Protokoll 1: Serielle Optimierung der Matrix Multiplikation auf CPUs

Gruppe 9: Marcel Beyer, Martin Carnein, Franz Lehmann

Aufgabe 1:

Berechnung der Peak Performance mit folgender Formel:

$$FLOPS = \frac{Instructions}{Cycle} * \frac{Clock\ Rate}{Core} * NumberOfCores$$

Laut Intel Webseite besitzt der *Intel Xeon Platinum 8470 Prozessor* zwei AVX-512 FMA-Einheiten, sodass sich daraus 64 Instruktionen pro Zyklus ergeben. Die Basisfrequenz pro Kern ist 2.00 GHz.
Damit ergibts sich folgende Lösung für einen Kern:

$$FLOPS = 64\frac{Instructions}{Cycle} * \frac{2.00\ GHz}{Core} * 1\ Core = 128\frac{GFLOP}{s}$$

Aufgabe 2:

- Schleifenvertauschung
  - man hat mehrere Schleifen miteinander verschachtelt bspw. eine for-Schleife innerhalb einer anderen, um zum Beispiel durch eine Matrix zu iterieren
  - je nachdem wie man durch die Matrix iteriert, erst durch alle Spalten oder durch alle Zeilen, kann es vorkommen, dass Zugriffe schlechter oder besser sind
  - so wäre es besser, wenn man so iteriert, dass man eine ganze Cache Zeile komplett durchläuft und nicht immer nur ein Element aus einer Cache Zeile nimmt und dann die Cache Zeile verdrängt, weil man das Element aus der nächsten Cache Zeile braucht
- Loop unrolling – Schleifen entrollen
  - wenn man davon ausgeht, dass man eine Schleife hat, in der man auf ein Array zugreift und mit diesem Berechnungen ausführt, wird für jede Berechnung ein Sprung und ein Vergleich benötigt => wenig Arbeit, viele Befehle
  - mit Loop Unrolling würde man in einem Schritt auf mehrere Stellen im Array zugreifen und darauf Berechnungen ausführen und dann in der Schleife direkt mehrere Schritte weiter gehen, somit müssen weniger Befehle für die gleiche Arbeit dekodiert werden (es wird aber auch mehr Befehlscache gebraucht)
- Blocking/Tiling
  - Bei großen Daten(mengen) kann es vorkommen, dass die Daten nicht alle auf einmal in den Cache passen und werden nur einmal genutzt bevor die verdrängt werden
  - Daten werden in Blöcke aufgeteilt und die Berechnungen werden auf den kleineren Blöcken ausgeführt, welche komplett in den Cache passen, sodass Elemente wiedergenutzt werden können
- Wiedernutzung von Werten statt Neuberechnung
  - Wenn Werte im Programm mehrfach berechnet werden, ist es sinnvoll diese bei der ersten Berechnung zwischenzuspeichern
  - damit braucht man zwar ein wenig mehr Speicher jedoch kann man sich damit Rechenleistung und ggf. auch Zeit sparen

Aufgabe 3:

Wir nehmen die Komplexität von Matrixmultiplikation, welche n^3ist. Außerdem nehmen wir Basisfrequenz pro Kern 2.00 GHz. Auf dieser Grundlage machen wir folgende Berechnung bzw. Abschätzung:

$$\frac{n^3}{2GHz} = 10\ s$$

$$n^3 = 20\ GHz * s$$

$$n = \sqrt[3]{20.000.000.000} \approx 2715$$

Das Ergebnis dieser Abschätzung zeigt, dass wir eine n*n Matrix mit n = 2715 benötigen um eine Laufzeit von mindestens 10 Sekunden zu erreichen.

Aufgabe 4:

Die korrekte Durchführung der Matrixmultiplikation haben wir durch Tests überprüft. Das Ergebnis der normalen Matrixmultiplikation ohne Optimierungen haben wir mit den Ergebnissen der Optimierungen verglichen und konnten somit herausfinden, ob unsere Algorithmen korrekt arbeiten. Somit konnten wir die korrekte Durchführung nach Änderungen im Code sicherstellen.

