TECHNISCHE UNIVERSITÄT DRESDEN

Physikalische Berechnungen mit General Purpose Graphics Processing Units (GPGPUs)

im Rahmen des Proseminars
„Technische Informatik“
Juni 2010
von
Hartmut Schweizer

Dresden, den 02.06.2010

01 Gliederung

- Einleitung/Begriffserklärung
- Motivation
- Entwicklung
- Hardware
- Programmierung
- Anwendungsbeispiele
- Zusammenfassung/Ausblick

02 Einleitung

Was versteht man unter einer GPGPU

- Entwicklung im Bereich der Grafik-Hardware
- Ausführung von Algorithmen für gewöhnliche (nicht das Rendering betreffende) Probleme auf Grund der höheren Geschwindigkeit auf der GPU anstatt auf der CPU
- Beschränkung der GPU auf spezielle Probleme ohne großen Verwaltungsaufwand -> Großteil der Transistoren für Rechenoperationen verwenden (Keine Steuerungsaufgaben und Caching)

-> optimale Leistung bei GPGPU-Anwendungen, die hohe arithmetische Dichte aufweisen (Algorithmen mit verhältnismäßig vielen Rechenoperationen und wenigen Lese-/Schreiboperationen) [1]

03 Motivation

- Wachsende Ansprüche an herkömmliche PCs im Bereich von Anwendungen mit hoher Datenparallelität und Streamcomputing wie z. B.
  - Physikalische Simulationen
  - Physikeffekte in Computerspielen
  - Kryptographie
  - Video-Encodierung
- Ausweitung der Nutzung von Grafikkarten als vorhandene, kostengünstige Ressource über bisherige Aufgaben hinaus

04 Entwicklung

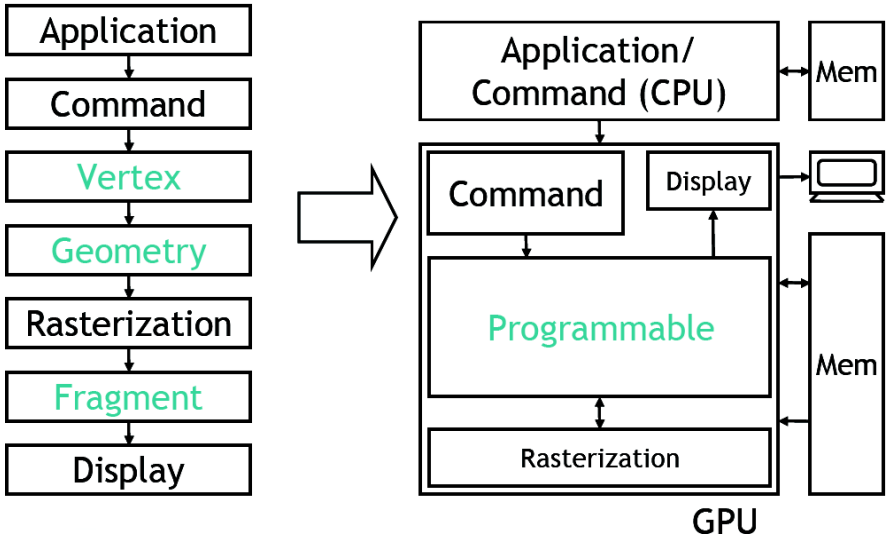
PhysX[2]

- Ageia Technologies, Inc.
  - US-amerikanisches Startup-Unternehmen
  - 2002 gegründet
- spezieller Prozessor (Physikbeschleuniger/Physics Processing Unit (PPU)
- zugehörige Schnittstelle (Physik-Engine) PhysX API (vormals NovodeX)
- 2008 von Nvidia übernommen

  -> Integration der PhysX-Engine in hauseigenes CUDA-System
- damit Nutzung von PhysX auch von der (Nvidia-)Grafikkarte möglich (wenn CUDA-unterstützt)

