Einführung in die technische Informatik

Rechnerarchitektur Praktikum 1

Praktikumsbericht Gruppe 1: Benjamin Probst, Tim Hanel

Aufgabe 1:

Für ein Kern FP/s (double-precision):

2 Ausführungseinheiten (FMA) * 2 Operationen / FMA-Einheit * 4 SIMD Operationen / (FMA Operation) (=256 bit Breite des Datenpfad / 64 bit double) * 3,3 Ghz Taktrate (max Frequency laut IntProducts) = 52.8 GFLOP/s

Für mehrere Kerne FP/s (double-precision):

52.8 GFLOP/s*12 Kerne = 633,6 GFLOPS/s

Aufgabe 2:

Schleifenvertauschen:

Durch das Vertauschen von Schleifen können Cache Zugriffe optimiert werden.
Zunächst wird beim Allokieren der Daten ein konsekutiver Speicherblock angelegt (im Normalfall).
Ladezugriffe finden (Cache-)Zeilenweise (64B) auf Speicher des Caches statt.
Häufig gibt es dabei auch Prefetcher, welche die nächste Zeile bereits mit laden (ausgelöst durch einen Cache Miss oder das Erkennen von Zugriffsmustern.)
Im schlimmsten Fall lädt jeder neue Ladezugriff eine neue Cachezeile was schnell zu Cache-Misses im L1 Cache des Prozessors führt (da die Anzahl der Cachezeile im L1 begrenzt ist) wodurch die Daten erst mit einer höheren Latenz aus dem L2 Cache oder aus noch höheren Ebenen geholt werden müssen. Durch das Vertauschen von Schleifenvariablen kann ein aufsteigender Zugriff innerhalb einer Cachezeile erzielt werden und Cache – Misses stark dezimiert, wodurch Ausführungseinheiten deutlich mehr Befehle pro Zeiteinheit Ausführungen können (da die Ladezeiten für die Operatoren deutlich geringer ausfallen).

Bei einer Matrix der Form n x n würde ein Zugriff der Form for i 0..n for j 0..n Matrix[j][i] spaltenweise über die Zeilen iterieren und dabei immer vollständige Zeilen laden. Wobei for i 0..n for j 0..n Matrix[i][j] die räumliche Lokalität der Cachezeilen ausnutzt und so Cache-Misses verhindert.

Loop - unrolling:

Durch Loop - unrolling kann der Einfluss von Branch – Misses verringert werden, indem der Anteil an Sprungbefehlen pro ausgeführtem Schleifendurchlauf reduziert wird.
Dabei wird der Befehl im Schleifenrumpf mehrfach untereinander ausgeführt und Inkrementieren der Zählvariablen entsprechend angepasst.
Loop-unrolling kann abhängig von Architektur und Compiler auch zusätzliche Parallelität schaffen, indem die Befehle eines einzelnen Schleifendurchlaufs besser die verfügbaren Ausführungseinheiten aufgeteilt werden (ersteres ist im Falle von General – Purpose - Prozessoren höchstwahrscheinlich relevanter).

Bei zu starkem Loop-unrolling können Cache – Misses impliziert werden, wenn der während eines Schleifendurchlaufs geladene Datensatz die Kapazität des L1 – Cache übersteigt.

Blocking/Tiling:

Bei dieser Technik wird die Matrixmultiplikation nacheinander jeweils für einen (Teil-)Block der gesamten Matrix ausgeführt. Diese Technik zielt ebenfalls auf die Verringerung von Cache Misses, indem der in einer bestimmten Zeit (die Dauer der Berechnung aller Werte des Blocks) genutzte Speicherbereich möglichst gering gehalten wird. Dieser Bereich sollte nach Möglichkeit kleiner sein als der L1-Cache des verwendeten Prozessors.

