Projektarbeit GPU
Matrix-Vektor-Produkt

Daniel Emil Giring

vorgelegt bei

Dr. Ralf Seidler

Fakultät für Mathematik und Informatik

Friedrich-Schiller-Universität Jena

Contents

1 Motivation

Matrix-Vektor-Operationen sind Elementar für eine Vielzahl von Berechnungen. Daher ist es von großer Bedeutung diese zu optimieren um Rechenzeit und andere Ressourcen zu sparen. Bei Matrix-Vektor Operation werden viele, bis auf den Indize, gleiche Operationen durchgeführt. Daher eignen sich Grafikarten gut für diese, da GPU sehr effizient bei hochparallelen Anwendungen mit gleichen Operationen sind. Im folgenden werden vier Algorithmen zu Matrix-Vektor vorgestellt, deren Implementierung in Cuda besprochen und deren Performance diskutiert sowie ein Vergleich zu einer CPU Implementierung gezogen wird.

2 Algorithmen zur Matrix-Vektor-Operation

Aus der linearen Algebra kennen wir das Matrix-Vektor-Produkts wie folgt: Sei $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$,. Dann errechnet sich das Matrix-Vektor-Produkt wie folgt: $b = Ax, b_j = \sum_{i=1}^{n} a_{i,j} \cdot x_i$. In den folgenden Implementierungen des Matrix-Vektor-Produkts auf Grafikkarten bekommt jeder thread ein Produkt $a_{i,j} \cdot x_i$ zur Berechnung. Diese Produkte werden dann mittels einer Summer über i reduziert, sodass das Endergebnis in einen Vektor b gespeichert werden kann.

2.1 Sharedmemory mit wiederholten Aufruf des Kernels, Kernel 1

Für diese Implementierung benötigen wir neben einer $n \times n$ Matrix A und eine n-dimensionalen Vektor x einen Speicher für das Ergebnis und die Zwischenergebnisse buff, eine boolvariable doComputation sowie eine Größe toreduce.
Bevor ein Kernel gestartet werden kann muss eine Blockgröße und ein grid definiert werden. Da wir mit einer zweidimensionalen Matrix rechnen eignen sich zweidimensionale Threadblöcke. In meiner Implementierung kann der Benutzer die Größe der Threadblöcke selbst einstellen, wobei sx die Anzahl der Spalten und sy die Anzahl der Zeilen eines threadblockes darstellt. Aus Performancegründen ist es wichtig für sx und sy zweier Potenzen einzugeben. Außerdem muss beachtet werden, dass $sx \cdot sy \leq 1024$, da ein Threadblock maximal 1024 threads enthalten kann. Die Anzahl an Thread-Blöcken, welche gestartet werden, werden in der dim3 Variable grid bzhw, itgrid gespeichert. Dies wird hier auch wieder in zwei Dimensionen dargestellt, da man somit die zweidimensionale Matrix gut in zweidimensionale Blöcke aufteilen kann.
Am Anfang wir die boolvariable doComputation auf true gesetzt, toreduce auf n, da jeweils n Produkte zu einer Summe zu reduzieren sind. Dem Kernel wird die Matrix A, x, doComputation, die Anzahl der Spalten der Matrix (size), sowie toreduce übergeben. Nun kann der Kernel das erste mal gestartet werden.
Es werden bei der ersten Ausführung des Kernels size/sx*size/sy viele Threadblöck gestartet. Mann kann sich die Gesamtheit der Threadblöcke wieder wie eine Matrix vorstellen. In jedem Threadblock wird zunächst von den verschiedenen threads das Produkt $a_{i,j} * x_i$ ausgerechnet. Dabei wird die Matrix A so aufgeteilt, dass die verschiedenen Threadblöcke wie ein Gitter über der Matrix liegen. Somit wird jeder Matrixeintrag $a_{i,j}$ in genau einen Threadblock von genauen einen Thread mit dem dazugehörigen Vektoreintrag x_i multipliziert

