Praktikum

Effizente Matrix Multiplikation

Tobias Sebastian Hahn

24. Januar 2016

Matrikelnummer: 3778426

Inhaltsverzeichnis

1 Aufgabe 1 **5**

2 Aufgabe 2 **10**

3 Aufgabe 3 **15**

4 Aufgabe 4 **23**

1 Aufgabe 1

Für die Multiplikation zweier nxn-Matrizen soll ein möglichst effizienter Algorithmus gefunden werden. Nutzen Sie dazu den vorgegebenen Quelltext, der bereits die Basisvariante und eine Zeitmessroutine enthält. Diese Basisvariante sollen Sie optimieren –zunächst ohne zusätzliche Compiler-Flags.

Der gegebene Quelltext ist in Abbildung 1.1 gegeben. Ein großes Problem ist der Indexzugriff auf die B Matrix ($k * dim + j$). Da k der Bezeichner für die innerste Schleife ist, kommt es durch die Multiplikation mit dim zu vielen Cachemisses. Dies kann umgangen werden indem die Matrix B vor der Berechnung transponiert wird. Der Quelltext der transponierung ist in Abbildung 1.2 geben. Der Quellcode der Optimierung sieht man in Abbildung 1.3. In diesem Quellcode wurden zusätzlich die Indexberechnungen aus der innersten Schleife herausgezogen. Dazu wurden die Variablen $idim$ und $jdim$ eingeführt.

Die innerste Schleife berechnet eine Multiplikation und eine Addition. Danach kommt eine bedingte Sprunganweisung. Um die arithmetische Intensität zu erhöhen rollen wir die innere Schleife aus. Da alle getesteten Matrixgrößen den gemeinsamen Teiler 32 haben, nehmen wir diesen Wert als Ausrollparameter gewählt. Der Ausrollparameter hat den Bezeichner $BLOCK\_SIZE$. Der Quelltext kann in Abbildung 1.4 gesehen werden. Für diese Funktion wird das Makro $ADDER$ definiert. Diese Makro multipliziert zwei Zellen der Matrizen A und B.

Als letzte Optimierung haben wir Tiling implementiert. Der Quellcode ist in Abbildung 1.5 dargestellt. Dieser Quelltext nutzt auch die Ausrollung der innersten Schleife und die Transponierung der Matrix B.

```
mat_mult_non_opt(double *A, double *B, double *C, const unsigned int dim)
{
  for ( uint32_t i = 0; i < dim; i++ )
  {
    for ( uint32_t j = 0; j < dim; j++ )
    {
        for ( uint32_t k = 0; k < dim; k++ )
        {
            // C[i][j] += A[i][k] * B[k][j]
            C[ i * dim + j ] += A[ i * dim + k ] * B[ k * dim + j ];
        }
    }
  }
}
```

Abbildung 1.1: Der Quelltext der nicht optimierten Matrixmultiplikation.

```
transpose_mat( double* mat, const unsigned int dim)
{
  int i, j;
  for( i=1; i<dim; ++i){
      for( j=0; j<i; ++j){
    double help = mat[ i * dim +j];
    mat[ i * dim + j ] = mat[ j * dim + i];
    mat[ j * dim + i ] = help;
      }
  }
}
```

Abbildung 1.2: Diese Funktion transponiert eine Matrix.

```
mat_mult_transpose(double *A, double *B, double *C, int dim)
{
  transpose_mat(B, dim);
  uint32_t j, i;
  uint32_t idim, jdim;
  double temp;
  for ( j = 0; j < dim; j++ )
  {
      jdim = j * dim;
      for (  i = 0; i < dim; i++ )
      {
    idim = i * dim;
    for (size_t k = 0; k < dim; k++ )
    {
        C[ idim + j ] += A[ idim + k ] * b[ jdim + k ];
    }
      }
  }
  transpose_mat(B, dim);
}
```

Abbildung 1.3: Die Funktion berechnet die Matrixmultiplikation. Vor und nach der Berechnung wird B transponiert.

```
mat_mult_unroll_transpose( double *A, double *B, double *C, int dim)
{
  #define ADDER(a) (A[ idimk + a] * B[ jdimk + a])
  transpose_mat(B, dim);
  uint32_t i, j, k;
  uint32_t idim, jdim, idimk, jdimk;
  for (  i = 0; i < dim; i++ )
  {
    idim = i*dim;
    for (  j = 0; j < dim; j++ )
    {
        jdim= j*dim;
        for (  k = 0; k < dim; k+=32 )
        {
            jdimk = jdim+k;
            idimk = idim+k;

            C[ idim + j] += ADDER(0)
                + ADDER(1)
                + ADDER(2)
                + ADDER(3)
                + ADDER(4)
                + ADDER(5)
                + ADDER(6)
                + ADDER(7)
                + ADDER(8)
                + ADDER(9)
                + ADDER(10)
                + ADDER(11)
                + ADDER(12)
                + ADDER(13)
                + ADDER(14)
                + ADDER(15)
                + ADDER(16)
                + ADDER(17)
                + ADDER(18)
                + ADDER(19)
                + ADDER(20)
                + ADDER(21)
                + ADDER(22)
                + ADDER(23)
                + ADDER(24)
                + ADDER(25)
                + ADDER(26)
                + ADDER(27)
                + ADDER(28)
                + ADDER(29)
                + ADDER(30)
                + ADDER(31);
        }
    }
  }
  transpose_mat(B, dim);
  #undef ADDER
}
```

