Serielle Optimierung der Matrix Multiplikation auf CPUs - Protokoll

Max Kurze, Vincent Melisch

December 2022

Alle Messungen wurden auf einer Intel(R) Xeon(R) E5-2680 v3 CPU durchgeführt.

Aufgabe 1

- FPPP, Single Precision, AVX, ohne Turbo:

$$12\ Cores\ \cdot\ 2.1\ \cdot\ 10^9\ \frac{Instructions}{Second}\ \cdot\ \frac{256\ Bit}{32\ Bit}\ (AVX)\ \cdot\ \frac{2\ FPUs}{Core}\ \cdot\ 2\ (FMA)$$
$$= 806,4 \cdot 10^9 FLOPS$$

- FPPP, Single Precision, AVX, mit Turbo:

$$12\ Cores\ \cdot\ 2.8\ \cdot\ 10^9 \frac{Instructions}{Second}\ \cdot\ \frac{256\ Bit}{32\ Bit}\ (AVX)\ \cdot\ \frac{2\ FPUs}{Core}\ \cdot\ 2\ (FMA)$$
$$= 1075,2\ \cdot\ 10^9\ FLOPS$$

Aufgabe 2

Implementierungen der einzelnen Techniken sind in Aufgabe 7 zu finden.

- Schleifenvertauschung

  Aufgrund von Lokalität, wird beim Laden einer Variablen aus dem Hauptspeicher nicht nur diese, sondern auch benachbarte Daten in den Cache geladen. Daher ist es performant, Code zu schreiben, der auf im Speicher benachbarte Daten nacheinander zugreift. Um das umzusetzen, kann eine Vertauschung der Schleifen helfen, um Zugriffsmuster anzupassen. (*Eine genaue Erklärung findet sich in Aufgabe 7*)

- Loopunrolling
  Beim Loopunrolling wird der Code von N Schleifeniterationen in einer einzigen Schleifeniteration durchgeführt. Natürlich muss dafür die Schrittweite der Schleife auf N gesetzt werden. Durch das Unrollen sparrt man sich den Schleifenoverhead (teste, ob n ¡ BOUNDARY, springe) und entlastet somit auch die Branch-Prediction.

- Blocking/Tiling Eine große Matrixmultiplikation wird in viele kleine Matrixmultiplikationen aufgebrochen. So lässt sich beispielsweise eine Multiplikation von zwei 4 x 4 Matrizen in die in die Berechnung von vier 2 x 2 Matrizen aufteilen. Die Berechnungen der kleinen Teil-Matrizen sind dabei unabhängig voneinander und können somit gut parallelisiert werden.
  Beim

- Wiedernutzung von Werten Werten statt Neuberechnung
  Indem man Zischenergebnisse, die mehrfach gebraucht werden, in separaten Variablen abspeichert, kann die Ausführungszeit verringert werden, da Werte nicht neu berechnet werden müssen.

Aufgabe 3

Referenzimplementierung

```
extern void matrix_mult(float a[SIZE][SIZE], float b[SIZE][SIZE], float
↪  res[SIZE][SIZE]) {
  for (int r = 0; r < SIZE; r++)
    for (int c = 0; c < SIZE; c++)
      for (int i = 0; i < SIZE; i++)
        res[r][c] += a[r][i] * b[i][c];
}
```

Listing 1: default.c

Formel zur Berechnung der Matrixgröße SIZE:

$$SIZE^2(2 \cdot SIZE - 1)$$

Single Core, AVX, Basefrequency

$$67.2 \times 10^9 \ FLOP \times s^{-1} \times 10 \cdot s = SIZE^2(2 \cdot SIZE - 1) \ FLOP$$
$$672 \times 10^9 \ FLOP = SIZE^2(2 \cdot SIZE - 1) \ FLOP$$
$$SIZE = \lceil 6952.2 \rceil = 6953 \ [SIZE]$$

12 Cores, AVX, Basefrequency

$$806.4 \times 10^9 FLOP \times s^{-1} \times 10 \cdot s = SIZE^2(2 \cdot SIZE - 1) \ FLOP$$
$$8064 \times 10^9 \ FLOP = SIZE^2(2 \cdot SIZE - 1) \ FLOP$$
$$SIZE = \lceil 15916.4 \rceil = 15917 \ [SIZE]$$

