SIMT/GPGPU - CUDA & OpenCL

Tobias Schiffmann

Betreuerin: Gregor Daiß

Zusammenfassung Ein schöner Abstract. Das ist einfach die Kurzzusammenfassung.

1 Gliederung

- Motivation / kurzer einstieg in motivation für GPUs 1 entworfen für Grafikanwendungen mit sehr großen Datenmengen –> im Grafikbereich: hohes Potential an Datenparallelität! –> keine Abhängigkeiten ==> heutiges Beispiel NNs 1 sehr hoher FP-Operation durchsatz, verteilt auf große Anzahl Threads 1 Damals noch mit komplexer Programmierumgebung (DirectX / OpenGL!! G!) - - State of Technology - SIMT –> Unterschied zu SIMD 2 introduced here !!!

relevant ? : 1 Transferoperationen brauchen lange, um Wartezeit zu verdecken, einzelne Threads der Instruktion ausführen 1 SIMD Thread-Scheduler wählt welcher thread als nächstes ausgeführt wird -> hat score board (Position der Ausführung / operanden in Register)

- GPU - Architektur(en) –> Basics des Aufbaus –> nur die neuste Architektur –> was verwenden Unterschiedliche Hersteller 1 multi-threaded SIMD-Prozessoren (NVIDEA: Streaming Multiprocessors [SMX]), können als unabhängige MIMD-Kerne betrachtet werden 1 jeder SIMD Prozessor hat mehrere SIMD-Funktionseinheiten(jede hat Int und FP - Einheit) (1 Fermi und Kepler architecture) 3 Turing architecture !

- Programmier Frameworks (CUDA / OpenCL) –> Unterschiede –> Beispiele 1 beide: Program Separierung in Host-Program(CPU [IO + User]) und Device-Program (GPU) 1 sollen GPU Programmierung vereinfachen 1 Interaktion: Host-Programm kopiert Daten in GPU Speicher und Device Funktionen aufrufen - CUDA 1 NVIDIA 1 device program = kernel-functions/kernels 1 Program startet mit host-Programm bis Kernel-Funktionsaufruf -> Kernel und Host laufen parallel - Aufruf erzeugt CUDA-Threads (zusammengefasst als Grid) CUDA ist nutzbar in verschiedenen PLs (C/C++/Fortran/Python) 1 erklärungen für C sind hier drin 1 Kernelaufruf enthält *execution configuration* –> gibt die organisation der generierten Threads im Grid der Kernel-Funktion an - unterteile Threads des Grids in Blöcke von Threads (3-dim, blockIDx.xyz) - Blöcke sind in threads aufgeteilt (3-dim, threadIDx.xyz) –> jeder einzelne Thread des Grids kann aufgerufen werden eine Kernel-Funktion auszuführen - um Threads zu synchronisieren gibt es synchronisationsfunktionen nur innerhalb eines Blocks 1 verschiedene Speichertypen - global memory(host-RW, Device-RW) - constant memory (host-RW, device-R) - register(nur thread kann drauf zu greifen) and shared memory(alle threads eines Blocks) (kurze Zugriffszeit) –> Schaubild !! (1) 1 Thread Scheduling - passiert nicht auf der Ebene einzelner Threads, sondern "Warps-

-> mehrere Threads eines Blocks (aufsteigende threadIDx) zusammengefasst - eine Instruktion nach der anderen für alle Threads eines Warps ausführen –> SIMT

- OpenCL 1 Mehrere partner (u.a. NVIDIA) 1 standardisiertes Programmiermodel - Vergleich von CUDA, OpenCL und (MIMD oder Sequentiel) - Beispiel für hohe Datenparallelität 1 hier werden einige Beispiele gennant - Beispiel mit geringer Datenparallelität –> Schaubilder für execution time aller - Conclusion / Discussion

1. [RR12] (1)
2. [LNOM08] (2)
3. [Bur20] (3)

2 Einleitung

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

2.1 Anmerkungen zur Einleitung

Hier kommt noch mehr Text. Wir verweisen dazu auf [Ich08].

Eine schöne Formel ist

$$u(\vec{x}) = \sum_{i=1}^{N} \alpha_i \varphi_i(\vec{x}) \,,$$

aber das geht auch inline als $u(\vec{x}) = \sum_{i=1}^{N} \alpha_i \varphi_i(\vec{x})$, also mitten im Text.

