \frac{\text{Flops}}{\text{Taktzyklus}} = 627,2 \frac{\text{Flops}}{\text{s}}$  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  $\approx 0.0058$ , 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 laden 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 Speicheroperation in Z durchführt. Die Codebalance ist also  $B_c = \frac{3+1}{4} = 1$ .

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](mailto: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>

## 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 Speicher gebunden. 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

**Table 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  $P = \min(P_{\max}, \frac{b_{\text{Stream}}}{B_c})$  abändern.

## 5 DAS BEACHTEN DES SPEICHERZUGRIFFSMUSTERS

Bei zwei Dimensionalen Arrays bestimmt die Programmiersprache die Reihenfolge in der die Elemente im Speicher abgelegt werden. Fortran, Julia und R werden beispielsweise in Column-Major-Order abgelegt, während C++ in Row-Major-Order abgelegt wird. Greift man auf ein Element zu, dann wird die zugehörige Cache Line geladen. Im besten Fall werden anschließend alle Elemente

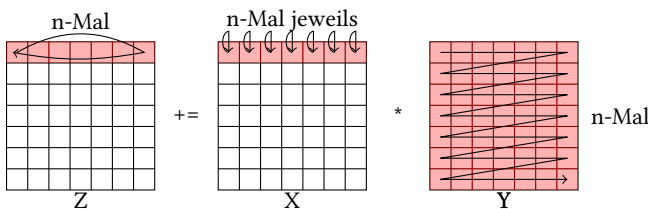
der Cache Line abgerufen dann die Cache Line verdrängt und nie wieder benötigt. Im schlechtesten Fall wird die Cache Line verdrängt bevor alle Elemente abgerufen wurden oder sogar nur ein Element genutzt, wodurch der Zweck des Caches verfehlt wird. In Abbildung 10 ist ein Beispiel für ein 2D Array in 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, was man Strided Zugriff nennt, weil man restlichen Elemente der Zeile überspringt. Um die Auswirkungen des Speicherzugriffsmusters zu demonstrieren, sind hier zwei C++ Varianten einer Matrix Vektor Multiplikation dargestellt. Die Elemente der Matrizen sind in Row Major abgespeichert und die Matrizen sind quadratisch und gleich groß.

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

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 greift Z mit einem Stride-1 auf die Elemente zu, dabei wird die selbe Zeile n-Mal durchgelaufen bis die nächste Zeile bei Z bearbeitet wird. X greift n-Mal auf das selbe Element zu bevor es mit einem Stride-1 auf das nächste Element zugreift. Man bleibt somit bei X auch in der selben Zeile also in der selben Cache Line. Y arbeitet sich Zeilenweise durch die Matrix und wiederholt jenes n-Mal. Die folgende Abbildung zeigt nochmal das Verhalten des Algorithmus.

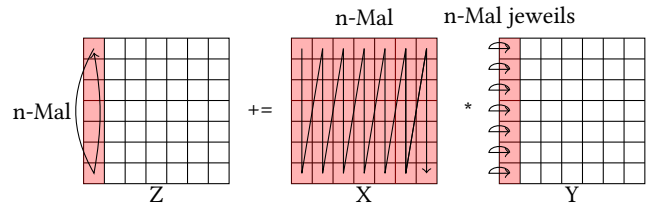
**Abbildung 11: Visualisierung der ersten Implementierung**



In der zweiten Implementierung 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 12 ist das Verhalten der zweiten Implementierung dargestellt.

**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 13 ist die Performance der beiden Implementierungen und die 4 restlichen Möglichkeiten der Matrix-Matrix Multiplikation dargestellt.

**Abbildung 13: Performance der verschiedenen Implementierungen der Matrix-Matrix Multiplikation**

Die zeitlichen Werte für die beiden vorgestellten Implementierungen, die auf dem i7-9700K Prozessor erzielt wurden, sind in Abbildung 14 dargestellt. **Abbildung 14: Vergleich der beiden Implementierungen der Matrix-Matrix Multiplikation auf dem i7-9700K**

In beiden Abbildungen ist zu erkennen das die Analyse der Zugriffsmuster und die Berechnung der Capacity Misses einen guten Hinweis darauf geben können welcher Code effizienter arbeitet und das das Zugriffsmuster starken Einfluss auf die Performance hat.

## **ACKNOWLEDGEMENTS**

## **REFERENCES**

- [1] Louis. 2021. Intel's New 4th Generation Processors Coming Soon to Stealth. <https://www.stealth.com/intels-new-4th-generation-processors-coming-soon-to-stealth/>