04 Entwicklung

- Erweiterung des klassischen Ablaufs des Grafik-Renderings
- Alternative zur Abfolge von Transformation der Objekt-3D-Koordinaten in Bildschirmkoordinaten/ Beleuchtungsberechnung/ Rasterisation/Texturierung mittels fester, in Hardware gegossener Routinen



[3]

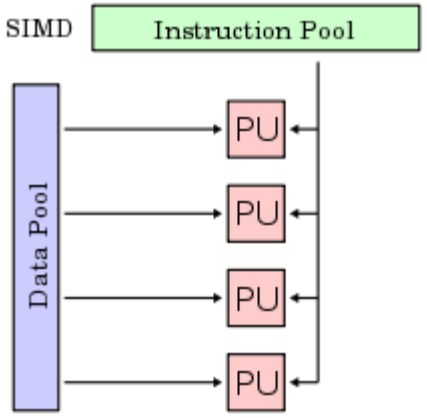
-> für jeden Vertex (Eckpunkt eines Dreiecks) und Pixel Ausführung von definierten kleinen Programmen

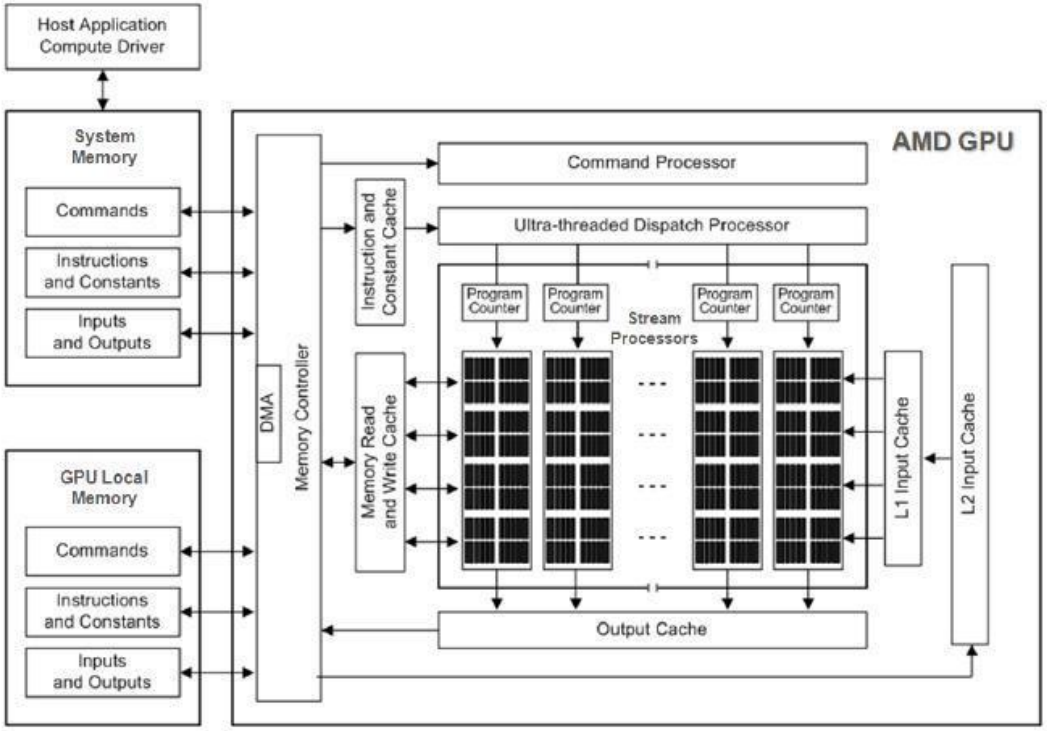
-> diese übernehmen Aufgabe und erzeugen aufwändige Effekte (Vertex- bzw. Pixelshader oder einfach Shader)