Abbildung 1.4: Der Quelltext transponiert die Matrix B und berechnet die Matrixmultiplikation. Die Berechnung der innersten Schleife wird 32 mal ausgerollt.

```
mat_mult_tilling( double *A, double *B, double *C, int dim)
{
  #define ADDER(a) (A[posXA_i_dim_posYA+ a] * B[posYB_j_dim_posYA + a ])
  transpose_mat(A, dim);
  uint32_t i, j, posXA, posYA, posYB;
  uint32_t posYB_j_dim, posXA_i, posYB_j_dim_posYA, posXA_i_dim_posYA ;
  for(posXA=0; posXA<dim; posXA +=BLOCK_SIZE)
  {
      for(posYA=0; posYA<dim; posYA += BLOCK_SIZE)
      {
    for(posYB=0; posYB<dim; posYB += BLOCK_SIZE)
    {
        for(i=0; i<BLOCK_SIZE; ++i)
        {
      posXA_i = posXA + i;
      posXA_i_dim_posYA = (posXA + i) * dim + posYA;
      for(j=0; j<BLOCK_SIZE; ++j)
      {
          posYB_j_dim = (posYB + j) * dim;
          posYB_j_dim_posYA =  (posYB + j) * dim + posYA;
          C[ posYB_j_dim  + (posXA_i)] += ADDER(0)
        + ADDER(1)
        + ADDER(2)
        + ADDER(3)
        + ADDER(4)
        + ADDER(5)
        + ADDER(6)
        + ADDER(7)
        + ADDER(8)
        + ADDER(9)
        + ADDER(10)
        + ADDER(11)
        + ADDER(12)
        + ADDER(13)
        + ADDER(14)
        + ADDER(15)
        + ADDER(16)
        + ADDER(17)
        + ADDER(18)
        + ADDER(19)
        + ADDER(20)
        + ADDER(21)
        + ADDER(22)
        + ADDER(23)
        + ADDER(24)
        + ADDER(25)
        + ADDER(26)
        + ADDER(27)
        + ADDER(28)
        + ADDER(29)
        + ADDER(30)
        + ADDER(31);
      }
        }
    }
      }
  }
  transpose_mat(A, dim);
  #undef ADDER
}
```


Abbildung 1.5: Matrixmultiplikation mit Tiling.

2 Aufgabe 2

Nutzen Sie die integrierte Zeitmessroutine um ihren Fortschritt bei der Optimierung zu bewerten.

Wir haben den Aufruf der Zeitmessung geringfügig angepasst, siehe Abbildung 2.1. Ziel ist es die Auswertung zu vereinfachen. Um die Matrixmultiplikation zu starten, haben wir ein Skript geschrieben. Dieses Skript ist in Abbildung 2.2 angegeben. Dieses Skript wurde mittels $sbatch$ gestartet.

In Abbildung 2.3 sieht man die benötigten Zeiten, für die Kompilierung mit gcc und in Abbildung 2.4 die Zeiten für den Intel Compiler. Berechnungen welche länger als 30 Sekunden benötigen werden ignoriert. Alle Zeiten können im Anhang nachgeschlagen werden. Die GFLOP/s können in Abbildung 2.5 und in Abbildung 2.6 abgebildet.

Zuerst betrachten wir die Zeiten des gcc compilierten Quelltextes. Durch das herausziehen der Indexberechnung und die transponierung der Matrix B, sinkt die benötigte Zeit auf rund ein drittel. Die GFLOP/s steigen von 0,25 auf 0,82. Der Abfall der Performance erfolgt später und weniger ausgeprägt. Bei einer Matrixgröße von 2048 unterscheiden sich die GFLOP/s um den Faktor 20. Das Ausrollen der inneren Schleife bringt zusätzlich einen Geschwindigkeitsgewinn. Die GFLOP/s steigen auf bis zu 1,5. Das Tiling ist langsamer als die Ausrollung. Die Optimierung Tiling hat maximal 1,44 GFLOP/s. Ein mögliche Erklärung ist, dass die gewählte Blockgröße schlecht ist.

Der Intel Compiler liefert Ergebnisse die vergleichbar, mit denen des gcc, sind.

```
// Nothing
t_start = gtod();
mat_mult_non_opt(A,B,C, dim);
t_end = gtod();
gflops = ( ( double )2 * dim * dim * dim / 1000000000.0 ) / ( t_end -
    t_start );
printf("%d;%0.4f;%0.2f;", dim, t_end - t_start, gflops );
free( C );

// Transpose
C = zero_mat( dim );
t_start = gtod();
mat_mult_transpose(A,B,C, dim);
t_end = gtod();
gflops = ( ( double )2 * dim * dim * dim / 1000000000.0 ) / ( t_end -
    t_start );
printf("%0.4f;%0.2f;",  t_end - t_start, gflops );
free( C );

// Unroll 32 Transpose
C = zero_mat( dim );
t_start = gtod();
mat_mult_unroll_transpose(A,B,C, dim);
t_end = gtod();
gflops = ( ( double )2 * dim * dim * dim / 1000000000.0 ) / ( t_end -
    t_start );
printf("%0.4f;%0.2f;", t_end - t_start, gflops );
free( C );

// Unroll 32 BLOCKING
C = zero_mat( dim );
t_start = gtod();
mat_mult_blocking(A,B,C, dim);
t_end = gtod();
gflops = ( ( double )2 * dim * dim * dim / 1000000000.0 ) / ( t_end -
    t_start );
printf("%0.4f;%0.2f;\n", t_end - t_start, gflops );
```

Abbildung 2.1: Quelltext der die Veränderung der Zeitmessroutine darstellt

```
#!/bin/bash
#SBATCH --time=01:00:00
#SBATCH --partition=sandy
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:0

module load gcc
module load intel

gcc -O0 -std=c99 ./main.c
./a.out > gccO0.csv
icc -O0 -std=c99 ./main.c
./a.out > iccO0.csv
```

Abbildung 2.2: Dieses Skript wird benutzt um den Auftrag auf Taurus zu starten.

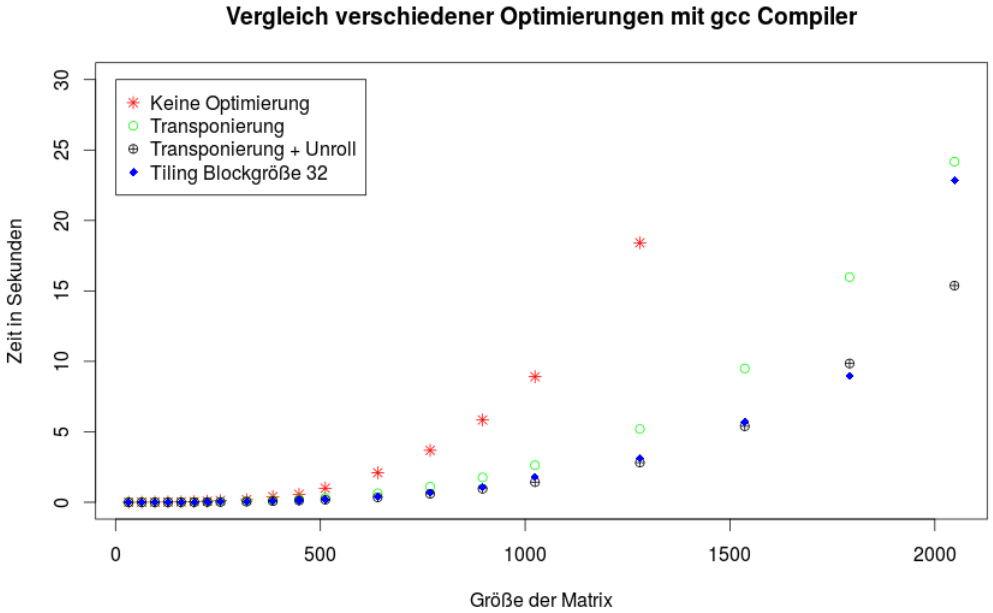

Abbildung 2.3: Vergleich der Laufzeiten verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O0 -std=c99 compiliert.