Bsp (link):

```
for (ii = 0; ii < n; ii+=B) {
  for (jj = 0; jj < n; jj+=B) {
    for (kk = 0; kk < n; kk+=B) {
      for (i = ii; i < ii+B; i++) {
        for (j = jj; j < jj+B; j++) {
          for (k = kk; k < kk+B; k++) {
            A[i,j] += X[i,k]*Y[k,j];
          }
        }
      }
    }
  }
}
```

Wiedernutzung von Werten statt neu Berechnung:

Beim Zugriff auf das Array des Resultats wird in jeder Iteration eine Pointer Berechnung durchgeführt und anschließend muss das Ergebnis in der Ergebnismatrix gespeichert werden, durch die Verwendung einer Hilfsvariable (d) kann man sich diese Berechnung sparen und die Zahl der Speicherzugriffe verringern.
z.B

```
for(int j=0;j<SIZE;++j){
          for(int i=0;i<SIZE;++i){
              double d=0;
               for(int k=0;k<SIZE;++k){
                  d+=input1[j*SIZE+k]*input2[k*SIZE+i];
               }
               output[j*SIZE+i]=d;
          }
    }
```

Hier wird nur n^2 statt n^3 Mal auf die Ergebnismatrix zugegriffen.

<u>Aufgabe 3:</u>

Insgesamt werden hier $2n^3$ Operationen ausgeführt um zwei Matrizen der Größe n×n miteinander zu multiplizieren. (Für das Skalar Produkt werden im Normalfall (2 x n – 1) Operationen benötigt, aber da wir auf eine 0 in der Ausgabematrix addieren sind es hier 2 x n Operationen für eine Zelle der Ausgabematrix (und damit 2 x n x n^2= $2n^3$ Operationen für die gesamte Ausgabematrix).

Also eine Größe von n = $\frac{1}{2}\sqrt[3]{52.8\ \mathrm{GFLOPS}}$ x 10 (SEK) = 18758 wird benötigt damit das Programm mindestens 10 Sekunden läuft.

Aufgabe 4:

t (in s) > $\frac{2\,x\,n}{\sqrt[3]{52.8 \times 10^9}}$

Aufgabe 6 UND 7:

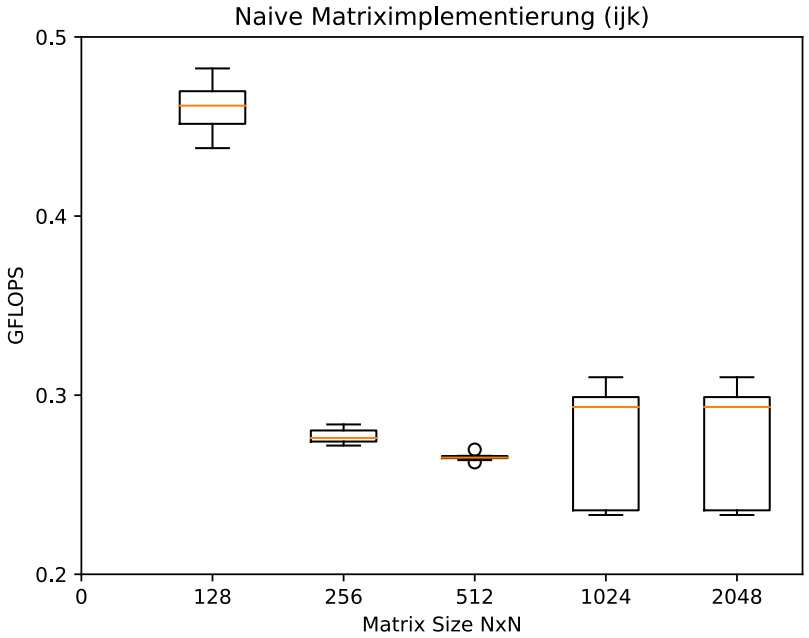
Verwendeter Compiler: GCC 7.1.0

Compilerflags: Keine

Anmerkung: Die unter https://www.openmp.org/spec-html/5.1/openmpsu53.html genutzten Direktiven sind ab OpenMP Version 5.1 verfügbar und werden derzeit noch nicht einmal in der aktuellen GCC 12 Version unterstützt. Auf dem Testsystem Taurus ist die höchste GCC Version die verwendete 7.1.0. Loop Unrolling und Tiling wurden letztendlich per Hand durchgeführt (Compilerflags wie -funroll-loops oder die Verwendung des #pragma GCC optimize („unroll-loops“--oder beides) haben leider keine spürbare Veränderung der Ausführungszeit gezeigt und wurden daher vor der Ausführung auf allen Matrixgrößen (128..2048) entfernt.).

