MATRIX MULTIPLIKATION MIT FPU

Optimierung der Matrix Multiplikation mit FPU und Streaming-Data Befehlen (MMX, SSE, SSE2, AVX).

30. Juli 2021

Boris Foko Kouti (s0559792)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Code-Verzeichnis

1 Einleitung

Die Matrix Multiplikation in der linearen Algebra ist eine binäre Operation, die eine Matrix aus zwei Matrizen erzeugt. Bei der Matrixmultiplikation muss die Anzahl der Spalten der ersten Matrix gleich der Anzahl der Zeilen der zweiten Matrix sein. Die resultierende Matrix, das so genannte Matrixprodukt, hat die Anzahl der Zeilen der ersten und die Anzahl der Spalten der zweiten Matrix [1, 2].
Die meisten Algorithmen zur Berechnung dieses Produkts haben eine Komplexität von $O(n^3)$ bei $n * n$ Matrizen. Bei wachsenden n-Werte wird benötigt dieser Algorithmus immer mehr Zeit und Energie. Im Rahmen der Lehrveranstaltung MWP Wahlpflichtmodul/GIT Green IT sind wir beauftragt worden, dieser Algorithmus mit Hilfe von FPU (über die Floating-Point Execution Unit [3]) zu optimieren und gegebenenfalls Operationen über SIMD (Single instruction, multiple data) zu parallelisieren. Es geht also bei der vorliegenden Arbeit nicht an der ersten Linie darum die Komplexität dieses Algorithmus zu reduzieren (auch wenn über Unrolling mit SIMD oder FPU mache Vergleich-Operationen entfallen), sondern darum die einzeln Operationen zu beschleunigen (FPU) oder zu parallelisieren (SIMD).
In den nächsten Abschnitten wird zunächst auf den Algorithmus für den Matrix-Produkt eingegangen, dann werden in dieser Reihenfolge FPU und SIMD (über Intel Intrinsics) kurz vorgestellt. Im Anschluss dazu werden sowohl die Umsetzung, als auch die erzielten Ergebnisse präsentiert. Zum Schluss wird noch einmal die gesamte Arbeit mit den wichtigsten Erkenntnissen zusammengefasst.

2 Grundlagen zu FPU und Intrinsics

Dieses Kapitel setzt die Grundlagen für weitere Bearbeitung dieser Thematik. Es wird sowohl auf dem Algorithmus für das Matrixprodukt, als auch auf FPU und SIMD mit Intel Intrinsics eingegangen.

2.1 Matrix Multiplikation

Zur mathematischen Definition einer Matrixmultiplikation werden hier zwei Matrizen A $(m * n)$ und B $(n * p)$ betrachtet:

$$\text{A}= \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & . & ... \\ a_{m1} & a_{m2} & ... & a_{mn} \end{pmatrix} \text{ und B}= \begin{pmatrix} b_{11} & b_{12} & ... & b_{1p} \\ b_{21} & b_{22} & ... & b_{2p} \\ ... & ... & . & ... \\ b_{n1} & b_{n2} & ... & b_{np} \end{pmatrix}$$

Das Produkt dieser beiden Matrizen C = AB von der Größe $(m * p)$ ist definiert als:

$$\text{C}= \begin{pmatrix} c_{11} & c_{12} & ... & c_{1p} \\ c_{21} & c_{22} & ... & c_{2p} \\ ... & ... & . & ... \\ c_{m1} & c_{m2} & ... & c_{mp} \end{pmatrix}, \text{sodass } c_{ij} = a_{ij}b_{ij}+a_{i2}b_{i2}+...+a_{in}b_{inj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

mit $i = 1, ..., m$ und $j = 1, ..., p$.

Dieser Vorgang wird algorithmisch in dem Pseudocode 1 wiedergegeben.