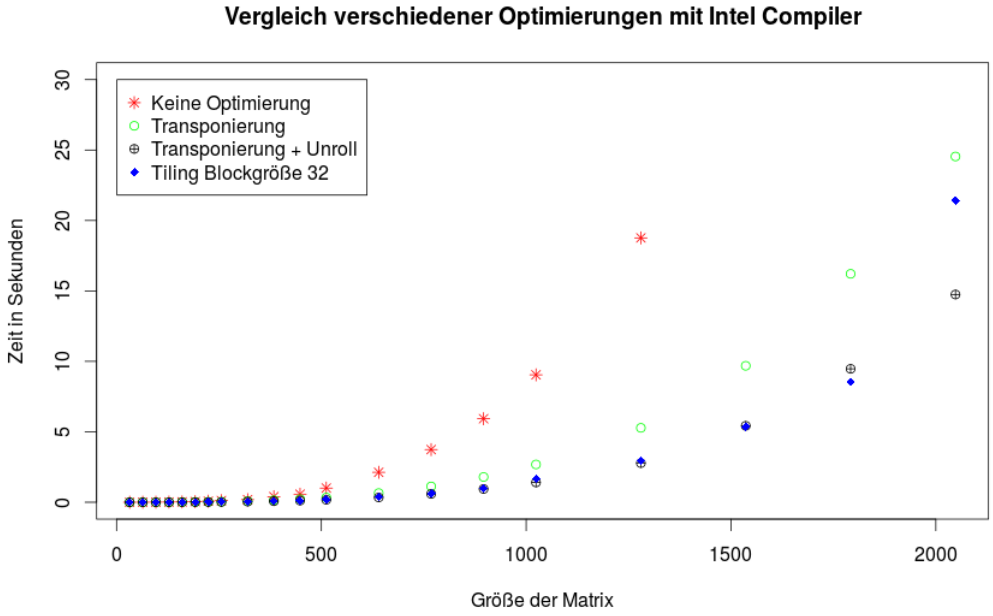

Abbildung 2.4: Vergleich der Laufzeiten verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O0 -std=c99 compiliert.

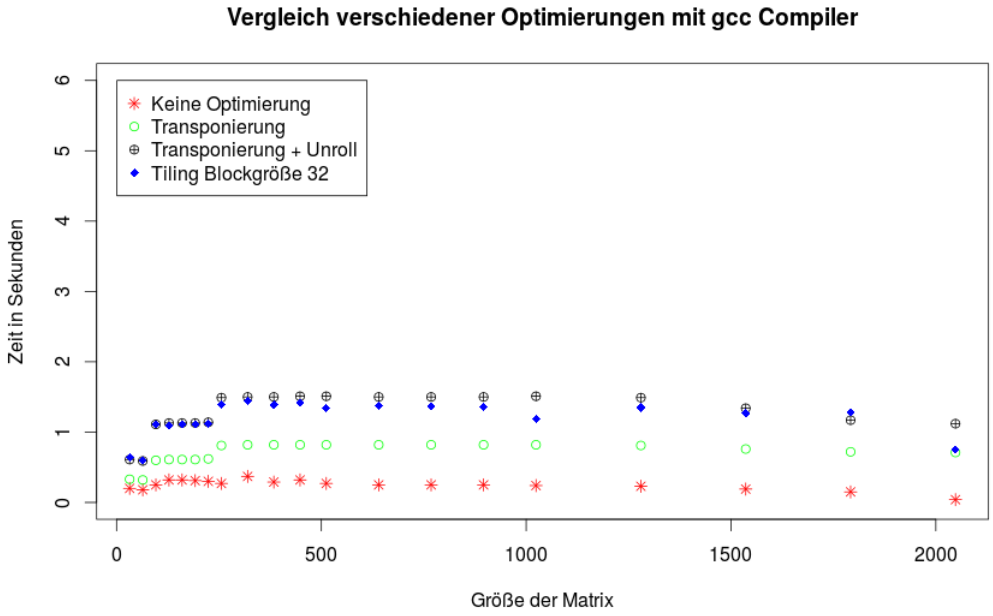

Abbildung 2.5: Vergleich der GFLOP/s verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O0 -std=c99 compiliert.

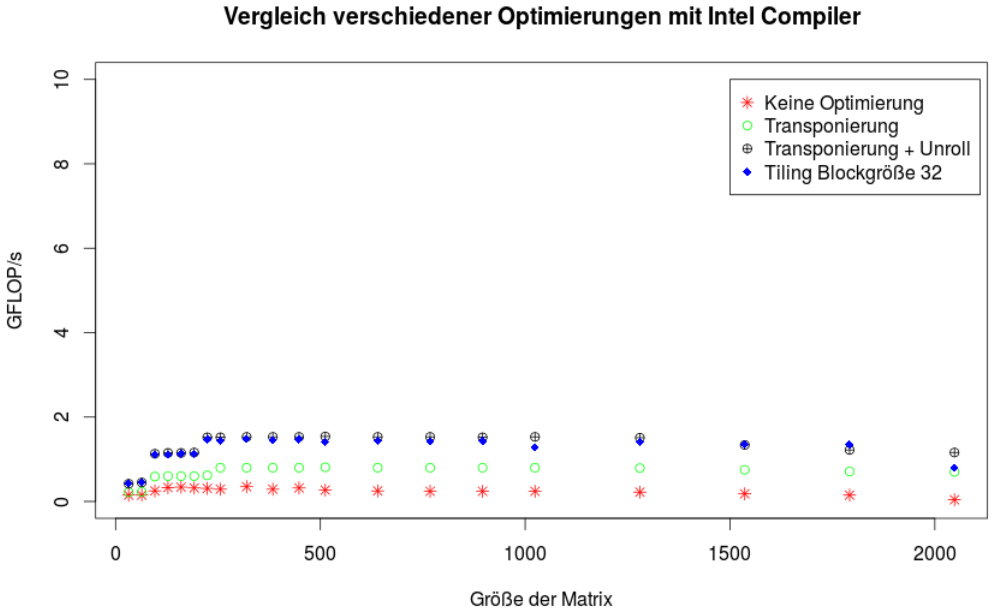

Abbildung 2.6: Vergleich der GFLOP/s verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O0 -std=c99 compiliert.