Die Daten in allen Diagrammen bestehen aus mindestens und in den meisten Fällen genau 10 Werten.
Die Matrixgrößen von 128-512 wurden mehrfach durchgeführt. Die Anzahl der Wiederholungen wurde abhängig von der Größe gewählt und war (x = Größe der Matrix – y= Größe der Schleife):

(128 – 200), (256 – 100), (512 -20), (1024 -1), (2048 -1).



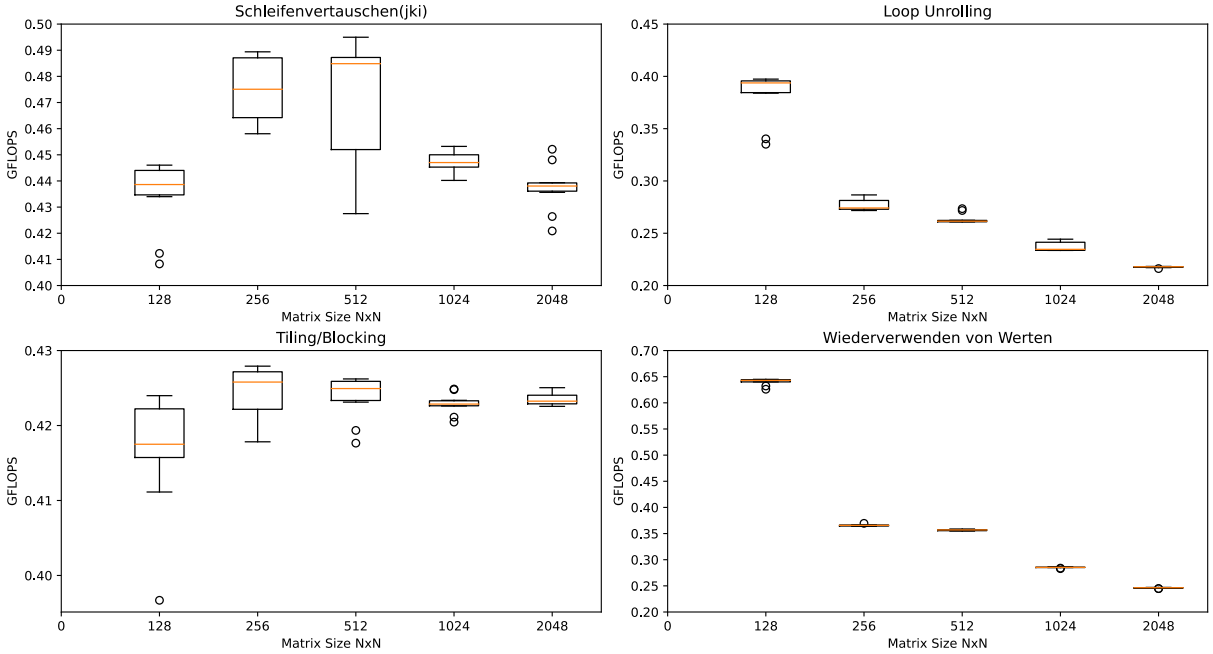
Zunächst wurde die Matrixmultiplikation in mit der Reihenfolge der Schleifenvariablen: ijk umgesetzt. Auf diese Implementierung bauen alle in Aufgabe 7 durchgeführten

Verbesserungen auf (d.h alle Iterationen außer „Veränderung von Schleifenvariablen“ verwenden ijk). Im Sinne der kritischen Betrachtung der Ergebnisse wären mehr als 10 Durchläufe besser gewesen, um eventuell einen höheren Streuungsgrad zu erreichen (insbesondere sichtbar bei Matrix Größe 512). Damit wären manche Ergebnisse besser innerhalb des Box – Plot Diagramm (und in den folgenden) sichtbar gewesen.