wird. Sei A eine $n * n$ Matrix, so starten wir $n/sx * n/sy$ Threadblöcke der Größe $sx * sy$. Jeder Threadblock rechnet somit sy Zeilen der Länge sx aus A. Das Ergebnis dieser Multiplikation schreiben die Threads dann in einen shared Memory innerhalb des Threadblocks. Der shared Memory wird so indiziert, dass dieser wieder als Matrix der Größe sx*sy gelesen werden kann. Dabei werden Produkte in eine Zeile geschrieben, beide den die Faktoren $a_{i,j}$ auch in der Ursprungsmatrix A innerhalb einer Zeile standen. Über diese Zeile kann jetzt innerhalb des threadblocks reduziert werden. Dafür wird sich die Reduzierung mittels Pointerarithmetik zu Nutzen gemacht. Jede Zeile wird in zwei Hälften geteilt, jeder Zeileneintrag ist einem Thread zugeordnet. Sei z gleich die länge der Zeile, jeder Thread mit threadId.x i, $i < z/2$ addiert auf den Wert in Spalte i den Wert aus Spalte $i + z/2$ auf. Ist dies getan wird die vorderen Hälfte der Spalte wieder in zwei Hälften geteilt und das selbe Verfahren auf die vordere Zeile angewendet. Dies wird solange durchgeführt, bis am Ende eine Zeile der länge 1 übrig bleibt. Diese liefert dann den Reduzierten Wert der Zeile.
Damit liefert jeder threadblock als Zwischenergebnis ein Spaltenvektor der Größe Länge sy. Diese Spaltenvektoren werden jetzt in einem Zwischenspeicher buff geschrieben. Dabei werden die Ergebnisse der untereinanderliegenden Threadblöcke in gleicher Reihenfolge in buff untereinander, die Ergebnisse der nebeneinanderliegenden Threadblöcke in buff nebeneinander gespeichert. Das Zwischenergebnis ist eine Matrix mit $size/sx$ vielen Spalten und $size$ vielen Zeilen.
Diese Matrix können wir nun wieder an den Kernel übergeben, dafür müssen wir zunächst noch ein paar Vorbereitungen treffen. Da wir von allen Einträgen $a_{i,j}$ das Produkt mit der entsprechenden Vektorkomponente ausgerechnet haben, wird die Variable doComputation auf false gesetzt. Da im Zwischenergebnis nur noch $size/sx$ viele Zeileneinträge zu reduzieren sind, wird toreduce auf size/sx gesetzt. Da das Zwischenergebnis nur toreduce=$size/sx$ viele Spalten besitzt werden nun weniger Threadblöcke in Zeilen benötigt. Somit wird die x. dimension des grids auf toreduce/sx gesetzt.(toreduce viele Einträge sind pro Zeile zu reduzieren, sx viele Zeileneinträge pro threadblock). buff, welcher das Zwischenergebnis enthällt, toreduce, doComputation und size wird an den kernel übergeben. Da doComputation auf false gesetzt ist werden im Kernel die zu den threadblock gehörigen Matrixeinträge des Zwischenspeichers buff direkt in den shared Memory geschrieben und es muss keine Berechnung dafür durchgeführt werden. Nun wird wie im ersten Schritt über die Zeilen des Shared Memory reduziert. Das Zwischenergebnis eines Threadsblocks ist wieder ein Spaltenvektor mit sy vielen Spalten. Die Spaltenvektoren werden wieder in den Zwischenspeicher buff geschrieben, wobei wir die Spaltenvektoren der Threadblöcke untereinander untereinander gespeichert werden, die Spaltenvektoren der Threadblöcke nebeneinander werden wieder nebeneinander gespeichert. Das Zwischenergebnis hiervon stellt ein Matrix mit sy vielen Zeilen und $toreduce/sx$ vielen Spalten dar. Am Ende wird toreduce dividiert durch sx da im neuen Zwischenergebnis nur noch $toreduce/sx$ viele Zeileneinträge reduziert werden müssen. Dieses Verfahren wenden wird solange an bis $toreduce = 1$. Ist dies erreicht, so haben wir alle Zeileneinträge auf eine Spalte reduziert, sodass wir das die vorderste Spalte des Zwischenergebnis in das Endergebnis speichert können.