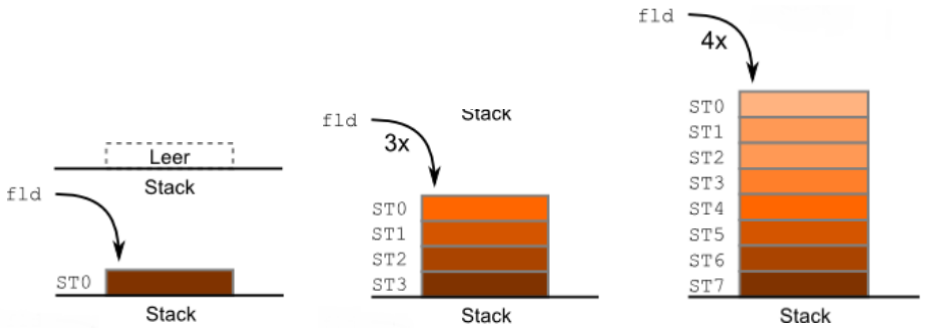
```
Input: matrices A and B
Let C be a new matrix of the appropriate size
For i from 1 to m:
    For j from 1 to p:
        Let sum = 0
        For k from 1 to n:
            Set sum = sum + A[i][k] x B[k][j]
        Set C[i][j] = sum
Return C
```

Code 1: Matrix Produkt Pseudo-Code

Die Standard-Implementierung (zum Beispiel in c++ oder c) der Operation $sum = sum + A[i][k] * B[k][j]$ bei realen Zahlen oder Gleitkommazahlen stellt sich eher als ineffizient und rechenaufwändiger heraus. Dies kann durch Einsatz der Floating-Point Execution Unit (FPU) verbessert werden. Im nächsten Abschnitt wird auf FPU genau eingegangen.

2.2 Floating Point Unit

Die arithmetisch logische Einheit (ALU) eines normalen Prozessors kann nur mit ganzzahlige Werte arbeiten. Dies mag bei manchen Operationen ausreichen, ist aber oft notwendig mit Dezimal- oder Fließkomma-Zahlen zu arbeiten. Dies kann zwar noch mit Assemblerroutinen basierend auf die Standard ALU-Operationen umgesetzt werden, allerdings sehr ineffizient. Zur effizienteren Lösung dieses Problem wurde ein hochspezialisierter Co-Prozessor der "Floating Point Unit" entwickelt. FPU ins Deutsch Fließkomma-Recheneinheit ermöglicht über seine acht 80-bits Registers (ST0 bis ST7) die Verarbeitung von Zahlen mit Nachkommastellen. Operanden sowie Ergebnisse werden in den acht verfügbaren Registern geladen und gespeichert. Das Laden eines Wertes in dem Register ST0 erfolgt immer über den Befehl **fld**. Dieser lädt den übergebenen Wert in ST0. Sollte aber davor schon ein Wert in ST0 gewesen sein, so wird dieser zunächst nach ST1 verschoben und erst danach wird der neue Wert in ST0 geladen. Dieser Prozess wird in der Abbildung 1 veranschaulicht. Im Debug Modus lassen sich diese Register auch in Microsoft Visual Studio über $Men\ddot{u} \rightarrow Debuggen \rightarrow Fenste \rightarrow Register \rightarrow Kontext - Men\ddot{u}(rechteMaustaste) \rightarrow FloatingPoint$.



(a) Start und 1 * fld (b) und 3 * fld (c) und 4 * fld

Abbildung 1: FPU Register Befüllung [4]

Sollte für eine Operation einen Wert benötigt sein, der in STN (mit 0 < N <= 7) liegt, kann der Werte in STO mit dem in STN üder den Befehl **fst Ziel** vertauscht werden (Ziel ist hier STN). Die Operation $Setsum = sum + A[i][k]xB[k][j]$ könnte dann wie folgt aussehen:

```
1 fld     dword ptr[esi + ecx * 4] // Load A[i][k]
2 fmul    dword ptr[edi + edx * 4] // Mult B[k][j]
3 mov     ecx, dword ptr[MATRIX_C] // Load C[i][j]
4 fadd    dword ptr[ecx + eax * 4] // C[i][j] + A[i][k]*B[k][j]
```

Code 2: Load Add und Mul mit FPU

2.3 SIMD-Operationen

Single instruction, multiple data (SIMD) ist eine Art der Parallelverarbeitung aus der Flynn Taxonomie [5]. SIMD beschreibt Computer mit mehreren Verarbeitungseinheiten, die dieselbe Operation für mehreren Datensätzen gleichzeitig durchführen. Dies basiert auf dem Prinzip der Parallelität auf Datenebene [6]. Diese Operationen haben sich über die Jahren von MMX in 1997 bis Core und weitere heute. Ein kleiner Auszug dieser Evolution für den Intel Prozessor in in der Abbildung 2 zu sehen.