12 Cores, AVX, Turbo

$$1075.2 \times 10^9 \ FLOP \times s^{-1} \times 10 \cdot s = SIZE^2(2 \cdot SIZE - 1) \ FLOP$$
$$10752 \times 10^9 \ FLOP = SIZE^2(2 \cdot SIZE - 1) \ FLOP$$
$$SIZE = \sqrt[3]{10752} = \lceil 17518.2 \rceil = 17519 \ [SIZE]$$

Führt man die Matrixmultiplikation parallel auf allen 12 Cores mit AVX und Turbo aus, benötigt man eine 17519 x 17519 Matrix, um eine Laufzeit von mindestens 10 Sekunden zu erhalten. Lässt man den Turbo weg und nimmt stattdessen die Basisfrequenz, verringert sich die Größe auf 15917 x 15917. Wird die Multiplikation auf einem einzelnen Core und ohne Turbo ausgeführt, reicht eine 6953 x 6953 Matrix aus, um eine Laufzeit von mindestens 10 Sekunden zu erlangen.

Aufgabe 4

Um die Korrektheit der Algorithmen zu überprüfen, werden diese auf zufälligen Testdaten ausgeführt und deren Ergebnisse mit einer Standard-Implementation verglichen. Wir haben uns dabei für die *Basic Linear Algebra Subprograms* (BLAS) Bibliothek entschieden. Hierbei wird die Multiplikation von der eigenen und der Biblitheksfunktion unabhängig voneinander ausgeführt und

anschließend jedes einzelne Element der Ergebnisse verglichen.

Der Quellcode in Listing 2 zeigt unser vorgehen. In Zeile 19 findet der Bibliotheksaufruf mit den zufällig initalisierten Matrizen statt. In Zeile 21 wird die eigene Implementierung aufgerufen und die Ergebnisse anschließend in der verschachtelten for-loop (Zeilen 23 - 27) mit der Bibliotheks-Variante verglichen.

```
5  float a[SIZE][SIZE];
6  float b[SIZE][SIZE];
7  float res[SIZE][SIZE];
8  float cblas_res[SIZE][SIZE];
9
10 extern void init_random(float[SIZE][SIZE]);
11 extern void init_zero(float[SIZE][SIZE]);
12 extern void matrix_mult(float[SIZE][SIZE], float[SIZE][SIZE], float[SIZE][SIZE]);
13
14 int main() {
15   init_random(a);
16   init_random(b);
17   init_zero(res);
18
19   cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, SIZE, SIZE, SIZE, 1.0,
20               &a[0][0], SIZE, &b[0][0], SIZE, 0.0, &cblas_res[0][0], SIZE);
21   matrix_mult(a, b, res);
22
23   for (int r = 0; r < SIZE; r++)
24     for (int c = 0; c < SIZE; c++)
25       if (cblas_res[r][c] != res[r][c]) {
26         printf("Check failed for row(%d) column(%d): cblas(%f) != res(%f)\n", r,
   ↪  c, cblas_res[r][c], res[r][c]);
27       }
28 }
```

Listing 2: testing.c

Der Code wird mit den fogenden Flags compiliert:
Dabei enthält die Umgebungsvariable die Größe der Matrix und die Variable den Namen der zu prüfenden Version (z.B. default oder unrolling).

Aufgabe 5

Zeitmessung: Berechnung der FLOPS der jeweiligen Matrixgrößen:

$$FLOPS = \frac{Anzahl\ Iterationen \cdot \frac{FLOP}{Iteration}}{Ausfuehrungszeit}$$

SIZE	128	256	512	1024	2048
MFLOPS	420.7	306.2	245.4	227.7	186.2
Diff.	0%	27.2%	19.9%	7.2%	18.2%

Es lässt sich erkennen, dass mit steigender Größe der Matrix die Mega-FLOPS von 420.7 (SIZE = 128) auf 186.2 (SIZE = 2048) abfallen. Mit Ausnahme von SIZE = 1024, lässt sich im Vergleich zur Vorgängergröße eine Reduzierung der Fließkommaperformance um etwa 20% feststellen. Unsere Vermutung ist, dass mit steigender Größe der Matrix, sich diese immer schlechter

