

Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-04-10

Struktur der Veranstaltung

I. Allgemeine Grundlagen

- Vorlesung / Übung
- Grundlegende parallele Algorithmen
- OpenCL

II. Spezielle Anwendungen

- Projekte
- Individuelle Themen in kleinen Gruppen
- Unabhängig voneinander
- Dafür Scheinerwerb, 9 ECTS

Organisation

➤ Vorlesung + Übung

- Mittwochs, 10:15 Uhr, 31/449a
- Gewichtung nach Bedarf
- Ausgabe / Vorbesprechung Übungsblatt: Nach Vorlesung
- Nachbesprechung: Termin darauf

➤ Mailingliste: pa13@list.serv.uni-osnabrueck.de

➤ Projekte

- Dauer: 3 Wochen
- Termin: Nach Abstimmung
- Alternativ(e) Termin(e): Möglich, aber weniger Betreuung
- Gemeinsame Abschlusspräsentation, 10 min p. P.

Testate

- Voraussetzung für Teilnahme an Projekten
- Übungsblätter sind in zweier Teams zu bearbeiten
- Dauer: 15 Minuten
- Registrieren für Testatverwaltung
- Mo, Di in Raum: 31/145
- Im Zweifel werden Einzelpersonen bewertet
- Mindestens 50% der Punkte sind pro Blatt zu erreichen
- Ein Joker

Begleitmaterial

- Videomitschnitt
 - Mp4
 - Flash
- Audiomitschnitt im mp3-Format
- Kein Skript
- Folien als PDF, ausführlicher als heute
- Literatur: Gleich...

<http://www-lehre.inf.uos.de/~pa>

Teil I: Grundlegende Parallele Algorithmen

Vorlesung + Übung

OpenCL

➤ **Open Compute Language:** API für einheitliche parallele Programmierung heterogener Systeme

- GPUs,
- CPUs,
- APUs,
- FPGAs
- ...
- Oder Kombinationen daraus



➤ Prozedural

➤ Initiiert 2008-12 durch Apple

➤ Danach betreut durch **KHRONOS**
GROUP

➤ Plattform, Betriebssystem & Sprachunabhängig

Beispiel: Vektoraddition

➤ Gegeben: $\mathbf{a} \in \mathbb{Z}^3$ $\mathbf{b} \in \mathbb{Z}^3$

➤ Gesucht: $\mathbf{c} \in \mathbb{Z}^3$, mit: $\mathbf{c} = \mathbf{a} + \mathbf{b}$

➤ Ergebnis:
$$\mathbf{c} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{pmatrix} = \begin{pmatrix} a_0 + b_0 \\ a_1 + b_1 \\ a_2 + b_2 \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

➤ Oder: $c_i = a_i + b_i, \text{ mit } i \in \{0, 1, 2\}$

Vektoraddition in Java

```
int dim = 3;  
int[] a = {5, 7, 9}, b = {2, 1, 1}, c = new int[dim];
```

Variante 1:

```
c[0] = a[0] + b[0];  
c[1] = a[1] + b[1];  
c[2] = a[2] + b[2];
```

Variante 2:

```
for(int i = 0; i < dim; i++){  
    c[i] = a[i] + b[i];  
}
```

Variante 3:

```
for(int i = dim-1; i >= 0; i--){  
    c[i] = a[i] + b[i];  
}
```

Variante 4:

```
c[2] = a[2] + b[2];  
c[0] = a[0] + b[0];  
c[1] = a[1] + b[1];
```

Parallele Vektoraddition

In : Vector a, b

Out: Vector c

```
For each ( $i \mid i \in \{0, \dots, \text{dim} - 1\}$ ) in parallel do  
  | c[i]  $\leftarrow$  a[i] + b[i]  
end
```

Pseudocode

In : Vector a, b

Out: Vector c

For each ($i \mid i \in \{0, \dots, \text{dim} - 1\}$) *in parallel do*

```
kernel void vec_add(  
  global int* a,  
  global int* b,  
  global int* c  
) {  
  int i = get_global_id(0);  
  c[i] = a[i] + b[i];  
}
```

OpenCL C

end

Beispiel: Scan, Inclusive, +

- In: Array mit Zahlen
- Out: Array mit Summe bis zum eigenen Index
- Beispiel:

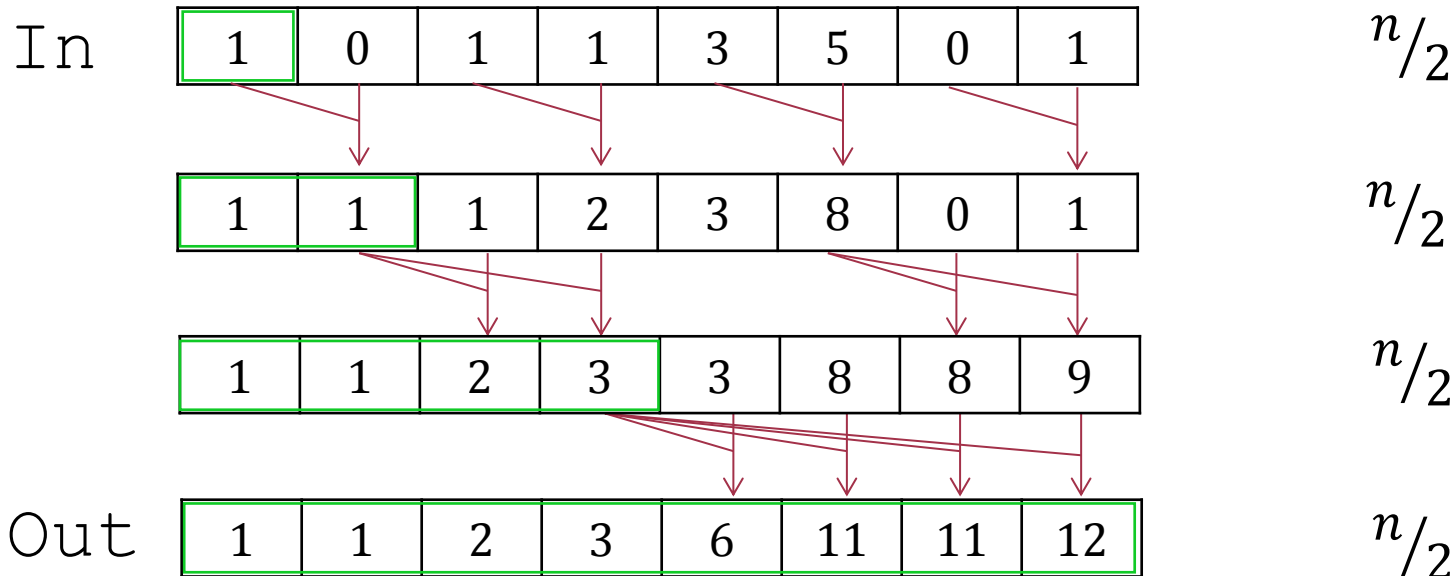
In	1	0	1	1	3	5	0	1
----	---	---	---	---	---	---	---	---

Out	1	1	2	3	6	11	11	12
-----	---	---	---	---	---	----	----	----

- Berechne: $\text{Out}[i] \leftarrow \text{Out}[i-1] + \text{In}[i]$
- Zur Berechnung aller $\text{Out}[i]$ ist $\text{Out}[i-1]$ nötig
- Algorithmus zwingend sequentiell?

Ein paralleler Ansatz







Berechnungen



- Laufzeit sequentieller Algorithmus: $O(n)$
- Diese parallele Variante
 - Pro Iteration: $O(n)$
 - Anzahl Iterationen: $\log_2 n$
 - Gesamtaufwand: $O(n \cdot \log n)$
 - Nicht *work-efficient*
 - Skalierbar







Reduction

- Gegeben: Array mit dynamisch entstehenden / verschwindenden Einträgen
- Etwa: Partikelsystem