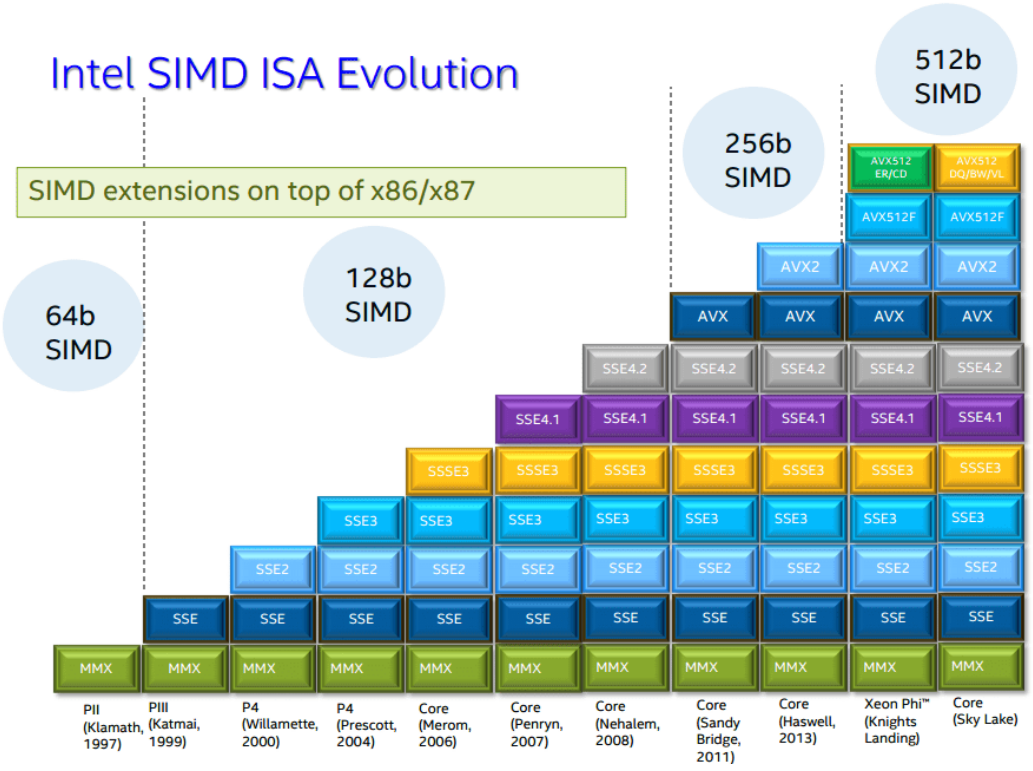


Abbildung 2: Evolution der Intel SIMD-Architekturen [7]

Die erste Variante MMX nutzt die acht 80-bits Register von FPU. Diese werden hier allerdings MM0 bis MM7 genannt. Damit lässt sich eine Operation gleichzeitig auf 8 *char* (8-Bit), 4 *short* (16-Bit), zwei *int* (32-Bit) und ein *int64* (64-Bit) durchführen. Diese Informationen über MMX und weitere sind in der Tabelle 1 zusammengefasst.

Tabelle 1: SIMD gleichzeitige Operanden nach Datentyp [8]

	char	short	int	int64	float	double
MMX	8	4	2	1	0	0
SSE	0	0	0	0	4	0
SSE2	16	8	4	2	4	2
AVX	16	8	4	2	8	4
AVX2	32	16	8	4	8	4
AVX512F	32	16	16	8	16	8
AVX512BW	64	32	16	8	16	8

Im Debug Modus lassen sich die Register für MMX und SSE auch in Microsoft Visual Studio über $Men\ddot{u} \rightarrow Debuggen \rightarrow Fenste \rightarrow Register \rightarrow Kontext - Men\ddot{u}(rechteMaustaste) \rightarrow MMX(oderSSE)$.

3 Implementierung

In diesem Kapitel werden die wichtigsten Implementierungsschritte vom Lösungsaufbau, über die Generierung der Matrizen (A und B) bis hinzu den Optimierungen mit FPU, MMX, SSE, SSE2 und AVX besprochen. Zur Umsetzung des hier vorgestellten Programms wurde die Programmiersprache C verwendet. Der Code wurde mit Hilfe der IDE MS Visual Studio 2019 entwickelt, kompiliert, debuggt und getestet.