Listing 3: wrapper.c

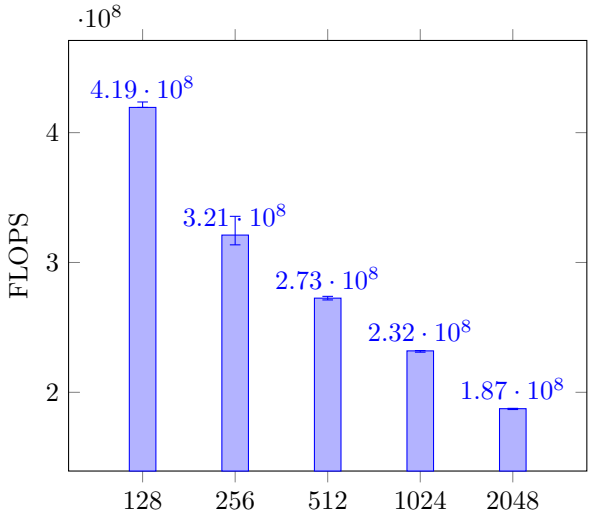
cachen lässt, wodurch häufig Daten aus dem Hauptspeicher geladen werden müssen. Generell nimmt jedoch die Performance ab, da die CPU während sie auf Daten wartet, nicht mit diesen rechnen kann.

Listing **??** zeigt den Code, welcher genutzt wird um die unterschiedlichen Implementationen auszuführen und deren FLOPS zu messen. Dazu werden zuerst die Arrays und mit zufälligen Werten initialisiert (Zeile 19-21) und danach an die jeweilige -Funktion (Zeile 25) übergeben. Um die Ausführungszeit zu messen wird die -Funktion aus der Standardbibliothek mit dem -Header genutzt. Das kompilieren erfolgt mit den folgenden Flags:

Wobei erneut genutzt wird um die Größe der Matrix anzugeben und selectiert welche Implementation gewählt wird.

Aufgabe 6

Der Graph stellt die in Aufgabe 5 gemessenen FLOPs dar. Auf der x-Achse sind die verschiedenen Matrixgrößen aufgetragen, auf der y-Achse die erreichten FLOPs.



SIZE	128	256	512	1024	2048
Median	419.9	304.4	246.0	224.5	186.0
Minimum	417.8	287.0	243.8	223.1	181.2
Maximum	426.1	331.6	246.0	244.5	190.3

Aufgabe 7

Optimierung: *Vertauschung der Schleifen*
Das Problem mit der Default-Implementierung ist, dass in jeder Iteration der innersten Schleife (Index k) auf b[i][c] zugegriffen wird. Da Matrizen in C im row-major Format abgespeichert werden, liegen aufeinanderfolgende Zugriffe (z.B. b[0][c] und b[1][c]) nicht nebeneinander im Speicher, was wiederum höchstwahrscheinlich in einem Cache Miss resultiert.
Indem man nun die i-Schleife mit der c-Schleife vertauscht, ändert sich das Zugriffsmuster zu folgendem: b[i][0], b[i][1], ..., b[i][SIZE-1]. Es wird also nacheinander auf im Speicher nebeneinanderliegende Werte zugeriffen, wodurch die Cache Misses deutlich reduziert werden.

```
extern void matrix_mult(float a[SIZE][SIZE], float b[SIZE][SIZE], float
↪  res[SIZE][SIZE]) {
  for (int r = 0; r < SIZE; r++)
    for (int i = 0; i < SIZE; i++)
      for (int c = 0; c < SIZE; c++)
        res[r][c] += a[r][i] * b[i][c];
}
```

Listing 4: loopswap.c

Optimierung: *Partial Loopunrolling*
Die Zahl der Iterationen der innersten Schleife werden um einen Faktor N (hier N = 4) rediziert, indem inerhalb eines Schleifendurchlaufs die Operationen von N Schleifendurchläufen ausgeführt werden. Durch das Unrolling wird die Zahl der Sprünge verringert.

```
extern void matrix_mult(float a[SIZE][SIZE], float b[SIZE][SIZE], float
↪  res[SIZE][SIZE]) {
  for (int r = 0; r < SIZE; r++)
    for (int c = 0; c < SIZE; c++)
      for (int i = 0; i < SIZE; i += 4) {
        res[r][c] += a[r][i + 0] * b[i + 0][c];
        res[r][c] += a[r][i + 1] * b[i + 1][c];
        res[r][c] += a[r][i + 2] * b[i + 2][c];
        res[r][c] += a[r][i + 3] * b[i + 3][c];
      }
}
```

