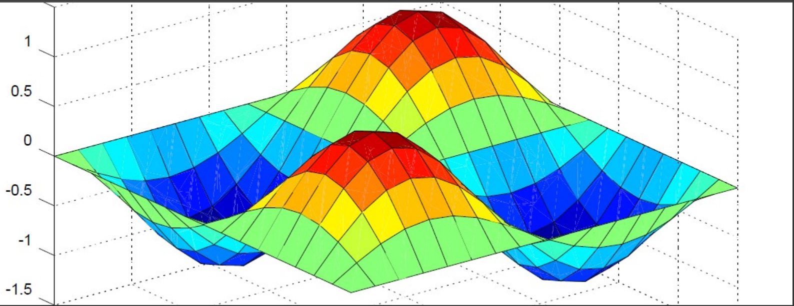
Karlsruher Institut für Technologie

Projekt 2

Arbeiten mit CUDA

Fabian Miltenberger, Sébastien Thill, Thore Mehr | 07.02.2017

LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELVERARBEITUNG (ITEC)



Gliederung

Struktur nach Aufgaben, aber mit grobem Einblick in unsere Vorgehensweise

Aufgabe 1 – Getting started

```
Information for GeForce GTX 960 (device 0):
Total global memory: 2092957696 ≈ 2 GB
Total const memory: 65536 ≈ 64 KB
Shared memory per block: 49152 ≈ 48 KB
Warp size: 32
Max threads per block: 1024
Max threads dimension: [ 1024, 1024, 64 ]
Max grid size: [ 2147483647, 65535, 65535 ]
```

→ Damit haben wir gearbeitet

Aufgabe 2 – Datentransferraten

- CudaMemcpy
  - 2,4 GB per s im RAM
  - 2,4 GB per s Up
  - 40 GB per s Trans
  - 1,4 GB per s Down
- Copy and Add
  - 2 GB per s im RAM
  - bis 30 GB per s Trans
  - fast linear mit Blockgröße

Aufgabe 3 – Gauß-Seidel-Verfahren

Implementierung: Sehr geradlinig, Synchronisation durch Kernelaufrufe

Beschleunigung gegenüber CPU (32 Kerne):

l	$T_{CPU}[s]$	$T_{GPU}[s]$	Beschleunigung $S[s]$
2	0,123	1,15	0,107
3	0,348	1,16	0,3
4	1,42	1,17	1,21
7	7,14	1,54	4,64
8	25,3	3,27	7,74
9	177	20,6	8,59

Aufgabe 3 – Asynchrone Parallelisierung

- **Approximate Computing**
  Verzicht auf hohe Genauigkeit und im Gegenzug an Geschwindigkeit und/oder Energieersparnisse in den Berechnungen zu gewinnen.

- **Asynchronous Parallelization**
  Ein Thread kann alle seine Iterationen durchlaufen kann ohne auf die Ergebnisse der anderen Threads zu warten
  $\rightarrow$ eleminieren der Wartezeiten

- **Relaxierte Parallelisierung**
  Unabhängig zwischen den entstandenen zeitlichen Unterteilungen Berechnungen durchführen.

Aufgabe 3 – Approximate Computing

Keine Verbesserung durch Flag `-use-fast-math`

Beschleunigung bei `float` anstatt `double`:

l	$T_{\texttt{double}}[s]$	$T_{\texttt{float}}[s]$	Beschleunigung $S[s]$
2	1,15	1,15	1,00
3	1,16	1,15	1,01
4	1,17	1,15	1,02
7	1,54	1,49	1,03
8	3,27	2,59	1,26
9	20,6	12,1	1,70

Weitere Beschleunigung für `half` möglich? (ab CUDA 7.7)

Aufgabe 3 – Programmierfehler

Dieser Code bricht manchmal zu früh ab:

```
...
int smallErr;
cudaMemcpy(&smallErr, smallErr_d, 1, DeviceToHost);
if (smallError) break;
...
```

Aufgabe 3 – Programmierfehler

Dieser Code bricht manchmal zu früh ab:

```
...
int smallErr;
cudaMemcpy(&smallErr, smallErr_d, 1, DeviceToHost);
if (smallError) break;
...
```

→`smallErr` hat zu großen Datentyp, muss entweder initialisiert werden, oder von Datentyp `char` sein

Aufgabe 3 – Weitere Architekturen

Einsatz von **Approximate Computing**
→ kürzere Rechenzeit auf Kosten der Genauigkeit

In modernen CPUs Gleitkommazahlen intern mit 80 Bit verarbeitet
→ kein zeitlicher Unterschied zwischen `double` und `float`

Aufgabe 3 – Weitere Architekturen 2

Ein Unterschied bei sehr großen Datenmengen?

