Hochleistungssimulationen

Übung 2

Niklas Beck (2582775), Paul Hollmann (2465070)

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2023/24
Abgabe 22. Dezember 2023

Matrix-Vektor-Multiplikation

Das Matrix-Vektor-Produkt ist eine wohl bekannte mathematische Operation, welche auch für große Matrizen gebraucht wird. Um die Rechenzeit möglichst gering zu halten, sind parallele Berechnungen sinnvoll zu wählen. Insbesondere wenn gleiche Operationen auf unterschiedlichen Daten ausgeführt werden sollen.

Aufgabe 1: Sequentielle Berechnung des Matrix-Vektor-Produkts

Wir entwickeln zunächst ein sequentielles Programm. Nach Eingabe eines Wertes m, wird eine quadratische Matrix und ein Vektors mit Zufallswerten gefüllt. Ziel der Aufgabe ist es die Berechnungszeit des Matrix-Vektor-Produkts zu untersuchen. Ohne Vorgabe eines Datentyps haben wir uns für den Typ float entschieden.

Für das Speichern der Eingabe- sowie Ausgabevektoren bzw. der Eingabe Matrix nutzen wir native Java Arrays. Wichtig für das Speichern der Matrix nutzen wir ein Array mit Stride von m zwischen einzelnen Zeilen. Das (i, j)-te Element findet sich demnach an Postion $i \cdot m + j$.

Die eigentliche Berechnung lässt sich mit dann mit zwei for-Schleifen realisieren (siehe Code).

Aufgabe 2: Parallele Berechnung des Matrix-Vektor-Produkts

Noch bevor Bearbeitung der nächsten Teilaufgabe haben wir uns die Dokumentationen von OpenCL und JOCL herausgesucht und anhand der Beispiele aus dem moodle-Kurs und von JOCL ein Verständnis zur Einbindung von OpenCL-Code erlangt. Dann haben wir, noch auf den eigenen Rechnern, einen Kernel in einer Extradatei ausgelagert, da es eine schönere Formatierung ermöglicht. Der ganze Code im Kernel ist ein C/C++ Dialekt, was hier noch wenige Auswirkungen hat, aber nicht vergessen werden sollte. Als Vorlage zum Kernel dient das Programm aus Aufgabe 1. Wichtige Ergänzungen im Funktionskopf sind hierbei zuerst einmal die Spezifikation als kernel, die Änderung von Arrays zu Pointern als auch die Angabe des Adressraums der Parameter. Sinnvoll in diesem ersten Programm ist der global-Adressbereich, da hier die Variablen von host und device verwendet werden können. Die Datentypen sind in diesem Fall noch die gleichen, es gibt aber zum Beispiel nicht einfach einen double Typ. Dieser scheint standardmäßig nicht auf allen Geräten verfügbar zu sein. Zum Aufsplitten der Berechnung von einzelnen Spalten wird die ID der jeweiligen Threads per get_global_id erhalten. Diese Zahl wird als Zeilennummer der Matrix und des Ergebnisses verwendet.

Nach diesem Schritt in OpenCL-Code wird die Einbindung in den Java-Code benötigt. Es muss eine platform, properties und damit ein device erstellt werden. Das JOCL Device-Object haben wir für ein paar Testdaten mal untersucht und besonders die maximale Anzahl an work_groups ist für spätere Ziele noch relevant. Ein Context wird für das Programm noch benötigt bevor der Kernel per clCreateProgramWithSource in den JOCL Code eingefügt werden kann. Der C++ Code muss gebuilded werden und kann im Anschluss, durch clCreateKernel, als Kernel erstellt werden. Eine Command Queue wird erschaffen, um den Kernel später für eine asynchrone Ausführung auf das Device zu schicken. Etwas mehr Aufmerksamkeit mussten wir der memory object Erstellung schenken. Buffer wurden für alle Parameter und die Rückgabe erstellt und die Größe der Dateitypen sollte dazu beachtet werden. Der Kernel wird mit den Buffer-Objekten und möglichen Konstanten verbunden, die Systemzeit wird einmal gemessen und dann geht es endlich zur Ausführung des Kernels: $clEnqueueNDRangeKernel(commandQueue, kernel, 1, null, global\_work\_size, local\_work\_size, 0, null, null)$; Im Ausführungsbefehl können, wichtig für spätere Ziele, die work_groups angepasst werden. Zuerst verwenden wir hier

local_work_size=NULL, da sich OpenCL dann selbst eine geeignete Größe bestimmt. Nach der Ausführung wird der Buffer mit dem Ergebnis gelesen und nochmal die Zeit gemessen. Zu guter Letzt, immer wichtig bei der Arbeit direkt mit Speicher, müssen die Objekte freigegeben werden. Mit den richtigen Timern können wir schon gleich die Zeiten der Berechnung sequentiell und parallel vergleichen.

