1. Hausübung - OpenCL

Dokumentation von Leon Bohmann (2493657) und Jonathan Stollberg (247775)
Tag der Einreichung: 4. Dezember 2022

Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1 Einführung

In dieser Hausübung ging es darum die Berechnung eines Matrix-Vektor-Produkts gemäß

$$c_i = A_{ij} b_j$$

in der Programmiersprache *Java* zu implementieren. Dabei soll die Matrix $\mathbf{A}$ stets quadratisch sein, d.h. $\mathbf{A} \in \mathbb{R}^{m \times m}$ und $\mathbf{b} \in \mathbb{R}^m$. Die Implementierung wurde sequentiell und parallel umgesetzt, wobei darauf geachtet wurde, dass das Programm sowohl auf der CPU mittels Streams als auch auf der GPU mittels der Schnittstelle *OpenCL* parallelisiert ist.

Das Programm kann über die Kommandozeile mit zwei optionalen (und untrennbaren) Argumenten aufgerufen werden. Das erste Argument repräsentiert dabei die Problemgröße m, das zweite Argument legt die *local size* fest, die der *OpenCL*-Kernel anfragt. Wird das Programm ganz ohne Argumente aufgerufen, so wird die Berechnung für verschiedene vordefinierte m und *local sizes* ausgeführt:

```
1 $ java -jar HighPerformanceSimulation.jar
2 $ java -jar HighPerformanceSimulation.jar 1000 10
```

Im Folgenden sollen nun kurz wichtige Aspekte der Implementierung erläutert und anschließend die Performance der sequentiellen und parallelen Berechnungen für verschiedene m analysiert und beurteilt werden.

2 Sequentieller Algorithmus

Zu Beginn des Programmaufrufs wird ausgehend von m eine Zufallsmatrix $\mathbf{A}$ sowie ein Zufallsvektor $\mathbf{b}$ generiert. Diese werden als eindimensionale short-Arrays gespeichert, sodass auf das Element A_{ij} über Index $i \times m + j$ zugegriffen wird. Für $\mathbf{b}$ ist der Zugriff über Indizes trivial.

Nachdem das Ergebnis-Array $\mathbf{c}$ der Länge m initialisiert wurde, lässt sich die sequentielle Matrix-Vektor-Multiplikation nun mit wenigen Zeilen Code analog zur Einstein'schen Summenkonvention mithilfe von for-Schleifen implementieren:

```
1 short[] c = new short[m];
2 for (int i = 0; i < m; i++) {
3     for (int j = 0; j < m; j++) {
4         c[i] += A[i * m + j] * b[j];
5     }
6 }
```

Selbstverständlich muss der Code noch in eine Klassenstruktur bzw. eine Methode eingebunden werden. Für Weitere Details zur Implementierung wird an dieser Stelle an den beiliegenden Code in der Datei *MatrixVector.java* verwiesen.

3 Paralleler Algorithmus mittels OpenCL

Zur Implementierung mit *OpenCL* waren einige Schritte zur Initialisierung nötig. Die Verwendete *Java OpenCL* Library (*JOCL*) ist ein Low-Level-API Wrapper. Es waren also mehrere aufwendige OpenCL-Calls nötig, um die gewünschte Funktionalität zu erreichen. Die Implementierung des Kernels erfolgte in einer *C99*-Notation, welche für den *OpenCL* Compiler vorgegeben war.

Funktion des Kernels

Folgend ist der Source-Code des Kernels dargestellt:

```
__kernel void matrix_vec(
    __global const short *lhs,
    __global const short *rhs,
    __global short *result,
    const int d) {
        int gid = get_global_id(0);
        result[gid] = 0;
        for(int i = 0; i < d; ++i) {
            result[gid] += lhs[gid * size + i] * rhs[i];
        }
};
```

Mit der Instruktion `__kernel` wird dem *OPCL* Compiler mitgeteilt, dass es sich um einen Einstiegspunkt handelt. Mit `__global` markierte Argumente sind Strukturen aus dem globalen Speicher, welcher von allen Kernels angesprochen werden kann. Bei dem letzten Argument handelt es sich um die Gesamt-Größe der übergebenen (1D-)Arrays, da der Kernel keine andere Möglichkeit hat die übergebenen Matrix-Größen zu erfahren (die Pointer `*lhs`, `*rhs` und `*result` zeigen nur auf den Anfang der entsprechenden Arrays).