Speicherbandbreite möglicherweise bei `double` als Engpass
Eher reine Multi-Core CPUs betroffen
Many-Core- und MIC-Architekturen weniger:

- größere Caches
- mehrere Speicheranbindungen

Aufgabe 4 – Einführung

Gesucht: Dreiecksmatrizen L, U, sodass

$$L \cdot U \approx A$$

Approximate Computing herangehensweise
$\rightarrow$ Incomplete LU-Decomposition (ILU)

Algorithmus 1: Fixpunktiteration
Algorithmus 2: Gauß

Aufgabe 4 – ILU-Zerlegung

Algorithmus 1

- Einträge komplett unabhängig
  $\rightarrow$ Sehr gut parallelisierbar
- Pufferung zwischen Iterierten
- Iterativ

Algorithmus 2

- Starke Abhängigkeit der Einträge
  $\rightarrow$ Schlecht parallelisierbar

Aufgabe 4 – Mathematische Sicht

Algorithmus 1

- Stabil
- Anzahl an benötigten Schritten nicht bekannt

Algorithmus 2

- Keine Pivotwahl
  $\rightarrow$ mögliche Stabilitätsprobleme
- Nach n Schritten fertig

Aufgabe 4 – Speichermethodik

Nach Algorithmus 2: $a_{ij} = 0 \Rightarrow l_{ij} = 0$ und $U = L^T$ (S_U entsprechend)
Belegung von L, U und A:

$$\mathcal{B} = \begin{pmatrix} * & * & 0 & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & * & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & * & 0 & 0 & 0 \\ * & 0 & 0 & * & * & 0 & * & 0 & 0 \\ 0 & * & 0 & * & * & * & 0 & * & 0 \\ 0 & 0 & * & 0 & * & * & 0 & 0 & * \\ 0 & 0 & 0 & * & 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & 0 & * & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & * & 0 & * & * \end{pmatrix}$$

$\rightarrow$Speicherung möglich als Array der Größe $5n$
Speicherverschnitt in $\mathcal{O}(\sqrt{n})$

Aufgabe 4 – Ergebnis

Stark abweichende Einträge in $L \cdot U$:

$$L \cdot U \approx \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & \color{red}{0.25} & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & \color{red}{0.27} & -1 & 0 & 0 & 0 \\ -1 & \color{red}{0.25} & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & \color{red}{0.27} & -1 & 4 & -1 & \color{red}{0.27} & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & \color{red}{0.29} & -1 \\ 0 & 0 & 0 & -1 & \color{red}{0.27} & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & \color{red}{0.29} & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix} \overset{!}{\approx} A$$

Für unsere folgende Vorkonditionierung *hoffentlich* vernachlässigbar

Aufgabe 4 – OpenMP

Für Algorithmus 1: Für kleine *n* würde eine Parallelisierung in OpenMP – also auf der CPU – möglicherweise Sinn machen, aufgrund des Mehraufwandes der für die GPU-Ausführung erforderlich ist. Für große *n* hingegen erwarten wir, dass die CUDA-Implementierung einen Performance-Vorteil gegenüber der CPU-Implementierung hat. Dies schließen wir vor allem daraus, dass sich der Algorithmus sehr gut für die Grafikkarte parallelisieren lässt, da innerhalb einer Iteration Einträge unabhängig von einander sind.

Aufgabe 5 – CG-Verfahren

Idee: Anstatt $B \approx A^{-1}$ berechnen wir $LU = A$ durch unvollständige Zerlegung
Nun können wir $Br = p$ bzw. $r = LAp$ berechnen durch:

$$L\hat{p} = r, Up = \hat{p}$$

In unserem Fall: mittels GSV

Aufgabe 5 – Poisson-Problem

Gefordert: $\epsilon_i \leq 10^{-5}$

Unsere Implementierung mit $l = 9$: $\epsilon_{max} = 1,257 \times 10^{-5}$

Aufgabe 5 – Poisson-Problem

Gefordert: $\epsilon_i \leq 10^{-5}$
Unsere Implementierung mit $l = 9$: $\epsilon_{max} = 1,257 \times 10^{-5}$