Listing 5: unrolling.c

Optimierung: *Tiling*
Eine große Matrixmultiplikation wird in viele kleine Matrixmultiplikationen aufgebrochen. Für die Messungen wurde eine von 32 verwendet

```
extern void matrix_mult(float a[SIZE][SIZE], float b[SIZE][SIZE], float
↪  res[SIZE][SIZE]) {
  // looping through all tiles
  for (int rt = 0; rt < SIZE/TILE_SIZE; rt++)
    for (int ct = 0; ct < SIZE/TILE_SIZE; ct++)
      for (int it = 0; it < SIZE/TILE_SIZE; it++)
        // small matrix multiplication to calculate tile
        for (int r = 0; r < TILE_SIZE; r++)
          for (int c = 0; c < TILE_SIZE; c++)
            for (int i = 0; i < TILE_SIZE; i++)
              res[rt*TILE_SIZE + r][ct*TILE_SIZE + c] += a[rt*TILE_SIZE +
↪  r][it*TILE_SIZE + i] * b[it*TILE_SIZE + i][ct*TILE_SIZE + c];
}
```

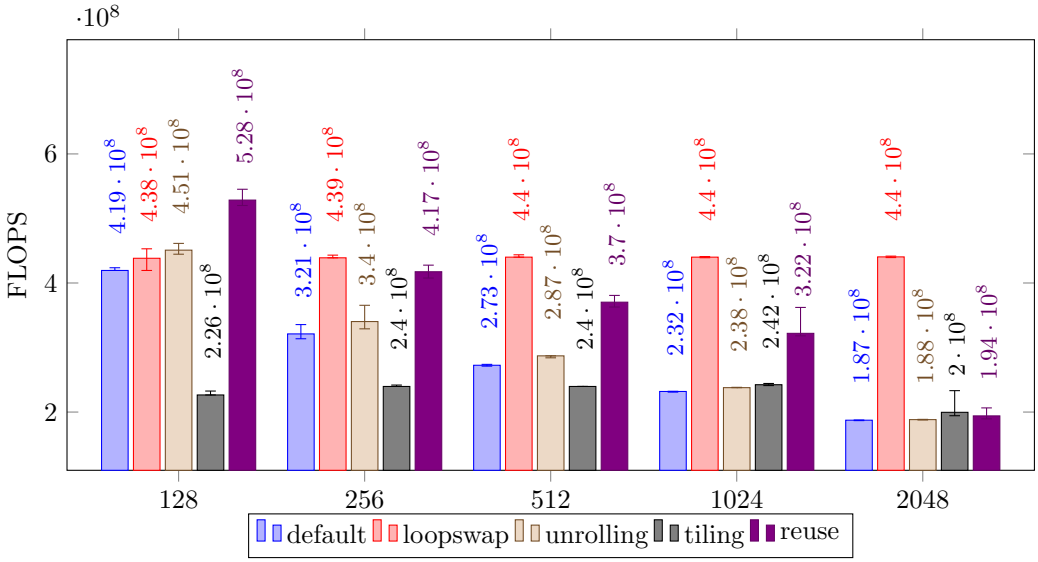
Listing 6: tiling.c

Optimierung: *Wiederverwendung von Werten*
Durch das Anlegen einer Akkumulatorvariablen *sum* wird beim Aufaddieren der einzelnen Summanden nicht immer auf den Speicher res[r][c] zugegriffen, sondern am Ende nur ein einziges Mal, wenn man *sum* nach res[r][c] speichert.

```c
#include "size.h"

extern void matrix_mult(float a[SIZE][SIZE], float b[SIZE][SIZE], float
↪  res[SIZE][SIZE]) {
  for (int r = 0; r < SIZE; r++)
    for (int c = 0; c < SIZE; c++){
      float sum = 0;
      for (int i = 0; i < SIZE; i++)
        sum += a[r][i] * b[i][c];
      res[r][c] = sum;
    }
}
```