3 Aufgabe 3

Überprüfen Sie, welche ihrer manuell durch geführten Optimierungen durch den Einsatz geeigneter Compiler-Flags bzw. Optimierungsstufen auch vom Compiler realisiert werden.

Als Compiler Flags haben wir $-O1$, $-O2$ und $-O3$ ausprobiert. Zuerst betrachten wir den gcc Compiler. Die Ausführzeiten für den gcc Compiler sind in den Abbildungen 3.1, 3.2 und 3.3 gezeigt. Die GFLOP/s sind in den Abbildungen 3.4, 3.5 und 3.6 zusammengefasst. Der mittels $-O1$ compilierte Quelltext ist im Vergleich zum $-O0$ compilierten Quelltext deutlich schneller. Die geringste Steigerung gibt es für den originalen Quelltext. Hier wird die Matrix nur rund 40% schneller ausgeführt. Die Steigerung, falls die B Matrix vorher transponiert wurde, beträgt ungefähr 325%. Das Ausrollen des Quelltextes ist 90% schneller, falls es mit $-O1$ compiliert wird. Das Tiling benötigt ungefähr halb soviel Zeit. Die Ausführzeiten zwischen $-O1$, $-O2$ und $-O3$ compilierten Quelltext sind identisch.

Die Zeiten für den Intel Compiler sind in den Abbildungen 3.7, 3.8 und 3.9 aufgeführt. Die GFLOP/s Performance ist in den Abbildungen 3.10, 3.11 und 3.12 zusammengefasst. Auf der $-O1$ Stufe ist der Intel Compiler mit dem gcc Compiler vergleichbar. Die Multiplikation und die Transponierung der B Matrix sind gleich schnell. Ein zusätzliches ausrollen der inneren Schleife wird von gcc Compiler besser übersetzt. Der gcc erzeugt rund 15% schnelleren Code. Bei der Matrixmultiplikation mit Tiling ist der gcc 0.5 GFLOP/s schneller. Das $-O2$ Flag erzeugt einen deutlichen Performancegewinn beim Intel Compiler. Am deutlichsten wird dies am Quelltext der unverändert compiliert wird. Dieser wird 15x schneller ausgeführt, als mit $-O1$ Flag, und erreicht 4.68 GFLOP/s. Die Transponierung der B Matrix ist mit $-O2$ Flag die schnellste Multiplikationsvariante. Diese Variante wird 2,5 mal schneller ausgeführt als mit $-O1$ Flag. Die Unroll variante ist ungefähr doppelt so schnell. Sie ist aber 1,6 GFLOP/s langsamer als die Transpositionierungs Variante. Bei der Blocking Multiplikation gibt es keine Änderung. Zwischen den $-O2$ und $-O3$ compilierten Varianten gibt es nur für die Ausrollung Variante einen Unterschied. Die Ausrollung ist mit $-O3$ Flag, 1,6 GFLOP/s langsamer als die mit $O2$ compilierten.

Das $-O1$ Flag des gcc Compiler aktiviert eine Vielzahl von Optimierungen. Beispiels weiße versucht $-fforward-propagate$ zwei Anweisungen zu kombinieren und zu vereinfachen. Das Flag $-freorder-blocks$ vertauscht Basisblöcke um die Codelokalität zu erhöhen. Das $-O2$ und das $-O3$ Flag erhöhen die Performance nicht. Der Größe unterschied zwischen den Intel Compiler und den gcc Compiler ist die $-O2$ Optimierungsstufe. Auf der Intel Homepage [1] steht:

"This option enables optimizations for speed. This is the generally recommended optimization level. The compiler vectorization is enabled at O2 and higher levels. With this option, the compiler performs some basic loop optimizations, inlining of intrinsic, Intra-file

[1] `https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler`

interprocedural optimization, and most common compiler optimization technologies."

	gcc			Intel		
	-O1	-O2	-O3	-O1	-O2	-O3
Keine Optimierung	6.5820	6.5765	6.5713	6.7796	0.4591	0.4527
Transponierung	0.8771	0.8773	0.8827	0.8768	0.3622	0.3617
Transponierung + Unroll	0.7882	0.7885	0.7764	0.8717	0.4965	0.7891
Blocking Blockgröße 32	0.8339	0.8369	0.8435	1.0509	1.0386	1.0414

Tabelle 3.1: Zusammenfassung der Ausführzeiten für eine Multiplikation von zwei Matrizen der Größe 1024. Die Zeiten sind in Sekunden.

	gcc			Intel		
	-O1	-O2	-O3	-O1	-O2	-O3
Keine Optimierung	0.33	0.33	0.33	0.29	4.68	4.74
Transponierung	2.45	2.45	2.43	2.34	5.93	5.94
Transponierung + Unroll	2.72	2.72	2.77	2.43	4.33	2.72
Blocking Blockgröße 32	2.58	2.57	2.55	2.72	2.07	2.06

Tabelle 3.2: Zusammenfassung der GFLOP/s für eine Multiplikation von zwei Matrizen der Größe 1024.