2.2 Sharedmemory mit Atomics, Kernel 2

In der zweiten Methode wird der Kernel nur einmal aufgerufen. Ähnlich wie in der Methode, in der wir nur mit shared Memory gearbeitet haben, wird hier zunächst die Matrix auf threadblöcke aufgeteilt, die entsprechenden Multiplikation werden in den Threadblöcken ausgeführt und jede threadblock hat als Zwischenergebnis einen Spaltenvektor der Größe sy. Jeodch werden die Einträge der Spaltenvektoren, der verschiedenen threadsblocks, welche aus der selben Zeile der Ursprungsmatrix hervorgehen, auf einen Wert in einem Speicher buff mittels atomicAdd Operationen aufaddiert. Somit liefert nach diesem Kernel buff einen Spaltenvektor der Länge sy, welche das Matrix-Vektor-Produkt enthält.

2.3 Nur Atomic Operationen, Kernel 3

Ähnlich wie in den anderen Methoden bekommt hier wieder jedem Eintrag aus der Matrix A genau ein thread zugeordnet, der das Produkt mit der entsprechenden Vektor Komponente ermittelt. Der speichert das Produkt in eine lokale Variable addsc ab. Es wurde vorher ein Speicher buff angelegt, welcher so viele belegbare Speicheradresse besitzt, wie die Matrix A Zeilen. Threads, welche das Produkt mittel des Matrixeintrag aus der gleichen Spalten errechnet haben addieren ihr Ergebnis jetzt auf den entsprechenden Eintrag in der variablen buff auf, sodass buff am Ende ein Spaltenvektor, welcher dem Matrix-Vektor Produkt entspricht, liefert.

2.4 Intra grid Groups, Kernel 4

2.5 Shuffle

Eine weitere Methode zur Lösung des Problems ist die Verwendung von shuffle Operationen. Diese Bedarf aber einiges mehr an Implementierungsaufwand und wird daher in dieser Stelle nicht näher erläutert. Jedoch ist diese Methode deutliche performanter auf Grafikkarten als die bisher gezeigt sodass ich sie nicht unerwähnt lassen möchte.

3 Performance

3.1 Verwendete Grafikkarten

Für die Performancemessungen der Algorithmen wurden die Grafikkarten des Lehrstuhls Nvidia GTX780, Nvidia RTX2070 und der das Ara-Clusters Nvidia P100 verwendet. Alle Berechnung wurden mit Daten des Datentyp floats, also in Single Precision ausgeführt.
Die Angaben zur theoretischen Performance in Table 1 der GTX780, P100 sind Herrstellerangaben. Die theoretische Performance der RTX2070super errechnet sich durch:

$$\text{SP Performance} = \text{fma}\cdot\#\{\text{Cuda Cores}\}\text{Takt} = 2\cdot 2560 * 1,61 Ghz = 8243 GFlops$$

Model	cuda Cores	H Takt GHz	theo. SP. Performace
GTX780	2304	0,87 (9 Boost)	3976 GFlops
P100	3584	1,33 (1,48 Boost)	9400 GFlops
RTX2070super	2560	1,61 (1,77 Boost)	8243 GFlops

