Uebung 2

Aufgabe 1

```
#ifdef VERSION_A
#define calcIndex(threadIndex, j) (((N) / (RESULT_COUNT)) * (threadIndex) + (j))
#else
#define calcIndex(threadIndex, j) ((threadIndex) + (j) * (RESULT_COUNT))
#endif

__global__ void sum(value_T* array, value_T* results){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if(idx < RESULT_COUNT){
        value_T sum = 0;
        for(int j = 0; j < (N / RESULT_COUNT); ++j){
            int index = calcIndex(idx, j);
            sum += array[index];
        }
        results[idx] = sum;
    }
}

int main() {
    int threads_per_block = 32 * pow(5, K);
    /* ... */
    sum<<<
        (RESULT_COUNT + threads_per_block - 1)/threads_per_block,
        threads_per_block
    >>>(device_array, device_results);
    /*...*/
}
```

Messdaten

Für die Messungen wurde eine GTX 970 verwendet.

Version A

Die Messwerte der 5 Testläufe `#1` bis `#5` mit dem gegebenen k und dem Durchschnitt.

k	avg	#1	#2	#3	#4	#5
0	38.6ms	39ms	38ms	40ms	37ms	39ms
1	53.6ms	54ms	54ms	54ms	52ms	54ms
2	49.4ms	48ms	48ms	51ms	49ms	51ms

k >= 3 stürzt ab. 32 * 5 ^ 3 = 4000. Zu viele Threads.

Version B

Die Messwerte der 5 Testläufe #1 bis #5 mit dem gegebenen k und dem Durchschnitt.

k	avg	#1	#2	#3	#4	#5
0	2ms	2ms	2ms	2ms	2ms	2ms
1	2ms	2ms	2ms	2ms	2ms	2ms
2	2ms	2ms	2ms	2ms	2ms	2ms

k >= 3 stürzt ab. 32 * 5 ^ 3 = 4000. Zu viele Threads.

Bewertung der Messdaten

Version A ist durchweg um Faktor ~20-25 langsamer als Version 2. Eine Änderung von k führt zwar zu anderen Messwerten in Version A, kann das grundlegende Problem jedoch nicht eliminieren. Eine Änderung von k führt zu keinen ersichtlichen Messwerten in Version B, das hat jedoch nichts zu bedeuten. Die Laufzeiten von Version B sind so gering, dass tatsächlich auftretende Änderungen durch Änderung von k einfach übersehen werden. Die größte Schwankung in Version A ist zwischen 37ms und 54ms. Das wäre eine Änderung von +46%. Diese Änderung wäre in Version B wahrscheinlich nicht sichtbar.

Eine Erklärung des Verhaltens ist eine bessere Ausnutzung der Speicheranbindung. Bei Variante A ist jeder Thread i alle nebeneinanderliegenden Speicherzellen ab 5000 * i zuständig. Das heißt, dass zu einem gegebenen Zeitpunkt t beispielsweise die Speicherzelle 5000 * i + t gelesen werden muss. Da Threads parallel ausgeführt werden, wird die Grafikkarte zum Zeitpunkt t alle Speicherzellen 5000 * i + t gleichzeitig laden müssen. Also beispielsweise t, 5000 + t, 10000 + t, 150000 + t, etc. Die zu ladenden Speicherzellen sind dabei immer exakt 5000 Zellen voneinander entfernt. Die ist eine schlechte Idee, da heutige Speicher extrem effizient sind, nebeneinanderliegende Speicherzellen zu laden.

Variante B nutzt die Speicheranbindung viel geschickter aus. Zum Zeitpunkt t werden die Speicherzellen i + t * 20000 ausgelesen. Beispielsweise t * 20000, 1 + t * 20000, 2 + t * 20000, etc. Wie man sieht, liegen die Speicherzellen zum Zeitpunkt t alle nebeneinander, was viel vorteilhafter in Hinsicht auf die Speicheranbindung ist.

Die Änderungen der Messdaten in Variante A, wenn man k erhöht, lassen sich vielleicht auch dadurch erklären, dass mit höherem k sich mehr Threads gegenseitig im Weg stehen. Je mehr Threads gleichzeitig diese schlechte Ladetechnik verwenden, desto länger müssen die einzelnen Threads auch warten. Das wäre auch eine mögliche Erklärung, wieso man keine Änderung der Zeiten in Variante B sieht. Die Threads stehen sich nicht wirklich gegenseitig im Weg.