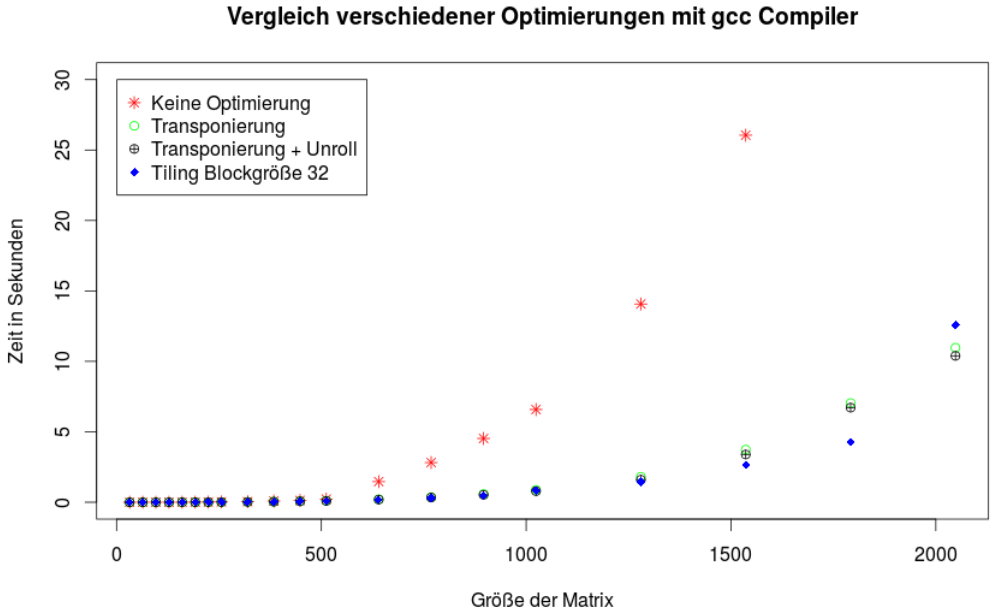

Abbildung 3.1: Vergleich der Laufzeiten verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O1 -std=c99 compiliert.

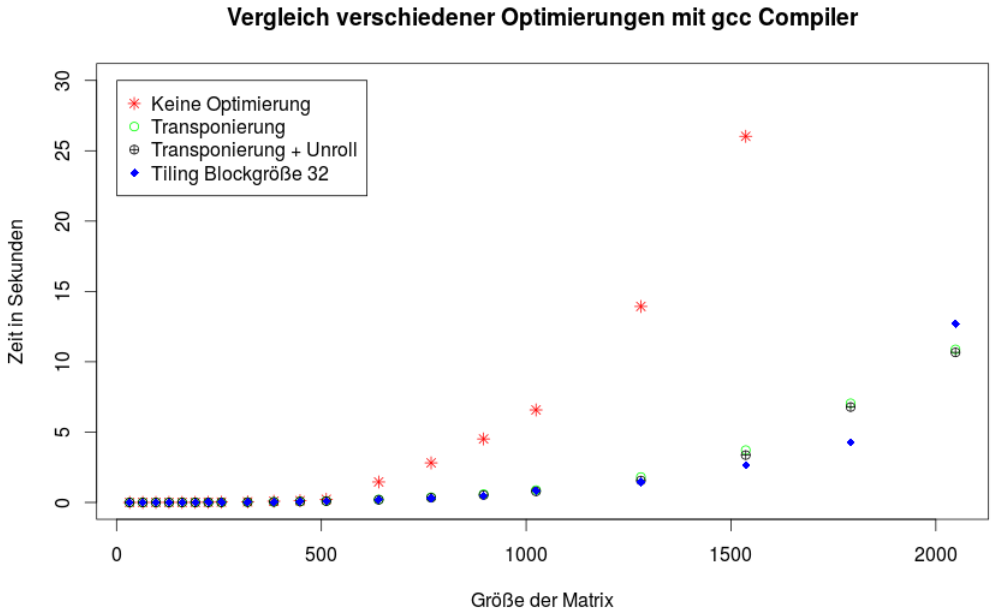

Abbildung 3.2: Vergleich der Laufzeiten verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O2 -std=c99 compiliert.

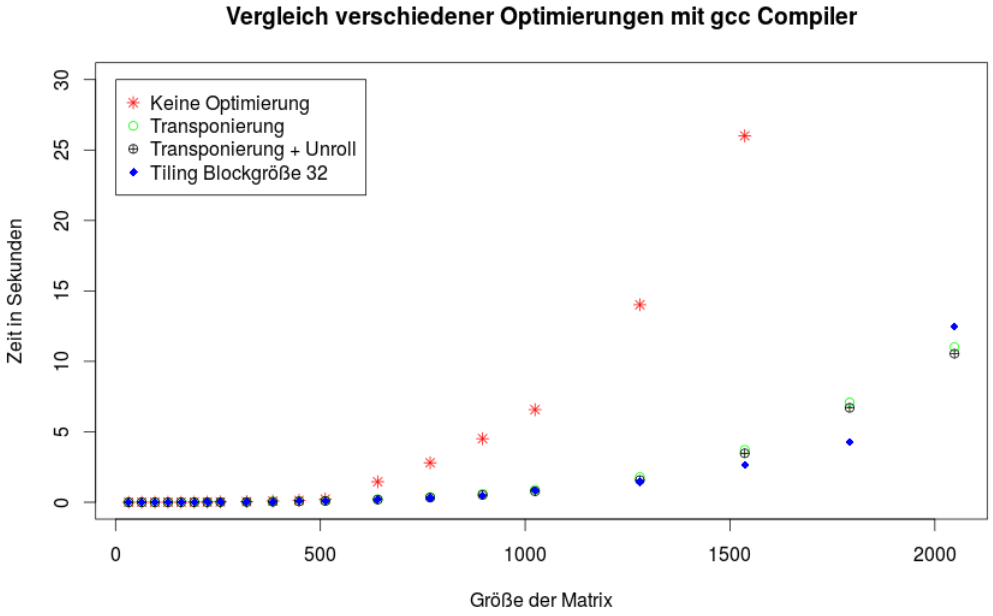

Abbildung 3.3: Vergleich der Laufzeiten verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O3 -std=c99 compiliert.

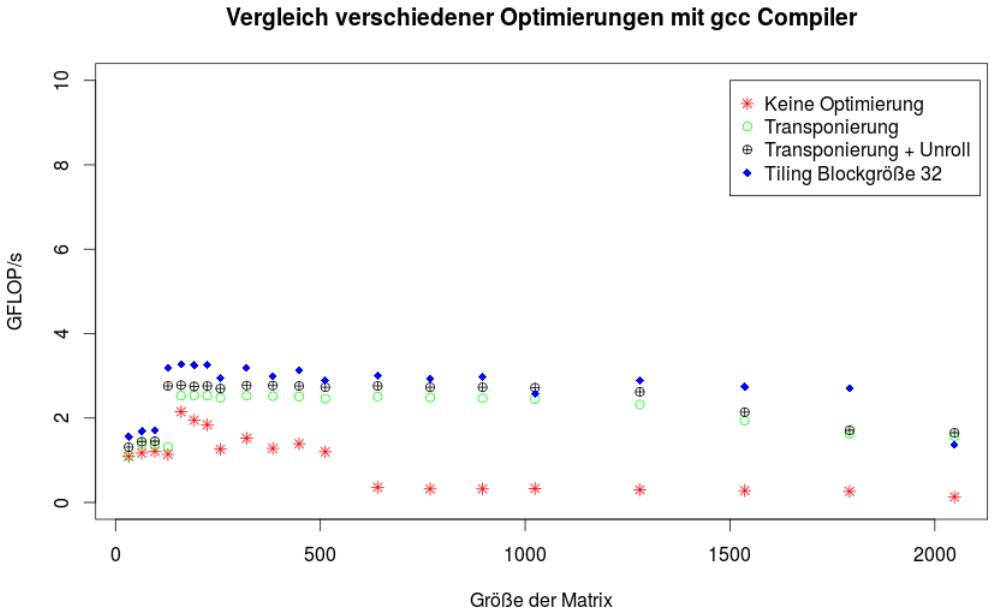

Abbildung 3.4: Vergleich der GFLOP/s verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O1 -std=c99 compiliert.