3.1 Daten Generierung

```
// Declaration of matrix for double and float operations
double *MATRIX_D_A, *MATRIX_D_B, *MATRIX_D_C;
float *MATRIX_F_A, *MATRIX_F_B, *MATRIX_F_C;
// Memory allocation with malloc
MATRIX_D_A = malloc(N * N * sizeof(double));
MATRIX_D_B = malloc(N * N * sizeof(double));
MATRIX_D_C = malloc(N * N * sizeof(double));
MATRIX_F_A = malloc(N * N * sizeof(float));
MATRIX_F_B = malloc(N * N * sizeof(float));
MATRIX_F_C = malloc(N * N * sizeof(float));
// Init Matrix with random value on false and 0 on true
initMatrixD(MATRIX_D_A, false, N);
initMatrixD(MATRIX_D_B, false, N);
initMatrixD(MATRIX_D_C, true, N);
// Copy Double values to Float values
copyMatrixDF(MATRIX_D_A, MATRIX_F_A, N);
copyMatrixDF(MATRIX_D_B, MATRIX_F_B, N);
copyMatrixDF(MATRIX_D_C, MATRIX_F_C, N);
```

Code 3: Generierung der Matrizen A und B

Die im Codestrück 3 verwendete Funktion *initMatrixD* wird hier implementiert:

```
void initMatrixD(double* MATRIX, bool initToZero, int N) {
  if (initToZero) {
    // Init MATRIX Value to 0 (see project)
  } else {
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        MATRIX[i * N + j] = random_double(MIN, MAX);
      }
    }
  }
}
```

Code 4: Init Funtionene für die Generierung der Matrizen A und B

3.2 Matrix Multiplikation Standard Version

Die Standard Version der Matrix Multiplikation ist analog zum Algorithmus 1 implementiert. Zur Messung der Rechenzeit wird ein *clock_t* aus der Klasse *time.h* verwendet.

```
int multipy(const double* MATRIX_A, const double* MATRIX_B,
    double* MATRIX_C, int N) {
  int msec = 0;
  clock_t start, finish;
  start = clock();

  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      // Multiply the row of MATRIX_A by the column of
      // MATRIX_B to get the row of MATRIC_C.
      for (int k = 0; k < N; k++) {
        MATRIX_C[i * N + j] += MATRIX_A[i * N + k] * MATRIX_B
    [k * N + j];
      }
    }
  }
  finish = clock();
  msec = 1000.0 * (finish - start) / CLOCKS_PER_SEC;

  return msec;
}
```