															
1	1	0	0	0	0	1	0	0	0	0	1	1	0	1	0



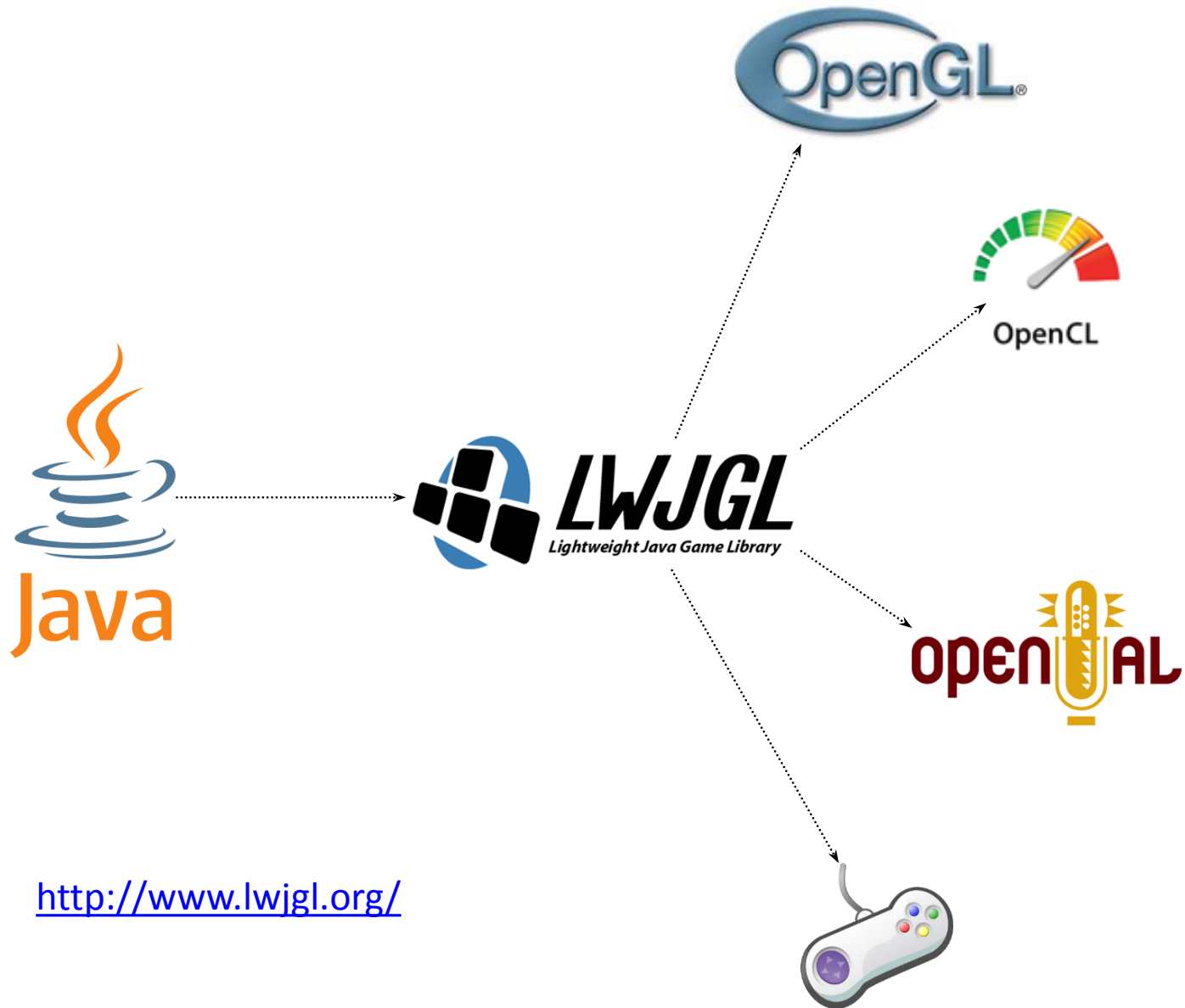
Exclusive Scan, +

0	1	2	2	2	2	2	3	3	3	3	3	4	5	5	6
															

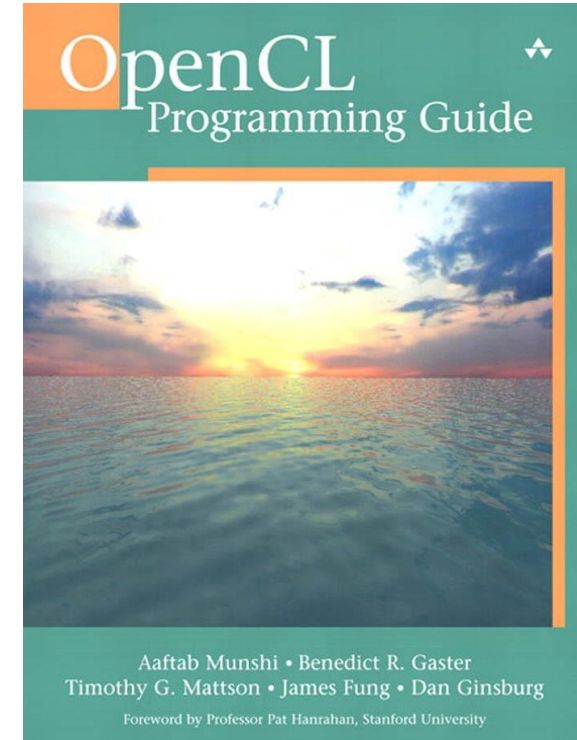
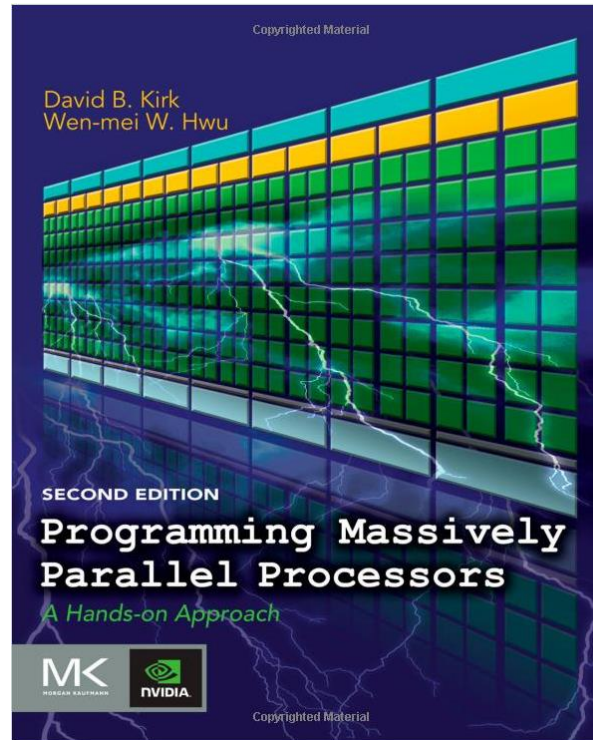
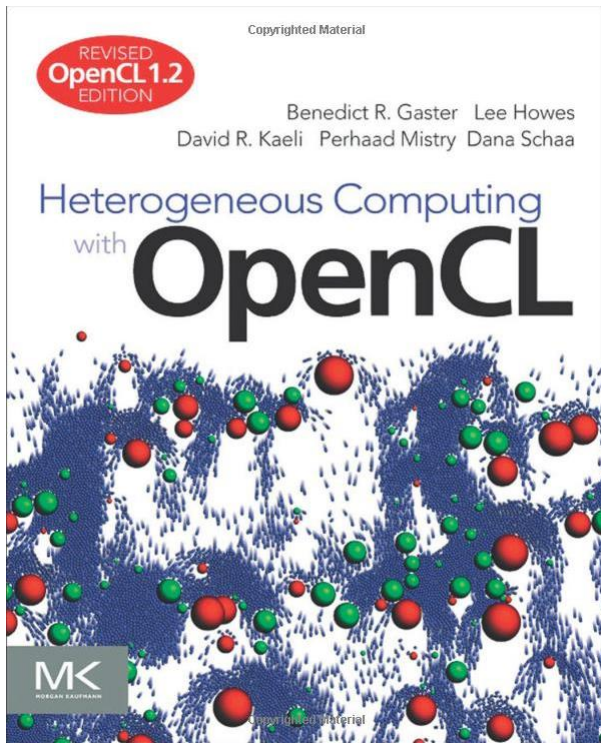
Themenüberblick - Grundlagen

- Hello OpenCL
- Konzepte der parallelen Programmierung
- Parallele Algorithmen & OpenCL
- Optimierung
- Dynamic Parallelism*
- Distributed Memory*
- Genauigkeit*

Java + LWJGL



Literatur



Safari Ebooks (Zugriff aus Uni-Netz):