Intention: $l = 10$ setzen
Würde $(2^{10} - 1)^3 \cdot 8$ Bytes $\approx$8 GB für Historie von u benötigen
$\rightarrow$Speicherproblem

Aufgabe 5 – Poisson-Problem

Gefordert: $\epsilon_i \leq 10^{-5}$
Unsere Implementierung mit $l = 9$: $\epsilon_{max} = 1,257 \times 10^{-5}$

Intention: $l = 10$ setzen
Würde $(2^{10} - 1)^3 \cdot 8$ Bytes $\approx$8 GB für Historie von u benötigen
$\rightarrow$Speicherproblem

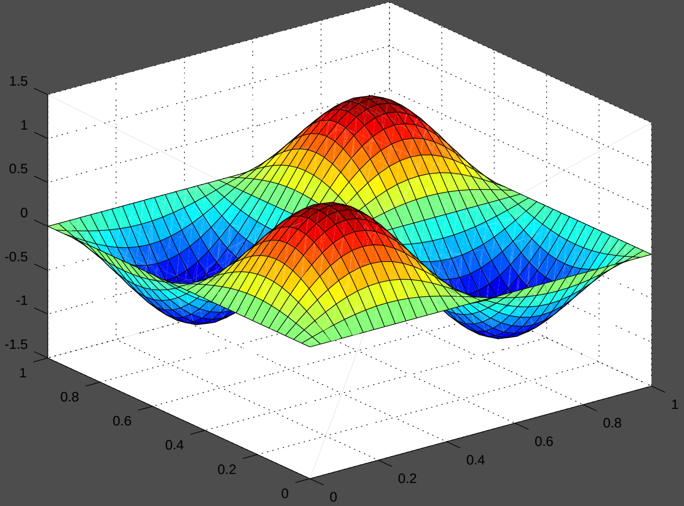
Lösung: Auf Historie verzichten
Speicherbedarf für u: $(2^{10} - 1)^2 \cdot 8$ Bytes $\approx$8 MB
Problem: Auswirkung unklar, aber funktioniert (Approximate Computing?)
Fehler jetzt: $\epsilon_{max} = 3,25 \times 10^{-6}$
Größere *blocksize* nötig?

Aufgabe 5 – Vergleich

l	Impl.	Iterationen	$T[s]$	$T_{iteration}[s]$	Fehler ϵ_i
7	CPU	243	$1,7$	7×10^{-3}	$2,01 \times 10^{-4}$
	GPU	91	$1,83$	$2,01 \times 10^{-2}$	$2,01 \times 10^{-4}$
8	CPU	483	2	$4,14 \times 10^{-3}$	$5,02 \times 10^{-5}$
	GPU	153	$7,04$	$4,60 \times 10^{-2}$	$5,21 \times 10^{-5}$
9	CPU	961	$2,5$	$2,6 \times 10^{-3}$	$1,26 \times 10^{-5}$
	GPU	285	$71,0$	$2,49 \times 10^{-1}$	$1,31 \times 10^{-5}$

$\rightarrow$ Weniger Iterationen, aber **mehr Zeitaufwand**

Aufgabe 5 – Poisson-Problem



Aufgabe 5 – Genauigkeit

Stellschrauben: $\varepsilon_{cg}, \varepsilon_{ILU}, \varepsilon_{GSV}$

Auswirkung auf den Fehler:

ε_{cg}	ε_{ILU}	ε_{GSV}	Fehler ϵ_{max}	$\Delta\epsilon_{max}$
10^{-3}	10^{-3}	10^{-3}	$2,20 \times 10^{-4}$	0
10^{-6}	10^{-3}	10^{-3}	$5,14 \times 10^{-5}$	$-1,69 \times 10^{-4}$
10^{-3}	10^{-6}	10^{-3}	$1,97 \times 10^{-4}$	$-2,30 \times 10^{-5}$
10^{-3}	10^{-3}	10^{-6}	$2,14 \times 10^{-4}$	$-6,02 \times 10^{-7}$
10^{-12}	10^{-3}	10^{-3}	$5,02 \times 10^{-5}$	$-1,70 \times 10^{-4}$
10^{-12}	10^{-12}	10^{-12}	$5,02 \times 10^{-5}$	$-1,70 \times 10^{-4}$

$\rightarrow \varepsilon_{cg}$ hat größte Relevant; naheliegend, da Rest Approximation