Code 5: Standard Version der Matrix Multiplikation

3.3 Matrix Multiplikation mit FPU über Inline Assembler

Die Matrix Multiplikation mit FPU verläuft analog zu der Standard Version mit dem Unterschied, dass diese zum Einen als Inline Assembler [9, 10] geschrieben ist und zum Anderen bei Multiplikation und Addition Operationen die FPU Register und Befehle einsetzt.

```
/*
By default start first loop
LN1 First for loop increment counter
LN2 First for loop compare i with N
After LN2 start the second loop
LN3 Second for loop increment counter
LN4 Second for loop compare i with N
After LN4 start the third loop
LN5 Third for loop increment counter
LN6 Multiplication and Addition
```

```
LN7 Close third for loop
LN8 Close second for loop
LN9 Close first for loop
LN10, LN11 Clean and return
*/
int multiply_fpu(float* MATRIX_A, float* MATRIX_B, float*
    MATRIX_C, int N) {
  int msec = 0;
  clock_t start, finish;
  start = clock();

  __asm {
  //
    push        ebp
    sub         esp, 0E4h
    push        ebx
    push        esi
    push        edi
    lea         edi, [ebp - 0E4h]
    mov         ecx, 39h
    mov         eax, 0CCCCCCCCh
    rep stos    dword ptr es : [edi]
    mov         dword ptr[ebp - 8], 0
    jmp LN2
    LN1 :
      mov eax, dword ptr[ebp - 8]
      add eax, 1
      mov dword ptr[ebp - 8], eax
    LN2 :
      mov         eax, dword ptr[ebp - 8]
      cmp         eax, dword ptr[N]
      jge LN10
      mov         dword ptr[ebp - 14h], 0
      jmp LN4
    LN3 :
      mov         eax, dword ptr[ebp - 14h]
      add         eax, 1
      mov         dword ptr[ebp - 14h], eax
    LN4 :
      mov         eax, dword ptr[ebp - 14h]
      cmp         eax, dword ptr[N]
      jge LN9
      mov         dword ptr[ebp - 20h], 0
      jmp         LN6
    LN5 :
      mov         eax, dword ptr[ebp - 20h]
      add         eax, 1
      mov         dword ptr[ebp - 20h], eax
    LN6 :
```

```
59        mov          eax, dword ptr[ebp - 20h]
60        cmp          eax, dword ptr[N]
61        jge          LN8
62        mov          eax, dword ptr[ebp - 8]
63        imul         eax, dword ptr[N]
64        add          eax, dword ptr[ebp - 14h]
65        mov          ecx, dword ptr[ebp - 8]
66        imul         ecx, dword ptr[N]
67        add          ecx, dword ptr[ebp - 20h]
68        mov          edx, dword ptr[ebp - 20h]
69        imul         edx, dword ptr[N]
70        add          edx, dword ptr[ebp - 14h]
71        mov          esi, dword ptr[MATRIX_A]
72        mov          edi, dword ptr[MATRIX_B]
73        fld      dword ptr[esi + ecx * 4]
74        fmul         dword ptr[edi + edx * 4]
75        mov          ecx, dword ptr[MATRIX_C]
76        fadd         dword ptr[ecx + eax * 4]
77        mov          edx, dword ptr[ebp - 8]
78        imul         edx, dword ptr[N]
79        add          edx, dword ptr[ebp - 14h]
80        mov          eax, dword ptr[MATRIX_C]
81        fstp         dword ptr[eax + edx * 4]
82      LN7 :
83        jmp LN5
84      LN8 :
85        jmp LN3
86      LN9 :
87        jmp LN1
88      LN10 :
89        pop          edi
90      LN11 :
91        pop esi
92        pop ebx
93        add          esp, 0E4h
94        cmp          ebp, esp
95        mov          esp, ebp
96        pop ebp
97        ret
98    }
99
100   finish = clock();
101   msec = 1000.0 * (finish - start) / CLOCKS_PER_SEC;
102
103   return msec;
104 }
```

Code 6: Matrix Multiplikation mit FPU

3.4 Matrix Multiplikation mit MMX Intel Intrinsics

Die MMX kann keine Fließkommazahlen verarbeiten, daher werden wir short(int16) als Input und in32 als Ergebnis verwenden. MMX Verarbeitet 4 short gleichzeitig. MMX ist aus diesem Grund nicht für die Optimierung relevant. Es wird hier trotzdem beispielhaft aufgeführt.

```
int multipy_mmx(const short* MATRIX_A, const short* MATRIX_B,
    int* MATRIX_C, int N)
{
  int msec = 0;
  clock_t start, finish;
  start = clock();
  __m64 a_line,  b_line;

  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      __m64 sum = _mm_setzero_si64();   // init sum to zero
      int sum_down = 0;
        // add 4 short at the same time using the MMX
   _m_pmaddwd function
        // equivalent to asm pmaddwd
      for (int k = 0; k < N; k+=4) {
        a_line = _mm_set_pi16(MATRIX_A[i * N + k],
        MATRIX_A[i * N + k + 1], MATRIX_A[i * N + k + 2],
        MATRIX_A[i * N + k + 3]);
        b_line = _mm_set_pi16(MATRIX_B[k * N + j],
        MATRIX_B[(k + 1) * N + j], MATRIX_B[(k + 2) * N + j],
        MATRIX_B[(k + 3) * N + j]);
        sum = _m_paddw(sum, _m_pmaddwd(a_line, b_line));
      }
      sum_down = _mm_cvtsi64_si32(sum);   // save low 32 bits
      sum = _m_psrlqi(sum, 32);     // shift right on 32 bits
      sum_down += _mm_cvtsi64_si32(sum);  // save low 32 bits
      MATRIX_C[i * N + j] = sum_down;
    }
  }
  // Clear the MMX registers and MMX state
  _m_empty();
  _mm_empty();
  finish = clock();
  msec = 1000.0 * (finish - start) / CLOCKS_PER_SEC;

  return msec;
}
```