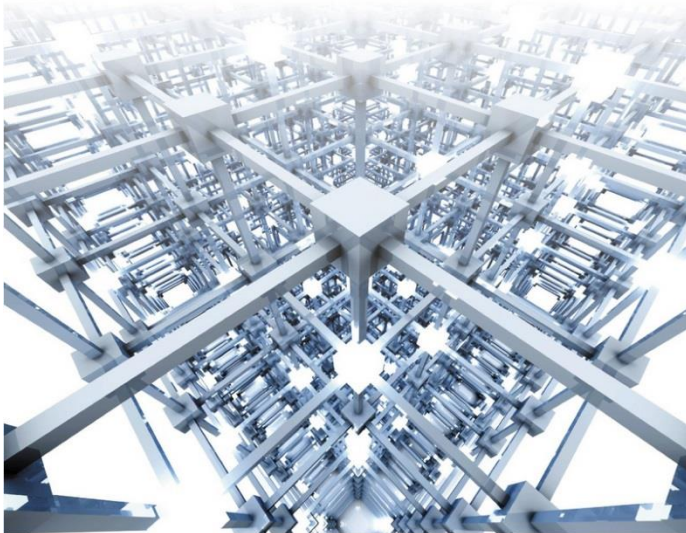
http://rzblx10.uni-regensburg.de/dbinfo/detail.php?bib_id=ubos&colors=&ocolors=&titel_id=3415

Literatur II

ALGORITHMS SEQUENTIAL & PARALLEL A Unified Approach

Third Edition

Russ Miller
Laurence Boxer



Urheberrechtlich geschütztes Material
Thomas Rauber
Gudula Rünger

Parallel Programming

for Multicore and Cluster Systems

 Springer

Urheberrechtlich geschütztes Material

Teil II: Spezielle Anwendungen

Projekte

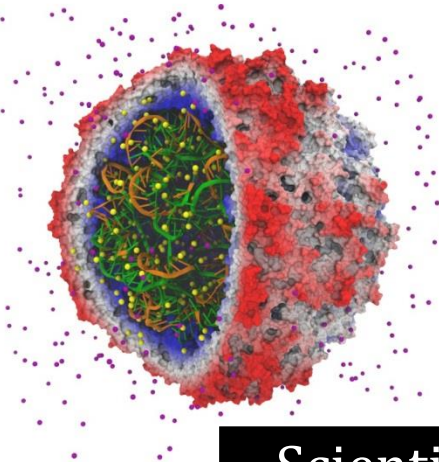
Vorschläge für Themengebiete

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -it_1 & 1 & 0 & -r_1 & 0 & 0 & 0 \\ -r_1 & 0 & 1 & -it_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -e^{-ikD} \\ 0 & -e^{-ikD} & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -it_2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -r_2 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_4 \\ a'_3 \\ a'_1 \\ a_2 \\ a_3 \end{pmatrix}$$

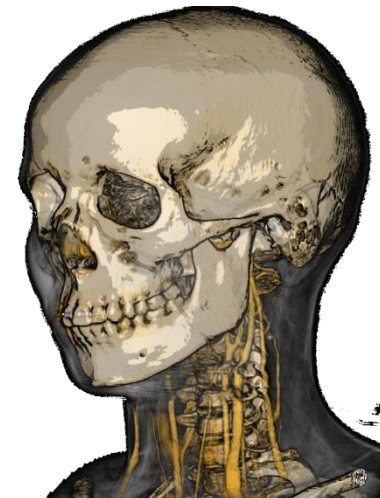
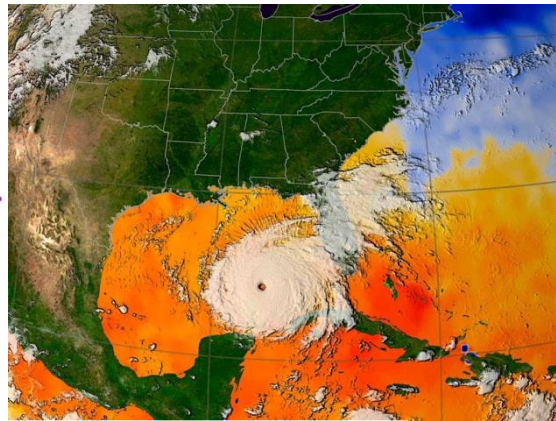
Numerik