Jedes Work-Item (1 Work-Item = 1 Kernel-Ausführung) erhält vom Host (*JOCL*) eine eindeutige ID. Diese erhält man mit `get_global_id`. Das Argument `0` definiert dabei die x-Richtung, da globale IDs auch mehrdimensional aufgereiht sein können. Mithilfe dieser ID wird eine Matrix-Zeile mit der Right-Hand-Side (`rhs`, dem Vektor) multipliziert. Das Ergebnis wird an der entsprechenden Stelle im Ergebnis-Array gespeichert. Mit anderen Worten: Die Matrix-Vektor-Multiplikation wird Zeilenweise in Kernels übergeben.

Ablauf der Initialisierung

Der Ablauf der Initialisierung verläuft wie folgt:

1. Erhalte Anzahl an Plattformen
2. Erhalte spezifische Plattform-IDs
3. Verwende 1. gefundene Plattform
4. Erhalte Anzahl an Devices auf Plattform
5. Verwende 1. gefundenes Device
6. Context für das Device erstellen
7. CommandQueue für das Device erstellen

Context und CommandQueue

Mit dem so generierten Context und der CommandQueue können dann jegliche weiteren Befehle ausgeführt werden. Der Context liefert dabei nötige infrastrukturelle Methoden und die CommandQueue dient der Kommunikation mit dem zugrundeliegenden Device. Im Regelfall sollten diese Objekte nach ihrer Verwendung verworfen und so für die Garbage Collection (GC) freigegeben werden. In diesem Fall wird die Referenz auf diese Objekte in der Klasse `OpenCL` gespeichert, um bei mehrfachem Aufruf erneute Instanzierungen zu vermeiden.

Kernel-Kompilierung

Um Code an die Grafik-Karte oder andere Compute Units (CU) zu übergeben, bedarf es noch weiterer Arbeit. Zunächst muss ein sogenanntes Programm erzeugt werden (`clCreateProgramWithSource`). Dazu verwendet man einen in *C99* verfassten Sourcecode, welcher durch *OpenCL* unterhalb von *JOCL* kompiliert wird (`clBuildProgram`). Das ausführbare Programm muss dann noch in einen Kernel überführt werden (`clCreateKernel`). Der Kernel wird letztlich von *JOCL* auf die CU übertragen und kann dann ausgeführt werden. Wie die Referenzen aus dem vorigen Absatz wird auch der Kernel nicht verworfen.

Speicher

Da sich Grafikkarte und CPU keinen Speicher teilen, müssen sogenannte Buffer implementiert werden, die Speicherzugriffe zwischen den beiden dedizierten Speichern koordinieren. Für weitere Details zur Implementierung wird an die Datei *MatrixVector.java* bzw. die Methode `MatrixVector.initParallel(long lws, int availableUnits)` verwiesen.

Um die zugrundeliegende Struktur in den globalen Speicher der Grafikkarte zu kopieren, werden Buffer erstellt, die den Inhalt des Host-Speichers (für den CPU verfügbarer Speicher) kopieren.

Kernel-Argumente

Um dem Host mitzuteilen, welche Argumente an die Kernel übergeben werden sollen, müssen diese noch mit `clSetKernelArg(...)` definiert werden. Hier ist darauf zu achten, dass die korrekten Objekt-Typen angegeben werden, da es sonst zu Memory-Leaks oder Schreiben in geschütztem Speicher kommt.

Starten der Kernel

Hat man die Initialisierung abgeschlossen, folgt das Starten der Kernels. Dazu wird eine globale und eine lokale Work-Size definiert. *Global* bedeutet hierbei den vollständigen Umfang der Work-Items (in diesem Fall die Menge an Zeilen in der Matrix) und *lokal* die Anzahl an gleichzeitig ausgeführten Work-Items (Kernel) innerhalb einer Work-Group. Aufgrund der Architektur von Grafikkarten (zumindest von NVIDIA) ist es ratsam, die Local-Work-Size(LWS) entsprechend der sogenannten Warp-Size zu definieren. Während Work-Groups auf sogenannten Threads ausgeführt werden, werden die beinhalteten Work-Items dann nochmal auf einzelne CU verteilt. Die Anzahl der so verteilten Work-Items nennt man Warp-Size. Bei häuslichen Tests auf einer *GTX 1060* mit einer Warp-Size von 32 erhält man eine LWS von ebenjenen 32. Bei kleineren LWS als 32 bleiben Rechenkapazitäten der GPU unbenutzt.

Die Kernels werden zuletzt mit `clEnqueueNDRangeKernel` an die CommandQueue übergeben. Hierbei werden die globalen und lokalen Arbeitsgrößen übergeben und der Host übernimmt die Verteilung.

Das übergeben der Kernels an die CommandQueue garantiert noch nicht, dass diese auch ausgeführt werden. Da der Host durch die übergebenen Kernel-Argumente weiß, welches Argument die Ausgabe des Kernels ist, kann er mit der Ausführung der Kernels warten, bis dieser angefordert wird.