Code 7: Matrix Multiplikation mit MMX

3.5 Matrix Multiplikation mit SSE Intel Intrinsics

Über SSE Register lassen sich 4 Floats gleichzeitig verarbeiten. SSE ist dadurch mindestens 4 mal schneller als der Standard.

```
1
2  int multipy_sse(const float* MATRIX_A, const float* MATRIX_B,
       float* MATRIX_C, int N)
3  {
4    int msec = 0;
5    clock_t start, finish;
6    start = clock();
7    for (int i = 0; i < N; i++) {
8      for (int j = 0; j < N; j += 4) {
9        __m128 sum = _mm_load_ps(MATRIX_C + i * N + j);
10       for (int k = 0; k < N; k ++) {
11         sum = _mm_add_ps(_mm_mul_ps(
12         _mm_set1_ps(MATRIX_A[i * N + k]),
13         _mm_load_ps(MATRIX_B + k * N + j)), sum);
14       }
15       _mm_store_ps(MATRIX_C + i * N + j, sum);
16     }
17   }
18   finish = clock();
19   msec = 1000.0 * (finish - start) / CLOCKS_PER_SEC;
20   return msec;
21 }
```

Code 8: Matrix Multiplikation mit SSE

3.6 Matrix Multiplikation mit SSE2 Intel Intrinsics

Da SSE2 dieselbe Anzahl von Float (nämlich 4) wie SSE gleichzeitig verarbeiten kann, ist hier keine Verbesserung zu Erwarten. Statt dessen wurde hier Testweise die Operation auf Double (2 Double gleichzeitig) durchgeführt.

```
1  int multipy_sse2(const double* MATRIX_A, const double*
       MATRIX_B, double* MATRIX_C, int N)
2  {
3    int msec = 0;
4    clock_t start, finish;
5    start = clock();
6    for (int i = 0; i < N; i++) {
7      for (int j = 0; j < N; j += 2) {
8        __m128d sum = _mm_load_pd(MATRIX_C + i * N + j);
9        for (int k = 0; k < N; k++) {
10         sum = _mm_add_pd(_mm_mul_pd(
11         _mm_set_pd1(MATRIX_A[i * N + k]),
12         _mm_load_pd(MATRIX_B + k * N + j)), sum);
```

```
      }
      _mm_store_pd(MATRIX_C + i * N + j, sum);
    }
  }
  finish = clock();
  msec = 1000.0 * (finish - start) / CLOCKS_PER_SEC;
  return msec;
}
```

Code 9: Matrix Multiplikation mit SSE2

3.7 Matrix Multiplikation mit AVX Intel Intrinsics

AVX ermöglicht die Verarbeitung von 8 Double und 16 Float. AVX ist also mindestens 16 mal schneller als die Standard Multiplikation mit Float.

```
int multipy_avx(const float* MATRIX_A, const float* MATRIX_B,
    float* MATRIX_C, int N)
{
  int msec = 0;
  clock_t start, finish;
  start = clock();
  const int block_width = N >= 256 ? 512 : 256;
  const int block_height = N >= 512 ? 8 : N >= 256 ? 16 : 32;
  for (int row_offset = 0; row_offset < N; row_offset +=
   block_height) {
    for (int column_offset = 0; column_offset < N;
   column_offset += block_width) {
      for (int i = 0; i < N; ++i) {
        for (int j = column_offset; j < column_offset +
   block_width && j < N; j += 8) {
          __m256 sum = _mm256_load_ps(MATRIX_C + i * N + j);
          for (int k = row_offset; k < row_offset +
   block_height && k < N; ++k) {
            sum = _mm256_fmadd_ps(_mm256_set1_ps(MATRIX_A[i *
    N + k]), _mm256_load_ps(MATRIX_B + k * N + j), sum);
          }
          _mm256_store_ps(MATRIX_C + i * N + j, sum);
        }
      }
    }
  }
  finish = clock();
  msec = 1000.0 * (finish - start) / CLOCKS_PER_SEC;

  return msec;
}
```

Code 10: Matrix Multiplikation mit AVX

4 Ergebnisse

Dieser Kapitel befasst sich mit den Ergebnissen, die im Laufe der mit der Anwendung durchgeführten Tests und Messungen gesammelt wurden. Diese Ergebnisse bestehen aus den gemessenen Rechenzeiten der verschiedenen Implementierungen, die mit VTune [11] gemessenen Clocks und den entsprechenden Energieverbrauch. Die in diesem Kapitel präsentierten Messungen sind auf einem Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, 2.30 GHz mit 8 logische Kern durchgeführt worden.