05 Hardware

Vereinfachter Aufbau einer aktuellen GPU

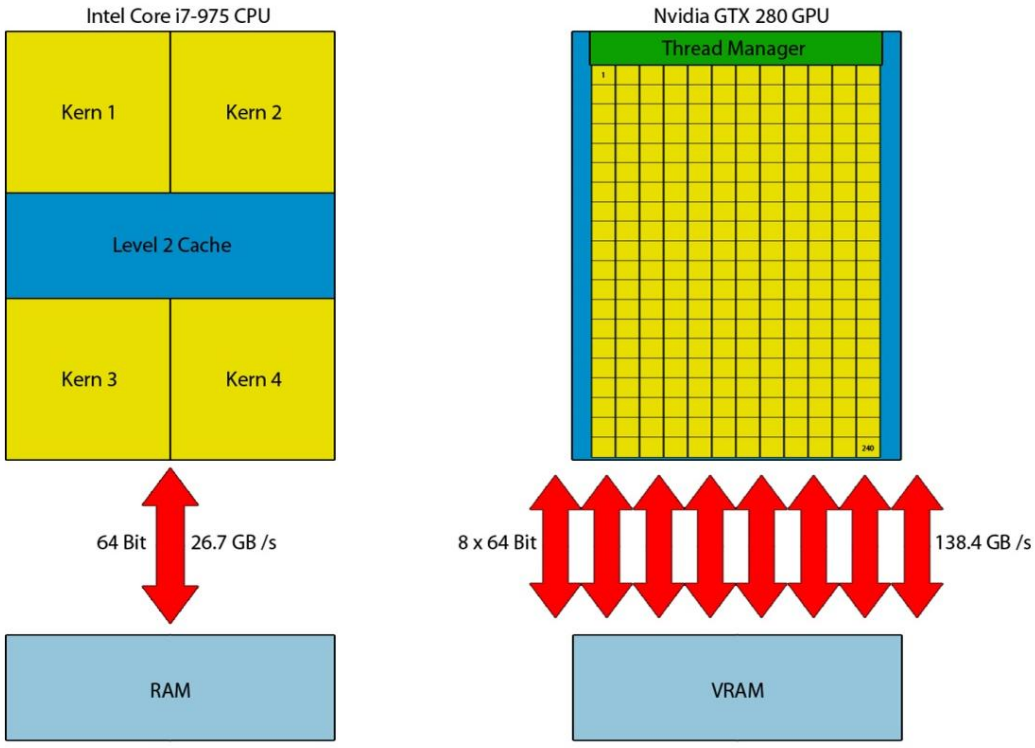
SIMD (Single Instruction, Multiple Data)





05 Hardware

Vergleich CPU/GPU (1)



05 Hardware

Vergleich CPU/GPU (2)

	Rechenleistung	Speicherbus-Datenrate
ATI RV770	1200 GFlops	n/a
NVIDIA GeForce GT200	1063 GFlops	25,3 Gbyte/s
NVIDIA GeForce 6800	60 GFlops	18 GByte/s
Intel Core 2 Quad Q6600	21,4 GFlops	n/a
Intel Pentium 4 mit SSE3, 3,6 GHz	14,4 GFlops	5 GByte/s

06 Programmierung

Programmiersprachen/-Konzepte

- Hardwareabhängig
  - AMD Stream Computing
  - CUDA (Nvidia)

- Hardwareunabhängig
  - DirectX Compute Shader (DirectX 11 Microsoft)
  - OpenCL (Khronos Group)

06 Programmierung

AMD Stream Computing

- aktuelle Version: 2.1
- unterstützte Betriebssysteme:
  - Windows XP SP3(32-bit) SP2(64-bit)
  - Windows Vista SP2 (32-bit/64-bit)
  - Windows 7 (32-bit/64-bit)
  - openSUSE™ 11.2 (32-bit/64-bit)
  - Ubuntu® 9.10 (32-bit/64-bit)
  - Red Hat® Enterprise Linux® 5.4 (32-bit/64-bit)
- benötigt:
  - ATI Grafikkarte
    - ATI RadeonHD 4350 oder höher (Mobility 4300)
    - ATI FirePro V3750 oder höher (Mobility M5800)
    - ATI FireStream (Konkurrenzprodukt zu nVideaTeslar)
  - aktueller Grafikkartentreiber
  - Compiler (MSVS 2008 /GCC 4.3/ICC 11x)