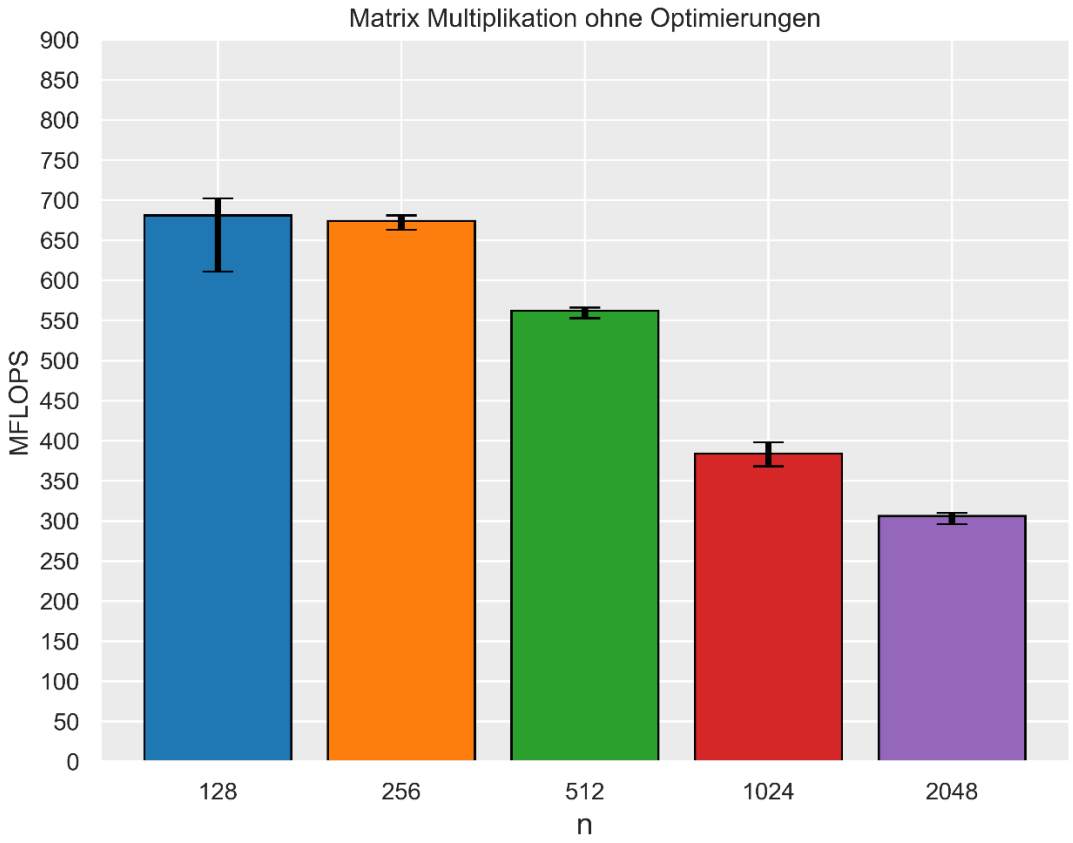
Aufgabe 5:

Folgend unsere Implementation für die Matrix – Matrix Multiplikation, wobei n die Größe der quadratischen Matrizen ist.

```
void normal_matrix_mul(const float* mat1, const float* mat2, float* mat_out, int
n) {
    for (int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            mat_out[i*n+j] = mat1[i*n] * mat2[j];
            for (int k = 1; k < n; k++) {
                mat_out[i*n+j] += mat1[i*n+k] * mat2[k*n+j];
            }
        }
    }
}
```

Aufgabe 6:

Die folgenden Bar-Plots zeigen die Ergebnisse unserer Messungen, wobei die Balken an sich den Median darstellen. Zusätzlich sind noch das Minimum und Maximum angegeben.

