Matrix-Vektor-Produkt

Jonas Gütter
Matrikelnr 152127

28. Juni 2017

Im Rahmen der Aufgabe wurden verschiedene Implementierungen eines Matrix-Vektor-Produkts $A*x$ im Hinblick auf ihre Laufzeit miteinander verglichen. Folgende Implementierungsmöglichkeiten wurden betrachtet:

- Serielles Berechnen von Ax auf der CPU
- Serielles Berechnen von A^Tx auf der CPU
- Paralleles Berechnen von Ax auf der GPU
- Paralleles Berechnen von A^Tx auf der GPU
- Paralleles Berechnen von Ax auf der GPU unter Nutzung von shared memory
- Paralleles Berechnen von A^Tx auf der GPU unter Nutzung von shared memory
- Paralleles Berechnen von Ax auf der GPU unter Nutzung von shared memory und dynamischer Parallelisierung
- Paralleles Berechnen von A^Tx auf der GPU unter Nutzung von shared memory und dynamischer Parallelisierung

Abbildung 0.1 zeigt die Laufzeiten der verschiedenen Implementierungen in Millisekunden für verschiedene Matrix-Größen. Bei der seriellen Implementierung zeigt sich, dass die Bildung des Produkts Ax um etwa eine Größenordnung schneller ist, als die Bildung des Produkts A^Tx. Dies war auch zu erwarten, da bei einem Speicherzugriff mehrere aufeinanderfolgende Elemente in den Cache geladen werden, die bei Ax auch direkt benötigt werden. Bei A^Tx dagegen werden keine aufeinanderfolgenden Elemente benötigt, in jeder Iteration muss also ein neuer Zugriff auf den Hauptspeicher erfolgen. Bei der parallelen Berechnung ohne shared memory und dynamischer Parallelisierung verhält es sich genau andersherum, dort ist die Berechnung von A^Tx deutlich schneller als die Berechnung von Ax. Dies liegt daran, dass jeder Thread eine Zeile der Matrix bearbeitet. Es werden also etwa gleichzeitig die ersten Elemente aller Zeilen abgerufen. Bei A^Tx befindet sich ein großer Teil davon nach dem ersten Speicherzugriff bereits im Cache, daher wird in diesem Fall weniger Zeit benötigt. Die Implementierung mit Hilfe von shared memory weist im Vergleich dazu nur kleine Unterschiede zwischen Ax und A^Tx auf, die Variante Ax ist jedoch durchgehend schneller. Da hier wieder mehrere Threads dieselbe Zeile bearbeiten, ist es wieder von Vorteil, dass aufeinanderfolgende Elemente sich unter Umständen bereits vor dem Speicherzugriff im Cache befinden. Dasselbe gilt für die Implementierung mit shared memory und dynamischer Parallelisierung.
In den Abbildungen 0.2, 0.3 und 0.4 sind die verschiedenen Laufzeiten in Abhängigkeit von der Matrixgröße dargestellt. Vor allem in Abbildung 0.4 ist zu erkennen, dass die parallele Berechnung ohne shared memory und ohne dynamischer Parallelisierung am wenigsten Zeit braucht. Dies liegt möglicherweise daran, dass für den Einsatz des shared memory das Programm teilweise wieder durch eine for-Schleife serialisiert wurde. Auch wurden atomare Operation eingesetzt, die die Laufzeit weiter erhöhen. Mit Abstand am langsamsten ist die serielle Implementierung in der Variante A^Tx, für Matrixgrößen ab

	nr_1	nr_2	nr_3	nr_4	nr_5
array size / 1024	1	2	4	8	16
Ax CPU	2.648388	10.844859	43.006201	171.924113	687.50807
A^Tx CPU	8.519093	86.301981	379.152365	1568.62075	6585.118184
Ax GPU	1.381536	2.760736	5.492832	21.878529	91.003487
A^Tx GPU	0.49696	0.990336	1.980096	4.138976	10.681856
Ax GPU shared mem	6.868512	13.728864	27.623072	57.452255	110.084351
A^Tx GPU shared mem	8.026976	16.109024	32.605919	79.533058	250.683838
Ax GPU shared mem dp	3.77552	4.148192	131.763428	448.153351	1530.365967
A^Tx GPU shared mem dp	4.357088	5.189088	132.323929	562.721436	2018.924438

Abbildung 0.1: Laufzeiten(ms) der verschiedenen Implementierungen für verschiedene Matrixgrößen

4096x4096 Einträgen wird auch der Einsatz der dynamischen Parallelisierung sehr langsam. Die Threadanzahl ist bei dieser Variante im Vergleich zu den anderen Implementierungen am größten, möglicherweise ist die Threadanzahl so groß, dass eine quasi-parallele Beabreitung durch Hyperthreading nicht mehr möglich ist.
Die einfache parallele Berechnung des Produkts $A^T x$ ohne weitere Konzepte ist also von den betrachteten Implementierungen am effektivsten. Der Einsatz des shared memmorys und der dynamischen Parallelisierung kann aber möglicherweise noch optimiert werden, so dass dort noch bessere Laufzeiten entstehen könnten.