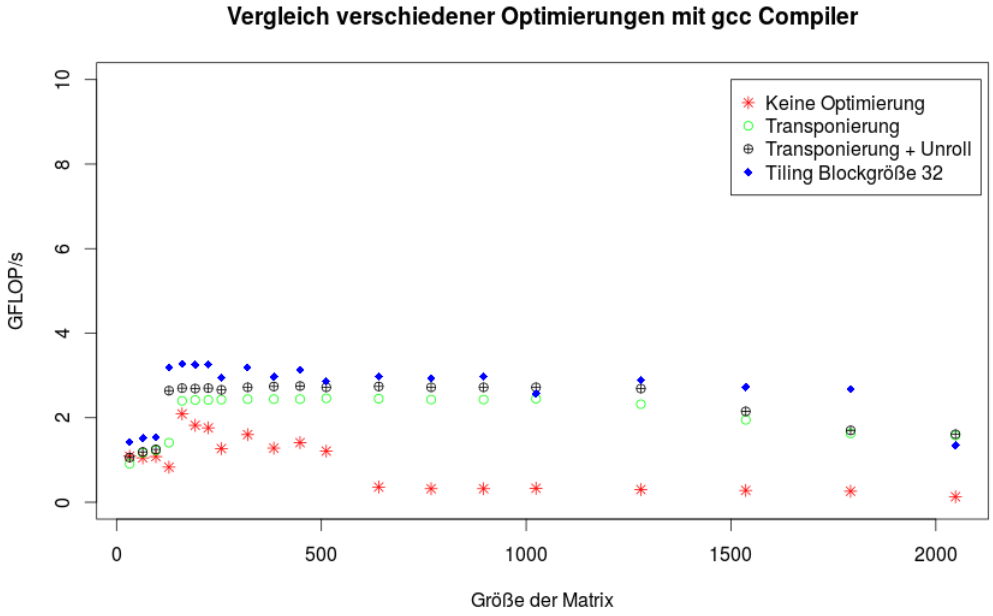

Abbildung 3.5: Vergleich der GFLOP/s verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O2 -std=c99 compiliert.

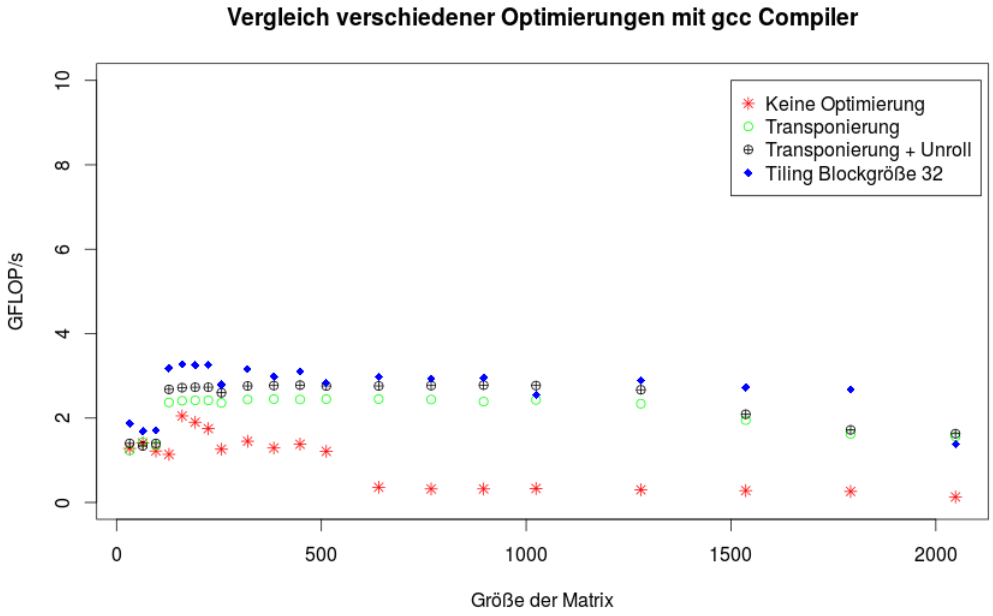

Abbildung 3.6: Vergleich der GFLOP/s verschiedener Optimierungen mit dem gcc Compiler. Der Quelltext wurde mit den Parametern -O3 -std=c99 compiliert.

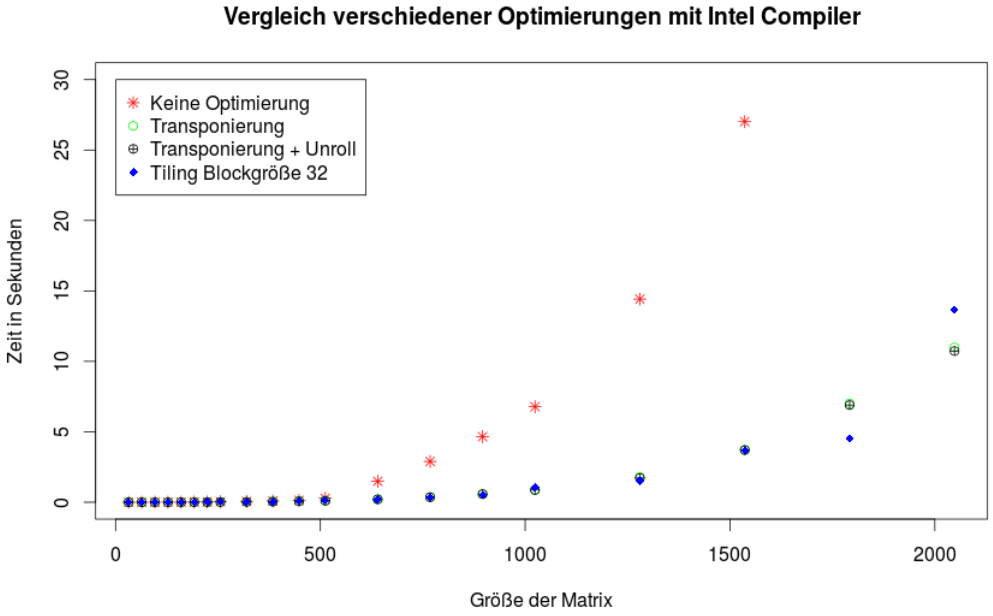

Abbildung 3.7: Vergleich der Laufzeiten verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O1 -std=c99 compiliert.