Vergleicht man die Ergebnisse zwischen Berechnungen auf den verschiedenen Geräten so stellt man minimale Unterschiede fest. Hier kann es sich verschiedene Gleitkomma Multiplizier- und Addiergenauigkeiten oder -optmierungen auf den verschiedenen Geräten handeln.

Aufgabe 3: Ausführung auf dem Lichtenberg-Hochleistungsrechner

Zur Ausführung auf dem Cluster wird ein Job Script gebraucht. Wir verwenden hier Maven, zum Compilieren und Bauen einer JAR Datei. Die Größe der Matrix wird als Kommandozeilenparameter übergeben. Das batch Dokument vecmat.sh kann per SLURM in die Processqueue übergeben werden. Dieses nimmt einige Einstellungen vor, lädt die notwendigen Module, baut das Programm unter Nutzung der Maven Bibliothek und führt es dann in unterschiedlichen Parameterkonfigurationen aus.

Aufgabe 4: Messungen und Analyse

In dieser Teilaufgabe ist es nun verlangt für einige Problemgrößen die Ausführungszeit auf dem Cluster zu messen.

Problemgröße m	**CPU Zeit** [ms]	**Kernelzeit** [ms]	**Komplette Devicezeit** [ms]
1	0.003343	0.037888	0.765934
10	0.007416	0.033792	0.783118
1000	18.08856	0.164864	6.884962
2000	16.543831	0.376832	18.906941
4000	37.728688	0.945152	67.266515
8000	123.748559	1.862656	248.687903
15000	409.879089	2.832384	825.980899
46340	3815.686135	35.605504	7729.755513

Tabelle 1: Vergleich CPU vs. GPU Ausführungen.

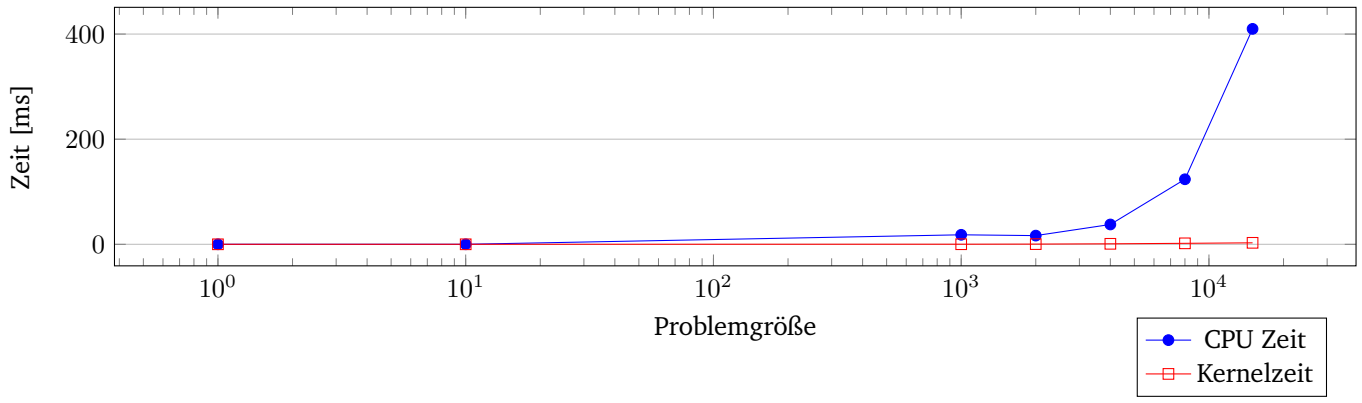

Abbildung 1: Daten aus Tabelle 1 als Plot.

Tabelle 1 stellt für verschiedene Problemgrößen die Rechenzeiten unterteilt in CPU Zeit, Kernelzeit und komplette Devicezeit[1] dar.

[1] Device Speicherallokation + Kopieren der Arrays hin und zurück sowie Aufruf und Ausführen des Kernels.