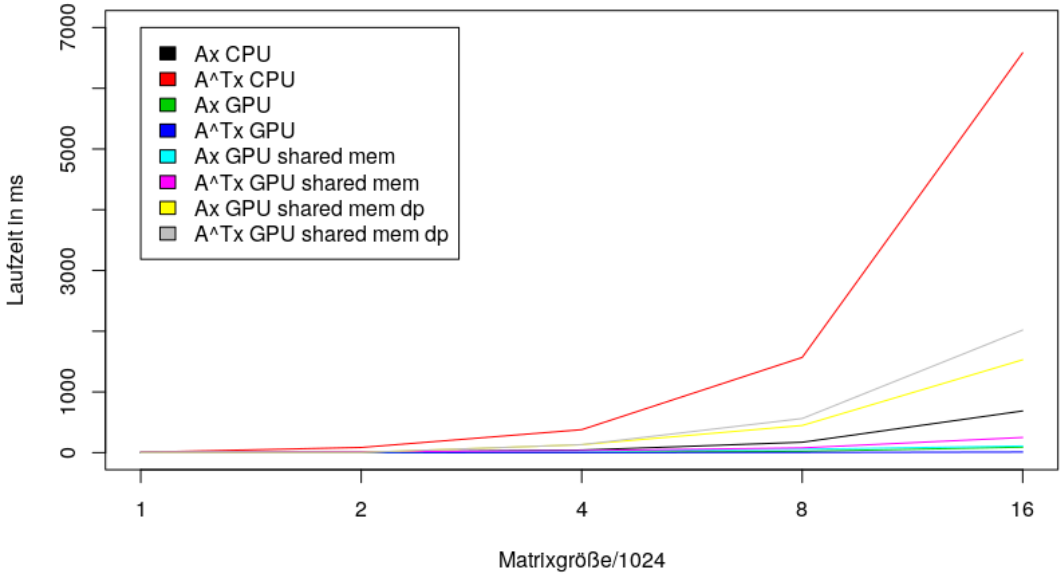

Abbildung 0.2: Entwicklung der Laufzeiten

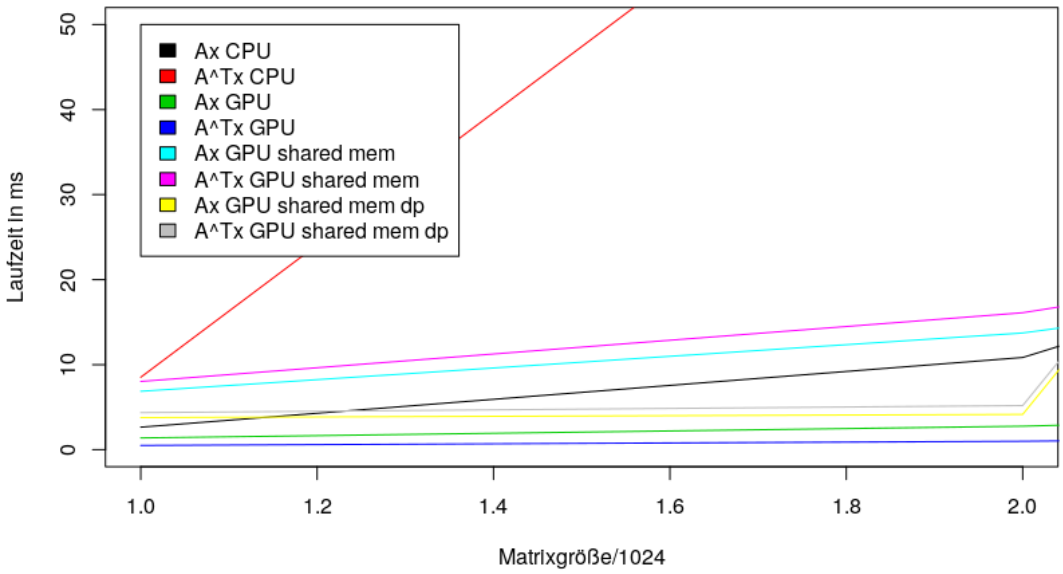

Abbildung 0.3: Entwicklung der Laufzeiten bei kleinen Problemgrößen

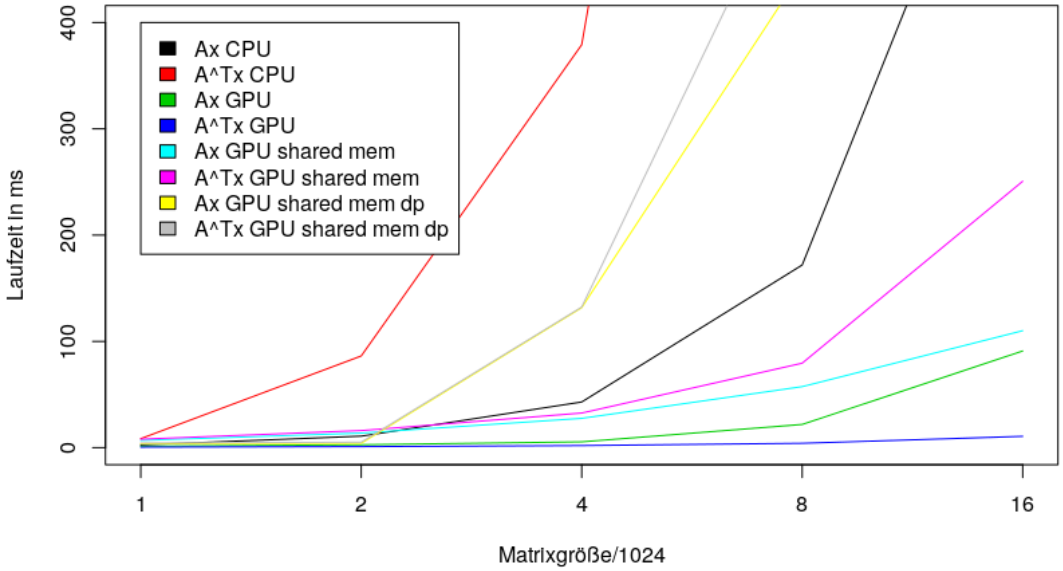

Abbildung 0.4: Entwicklung der Laufzeiten unter 400 ms