Übungen zur Vorlesung

Paralleles Höchstleistungsrechnen

Dr. S. Lang, D. Popović

Abgabe: 02. Februar 2010 in der Übung

Übung 31 Matrix-Matrix Multplikation mit Cuda (5 Punkte)

In Übung 2 haben wir zwei quadratische Matrizen miteindander multipliziert und die $MFLOP$-Rate gemessen. Wir wollen diesen Algorithmus nun mittels *Cuda* parallelisieren und die Performance messen. Auf der Homepage finden Sie ein Gerüst für das Projekt, das die Dateien `mm_cuda.cu` und ein für den Pool passendes Makefile enthält. Die *Cuda*-Applikationsdatei kann mit dem Makefile kompiliert werden und führt das sequentielle Programm auf einer CPU aus. Die Funktion dazu heisst `runVanilla`.

Schreiben Sie nun eine Funktion `runCuda`, die einen Cuda-Kernel zur Matrix-Matrix-Multiplikation für Matrizen aus $\mathbb{R}^{n\times n}$ ausführt. Sie sollten mit der Funktion die gesamte verstrichene Zeit für das Programm messen können (elapsed time in s). Ihr Programm sollte ausserdem mindestens folgende Punkt realisieren:

- Um das Zeit-intensive Laden aus dem globalen Device-Speicher zu vermeiden, verwendet der Kernel ein Tiling (Tilesize t),
- mit dem die Threads die verschiedenen Tiles in den *Shared memory* laden, von wo aus die anderen Threads die Daten wiederverwenden können.
- Der Einfachheit halber gilt $n \mod t = 0$.

Messen Sie nun die $MFLOP$-Rate beider Varianten. Verwenden Sie hinreichend grosse Matrizen. Nehmen Sie zuletzt noch eine Messreihe für die paralllele Variante mit verschiedener Tilesize t auf $(t = 4, 8, 16, 32, 64)$. Wie verhält sich nun die Laufzeit in Abhängigkeit von t, kann das Programm immer ausgeführt werden?

Hinweise:

- Eventuell müssen Sie im Pool den Pfad zur Bibliothek `libcudart.so.2` explizit setzen: `export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda/lib/`.
- Falls Sie Probleme mit der Aufgabe haben, lesen Sie sich das Textbook *Cuda Programming Model* von D. Kirk und W. Hwu durch, insbesondere in Kapitel 4 stehen sehr wertvolle Hinweise. Das Buch im WWW gibt es zum Beispiel hier: `http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook`.
- Für die Zeitmessung verwenden wir die verstrichene Gesamt-Zeit in s. Die Rechnungen sollten daher so umfangreich sein, dass eine sinnvolle Zeitmessung möglich ist. Hierbei sollten Sie auch sicherstellen, dass die Zeitmessung nicht zu sehr durch andere Prozesse gestört wird.

Übung 32 Cuda – Limiting factors (5 Punkte)

Die verschiedenen *Cuda*-Speicher sind natürlich durch die Hardware beschränkt. Mit den Grenzen wollen wir uns in dieser Übung beschäftigen, denn die Speichergrenzen limitieren ihrerseits die Anzahl der möglichen Threads in einem Streaming Mulitprozessor (SM).

Wir betachten exemplarisch die Daten der GPU *GeForce 8800 GTX* aus dem Jahr 2006. Diese hat z.B. insgesamt $128K$ *Registerspeicher*, die aufgeteilt sind in $8K$-Register pro SM.

- Wenn die maximale Anzahl Threads pro SM 768 ist, wieviel Register darf jeder Thread maximal verwenden, um die maximale Anzahl verwendeter Threads zu ermöglichen?

- Wenn die Threads mehr Register verwenden, reduziert sich die Anzahl verwendet Threads pro SM. Wenn diese Reduktion auf Block-Granularitätsebene geschieht, und jeder Block 256 Threads enthält, wieviele Threads können dann maximal pro SM verwendet werden? Wie gross ist die relative Reduktion?

Auch der *Shared memory* kann ein limitierender Faktor werden. Die genannte Karte hat $16K$ *Shared memory*. Da jeder SM aus bis zu 8 Blöcken besteht, bedeutet dies maximal $2K$ pro Block. Brauchen die Blocks mehr gemeinsamen Speicher, reduziert dies die Anzahl der möglichen Blöcke. Betrachten wir dazu das Matrix-Matrix-Multiplikationsbeispiel.

- Wenn die Tilesize $t = 16$ ist, wieviel braucht dann jeder Block an gemeinsamen Speicher, falls jeweils ein Block eine Tile auf einmal bearbeitet? Wieviele Blöcke können also pro SM verwendet werden? Ist der gemeinsame Speicher für dieses Set-Up ein limitierender Faktor?
- Nun werde t auf 32 erhöht. Wieviel *Shared memory* braucht nun jeder Block? Wieviele Blöcke kann ein SM nun beherbergen? Ist der gemeinsame Speicher für dieses Set-Up ein limitierender Faktor?