Aufgabe 2

Annahme: - N ist immer gerade

Die Idee ist, dass `N/2` Threads erzeugt werden und sich jeder Thread um zwei Zeilen kuemmert. Thread `i` kuemmert sich um Zeile `i` und `(N - i - 1c)`. Das führt dazu, dass jeder Thread in Zeile `i` `i` Nullwerte und `N - i` Zufallswerte hat; und in Zeile `(N - i - 1) (N - i - 1)` Nullwerte und `N - (N - i - 1) = i + 1` Zufallswerte. Das sind zusammen `i + N - i - 1 = N - 1` Nullwerte und `N - i + i + 1 = N + 1` Zufallswerte. D.h. jeder Thread hat exakt gleich viele Zufallswerte und Nullwerte, da diese nicht mehr abhängig von der Threadnummer `i` sind. Durch geschickte Schachtelung werden Berechnungen in dem 0-Werte Bereich der Matrix verhindert.

```
void mat_vec_mul(float *res, float *mat, float *vec)
{
    // Note: this only works for N%2 == 0
#ifdef OMP_ENABLED
#pragma omp parallel for num_threads(6)
#endif
    for (uint64_t i = 0; i < N / 2; i++)
    {
        // top row
        int y_top = i; // this happens to indicate the "line-coordinate" for the 0-value lower part.
        // bottom row
        int y_bot = N - 1 - i; // move from bottom
```

```
        // Calc top row
        float sum_top = 0.0f;
        for (uint64_t x = y_top; x < N; x++)
            sum_top += mat[y_top * N + x] * vec[x];
        res[y_top] = sum_top;

        // Calc bottom row
        float sum_bot = 0.0f;
        for (uint64_t x = y_bot; x < N; x++)
            sum_bot += mat[y_bot * N + x] * vec[x];
        res[y_bot] = sum_bot;
    }
}
```

Ausnahme fuer ungerade N: Ist `N` ungerade, so haben wird der Thread `N / 2` sich nur um eine Zeile in der Mitte kümmern müssen. Das das aber maximal einmal passiert und `N` groß gewählt wird, bewerten wir es ähnlich, wie wenn man 5 Threads auf 4 CPU-Kernen aufteilen will. In der Implementierung `ex3_2.c` wurde die oben Beschriebene Loesung implementiert, in `ex3_2_pat.cpp` die komplette Implementation.

```
#ifdef OMP_ENABLED
#pragma omp parallel for
#endif
for (int thread = 0; thread <= N / 2; ++thread) {
    int index_row_top = thread;
    int index_row_bottom = N - thread - 1;
    value_t *current_row_matrix;
    if (thread == N / 2) {
        current_row_matrix = matrix[thread];
        value_t sum = 0;
        for (int x = N - 1; x >= N / 2; --x) {
            sum += current_row_matrix[x] * vector[x];
        }
        result[thread] = sum;
    } else {
        current_row_matrix = matrix[index_row_top];
        value_t sum = 0;
        for (int i = thread; i < N; ++i) {
            sum += current_row_matrix[i] * vector[i];
        }
        result[index_row_top] = sum;
        sum = 0;
        current_row_matrix = matrix[index_row_bottom];
        for (int i = N - 1 - thread; i < N; ++i) {
            sum += current_row_matrix[i] * vector[i];
        }
        result[index_row_bottom] = sum;
    }
}
```

Messdaten

X86 basierte CPU

Für die Messungen wurde ein Ryzen 7 3800X (8 Kerne/16 Threads) mit 16GB Arbeitsspeicher verwendet. N wurde auf `36000` gesetzt, was bei `double` Datentype ungefähr 10GB für die Matrix entspricht. Es wurden immer 5 Testläufe gemacht und gemessen wurde mit `double omp_get_wtime(void);`.

Typ	Avg	#1	#2	#3	#4	#5
parallel	0.164s	0.167s	0.155s	0.156s	0.187s	0.156s
sequentiell	1.627s	1.660s	1.613s	1.630s	1.614s	1.616s

Der durchschnittliche Speedup ist demnach `9.92`, was im Bereich des Erwarteten liegt.

ARM basierte CPU

Eine weitere Messereihe mit einem ARM-basierten Prozessor (M1 Pro, 6P-Cores) und `N = 36000` ergibt einen Speedup von `5.6`:

Ohne OMP:

```
Compute time 2447.359000ms
Compute time 2233.519000ms
Compute time 2231.694000ms
Compute time 2232.839000ms
Compute time 2231.556000ms
Avg compute time 2275.393400ms
```

Mit OMP:

```
Compute time 428.901000ms
Compute time 399.094000ms
Compute time 400.846000ms
Compute time 400.473000ms
Compute time 400.540000ms
Avg compute time 405.970800ms
```

Aufgabe 3

Idee

Die grundsätzliche Idee ist dieselbe, wie in Aufgabe 2: Immer eine Zeile oben und eine Zeile unten im selben Thread berechnen, sodass alle Threads die gleiche Arbeit haben.

Einziger Unterschied ist, dass wir im Vergleich zu Aufgabe 2 ein eindimensionales Array statt einem zweidimensionalen verwendet haben.