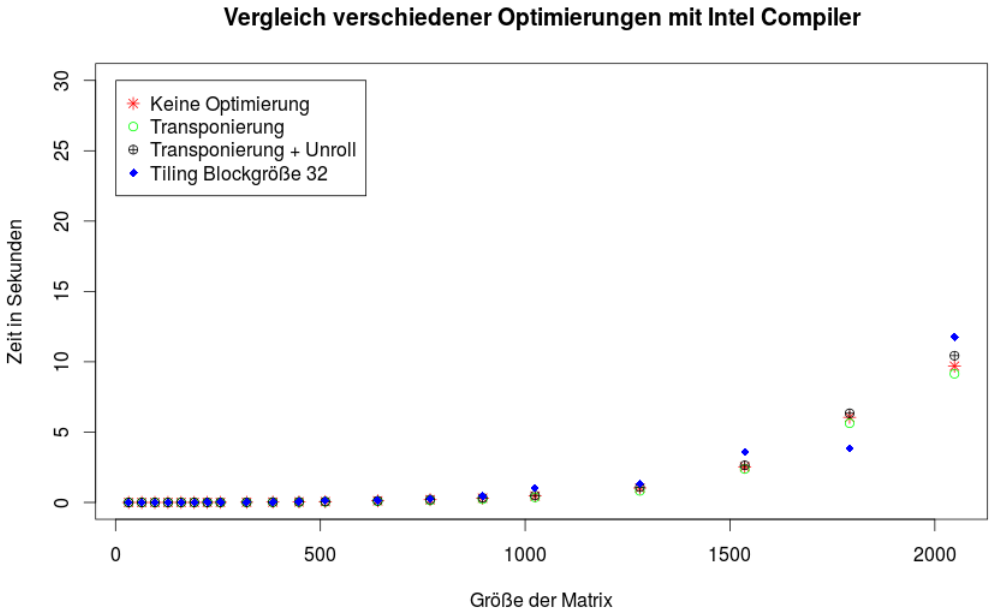

Abbildung 3.8: Vergleich der Laufzeiten verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O2 -std=c99 compiliert.

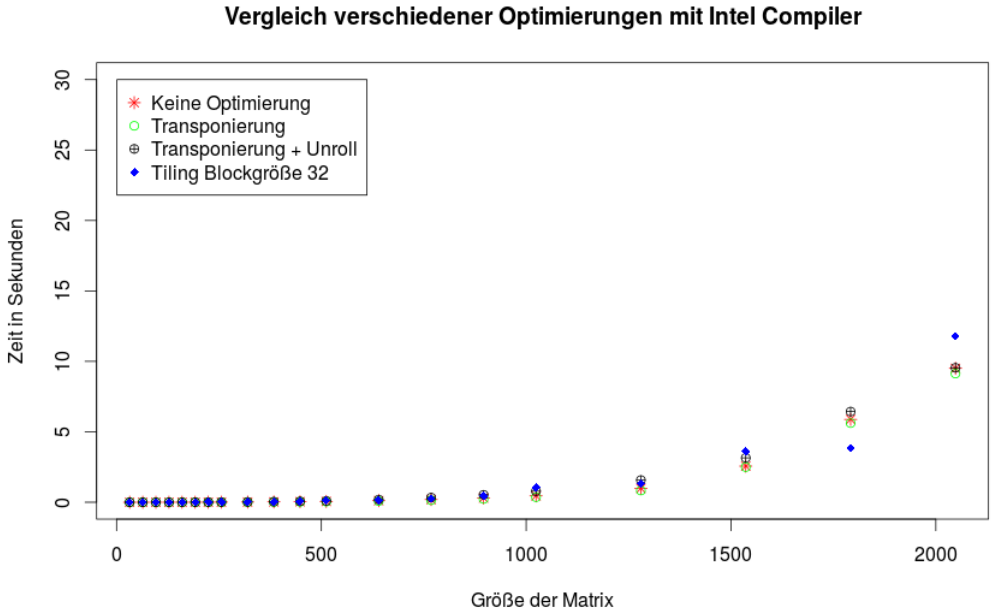

Abbildung 3.9: Vergleich der Laufzeiten verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O3 -std=c99 compiliert.

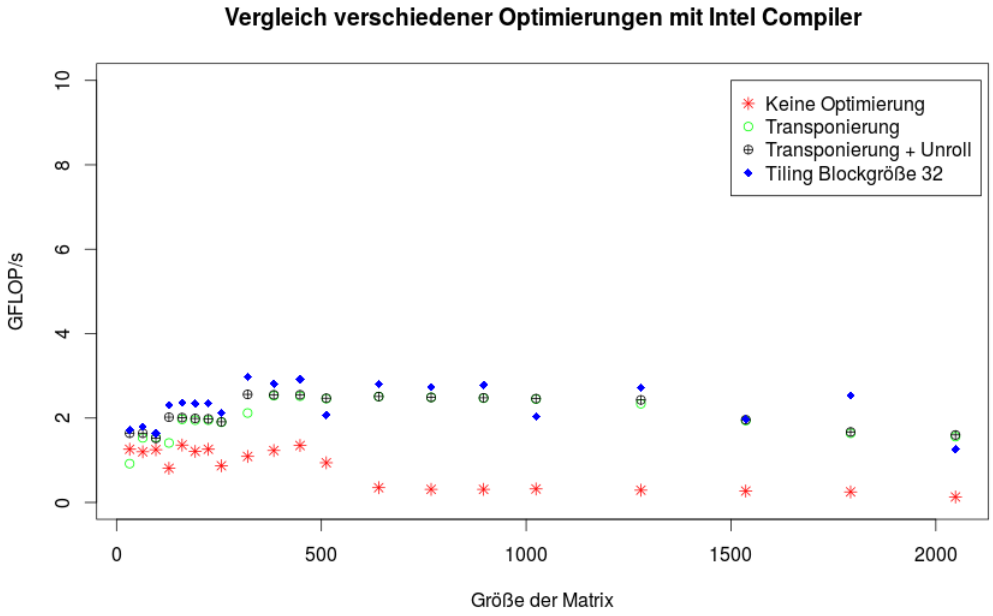

Abbildung 3.10: Vergleich der GFLOP/s verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O1 -std=c99 compiliert.