Ein weiters Problem kann die *Divergenz* verschiedener Threads bedeuten. Diese kann z.B. auftreten, wenn ein Verzweigungsbedingung eine Funktion der Thread-ID ist. Betrachten Sie folgendes Code-Fragment. Es erzeugt zwei Ausführungspfade für die Threads eines Blocks:

```
if(threadIdx.x > 2) {// do sth.;} else{// do sth. else;}
```

Folgen aber die Threads eines Warps in diesem Beispiel immer zwingend demselben Ausführungspfad? Wie könnten Sie dieses Beispiel „Divergenz-frei“ machen, d.h., alle Threads eines Warps folgen demselben Ausführungspfad, dennoch werden pro Block zwei Ausführungspfade angeboten?

Übung 33 Reduktionsoperationen mit Cuda (10 Punkte)

Lesen Sie das bereitgestellte Dokument *Optimizing Parallel Reduction in Cuda* (`reduction.pdf`). Dort wird die parallele Reduktion am Beispiel der Summenbildung beschrieben und sieben verschiedene Varianten der Optimierung gezeigt, und zwar

1. Interleaved Addressing + divergente Branches,
2. Interleaved Addressing + nicht-divergente Branches, dafür aber Bank-Konflikte,
3. Sequentielle Addressierung,
4. Addieren schon beim ersten Loading,
5. Unrolling des letzten Warps,
6. Komplettes Unrolling,
7. Multiple Elemente pro Thread (max 64 Blocks).

Uns interessieren die Varianten 1, 3 und 6. Implementieren Sie ein *Cuda*-paralleles Programm, dass N Zahlen, die in einem Feld gespeichert sind, addieren kann (alternativ: Bilden des Maximums) und zwar in den oben genannten drei Varianten. Das Synchronisieren über die Threads ist kein Problem, da mit `__syncthreads()` möglich, Sie müssen Sich aber um die globale Reduktion kümmern. Messen Sie die Laufzeit in Abhängigkeit der Anzahl an Elementen und vergleichen Sie die Varianten untereinander, wie es auch im Dokument zu sehen ist.

Übung 34 Gauss-Seidel-Verfahren mit Cuda (15 Zusatz- Punkte)

In den bisherigen Übungen haben wir das Gauss-Seidel-Verfahren für die Poisson-Gleichung kennengelernt, wodurch in jedem Iterationschritt m folgendes Gleichungssystem berechnet werden muss:

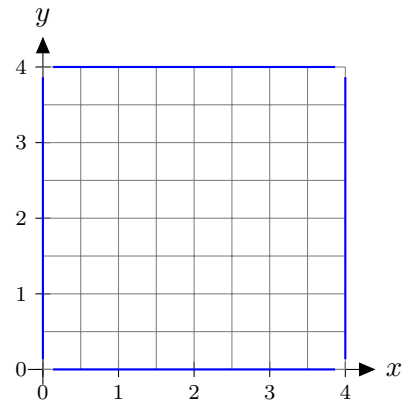
$$A\mathbf{x} = \mathbf{b} \quad \Longleftrightarrow \quad \left(\sum_{j=1}^{i-1} a_{ij} x_j^{m+1}\right) + a_{ii} x_i^{m+1} = b_i - \sum_{j=i+1}^{n} a_{ij} x_j^m.$$

In einer weiteren Übung haben Sie ausserdem die *Rot-Schwarz-* oder *Schachbrett-Musterung* für das Verfahren kennengelernt. Dieses Verfahren wollen wir nun mit *Cuda* auf der GPU implementieren.

Gegeben sei die partielle Differential-Gleichung (Poisson-Gleichung)

$$\begin{aligned} \triangle u &= -2\big((y^2-1)+(x^2-1)\big) && \text{in } \Omega, \\ u &= 0 && \text{auf } \partial\Omega. \end{aligned}$$

Das Gebiet Ω sei das Einheitsquadrat, wie auf den ersten beiden Übungszetteln definiert (siehe Abbildung rechts, dort aber skaliert). Die Gleichung hat dann die analytische Lösung $u(x, y) = (x^2 - 1) \cdot (y^2 - 1)$.



Implementieren Sie das Gauss-Seidel-Verfahren mit *Rot-Schwarz-Musterung* für diese Gleichung mit *Cuda* auf einem äquidistanten Gitter mit Schrittweite h. Testen Sie nach jedem (oder auch nur nach mehreren Schritten, um Last zu minimieren) Schritt auf Konvergenz, indem Sie eine Norm der (diskreten) Lösung, z.B. die L^2-Norm

$$\|u\|_{L^2}(\Omega) := \sqrt{\int_\Omega u^2 \, dx},$$

berechnen und abbrechen, falls diese kleiner einer frei wählbaren Toleranz ist. Überprüfen Sie zudem, mit welcher Asymptotik die Lösung für eine kleiner werdende Gitterweite, $h \to 0$, gegen die analytische Lösung konvergiert, ebenfalls in der gewählten Norm. Lassen Sie sich nach jedem Schritt die $MFLOP$-Rate ausgeben, so dass wir Sie mit den Ergebnissen vom erste Übungsblatt vergleichen können.

Für diese Übung können Sie auf die vorherigen Aufgaben zurückgreifen: Die Gauss-Seideliteration ist eine Matrix-Matrix-Multiplikation, allerdings mit variabler Matrixgrösse (der Vektor ist keine quadratische Matrix), und die Norm kann man mit der Reduktionsoperation implementieren.