Kernel-Ergebnisse auslesen

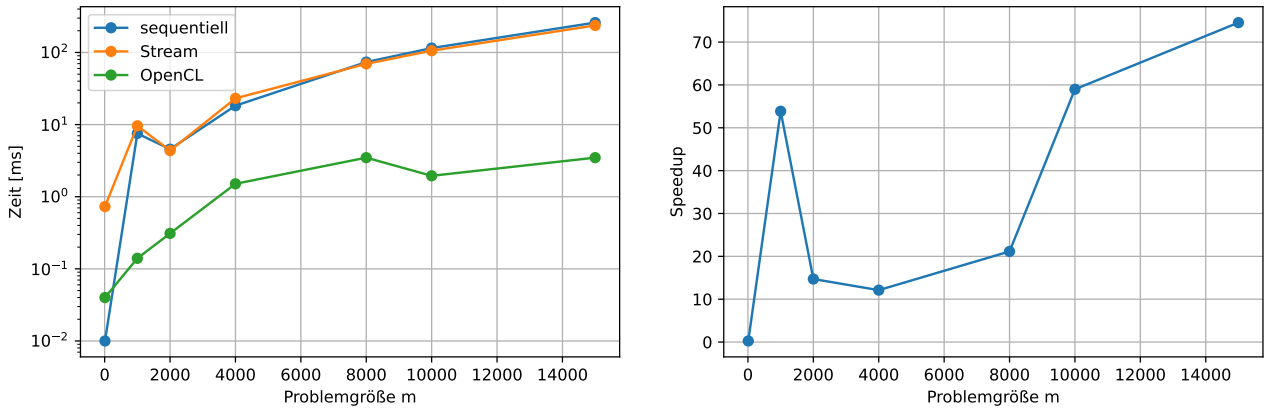
Ähnlich wie das kopieren vom Host-Speicher in den Ziel-Speicher müssen beim Lesen der Ergebnisse die Arrays vom Ziel-Speicher zurück in den Host-Speicher geladen werden. Dies geschieht mit `clEnqueueReadBuffer`.

Durch das Anfordern des Ergebnis-Speichers werden die zuvor eingereihten Kernels ausgeführt. Dieses Verhalten ist zur Messung nicht vorteilhaft, da auch das Auslesen des Speichers eine gewisse Zeit in Anspruch nehmen kann. Die Ausführung der Kernels wird daher nach `clEnqueueNDRangeKernel` mit Aufruf von `clFinish(OpenCL.commandQueue)` forciert (siehe `MatrixVector.parallel(..)`).

4 Performance-Analyse

Die Performance-Tests wurden auf dem Lichtenberg-Hochleistungsrechner der Technischen Universität Darmstadt durchgeführt. Zu diesem Zwecke wurde ein SLURM-Skript geschrieben, welches einen Rechenknoten mit einer GPU des Typs *NVIDIA A100-PCIE-40GB* sowie 30 GB Speicher pro CPU anfordert. Damit wurden die Tests für $m \in \{10, 1000, 2000, 4000, 8000, 10000, 15000\}$ gerechnet. Die `local_work_size` wurde dabei zunächst von *OpenCL* selbst festgelegt, anschließend aber auch noch variiert. Um die Rechenzeit möglichst unverfälscht zu zu untersuchen, wurde der Warmup der *OpenCL*-Parallelisierung in allen Messungen ignoriert. Außerdem wurden folgende Hardwarespezifikationen von *OpenCL* ausgelesen:

- `CL_DEVICE_MAX_WORK_GROUP_SIZE` = 262144
- `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS` = 3
- `CL_DEVICE_MAX_WORK_ITEM_SIZES` = 1024, 1024, 64
- `CL_DEVICE_MAX_COMPUTE_UNITS` = 108



(a) Rechenzeit.

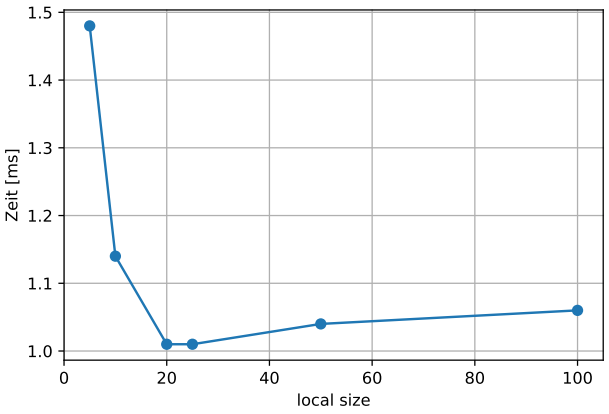
(b) Speedup durch die GPU-Parallelisierung.