Was noch fehlt ist ein Bild, z.B. das aus Abbildung 1 oder Abbildung 2. Wir können dazu prima die tollen Makros, die oben im Vorspann definiert wurden, verwenden. Beispielsweise mit folgenden Befehlen:

```
\bild{figures/grid_l2_brd}{fig:grid1}{Dies ist ein sogenanntes dünnes
Gitter zum Level 2.}{Die Kurzform lasse ich meist leer}
\bildbreite{figures/grid_l2_brd_B}{2cm}{fig:grid2}{Dies ist ein sogenanntes dünnes
Gitter zum Level 2 in 2cm Breite.}{}
```

Die Bilder werden automatisch nach vernünftigen Kriterien platziert, daher immer im Text mit `\ref{}` drauf verweisen (bei den Beispielen mit `\ref{fig:grid1}` und `\ref{fig:grid2}`).

Was wir hin und wieder noch brauchen ist eine Tabelle, wie z.B. Tabelle 1.

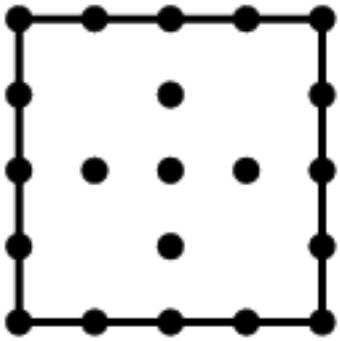

Abbildung 1. Dies ist ein sogenanntes dünnes Gitter zum Level 2.

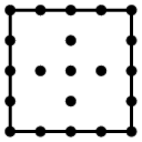

Abbildung 2. Dies ist ein sogenanntes dünnes Gitter zum Level 2 in 2cm Breite.

Tabelle 1. Diese Tabelle zeigt nicht die Daten von etwas Sinnvollem, sondern einfach irgend etwas. Tabellenbeschriftungen sind oft drüber.

Spalten			Absatz 5cm
linksbündig	rechtsbündig	zentriert	
1.0	-1.1	1.2	toller Text, der nach 5cm umbricht und dafür brauchen wir einfach mehr Text.
4321.1	6543.2	7654.3	mehr Text
2.44	4.66	6.88	8.00

2.2 Quellcode

Code-Beispiele können mittels `lstlisting`-Environment eingebunden werden. Siehe Listing 1 als Beispiel. Alternativen wie `minted` sind selbstverständlich auch erlaubt, solange sie Features wie Syntax-Highlighting und Zeilennummern mitbringen. Code-Beispiele sollten minimal sein, d.h. auf den Punkt gebracht und keinen überflüssigen Code beinhalten. Es muss standardkonformer Code sein und mit hinzugefügtem Boilerplate-Code (main, Auslassungen von Überflüssigem, ...) ohne Fehler compilierbar sein.

Quellcode aus Dateien kann per `lstinputlisting` einbezogen werden. Für Inline-Code `lstinline` verwenden. Für abstrakte Algorithmen (kein C++-Code) besser eines der algorithm-Packages verwenden.

Listing 1. Example using Lstlisting

```
1 template <typename T>
2 struct LessThan {
3   bool operator(T a, T b) { return a < b; };
4 };
5
6 std::vector<int> v = { 5, 4, 3, 2, 1 };
7 std::sort(v.begin(), v.end(), LessThan<int>());
```

2.3 Zum Schluss

... viel Spaß!

Literatur

Bur20. Burgess, John: RTX on—The NVIDIA Turing GPU. In: *IEEE Micro* 40 (2020), Nr. 2, S. 36–44. http://dx.doi.org/10.1109/MM.2020.2971677. – DOI 10.1109/MM.2020.2971677. – ISSN 0272–1732

Ich08. Ich: Vorlage für das Hauptseminar. In: *Diese Zeitschrift* (2008). – Dieses Dokument solle sich selbst verlinken (Hilfe, Endlosrekursion!) **./Ausarbeitung_Vorlage.pdf**

LNOM08. Lindholm, Erik ; Nickolls, John ; Oberman, Stuart ; Montrym, John: NVIDIA Tesla: A Unified Graphics and Computing Architecture. In: *IEEE Micro* 28 (2008), Nr. 2, S. 39–55. http://dx.doi.org/10.1109/MM.2008.31. – DOI 10.1109/MM.2008.31. – ISSN 0272–1732

RR12. Rauber, Thomas ; Rünger, Gudula: *Parallele Programmierung*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. http://dx.doi.org/10.1007/978-3-642-13604-7. http://dx.doi.org/10.1007/978-3-642-13604-7. – ISBN 978–3–642–13603–0