

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	3
2	Algorithmen zur Matrix-Vektor-Operation	3
2.1	Sharedmemory mit wiederholten Aufruf des Kernels	3
2.2	Sharedmemory mit Atomics	4
2.3	Nur Atomic Operationen	5
2.4	Intragrids	5

1 Motivation

Matrix Vektor Operationen gehören sind Elementar für verschiedene 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 Grafikkarten 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.

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

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. Auß 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 bzw, itgrid gespeichert. Dies wird hier auch wieder in zwei Dimensionen dargestellt, da somit man 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 Threadblock 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 das jeder Matriceintrag $a_{i,j}$ 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 thread Blocks. Der shared Memory wird so indeziert, dass dieser wieder als Matrix der Größe $sx*sy$ gelesen werden kann. Dabei werden Produkte in eine Zeile geschrieben, beiden 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 ... Pointer Reduzierung zu Nutzen gemacht. 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 $toreduce/sx$ gesetzt. ($toreduce$ viele Einträge sind pro Zeile zu reduzieren, sx viele Zeileneinträge pro threadblock). buff, welcher das Zwischenergebnis enthält, `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 Threadblocks ist wieder ein Spaltenvektor mit sy vielen Spalten. Die Spaltenvektoren werden wieder in den Zwischenspeicher buff geschrieben, wobei wie 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 spalten und $toreduce/sx$ vielen Zeilen durch. 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 einen reduziert, sodass wir das die vorderste Spalte des Zwischenergebnis in das Endergebnis speichern können.

2.2 Sharedmemory mit Atomics

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 . Jedoch werden die Einträge der Spaltenvektoren, der verschiedenen threadblocks, welche aus der selben Zeile der Ursprungsmatrix hervorgehen, auf einen Wert in einen 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

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. Das Produkt, speichert der thread in eine lokale Variable `addsc` ab. Es wurde vorher ein Speicher `buff` angelegt, welcher so viele Einträge hat, 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