Cuda kernel:

```
__global__ void mult(value_T *matrix, value_T *vector, value_T *results){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    value_T sum = 0;
    if(idx < N / 2){
        for (int i = idx; i < N; ++i) {
            sum += matrix[N * idx + i] * vector[i];
        }
        results[idx] = sum;
        sum = 0;
        for (int i = N - 1 - idx; i < N; ++i) {
            sum += matrix[N * (N - idx - 1) + i] * vector[i];
        }
        results[N - idx - 1] = sum;
        return;
    }else if(idx == N / 2){
        for (int x = N - 1; x >= N / 2; --x) {
            sum += matrix[N * idx + x] * vector[x];
        }
        results[idx] = sum;
```

```
        return;
    }
}
```

Messdaten

Gemessen wurde mit einer GTX970, einem I5-6600K (4Kerne/4Threads) und 16GB Arbeitsspeicher. N wurde auf 20000 gelegt, was knapp über 3GB Speicherverbrauch bedeutet. Die GTX970 besitzt zwar 4GB Speicher, jedoch hat sie (wie alle GTX970) einen Designfehler, der dazu führt, dass nur die ersten 3.5GB performant angesprochen werden können. Deswegen sind wir unter den theoretisch möglichen 4GB geblieben, um die Messung nicht durch diesen Designfehler zu verfälschen.

Da die Grafikkarte in einem anderen PC, als der Test-PC in Aufgabe 2 eingebaut ist, haben wir auch die Tests der Aufgabe 2 auf diesem PC wiederholt. Außerdem musste, wegen dem kleinen Grafikspeicher, N sowieso kleiner gewählt werden, was auch zum erneuten Durchführen der Tests geführt hat.

In der nachfolgenden Tabelle stehen für alle 4 möglichen Messarten der Speedup im Vergleich zum sequentiellen Programm, die Durchschnittsdauer in Sekunden und die tatsächlichen Messwerte der 5 Testläufe.

Da wir CPU und GPU Programme miteinander vergleichen, haben wir das GPU Programm auf 2 Arten gemessen. Einmal wurde nur die notwendige Rechenzeit im Kernel, ein anderes Mal die Rechenzeit im Kernel plus Allokieren des Grafikspeichers und Kopieren zum/vom Grafikspeicher. Die erste Messart macht in der Realität Sinn, wenn z.B.: - die Daten sowieso schon im Grafikspeicher sind - die Daten dort bleiben werden - die Berechnung öfters aufgerufen werden muss.

Die zweite Messart macht Sinn, wenn dem nicht so ist. Beispielsweise unser Programm, welches die Berechnung einmal durchführt, das Ergebnis entgegennimmt und sich dann beendet. Den Extraaufwand die Grafikkarte anzusprechen, ist dann real existierender Aufwand, den man in der Überlegung, ob man eine Grafikkarte benutzt, einbeziehen muss. Beide Messarten haben demnach ihre Berechtigung und es kommt auf das restliche Programm an, welche der Messarten eher der Realität entsprechen.

Typ	Speedup	Avg	#1	#2	#3	#4	#5
parallel	3.994	0.166	0.166	0.166	0.166	0.166	0.166
sequentiell	1.000	0.663	0.663	0.663	0.663	0.663	0.663
cuda	8.610	0.077	0.077	0.077	0.077	0.077	0.077
cuda + mem	0.994	0.667	0.671	0.672	0.658	0.662	0.672

Deutung der Messdaten

Zuallererst ist erfreulich, dass der Speedup des parallelen OpenMP Programms sich nahezu perfekt an den zweiten Test-PC angepasst hat: Ein Vierkerner mit 4 Threads, der einen Speedup um 3.994 durch Multicoreprogrammierung erhält.

Außerdem interessant ist es, dass die Werte vom parallelen und vom sequentiellen Programm so nah aneinander sind. Sie haben sich oft ab der 4. Nachkommastelle unterschieden, aber so genau braucht man es hierfür nicht.

Betrachtet man die Messart `cuda` so sieht man einen Speedup von 8.61. Mehr als doppelt so viel, wie `parallel`. Das wäre schon beachtlich, wenn wir in Aufgabe 2 nicht einen Speedup von 9.92 mit einer aktuelleren CPU erreicht hätten. Schließt man dann noch die Speicherallozierung und das Kopieren mit ein, hat man einen Performancenachteil sogar gegenüber dem sequentiellen Programm. Fairerweise muss man aber auch dazu sagen, dass die CPU in Aufgabe 2 um einiges neuer ist als alle Hardware in dieser Messreihe.

Trotzdem würden wir die Performance als positiv bewerten, da: - das CUDA Programm noch optimiert werden kann. Die Technik, welche in Aufgabe 1 einen Performancebonus von Faktor 20-25 gebracht hat, kann noch verwendet werden. Außerdem gibt es noch viele andere Techniken den CUDA Code zu beschleunigen. Die Blockgröße könnte man noch tunen, und und und... - Verglichen mit der CPU mit ähnlichem Baujahr mehr als eine doppelte Geschwindigkeit in der Berechnung rausgeholt werden konnte. Eine neuere Grafikkarte könnte bestimmt einen besseren Speedup rausholen, aber dafür fehlt uns die Hardware.

Aufgabe 4

`idx = blockIdx.x + threadIdx.x * gridDim.x`

`(threadIdx.x * gridDim.x) mod 3 == 0`, da `B == gridDim.x` und `B mod 3 == 0`.

Damit gilt `idx mod 3 == blockIdx.x mod 3`.