4.1 Rechenzeiten der verschiedenen Varianten

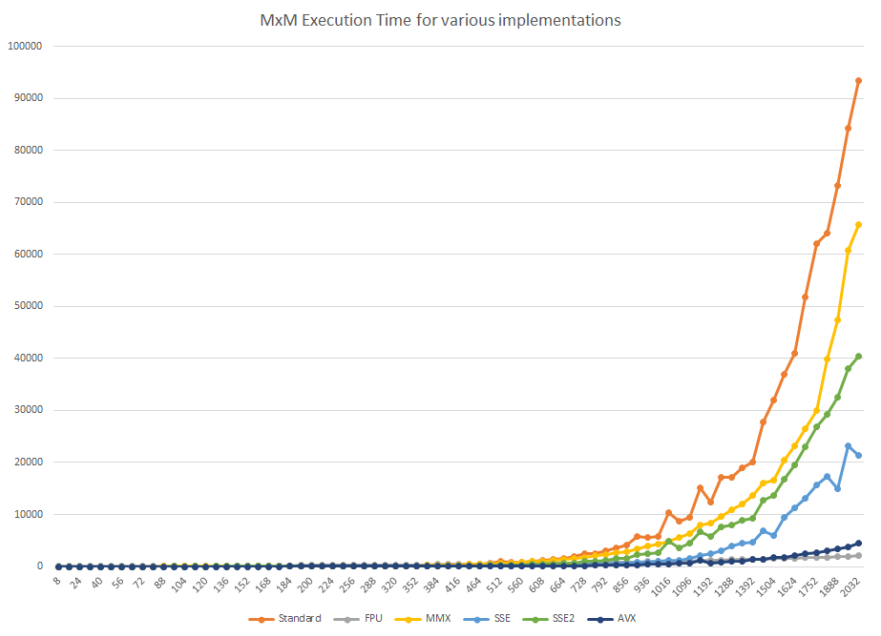


Abbildung 3: Darstellung der Rechenzeiten der verschiedenen Implementierungen (Standard, FPU, MMX, SSE, SSE2, AVX)

Die Kurven in Abb 3 sind aus den Daten aus dem Projekt MxM_Streaming _Data abgeleitet. Die “statistic.xlsx” Datei dazu wird mit dieser Dokumentation mit geliefert. Zu den Ergebnissen selbst muss unterzeichnet werden, dass MMX und SSE2 hier nicht von großer Relevanz sind, da bei dem Ersten keine Floating-Point Operationen möglich sind (wurde ausschließlich für Testzwecke implementiert), und bei dem Zweiten hier SSE2 Double verwendet wurden (da SSE2 genauso wie SSE 4 Float verarbeiten kann, besteht da kein Mehrwert). Auf der Grafik sind zwei klare Gewinner zu erkennen,

nämlich FPU und AVX, während die Standard Version exponentiell steigt, sind FPU und AVX eher linear mit einer sehr kleinen Steigung.

4.2 VTune Messungen der Clocks und weiteres

Hier wird ein Auszug aus den Messungen vorgestellt. Die vollständige Messungen sind als Kommentar in dem Code abgegeben worden.

Tabelle 2: Clocks und weitere Messungen mit V-Tune in Debug Mode

Function	CPU Time	Clockticks	Instructions Retired	CPI Rate
multipy	14.600s	48,727,800,000	29,050,200,000	1.677
multiply_fpu	6.874s	27,883,800,000	29,030,400,000	0.961
multipy_mmx	5.000s	18,208,800,000	18,304,200,000	0.995
multipy_sse2	5.267s	17,299,800,000	12,385,800,000	1.397
multipy_sse	1.249s	4,352,400,000	6,179,400,000	0.704
multipy_avx	0.468s	1,742,400,000	4, 163,400,000	0.419

Tabelle 3: Clocks und weitere Messungen mit V-Tune in Release Mode

Function	CPU Time	Clockticks	Instructions Retired	CPI Rate
multipy	3.166s	13,559,400,000	8,616,600,000	1.574
multipy_mmx	3.101s	12,585,600,000	6,741,000,000	1.867
multipy_sse2	1.590s	5,414,400,000	4,842,000,000	1.118
multipy_sse	0.495s	1,733,400,000	2,421,000,000	0.716
multipy_avx	0.151s	552,600,000	2,188,800,000	0.252
multiply_fpu	keine Messung da es Inline ASM			

4.3 VTune Messung des Energieverbrauchs

Eine Messung des Energieverbrauchs ist ausschließlich für die Standard-Umsetzung und die optimierte FPU Variante durchgeführt worden.

