

$$\begin{pmatrix} 2 & 5 & 7 \\ 8 & 9 & 10 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 4 \\ 5 \end{pmatrix}$$
$$= \begin{pmatrix} 2 \cdot 2 + 5 \cdot 4 + 7 \cdot 5 \\ 8 \cdot 2 + 9 \cdot 4 + 10 \cdot 5 \\ 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 5 \end{pmatrix}$$
$$= \begin{pmatrix} 59 \\ 102 \\ 25 \end{pmatrix}$$

Jennifer Vormann | s0559531

## Matrixmultiplikation

Code with Streaming SIMD Extensions

GreenIT – AI (M)

Prof. Dr. em. Vesselin Iossifov

12.07.21

# Motivation

- Meine Arbeit bei 50Hertz
- Viel Energie notwendig für Programme, Datenzentren, Dienste etc.
- Serverfabriken oft noch mit Kohlekraft betrieben
- Verantwortung als Developer übernehmen
- Nachhaltigkeit fester Bestandteil meines Lebens



[https://3.bp.blogspot.com/-VF540Js2h2A/WnYI9L8FRGI/AAAAAAAAG0/Y-g61DwFaNsna12XTojGol\\_yai-sCMz5kQCIcBGAs/s320/59734170.jpg](https://3.bp.blogspot.com/-VF540Js2h2A/WnYI9L8FRGI/AAAAAAAAG0/Y-g61DwFaNsna12XTojGol_yai-sCMz5kQCIcBGAs/s320/59734170.jpg)

# Matrixmultiplikation

## Aufgabenstellung:

- Matrixmultiplikation – Sequentiell
- Matrixmultiplikation – Mit SIMD Extensions (Intrinsics oder Assembler)

## Lösungsweg:

- DEBUG & RELEASE Modus
- Matrixmultiplikation umgesetzt mit naivem Ansatz und kleiner Matrix
- Getestet (mit Prints im DEBUG Modus)
- DEBUG FAST & RELEASE FAST Modus ergänzt
- Weitere Funktionen mit SIMD und Intrinsics implementiert
- Getestet und vermessen

# Matrix-multiplikation

## 1. Sequentielle Implementierung

Wo? **matrix/multiply.c**

Was?

- Spalten & Zeilen
- Dynamic mem allocation
- Multiplikation
- return matrix

```
#else
matrix_t matrix_multiply(matrix_t a, matrix_t b)
{
    size_t a_m = a.nrows;
    size_t a_n = a.ncols;
    size_t b_n = b.ncols;

    int *A = a.values;
    int *B = b.values;
    int *C = matrix_malloc(a_m * b_n * sizeof(int));

    memset(C, 0, a_m * b_n * sizeof(int));

    for (size_t i = 0; i < a_m; i++) {
        for (size_t j = 0; j < b_n; j++) {
            for (size_t k = 0; k < a_n; k++) {
                C[i * b_n + j] += A[i * a_n + k] * B[k * b_n + j];
            }
        }
    }

    return (matrix_t) { .nrows = a_m, .ncols = b_n, .values = C };
}
#endif // FAST
```

# Matrixmultiplikation

## 2. SIMD Implementierung (single instruction multiple data) mit Intrinsics

**Zweck:** Programme durch Parallelisierung auf Instruktionslevel beschleunigen

Wo? **matrix/multiply.c**

Was ist **anders**?

- memcpy -> Kopieren von Matrix, weil transponieren in-place arbeitet
- matrix\_transpose -> Matrix transponieren, für Cache Optimierungen
- vector\_dot\_product\_4x1 -> Dot-Produkt von 4-dimensionalen Vektoren
- unrolling -> Verarbeiten der rows in 16-int-Blöcken wenn möglich

# Matrixmultiplikation

Die Funktion mit SIMD & Intrinsics im FAST Modus:

```
static inline int vector_dot_product_4x1(int *a0, int *b0)
```

```
__m128i a = _mm_load_si128((__m128i *)a0);
__m128i b = _mm_load_si128((__m128i *)b0);
__m128i products = _mm_mullo_epi32(a, b);    // pairwise multiplication of 32-bit signed integers
__m128i sum = _mm_hadd_epi32(products, products); // horizontal addition
sum = _mm_hadd_epi32(sum, sum); // creates 4 times all sums multiplied
return _mm_cvtsi128_si32(sum); // convert lower 32 bits to a 32-bit signed integer
```

# Matrixmultiplikation

**Unroll** in **matrix\_t matrix\_multiply(matrix\_t a, matrix\_t b)**

```
// process the row in 16-int chunks
for (size_t k = 0; k + 15 < a_n; k += 16) {
    size_t offset = i * a_n + k;
    //unroll
    C[i * b_n + j] += vector_dot_product_4x1(&A[offset], &B[offset]);
    C[i * b_n + j] += vector_dot_product_4x1(&A[offset + 4], &B[offset + 4]);
    C[i * b_n + j] += vector_dot_product_4x1(&A[offset + 8], &B[offset + 8]);
    C[i * b_n + j] += vector_dot_product_4x1(&A[offset + 12], &B[offset + 12]);
}
// process the remainder (if any) in 4-int chunks
for (size_t k = a_n & -16; k + 3 < a_n; k += 4) {
    C[i * b_n + j] += vector_dot_product_4x1(&A[i * a_n + k], &B[j * a_n + k]);
}
// process the remainder (if any) in 1-int chunks
for (size_t k = a_n & -4; k < a_n; k++) {
    C[i * b_n + j] += A[i * a_n + k] * B[j * a_n + k];
}
```