Computer Vision

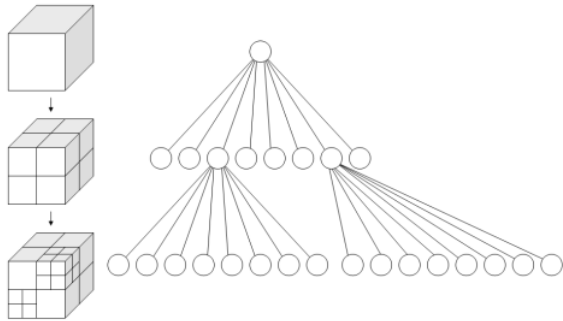


Scientific Simulation

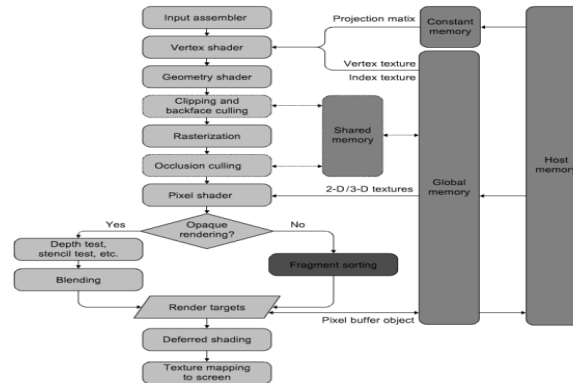


Scientific Visualization

Vorschläge für Themengebiete II



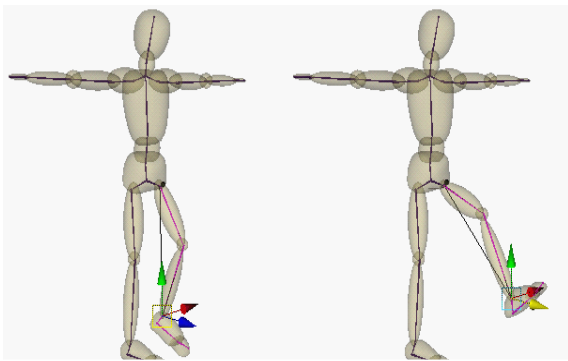
RealTime Ray Tracing



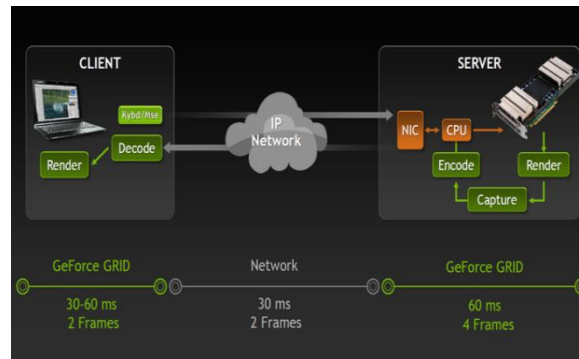
Programmable Graphics



„Physik für Spiele“



Animation



Remote Rendering



Eigene Vorschläge

Literatur III

Graphics Hardware (2007)
M. Monner, R.-D. Schoder (Editors)

KD-Tree Acceleration Structures for a GPU Raytracer

Tim Foley and Jeremy Sugerman¹
Stanford University

Abstract
Modern graphics hardware architectures excel at computer-intensive tasks such as ray-triangle intersection. However, they are not well suited for ray-triangle intersection in CPU-based ray-tracing applications. In contrast, the latter has gained widespread use in CPU-based ray-tracing applications. We present a novel GPU-based ray-tracing architecture that is well suited for GPU implementation and compare it to a ray-tracing architecture that is well suited for CPU implementation. Our hybrid architecture is up to 8 times faster than the CPU-based architecture and is well suited for GPU implementation.

Graphics Hardware (2007)
M. Monner, R.-D. Schoder (Editors)

Fully Procedural Graphics

T. N. Pong and J. Koppa
Microsoft Research, Redmond, WA

Abstract
The process of generating graphics data is often a complex task. This paper describes a fully procedural graphics system that generates graphics data in a highly efficient manner.