Code für die Naive Matriximplementierung (ijk):

```
void matmuljik(double * input1,double * input2,double * output){
    for(int i=0;i<SIZE;++i)
        for(int j=0;j<SIZE;++j)
            for(int k=0;k<SIZE;++k)
                output[j*SIZE+i]+=input1[j*SIZE+k]*input2[k*SIZE+i];
}
```

7. Verbesserungen der Matrixmultiplikation



Vertauschen von Schleifenvariablen

Alle Varianten zur Anordnung der Schleifenvariablen wurden untersucht mit dem Ergebnis, dass (jki) das beste Resultat liefert. Auffällig ist, dass diese Optimierung mit Abstand die besten Ergebnisse von allen untersuchten Verbesserungen liefert.

Code für Vertauschen von Schleifenvariablen:

```
void matmuljki(double * input1,double * input2,double * output){
    for(int j=0;j<SIZE;++j){          //jki
```

```
                for(int k=0;k<SIZE;++k){
                  for(int i=0;i<SIZE;++i){
                   output[j*SIZE+i]+=input1[j*SIZE+k]*input2[k*SIZE+i]
}
```

Loop Unrolling

Wie bereits erwähnt wurde Loop Unrolling und Tiling per Hand durchgeführt. Übersetzt in OpenMP 5.1 würde man den verwendeten Programmcode wie folgt ausdrücken.

```
void matmulUnroll(double * input1,double * input2,double * output){
        for(int j=0;j<SIZE;++j){
                #pragma omp unroll partial(4)
                  for(int i=0;i<SIZE;++i){
                        #pragma omp unroll partial(8)
                         for(int k=0;k<SIZE;++k){

                                output[j*SIZE+i]+=input1[j*SIZE+k]*input2[k*SIZE+i]

                         }
```

Loop unrolling performt sehr schlecht und liegt sogar unter der ersten Implementierung. Das Verändern der Loop Größe oder das „unrollen“ von noch viel größeren Loops ist per Hand nur schwer möglich und hier würden Compiler Direktiven mit Sicherheit oder OpenMp mit Sicherheit bessere Resultate liefern.

Tiling/Blocking

Obwohl Tiling ohne die von Compiler oder OpenMP unterstützten Directiven durchgeführt wurde liefert es deutlich bessere Resultate gegenüber der naiven Implementierung.
Tiling wurde im #pragma omp tile size(8,8,8) Format durchgeführt.

Code für Tiling/Blocking:

```
void matmulTiling(double * input1,double * input2,double * output){
    for(int i=0;i<SIZE;i+=8)
        for(int j=0;j<SIZE;j+=8)
            for(int k=0;k<SIZE;k+=8)
                for (int i2 = i; i2 < i + 8; i2 += 1)
                    for (int j2 = j; j2 < j + 8; j2 += 1)
                        for (int k2 = k; k2 < k + 8; k2 += 1)
                            output[j*SIZE+i]+=input1[j*SIZE+k]*input2[k*SIZE+i];

}
```

Wiederverwenden von Variablen:

Diese Methode liefert für niedrige Matrixgrößen zunächst deutlich bessere Werte als die Naive Implementierung, allerdings fällt die Performance für höhere Matrixgrößen (1024,2048) auf das Niveau des anfänglichen Versuchs(naiv) oder auf Loop unrolling.

Code für Wiederverwenden von Variablen:

```
void matmulParam(double * input1,double * input2,double * output){
        for(int j=0;j<SIZE;++j)     //jik
            for(int i=0;i<SIZE;++i){
                double d=0;
                for(int k=0;k<SIZE;++k){
                    d+=input1[j*SIZE+k]*input2[k*SIZE+i];
                }
                output[j*SIZE+i]=d;
            }
}
```

Aufgabe 8:

Jede Optimierung (außer O0) fügt eine Reihe von Compilerflags hinzu welche auch unter GccWeb zu finden sind. Jedes dieser Compilerflags stellt eine spezifische Optimierung dar, die darauf abzielt die Laufzeit oder den Code zu verbessern. Diese Optimierungen sind möglichst platformunabhängig.