Listing 7: reuse.c

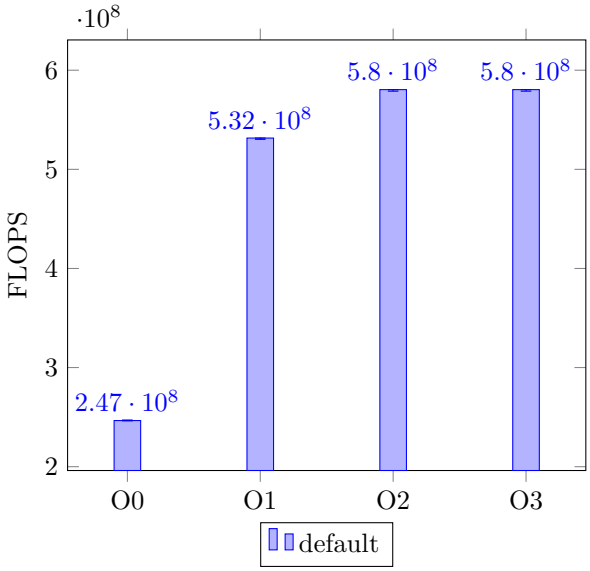


Wie bereits in Aufgabe 5 festgestellt, sinken die FLOPS der Standardimplementierung mit der Verdopplung der Matrixgröße um etwa 20%. Beim Loopswap hingegen bleiben die FLOPS über die verschiedenen Matrixgrößen fast konstant bei rund 440 Giga FLOPS. Unsere Vermutung ist, dass dies daher kommt, da beim Loopswap das Caching verbessert wird, indem die Lokalität besser ausgenutzt wird. Indem alle Werte einer Cachezeile nacheinander genutzt werden und es somit wenige bis keine Zugriffe auf ungecachte Daten gibt, können in dieser Zeit die neuen Daten aus dem Hauptspeicher geholt und somit die Speicherlatenzen verdeckt werden.
Unsere Implementierung von Tiling liefert über alle Matrizengrößen eine schlechte Fließkommaperformance von durchschnittlich 229.9 Giga FLOPS. Unsere Vermutung wäre, dass die Indexberechnung in der innersten Schleife die Performance nach unten drücken könnte. Außerdem gilt auch wie bei der Standardimplementierung ohne Loopswap, dass sich die Zugriffe auf b[it*TITLE_SIZE + i][ct*TITLE_SIZE + c] schlecht cachen lassen, was zu hohen Latzen führt. Bei der letzten Optimierung, der *Wiederverwendung von Werten*, erhalten wir mit einer Matrixgröße von 128 die meisten FLOPS überhaupt.

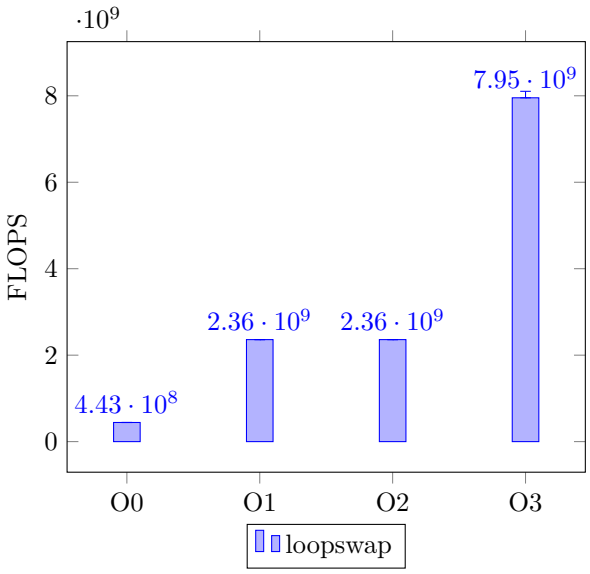
Aufgabe 8

Alle Plots stellen die leistung in FLOP/s in abhängigkeit zur genutzten compiler Optimierungs-Flag dar. Der compiler Auruf folgt diesem Schema:

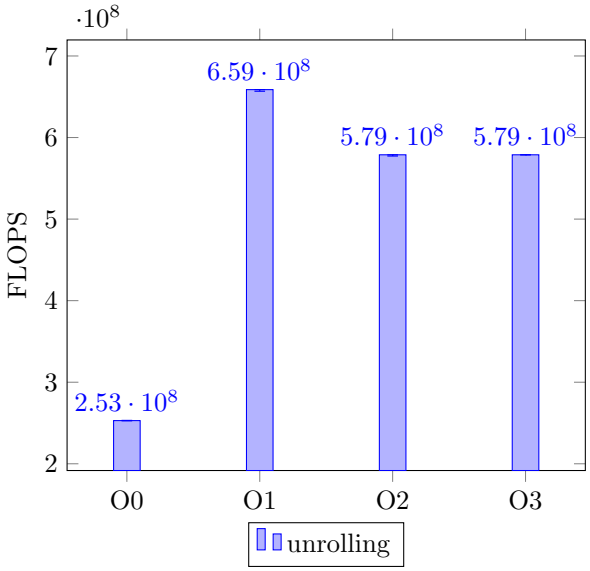
Dabei ist zu beachten, dass für alle Berechnungen eine Matrixgröße von 1024 verwendet wurde und die Werte O0, O1, O2 oder O3 annehmen kann.