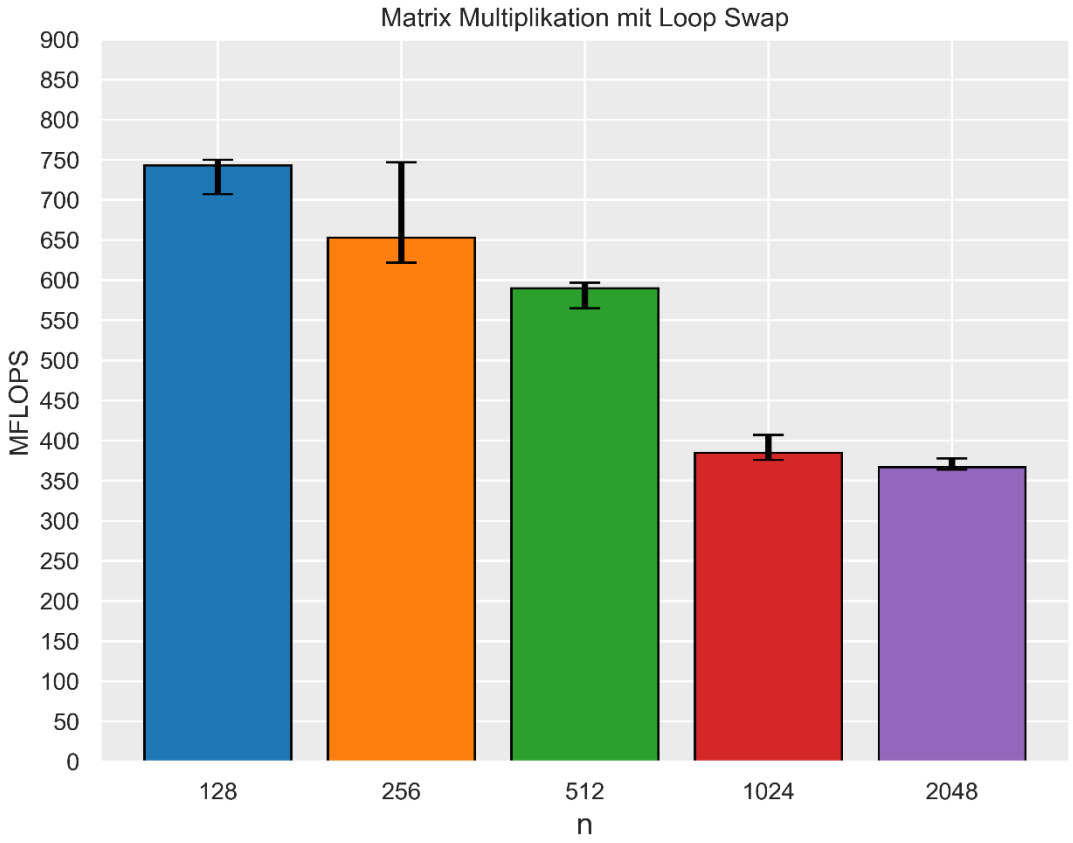


Bei der Matrixmultiplikation ohne Optimierungen kann man gut erkennen, dass umso größer die Matrizen werden, umso weniger FLOPS erreicht werden. Dies liegt daran, dass bei sehr großen Matrizen die Zeilen bzw. Spalten nicht mehr komplett in den Cache passen und somit für eine Berechnung mehrfach auf den Speicher zugegriffen werden muss, was Geschwindigkeitseinbuße bedeutet.

Aufgabe 7:

Im folgenden ist der Quellcode für die unterschiedlichen Methoden zur Optimierung der Matrixmultiplikation zu finden:

Schleifenvertauschung/Loop Swap:

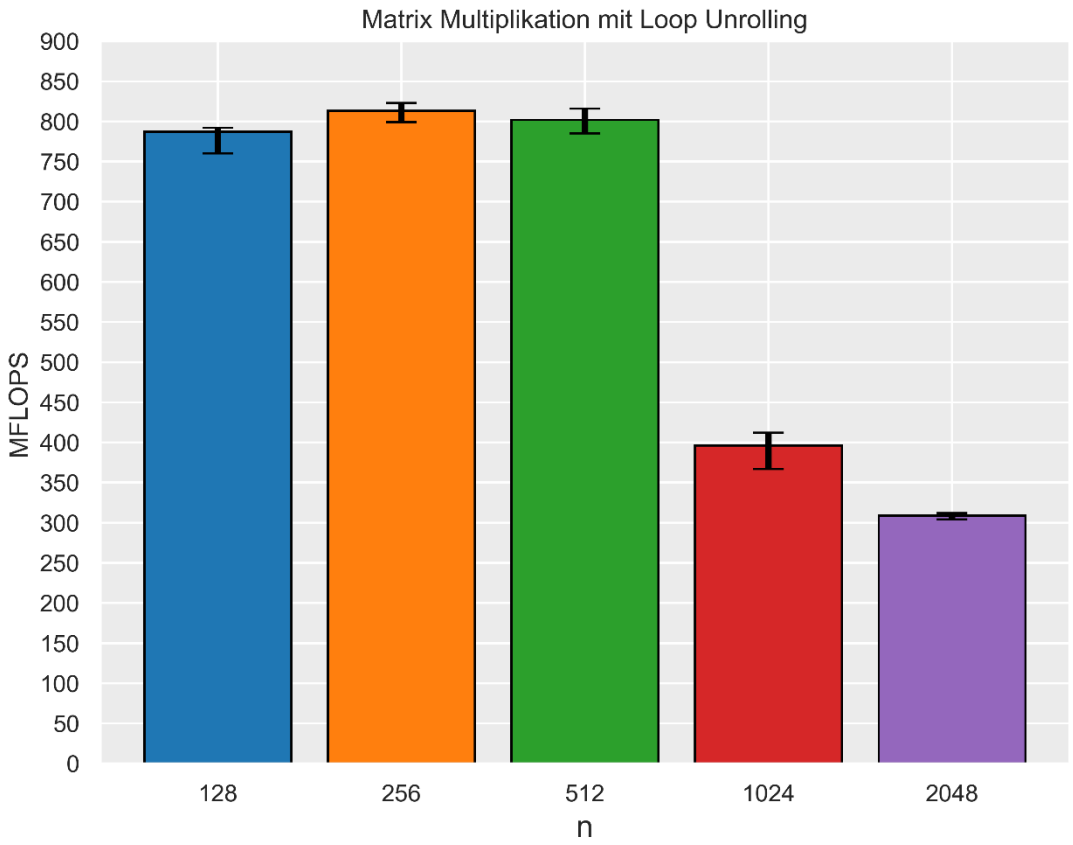



```
void loop_swap(const float* mat1, const float* mat2, float* mat_out, int n) {
    for (int j = 0; j < n; j++) {
        for(int i = 0; i < n; i++) {
            mat_out[i*n+j] = mat1[i*n] * mat2[j];
            for (int k = 1; k < n; k++) {
                mat_out[i*n+j] += mat1[i*n+k] * mat2[k*n+j];
            }
        }
    }
}
```

Bei der Matrixmultiplikation hat man ineinander geschachtelte Schleifen um jeweils über die Matrizen zu iterieren. Prinzipiell wird durch die verwendeten for-Schleifen über eine Input Matrix günstig iteriert und über die andere ungünstig. Dies ist auch mit Loop Swap so. Der Unterschied liegt dann dabei, wie man über die Output Matrix iteriert. In unserem Fall haben wir bei der nicht optimierten Variante die günstige Iteration und somit hätten wir erwartet, dass wir mit Loop Swap eine schlechtere Performance erreichen.
Jedoch zeigt das Ergebnis, dass es mit Loop Swap in manchen Fällen, sogar bessere Ergebnisse erzielt.

zu Aufgabe 7:

Loopunrolling:

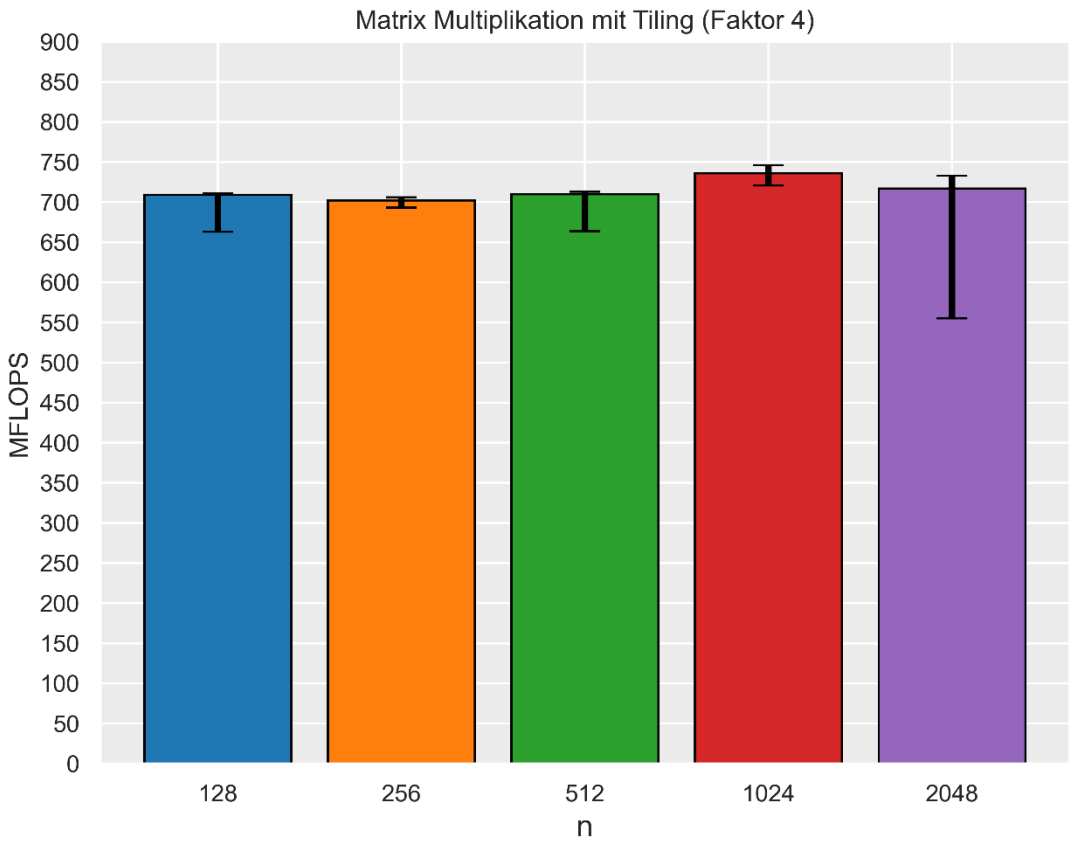



```
void loop_unrolling(const float* mat1, const float* mat2, float* mat_out, int n)
{
    for (int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            for (int k = 0; k < n; k+=4) {
                mat_out[i*n+j] += mat1[i*n+k] * mat2[k*n+j];
                mat_out[i*n+j] += mat1[i*n+(k+1)] * mat2[(k+1)*n+j];
                mat_out[i*n+j] += mat1[i*n+(k+2)] * mat2[(k+2)*n+j];
                mat_out[i*n+j] += mat1[i*n+(k+3)] * mat2[(k+3)*n+j];
            }
        }
    }
}
```

Beim Loop Unrolling greift man pro Iteration direkt auf mehrere Stellen in den Matrizen zu und kann somit pro Iteration mehr Rechnen. Dafür springt man dann eben direkt mehrere Stellen weiter in den Matrizen. Das erwartete Ergebnis hier ist eine Steigerung der Performance. Dies ist auch das tatsächliche Ergebnis, wie man im Diagramm sehen kann. Wird die Matrix zu groß, hat man nach wie vor das Problem, dass nicht alles in den Cache passt. Durch die zusätzlichen Speicherzugriffe verliert man dann Geschwindigkeit.

zu Aufgabe 7:

Blocking/Tiling:



Beim Blocking/Tiling zerlegen wir die Matrix in Blöcke und rechnen dann mit den kleineren Teilen der Matrizen. Das bringt vor allem Vorteile bei sehr großen Matrizen, da somit das Problem gelöst wird, dass nicht alles in den Cache passt und dadurch zusätzliche Speicherzugriffe nötig (= langsamer) sind. Dadurch war zu erwarten, dass bei den größeren Matrizen (n = 1024, 2048) mehr Leistung erreicht wird. Die Erwartungen wurden erfüllt, wie man im Diagramm sehen kann.