O0: Compilierung ohne Optimierungen, hauptsächlich zum Debuggen verwendet.

O1: Optimierungen der Laufzeit und des Codes mit vergleichsweise geringer Kompilierungsdauer

-implementiert Wiederverwenden von Variablen mit z.B -fmove-loop-invariants

O2: Enthält Flags mit höherer Kompilierungsdauer, welche insbesondere die Performance erhöhen.

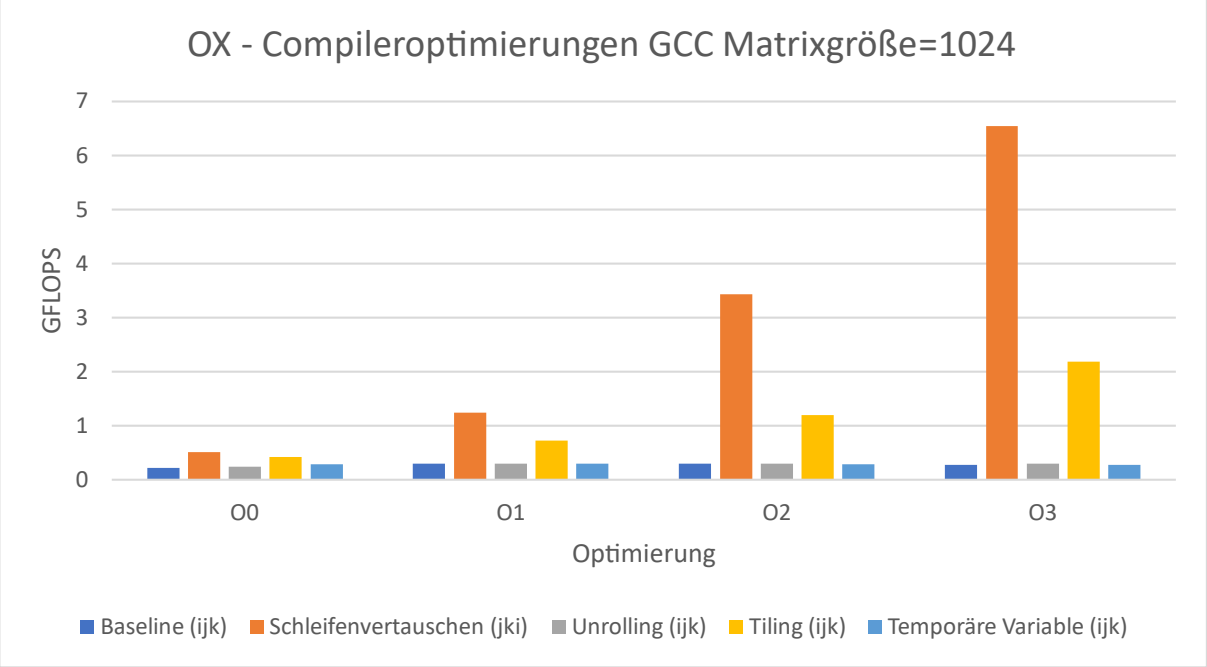
-unterstützt Tiling mit z.B -ftree-loop-vectorize

O3: Zusätzliche Flags welche auch den Speicherbedarf erhöhen können im Austausch gegen Performance.

- unterstützt Loop Unrolling mit z.B -floop-unroll-and-jam

- unterstützt Schleifenvertausch mit z.B -funswitch-loops

Architekturspezifische Optimierungen können z.B in der GCC manpage in der Form von Compilerflags gefunden werden für x86 Architekturen wie Haswell -- z.B unter x86 options.



Die Compileroptimierungen wurden mit einer Matrixgröße von 1024 einmalig und ohne Wiederholungen ausgeführt (da es diesbezüglich keine Einschränkungen gab). Dennoch lässt sich feststellen, dass die Compileroptimierungen nur für die Methoden Tiling und Schleifenvertauschen einen spürbaren Effekt haben. Außerdem haben die Ergebnisse gezeigt, dass O2 oder O3 zum Teil sogar leicht schlechtere Ergebnisse erzielen können (wird aus dem Diagramm nicht ersichtlich).