Abbildung 4.1: Matrix-Vektor-Multiplikation mit von *OpenCL* automatisch festgelegter `local_work_size`.

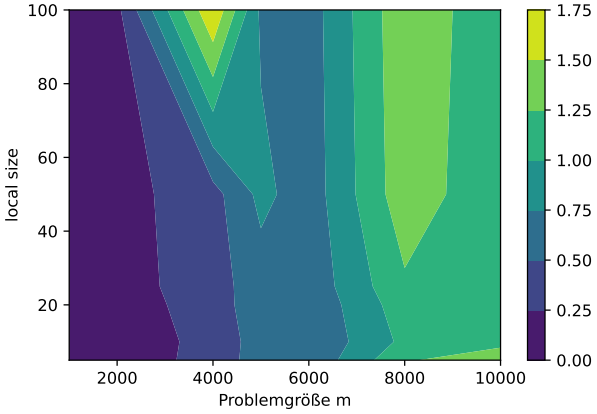
In Abb. 4.1 sind die Rechenzeit und der Speedup für die unterschiedlichen m und eine von *OpenCL* festgelegte `local_work_size` dargestellt. Es ist zu erkennen, dass die parallele Matrix-Vektor-Multiplikation insbesondere für sehr große m mit wesentlich geringerem Zeitaufwand durchgeführt wird, als im sequentiellen Fall. Somit wird im Falle $m = 15000$ bereits ein Speedup $S_p \approx 75$ erreicht. Dies ist darauf zurückzuführen, dass jede Zeile der Matrix $\mathbf{A}$ als eigenes *Work-Item* behandelt wird und somit mehrere Zeilen gleichzeitig abgewickelt

werden können. Lediglich für sehr kleine m, wie in diesem Falle $m = 10$ wird das sequentielle Programm schneller ausgeführt, da der Overhead die ansonsten vorliegende Zeiteinsparung überwiegt. Auffällig ist, dass es einen Ausreißer für $m = 1000$ gibt und die durch Streams parallelisierte Berechnung nur minimal schneller durchgeführt werden konnte als im sequentiellen Fall. Dies ist möglicherweise auf die allgemeine CPU-Auslastung zurückzuführen und bedarf nochmals einigen Untersuchungen. Da die CPU-Parallelisierung jedoch nicht Gegenstand der Aufgabenstellung ist, haben wir in diesem Fall zunächst darauf verzichtet.

Nun wurde die Anzahl der Work-Groups durch Variation der `local_work_size` $\in \{5, 10, 20, 25, 50, 100\}$ verändert. Belässt man die Problemgröße konstant, z.B. $m = 10000$, zeigt sich in Abb. 4.2a, dass nicht automatisch besonders viele oder besonders wenige Work-Groups die besten Ergebnisse liefern. So ist in unserem Beispiel `local_work_size = 20` eine besonders gute Wahl. Hier wurde im Vergleich zur zuvor durchgeführten Berechnung mit automatisch gewählter *local size* sogar nochmals etwas Zeit eingespart. Zu einem ähnlichen Ergebnis kommt man bei Betrachtung von Abb. 4.2b. Es ist bspw. erkennbar, dass sich die Rechenzeit für $m = 4000$ mit steigender `local_work_size` vergößert. Selbiges gilt auch für $m = 8000$. Um eine noch fundiertere Aussage treffen zu können, sollte man die *local size* sowie die Problemgröße allerdings nochmals etwas kleinschrittiger variieren.



(a) Ergebnisse für $m = 10000$.



(b) Rechenzeit in ms.

Abbildung 4.2: Variation der `local_work_size` und somit auch der Anzahl der Work-Groups.

5 Fazit

In dieser Übung wurde eine Matrix-Vektor-Multiplikation in Form eines klassischen, sequentiellen Algorithmus als auch eines durch *OpenCL* parallelisierten Algorithmus implementiert und die Performance der beiden Ansätze untersucht und verglichen. Es konnte festgestellt werden, dass durch die parallele Berechnung auf der GPU eine erhebliche Laufzeiteinsparung, insbesondere für sehr große Matrizen erreicht werden konnte. Desweiteren wurde gezeigt, dass eine geeignete Anzahl von *OpenCL*-Work-Groups nicht einfach naiv gewählt werden kann, sondern einer Voruntersuchung bedarf. Insgesamt kann jedoch mithilfe von *OpenCL* paralleler Code geschrieben werden, welcher im Vergleich zu seinem sequentiellen Gegenstück bei ausreichend zur Verfügung stehender Rechenleistung weitaus performanter ist. Der Mehraufwand bzgl. der Implementierung lohnt sich also deutlich.