Erkennbar ist, dass die Parallelisierung bis Problemgröße $m = 10$, keine Beschleunigung der Multiplikation darstellt. Die Erstellung der Threads ist bei $m = 10$ noch zu aufwendig, dass es ein Vielfaches der Zeit des sequentiellen Codes braucht. Bei höheren Problemgrößen gewinnt der Kernel alleine gegen die sequentielle Funktion, allerdings gibt es Overhead bei der Erstellung der Objekte und dem Kopieren der großen Arrays in den device-memory. Für das Problem alleine, scheint sich die Parallelisierung noch nicht zu lohnen. Kann aber mit den Daten auf dem Device weitergearbeitet fallen die Kopiervorgänge weitestgehend weg, und dann bietet die einen echten zeitlichen Vorteil.

Aufgabe 5: Variation der Anzahl der work-groups

Als letztes Ziel wollen wir üben eine gute Anzahl an local-work-groups vorzugeben. Für die CL_DEVICE_MAX_WORK_GROUP_SIZE erhalten wir auf der NVIDIA V100 1024. Wir behalten eine konstante Problemgröße von $m = 10000$. Da mit eigener gewählter local_work_size (nennen wir sie l) die global_work_size (nennen wir sie g) durch die local_work_size teilbar sein sollte, muss $g = m \mod l == 0 \ ? \ m : (\lfloor m/l \rfloor + 1) \cdot l$ gelten. Überschüssige Elemente im Kernel deren globale id $\geq m$ ist werden in diesem ignoriert.

Wir variieren nun local_work_size so ergibt sich Abbildung 2.

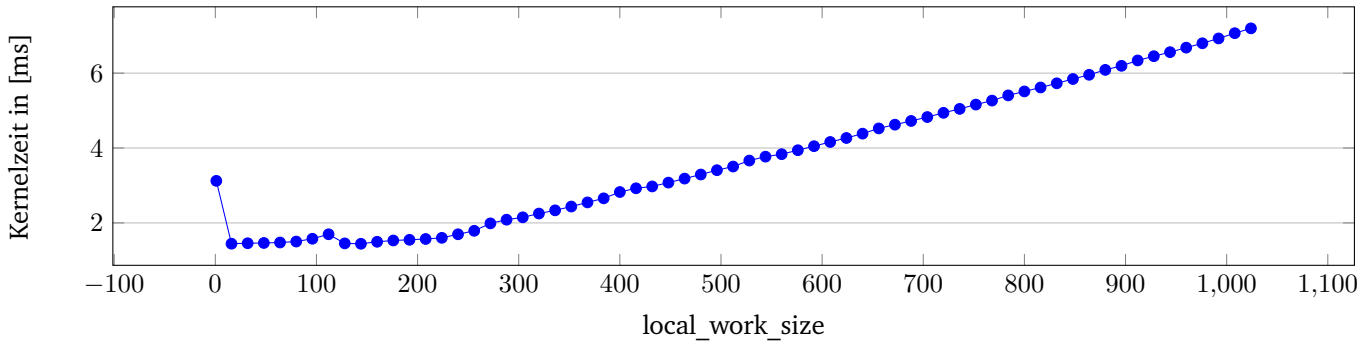

Abbildung 2: Kernelzeit in Abhängigkeit der local_work_size für ein konstantes $m = 10000$.

Es ist zu erkennen das für eine möglichst geringe Kernellaufzeit hier eine local_work_size zwischen 4 und 160 gewählt werden sollte. Diese ergibt sich aus der Hardware und Architektur dieser. Faktoren die dies in Kombination mit obigen beeinflussen können sind z.B. Speicherzugriffe. Je nach dem können zu viele oder zu wenige Threads einen Overhead erzeugen in dem die Hardware nicht mit ihrer vollen Kapazität genutzt wird.

A. Appendix

m	local_	**gws**	**kt**
10000	1	10000	3.1232
10000	16	10000	1.444864
10000	32	10016	1.457152
10000	48	10032	1.46432
10000	64	10048	1.476608
10000	80	10000	1.50224
10000	96	10080	1.57696
10000	112	10080	1.696768
10000	128	10112	1.453056
10000	144	10080	1.442816
10000	160	10080	1.496064
10000	176	10032	1.53088
10000	192	10176	1.549312
10000	208	10192	1.570784
10000	224	10080	1.601536
10000	240	10080	1.696768
10000	256	10240	1.788928
10000	512	10240	3.505152
10000	800	10400	5.511168
10000	816	10608	5.61664
10000	832	10816	5.728224
10000	848	10176	5.842944
10000	864	10368	5.955584
10000	880	10560	6.086656
10000	896	10752	6.1952
10000	912	10032	6.33856
10000	1024	10240	7.195648

Tabelle 2: Daten-Tabelle zu 2 (Auswahl).