Parallel Graph Component Labeling with GPUs and CUDA

K.A. Hordick*, A. Liot and D.P. Pagan
Institute of Information and Mathematical Sciences
Massey University - Albany, North Shore 107 080, Auckland, New Zealand
Email: {k.a.hordick, a.liot, d.p.pagan}@massey.ac.nz
Tel: +64 9 832 3400 Fax: +64 9 832 3181

Abstract
Graph component labeling, which is a subset of the general graph coloring problem, is a computationally intensive operation that is of importance in many applications and simulations. A number of data-parallel algorithms have been proposed for the component labeling problem on GPUs using CUDA. We discuss the performance of these algorithms and compare them with the performance of a sequential algorithm. We also discuss the performance of a novel algorithm that is based on the use of a hierarchical graph structure. The results show that the novel algorithm is well suited for GPU implementation and can be used to solve large-scale graph component labeling problems.

Key words: graph, component label, mesh, GPU, CUDA, data-parallel

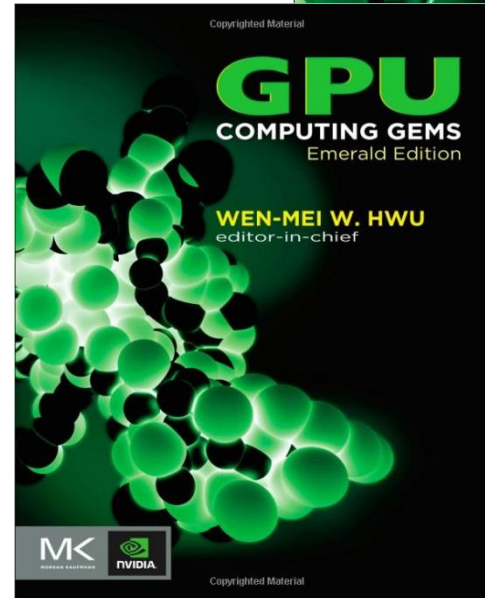
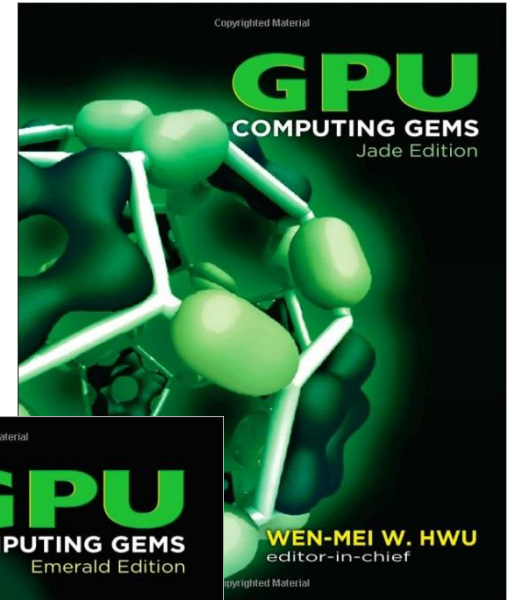
1. Introduction

Graph component labeling is an important algorithmic problem that arises in many applications. Generally, the problem involves identifying which nodes in a graph belong to the same connected cluster or component. In many applications, the graph of interest has a hierarchical structure. For example, the graph may be a hierarchical tree structure, or it may be a graph that is composed of many smaller graphs. In such cases, the graph component labeling problem can be solved by using a hierarchical approach. The results show that the novel algorithm is well suited for GPU implementation and can be used to solve large-scale graph component labeling problems.

Figure 2 illustrates how component labeling can be used to analyze the results of a simulation. The figure shows a Keweenaw spin exchange long model simulation [1] where a mean-field mixture of spin (colored dark blue and white) has been quenched to a low temperature and clusters or components have formed in the model system. Models such as these exhibit phase transitions and the component size distribution can be used to examine and quantify the location and properties of the resulting phase transitions.

¹ Author for correspondence
Presented at Parallel Computing

April 10, 2010



Das Team

Sascha Kolodzey, skolodze@uos.de

Übungsleiter

Sascha,

Nils Vollmer, nvollmer@uni-osnabrueck.de

Tutoren

Henning Wenke, Raum 31/323, hewenke@uos.de

Dozent