Table 1: GPU Daten

3.2 Vergleich der Algorithmen

Auf alle Grafikkarten ist Kernel 3, welcher nur Atomic Operationen zur Reduktion benutzt, der Langsamste. Bei Verwendung der Atomic Operation müssen verschiedene threads auf die selben Speicherzellen schreiben, welche keinen geteilten Speicher enthalten. Dies hat zur Folge, dass die entsprechenden Speicherzellen immer wieder neu von threads geladen und beschrieben werden müssen während des die Anderen threads warten müssen, bis sie auf den Speicher zugreifen können. Dieses Problem tritt bei Kernel 3 bei jeder Addition auf, sodass für die Addition weder die threads innerhalb eines threads Blocks, noch die threads über die Threadblöcke verteilt, welche auf die selbe Matrix Zeile zugreifen, parallel addieren können. Das Resultat ist eine recht langsame Performance von maximal 13 GFlops auf der RTX2070super, 4GFlops auf der P100 und 2,2 GFlops auf der GTX780.
Die Methode mit intragrids zeigt eine etwas, aber noch nicht viel schnellere Methode als die des Kernel 3. So erreicht die RTX2070super bis zu 20Gflops, die P100 bis zu 11 GFlops. Auf der GTX780 ist die eine korrekt Ausführung des Kernels 4 leider noch nicht möglich.
Deutlich schneller hingegegen sind die Ausführungen von Kernel 1 und 2. Interessant zu beobachten ist, dass auf der GTX780 Kernel 1 und Kernel 2 ähnlich schnell sind. Auf der RTX2070super Kernel 2 meist etwas schneller ist als Kernel 1. Auf der P100 hingegen ist Kernel 1 hingegen deutlich schneller als Kernel 2. Ein mögliche Erklärung hierbei findet sich anhand der Hardwarespezifikationen. Die P100 hat deutlich mehr Cuda Cores als die GTX780, RTX2070super, wodurch sie mehr Threadblöcke gleichzeitig starten kann, was ihr bei der Ausführung von viele Threadblöcke, also auch beim Wiederaufruf des Kernels zu Gute kommt. Die RTX2070super ist zudem höher getaktet als die P100, was ihr bei den Atomic Operationen hilft. Jedoch lässt es sich nicht allein auf dieses Argument herunterbrechen, da die GTX780 deutlich niedriger Taktet als die anderen beiden GPUs.
Somit liefert die P100 mit dem Kernel 1 bis zu 70 Gflops, mit dem Kernel 2 bis zu 48 GFlops.
Die RTX2070super liefert mit dem Kernel 1 bis zu 33 Gflops, mit dem Kernel 2 bis zu 47GFlops. Auf der GTX780 erreicht man mit dem Kernel 1 bis zu 21 Gflops, mit der Kernel 2 bis zu 19 Gflops. Die schnellst Methode findet sich jedoch im Algorithmus mit shuffle Operationen, welcher nochmal eine bis zu drei mal bessere Performance liefert.

Vergleich der Grafikkarten

3.2.1 Sharedmemory Methode

Bei der Verwendung der Methode mit Shared Memory ist die P100 deutlich schneller als die RTX2070super und die GTX780. Hierbei wird vor allem die größere Anzahl an Cuda Cores ihr helfen, da somit mehr thread Blöck gleichzeitig ausgeführt werden können.

3.2.2 Kernel 2

Bei der Ausführung von Kernel 2 liefert die P100 und die RTX2070super eine vergleichbare Performance. Die GTX780 ist deutlich langsamer als die anderen Karten was auf die geringer Anzahl an Cuda Cores und den geringer Takt zurückzuführen ist.

3.3 Atomic Operationen

Hier hat die RTX2070super deutlich schneller als die anderen beiden GPUs. Der schnellere Takt wird einen großen Einfluss dabei gespielt haben.

3.4 Einfluss der Problemgröße

3.5 Kernel 1 und Kernel 2

Bei den meisten Versuchen bei der Verwendung von Kernel 1 und Kernel 2 ist mit Vergrößerung der Problemgröße die Compute performance leicht gestiegen. Ein signifikanten Anstieg ist aber meist nicht zu sehen. Vereinzelt gab es aber auch hier leicht Abfälle der Performance mit größerer Problemgröße

3.6 Kernel 3

Besonders bei der P100 fällt die Performance mit Vergrößerung der Problemgröße bei Verwendung von Kernel 3 ab.
Bei der GTX780 ist ein leichter Verfall der Performance mit Vergrößerung der Problemgröße zu beobachten.
Bei Verwendung der RTX2070super je nach Threadblockgröße (sx) ein leichter oder starker Abstieg der Performance mit Vergrößerung der Problemgröße zu beobachten.