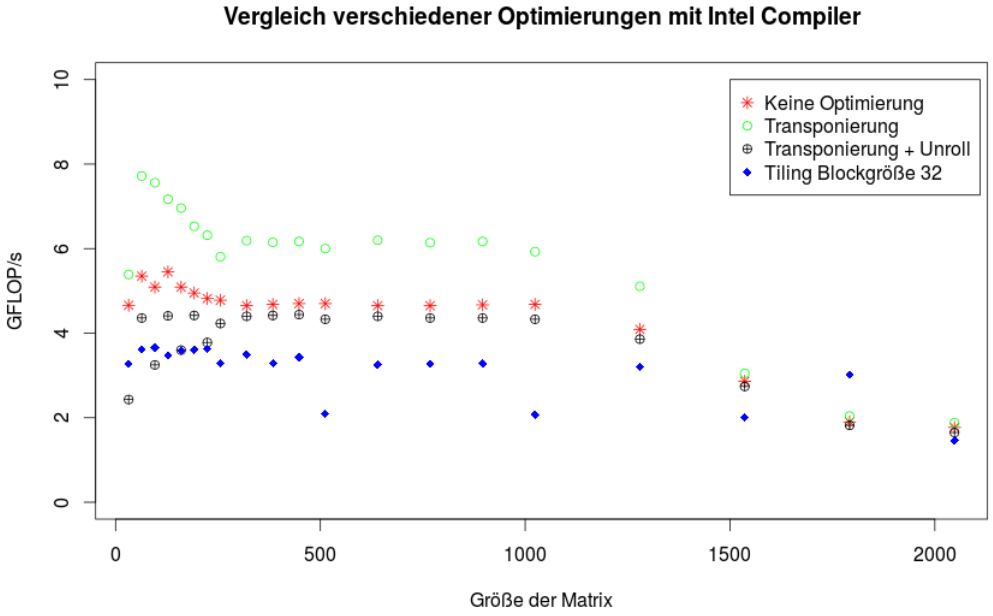

Abbildung 3.11: Vergleich der GFLOP/s verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O2 -std=c99 compiliert.

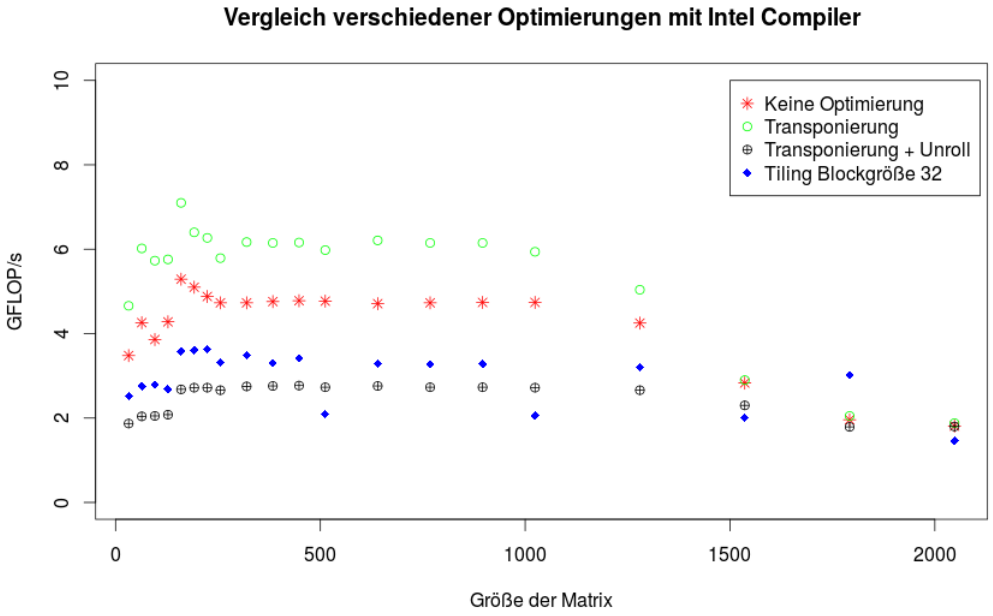

Abbildung 3.12: Vergleich der GFLOP/s verschiedener Optimierungen mit dem Intel Compiler. Der Quelltext wurde mit den Parametern -O3 -std=c99 compiliert.

4 Aufgabe 4

Berechnen Sie die theoretische Floating-Point-Peak-Performance des Prozessors. Bewerten und begründen Sie die Unterschiede der Leistung ihrer Implementierung im Vergleich zur maximal erreichbaren Leistung.

Der Intel Xeon E5-2690 hat eine Taktfrequenz von 2,9GHz. Er hat AVX, d.h. er kann 8 single precision Werte Gleichzeitig verarbeiten. Der Sandy Bridge Prozessor besitzt 8 Kerne. Für Single Precision wird Fused multiply add unterstützt. Die GFLOPS können wie folgt berechnet werden.

$$2,9 * 8 * 8 * 2 = 371,2 GFLOP/s$$

Im besten Fall erreichen wir Zwei Prozent der maximal möglichen Leistung. Der erste große unterschied der auffällt ist, das wir auf einem Kern statt auf acht rechnen. Der zweite unterschied ist, dass wir Double Precision statt single Presicion benutzen. Hierfür wird Fused multiply add nicht unterstützt. Wir können also eine neue Rechnung aufstellen. Für unser Programm ergibt sich eine maximale Performance von:

$$2,9 * 8 = 23,2 GFLOP/s$$

Die beste Variante erreicht 25% der Leistung.

Gründe warum die Leistung in unserer Implementierung niedriger sind, als die theoretische Leistung sind:

1. Die B Matrix wird vor und nach der Berechnung transponiert. Dies benötigt quadratisch Zeit, in Bezug auf die Matrixgröße.
2. Durch die Zählschleifen werden kubisch viele Sprünge durchgeführt.
3. Die größeren Matrizen passen nicht in die Caches. Die benötigten Zahlen müssen aus dem Hauptspeicher geladen werden. Dies ist teuer.