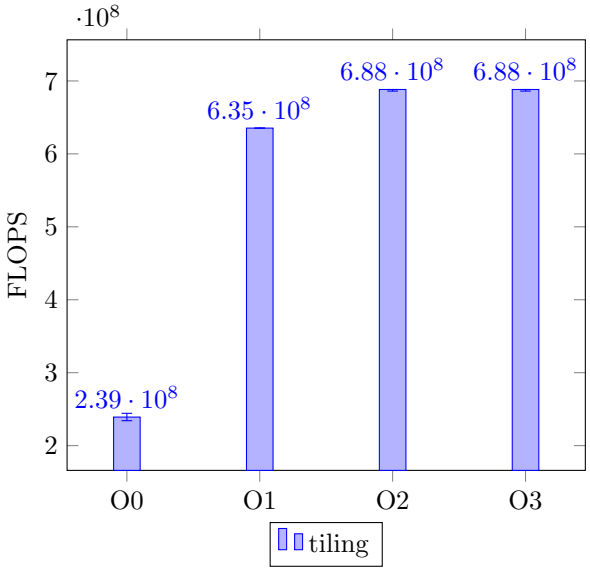
Aus der Standardimplementation konnte der compiler bereits mit O1 viele Optimierungen herausholen.



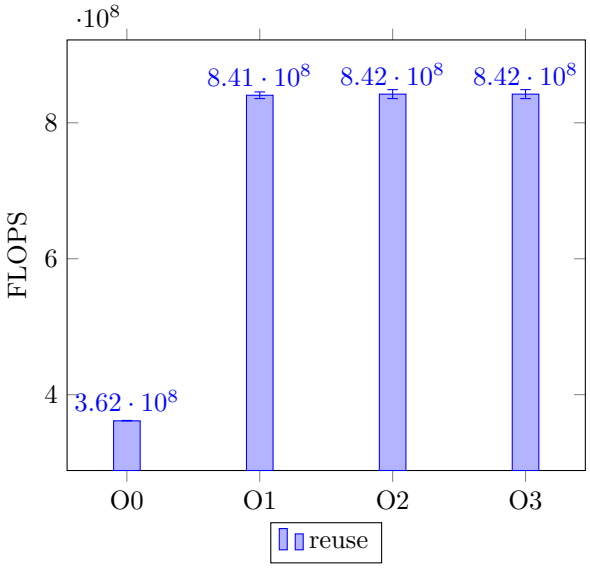
Bei der Implementation mit den vertauschten Schleifen erreicht der compiler durch die Flags O1 und O2 schon wesentlich höhere Werte als bei der default-version und auch durch die flag O3 werden die FLOPS noch einmal signifikant erhöht.



Auf die Schleifenentrollung hat die Flag O1 einen ziemlich guten Einfluss. Dahingegen wird die Performance durch die weiteren Optimierungen von O2 und O3 sogar wieder gesenkt.



Auf das tiling scheinen die Compiler-Flags einen sehr ähnlichen Einfluss wie auf die Standardimplementation zu haben. Dies könnte daran liegen, dass die beiden Versionen sich im Code sehr ähnlich sind, da auch beim Tiling die innersten 3 schleifen eine default-matrix-multiplikation umsetzen.



Auch bei der Wiederverwendung von Variablen ist ein ähnliches Muster zu erkennen. Die erste Optimierung zeit hier den einzigen Effekt.
Im allgemeinen lässt sich sagen, dass die Vertauschung der Schleifen auch im Bezug auf die Compiler-optimierungen den besten Effekt zeigt. Bei dieser Version wird einerseits die beste Performance erreicht und andererseits haben hier die Flags die größte Wirkung gezeigt.