3.7 Einfluss der Blockgröße

Die Performance Messung wurden mit fester Blockgröße bezüglich der Zeilen eines Threadsblocks $sy = 16$ aber Variabler Blockgröße sx bezüglich der Spalten eines Threadsblocks durchgeführt. Dabei war $sx \in \{4, 8, 16, 32, 64\}$. Je nach Kernel, GPU und Problemgröße war zwischen den verschieden Blockgröße sx ein Performanceunterschied vom Faktor bis zu vier zu sehen. Bei anderen Versuchen. Interessant dabei war zu beobachten, dass bei Kernel 1 und 2 auf der RTX2070super die Ausführung mit $sx = 8$, auf der GTX780 meist die Ausführung mit $sx = 16$ am schnellst war. Auf der P100 hingegen war für Kernel 1 bei kleiner Problemgröße die Ausführung für $sx = 8$ am schnellst und für große Problemgröße die Ausführung mit $sx = 16$ leicht schneller als die mit $sx = 8$.

Bei Kernel 2 hingegen war die Ausführung mit $sx = 32$ deutlich schneller auf der P100 als die anderen Ausführungen. Das liegt wahrscheinlich daran, dass die P100 recht langsam bezüglich der Atomic add Operationen ist. Wird sx größer, so müssen weniger Atomic Add Operationen durchgeführt werden, da die Threadblöcke größer werden und nur zwischen den Threadblöcken die Atomicadd Operation ausgeführt wird. Werden die Threadblöcke größe, so benötigt es weniger Threadblöcke um die Matrix abzudecken, demzufolge gibt es weniger Threadblöcke und es werden weniger Atomicadd Operationen ausgeführt. Eine weiter Erhöhung der Blockgröße auf $sx = 64$ führt jedoch zu einer schlechteren Performance.
Eine Richtlinie, welche Blockgröße generell Ideal ist konnte ich aber wegen der sehr verschiedenen Ergebnissen auf den verschiedenen Grafikkarten nicht feststellen.

3.8 Einfluss der Größe von Shared Memory/ L1 Memory

Kernel 1, 2 und 3 und wurden mit verschiedenen Cache Konfiguration durchgeführt.

```
(1) cudaFuncSetCacheConfig(kernel, cudaFuncCachePreferL1);
(2) cudaFuncSetCacheConfig(kernel, cudaFuncCachePreferShared);
(3) cudaFuncSetCacheConfig(kernel, cudaFuncCachePreferNone);
```

Dabei wird 64 kByte in Shared Memory und L1 Memory aufgeteilt. In Konfiguration 1 wird 48Kbyte dem L1 Speicher, 16 KByte dem shared memory, in Konfiguration 2 wird 16 kByte dem L1 Speicher, 48kByte dem shared memory und in Konfiguration 3 je 32 kByte den beiden Speicher zugeordnet.
Zwischen den Ausführungen mit verschieden Shared Memory Konfiguration sieht man meist keine großen Unterschiede. Das liegt daran, dass in jeder Implementierung nur ein shared Memory Objekt im Kernel definiert wurde. Dies hat maximal die Größe $1024 \cdot \text{sizeof(float)}$, also ist Maximal 4 kByte groß sodass es Problemlos auch in Konfiguration 1 mit dem kleinen shared Memory passt.

3.9 Differenz zur theoretischen Peak Performance

Bei allen Algorithmen und allen GPUs ist die erreicht Performance weit von der theoretischen Peak Performance entfernt. So erreicht man mit der P100 im shuffle Algorithmus maximal 240 GFlops, mit Kernel 1 maximal 70 Gflops, was nur ein hundertstel der theoretischen Peakperformance von 9400 Gflops entspricht. Die Differenz bei der RTX2070super und der GTX780 liegt in einer ähnlichen Größenordnung. Ein Grund hierfür ist, dass die geladenen Matrixeinträge der Ursprungsmatrix A nur einmal zur Berechnung benutzt werden und auch bei Grafikkarten die Recheneinheiten deutlich mehr Daten verarbeiten könnten, als die Speichereinheiten Daten liefern können. Bei einer Matrix-Matrix-Multiplikation müssen die Daten der Ursprungsmatrix deutlich häufiger verwendet werden als bei einer Matrix-Vektor-Multiplikation, da für $C+ = AB$ jede Zeile von Matrix A mit jeder Spalte aus Matrix B multipliziert wird. Somit bin ich in Hausaufgabe 3, der Matrix-Matrix-Multiplikation, auf eine Performance von knapp 600 GFlops mit der RTX2070super gekommen. Dies entspricht der dreifachen Performance der Implemtierung des shuffle Algorithmus, und sogar der 12-fachen Performance der Implemtierung von Kernel 2(47 GFlops) und der 18 fachen Performance der Implemtierung von Kernel 1 (33 Gflops).