Tabelle 4: Messung des Energieverbrauchs

Executable	Debug Energy Consumption (mJ)	RELEASE Energy Consumption (mJ)
MxM_Standard	140,469.360	59,315.186
Executable	130,589.722	2,218.933

5 Zusammenfassung

Ziel dieser Arbeit ist es gewesen, mit FPU und SIMD-Operatoren die Matrix Multiplikation zu optimieren. Es wurde im Kapitel 2 zunächst den Versuch unternommen, die Matrix Multiplikation selbst sowohl mathematisch als auch algorithmisch genau zu definieren. Dann wurden abwechselnd FPU und SIMD mit Intel Intrinsics genau erörtert. So konnte der Grundstein für die Implementierung der verschiedenen Varianten (FPU und SIMD mit MMX, SSE, SSE2 und AVX) gelegt werden. Die Umsetzung erfolgte im C Code mit der IDE MS Visual Studio 2019 (siehe Kapitel 3). Mit den aufgenommenen Messungen konnte gezeigt werden, wie effizienter und schneller die FPU Variante war im Vergleich zu der Standard-Version, sowohl was die Laufzeit angeht, die Anzahl der Clocks, als auch den Energieverbrauch. Es konnte durch die Kurven in Abb. 3 eine gewisse Tendenz, was das Verhältnis der Varianten zu der Größe der Input Matrizen angeht. Die Standard Version verhält sich exponentiell, während die FPU und AVX Varianten eher linear mit einer sehr kleinen Steigung verlaufen.

Literatur

[1] *Math Insight: Multiplying matrices and vectors.* URL: https://mathinsight.org/matrix_vector_multiplication. (visited on 25.07.2021).

[2] *Algebra Symbols: A comprehensive collection of 225+ symbols used in algebra, categorized by subject and type into tables along with each symbol's name, usage and example.* URL: https://mathvault.ca/hub/higher-math/math-symbols/algebra-symbols/. (visited on 25.07.2021).

[3] S. F. Anderson u. a. "The IBM System/360 Model 91: Floating-Point Execution Unit". In: *IBM Journal of Research and Development* 11.1 (1967), S. 34–53. DOI: 10.1147/rd.111.0034.

[4] Thorsten Thormählen. *Technische Informatik I FPU, MMX, SSE, x86-64.* URL: https://www.mathematik.uni-marburg.de/~thormae/lectures/ti1/ti_10_3_ger_web.html#1. (visited on 25.07.2021).

[5] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), S. 948–960. DOI: 10.1109/TC.1972.5009071.

[6] W. Daniel Hillis und Guy L. Steele. "Data Parallel Algorithms". In: *Commun. ACM* 29.12 (Dez. 1986), S. 1170–1183. ISSN: 0001-0782. DOI: 10.1145/7902.7903. URL: https://doi.org/10.1145/7902.7903.

[7] Mostafa Soliman. "State-of-the-Art in Processor Architecture". In: (Jan. 2016), S. 13–14. DOI: 10.13140/RG.2.1.3105.8000. URL: https://www.researchgate.net/profile/Mostafa_Soliman3/publication/288993171_State-of-the-%20Art_in_Processor_Architecture/links/56881ca308ae19758398eb08/State-of-the-Art-in-Processor-Architecture.pdf.

[8] *Intel® Intrinsics Guide: The Intel® Intrinsics Guide is a reference tool for Intel intrinsics.* URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#. (visited on 25.07.2021).

[9] *MASM for x64 (ml64.exe): Visual Studio includes both 32-bit and 64-bit hosted versions of Microsoft Assembler (MASM) to target x64 code.* URL: https://docs.microsoft.com/en-us/cpp/assembler/masm/masm-for-x64-ml64-exe?view=msvc-160. (visited on 25.07.2021).

[10] *asm declaration: asm-declaration gives the ability to embed assembly language source code within a C++ program.* URL: https://en.cppreference.com/w/cpp/language/asm. (visited on 25.07.2021).

[11] *Intel® VTune™ Profiler: Quickly Find and Fix Performance Bottlenecks and Realize All the Value of Your Hardware.* URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html#gs.7n9917. (visited on 25.07.2021).