# **Messungen - Overview**

## **Sequentiell versus SIMD Extensions**

### **Fakten Messungen allgemein:**

CPU: Intel(R) Processor code named Kabylake ULX  
Frequency: 1,8 GHz  
Local CPU count: 8  
Date: 22.06.2021, 22 C°

### **Fakten Messungen Power (mW) & Energy (mJ):**

CPU Intel's CPU i5 11.Gen Tigerlake  
Frequency: 2,4 GHz  
Date: 22.06.2021, 19 C°

# Messungen – Detail Sequentiell

Elapsed Time <small>⌚</small> : 17.569s <small>⌚</small>		
Clockticks:	50,913,600,000	
Instructions Retired:	30,643,200,000	
CPI Rate <small>⌚</small> :	1.661 <small>⌚</small>	
MUX Reliability <small>⌚</small> :	0.968	
⌚ Retiring <small>⌚</small> :	15.9%	of Pipeline Slots
⌚ Front-End Bound <small>⌚</small> :	0.9%	of Pipeline Slots
⌚ Bad Speculation <small>⌚</small> :	0.2%	of Pipeline Slots
⌚ Back-End Bound <small>⌚</small> :	82.9% <small>⌚</small>	of Pipeline Slots

Sequentielle Implementierung

TESTEN mit x64 Release

Power (mW)  
10820.214

Energy (mJ)  
55503.845

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Average CPU Frequency
[Loop at line 99 in matrix_multiply]	17.192s	50,550,400,000	30,155,200,000	1.676	2.9 GHz
[Loop at line 20 in matrix_create_random]	0.014s	52,800,000	180,800,000	0.292	3.7 GHz
[Loop at line 98 in matrix_multiply]	0.004s	6,400,000	4,800,000	1.333	1.4 GHz

# Messungen – Detail

## SIMD Extensions ohne Unroll

Elapsed Time <small>⌚</small> : 2.519s		
Clockticks:	8,451,200,000	
Instructions Retired:	15,531,200,000	
CPI Rate <small>⌚</small> :	0.544	
MUX Reliability <small>⌚</small> :	0.987	
⌚ Retiring <small>⌚</small> :	53.8%	of Pipeline Slots
⌚ Front-End Bound <small>⌚</small> :	1.8%	of Pipeline Slots
⌚ Bad Speculation <small>⌚</small> :	0.5%	of Pipeline Slots
⌚ Back-End Bound <small>⌚</small> :	43.8%	of Pipeline Slots

---

SIMD Implementierung ohne Unroll

---

TESTEN mit x64 Release FAST

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Average CPU Frequency
vector_dot_product_4x1	1.689s	5,720,000,000	10,185,600,000	0.562	3.4 GHz
[Loop at line 87 in matrix_multiply]	0.736s	2,505,600,000	4,870,400,000	0.514	3.4 GHz
[Loop at line 19 in matrix_create_random]	0.018s	36,800,000	161,600,000	0.228	2.1 GHz
[Loop at line 42 in matrix_transpose]	0.006s	22,400,000	14,400,000	1.556	3.6 GHz
[Loop at line 67 in matrix_multiply]	0.001s	3,200,000	4,800,000	0.667	3.6 GHz

# Messungen – Detail

## SIMD Extensions mit Unroll

Elapsed Time <small>⌚</small> : 2.297s <small>⬇️</small>		
Clockticks:	7,686,400,000	
Instructions Retired:	13,140,800,000	
CPI Rate <small>⌚</small> :	0.585	
MUX Reliability <small>⌚</small> :	0.911	
⌚ Retiring <small>⌚</small> :	46.2%	of Pipeline Slots
⌚ Front-End Bound <small>⌚</small> :	1.4%	of Pipeline Slots
⌚ Bad Speculation <small>⌚</small> :	0.4%	of Pipeline Slots
⌚ Back-End Bound <small>⌚</small> :	52.1%	⬇️ of Pipeline Slots

Intrinsics Implementierung mit Unroll

TESTEN mit x64 Release

Power (mW)  
11445.789

Energy (mJ)  
17713.501

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Average CPU Frequency
vector_dot_product_4x1	1.669s	5,624,000,000	9,488,000,000	0.593	3.4 GHz
[Loop at line 62 in matrix_multiply]	0.526s	1,835,200,000	3,155,200,000	0.582	3.5 GHz
[Loop at line 20 in matrix_create_random]	0.015s	57,600,000	195,200,000	0.295	3.8 GHz
[Loop at line 35 in matrix_transpose]	0.008s	27,200,000	16,000,000	1.700	3.4 GHz
[Loop at line 60 in matrix_multiply]	0.003s	3,200,000	8,000,000	0.400	1.2 GHz
[Loop at line 70 in matrix_multiply]	0.001s	0	3,200,000	0.000	0.0 MHz
[Loop at line 74 in matrix_multiply]	0s	0	0	0.000	0.0 MHz



Repository: <https://github.com/FrauMauz/Matrixmultiplikation SIMD Github.git>  
Quelle: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>