3.10 Performance Vergleich CPU

Ein Single Core CPU Implementation das Matrix-Vektor-Produktes, bei der die Matrix in Teilmatrizen aufgeteilt war für effektiver Transfer bezüglich Cache Lines, hat auf meinem System, mit AMD Ryzen 5 3600 eine Performance zwischen 0,1 und 3 Gflops erreicht. Besonders bei großer Problemgröße war diese sehr langsam. So hat sie für das Matrix-Vektorprodukt mit Dimension 16*1024 Werte zwischen 0,01 und 0,42 GFLOPS, je nach Größe der Teilmatrix.
Vor der Krise bezüglich der Verfügbarkeit von Grafikkarten hat man für ungefähr den gleichen Preis ein Nvidia RTX2070 super oder einen AMD Ryzen 9 3900X bekommen. Letzterer ist eine 12-core CPU mit einer vergleichbaren single core Performance zum AMD Ryzen 5 3600.
Geht man von einer Perfekten Parallelisierung das Programms aus und dem Idealfall der Singlecore Impletmentierung von 0,42 GFlops, so erhält man mit dem Ryzen 9 3900X eine Performance von $0,42 GFlops * 12 = 5,04 GFlops$. Dies wäre um ein Faktor 9 schlechter, als die Performance des Matrix Vektor Produkts Kernel 1.2 auf der RTX2070super, sx=8, size=16*1024, und sogar um ein Faktor 35 schlechter als der shuffle Algorithmus auf der RTX2070super (185 Gflops) für size=16*1024.
Ähnliches erkennt man bei der theoretischen Peak Performance. Der AMD Ryzen 9 3900x unterstüzt AVX2 und FMA Operation. Kann also 2 Operatioen auf 256bit pro Takt ausführen. 256bit entsprechen der Größe von 8 floating Point Datentypen. Die Theoretische performance errechnet sich hierdurch durch:

$$\text{SP Performance} = \text{fma} \cdot \#\{\text{Cores}\} \cdot \text{Values per cycle} \cdot \text{Takt}$$

$$= 2 * 12 * 8 * 3,8 GHz = 729,6 GFlops$$

Die theoretische Peakperformance des Ryzen 9 3900x liegt ist mit 730GFlops ist um einen Faktor 11 schlechter als die der Nvidia RTX2070super mit theoretischen 8243 GFlops, Table 1 Somit ist auszugehen, dass der Performanceunterschied auch bei weiteren Optimierung des Codes erhalten bleibt.
Anhand dieses Beispiels man gut, warum für Berechnungen von großen Matrix-Matrix oder Matrix-Vektorprodukten oft GPUs anstatt CPUs verwendet werden. Bei gleichbleibenden Operationen auf großen Datensätzen bieten die GPUs aufgrund der hohen Anzahl an Cuda Cores enorme Performance Vorteile gegenüber CPUs. CPUs werden trotzdem weiterhin eine zentrale Rolle in Computer spielen, da sie deutlich schneller bei der Ausführung von vielen verschiedenen Instruktionen nacheinander sind und der Großteil der Software auf CPU Ausführung programmiert ist.

4 Quellen

- https://www.nvidia.de/gtx-700-graphics-cards/gtx-780/ -https://www.computerbase.de/2016-04/nvidia-tesla-p100-gp100-als-grosser-pascal-soll-all-in-fuer-hpc-markt-gehen/ - https://en.wikichip.org/wiki/amd/ryzen$_9$/3900x