Aufgabe 5 – Beschleuniger

Bisher: Alles (außer Berechnung von *Br* (2×GSV)) auf CPU per OpenMP

Wenig sinnvoll, da:

1. Overhead durch Datenaustausch
2. Keine gleichzeitige Auslastung
3. GPU vermutlich schneller in den CPU-Aufgaben (Vektoroperationen)

Aufgabe 5 – Beschleunigung

Karlsruher Institut für Technologie

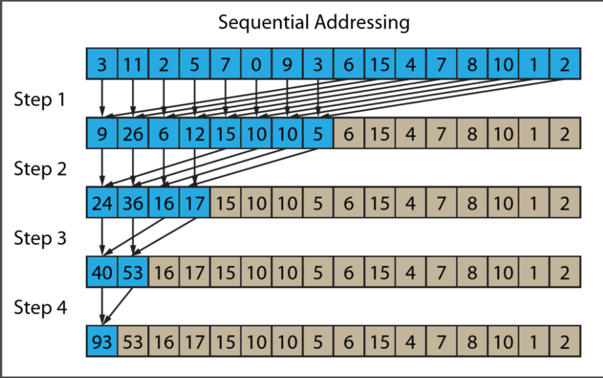
l	$S_{CPU \rightarrow GPU_a}[s]$	$S_{GPU_a \rightarrow GPU_c}[s]$
7	0,929	1,06
8	0,284	1,14
9	0,0352	1,11

Vorteil geringer als gedacht...
Br teuerster Berechnungsschritt

Aufgabe 5 – Skalarprodukt auf der GPU

Möglich mittels `atomic`-Operationen, aber nicht optimal

Bessere Lösung: *Reduction* (wie auch in OpenMP)



Aufgabe 6 – Lattice-Blotzmann-Methode

Karlsruher Institut für Technologie

Maximale Größe des Gitters?

Verfügbarer Speicherplatz auf der GPU $\approx$ 2 GB

Aufgabe 6 – Lattice-Blotzmann-Methode

Maximale Größe des Gitters?
Verfügbarer Speicherplatz auf der GPU $\approx$ 2 GB

Speicherbedarf für `rawdata1`, `rawdata2` und `u_0`:

$$\begin{aligned} M(\texttt{rawdata1}) &= M(\texttt{rawdata2}) \\ &= max_x \cdot max_y \cdot max_z \cdot 19 \cdot 8 \text{ Bytes} \\ M(\texttt{u\_0}) &= max_x \cdot max_y \cdot 8 \text{ Bytes} \end{aligned}$$

Aufgabe 6 – Lattice-Blotzmann-Methode

Karlsruher Institut für Technologie

Maximale Größe des Gitters?
Verfügbarer Speicherplatz auf der GPU $\approx$ 2 GB

Speicherbedarf für `rawdata1`, `rawdata2` und `u_0`:

$$\begin{aligned} M(\texttt{rawdata1}) &= M(\texttt{rawdata2}) \\ &= max_x \cdot max_y \cdot max_z \cdot 19 \cdot 8 \text{ Bytes} \\ M(\texttt{u\_0}) &= max_x \cdot max_y \cdot 8 \text{ Bytes} \end{aligned}$$

Bedingung: $M(\texttt{rawdata1}) + M(\texttt{rawdata2}) + M(\texttt{u\_0}) \leq 2.092.957.696$
Lösung: $max_x = max_y = max_z = 190$

Aufgabe 6 – Große Würfel

Würfel noch größer $\rightarrow$ passt nicht mehr in VRAM

Überblick – Projekt 1

- OpenMP
- Testtools (Intel Thread Checker, OpenMP Profiler)
- Quantitative Maßzahlen im Kontext von Parallelrechner
- Vorteile und Gefahren der Parallelisierung
- Verschiedene Architekturen und ihre Unterschiede
- Gauß-Seidel-Verfahren
- Partielle Differenzialgleichungen
- Methode der Finiten Elemente
- Krylow-Unterraumverfahren
- CG-Verfahren

Überblick – Projekt 2

- CUDA – GPU Architektur
- Datentransfer zwischen Host und Device
- Verschiedene Parallelisierungsmethoden
- Gelerntes auf CPU-Architekturen übertragen
- Vorkonditionierung
- ILU-Zerlegung
- Kondition und Stabilität

Fazit

Mit *OpenMP* und *CUDA* lässt es sich parallelisieren.