Blocking/Tiling:

```
void tiling_matrix_mul(const float* mat1, const float* mat2, float* mat_out, int n, int tiling_factor, int
tile_size) {
    float* result_tile = malloc(tile_size*tile_size*sizeof(float));
    float* mat1_tile = malloc(tile_size*tile_size*sizeof(float));
    float* mat2_tile = malloc(tile_size*tile_size*sizeof(float));
    for (int tx_out = 0; tx_out < tiling_factor; tx_out++) {
        for (int ty_out = 0; ty_out < tiling_factor; ty_out++) {
            for (int tile_index = 0; tile_index < tiling_factor; tile_index++){
                // copy tile from mat1
                for (int i = 0; i < tile_size; i++) {
                    for (int j = 0; j < tile_size; j++) {
                        mat1_tile[i*tile_size+j] = mat1[(((ty_out*tile_size)+i)*n)+((tile_index * tile_size) + j)];
                    }
                }

                // copy tile from mat2
                for (int i = 0; i < tile_size; i++) {
                    for (int j = 0; j < tile_size; j++) {
                        mat2_tile[i*tile_size+j] = mat2[(((tile_index * tile_size) + i) * n) + ((tx_out * tile_size)
+ j)];
                    }
                }
                // multiply tiles
                normal_matrix_mul(mat1_tile, mat2_tile, result_tile, tile_size);
                // copy result tile to mat_out
                for (int i = 0; i < tile_size; i++) {
                    for (int j = 0; j < tile_size; j++) {
                        mat_out[(((ty_out*tile_size)+i)*n)+((tx_out * tile_size) + j)] +=
result_tile[i*tile_size+j];
                    }
                }
            }
        }
    }
    free(result_tile);
    free(mat1_tile);
    free(mat2_tile);
}
```

Aufgabe 8:

Mit dem Flag - O probiert der Compiler Codegröße und Ausführungszeit zu reduzieren, ohne irgendwelche Optimierungen zu verwenden, welche viel Zeit bei der Kompilierung einnehmen würden.

- O1:
Bei der ersten Stufe werden die gleichen Optimierungen ausgeführt wie bei dem Flag – O. Also siehe Erklärung oben. - O und - O1 sind die gleichen Flags, sie bewirken das gleiche.

- O2:
Bei der zweiten Stufe werden zusätzlich zur ersten Stufe weitere Optimierungen durchgeführt. GCC führt hier so gut wie alle Optimierungen aus, welche keinenSpeed-Accuracy-Tradeoff beinhalten. In Vergleich zu Stufe 1 verbessert diese Option die Kompilierungszeit als auch die Performance des generierten Codes.

- O3:
Diese Stufe optimiert noch weiter, es werden alle aus Stufe 2 ausgeführt und noch weitere Optimierungsflags. (die genauen Flags können auf der *man gcc* nachgelesen werden)
Hier wird Loop Swap und Loop Unroll verwendet.

Blocking/Tiling haben wir bei den Flags nicht gefunden.

Wir haben die verschiedenen Flags - O Stufe 1 bis Stufe 3 getestet und haben auf jeder Stufe eine Steigerung der Leistung feststellen können.

Bei den prozessorarchitekturspezifischen Optimierungen kann man beispielsweise *-march=...* verwenden womit man die Architektur angeben kann. Ein anderer Befehl wäre *-xcode*, welcher Intel spezifisch ist und damit ggf. Optimierungen erreichen kann.