06 Programmierung

CUDA (Compute Unified Device Architecture)

- aktuelle Version: 3.0
- unterstützte Betriebssysteme:
  - Windows XP (32-bit/64-bit)
  - Linux (32-bit/64-bit)
  - Windows Vista
  - Windows 7
  - Apple Mac OS
- benötigt:
  - Nvidia Grafikkarte
    - ab „GeForce 8“-Serie
    - Quadro FX
    - nVidea Teslar
  - aktueller Grafikkartentreiber
  - Compiler

06 Programmierung

DirectX Compute Shader

- aktuelle Versionen:
  - Version 4.0 (DirectX 10) und 4.1 (DirectX 10.1) mit eingeschränktem Funktionsumfang
  - Vollversion 5.0 für DirectX 11
- unterstützte Betriebssysteme:
  - Windows XP (32-bit/64-bit)
  - Windows Vista (32-bit/64-bit)
  - Windows 7 (32-bit/64-bit)
- benötigt:
  - DirectX-fähige Grafikkarte ab DirectX 10
  - aktueller Grafikkartentreiber
  - Compiler

06 Programmierung

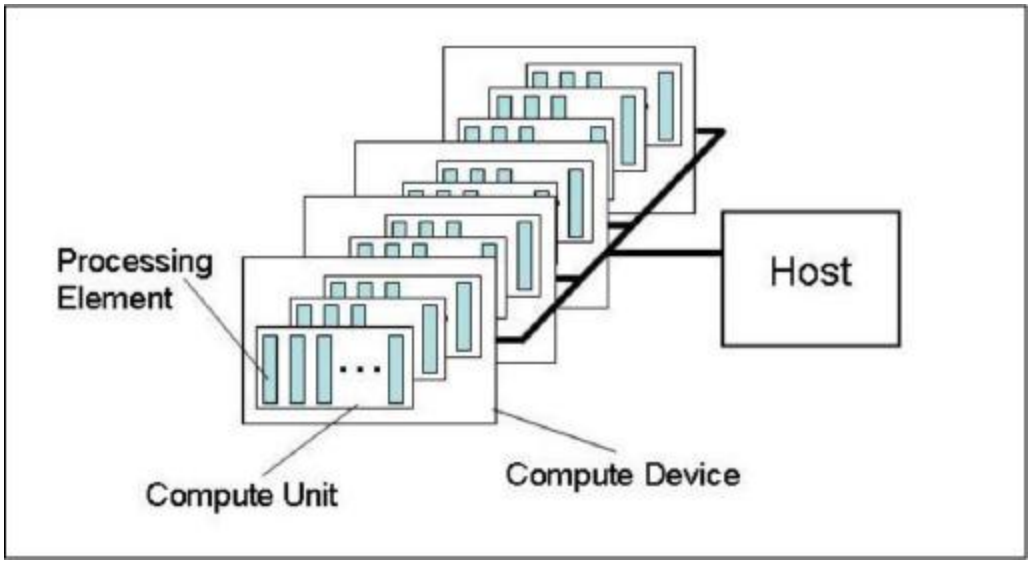
OpenCL

- aktuelle Version:
  - OpenCL Version 1.0
- unterstützte Betriebssysteme:
  - kann für beliebige Betriebssysteme implementiert werden
  - Nvidia bietet Implementierung für
    - Linux
    - Windows
    - Mac OS X
  - AMD ermöglicht Nutzung
    - über Stream für GPUs
    - über SSE3 für CPUs
    - Für Windows und Linux
  - IBM bietet Implementierung für
    - Power-Architektur
    - Cell Broadband Engine

06 Programmierung

OpenCL – Platform Model

- Host hat ein oder mehrere Compute Devices (z. B. Grafikkarte(n))
- Compute Device besteht aus mehreren Compute Units
- Compute Unit besteht mehreren Processing Units

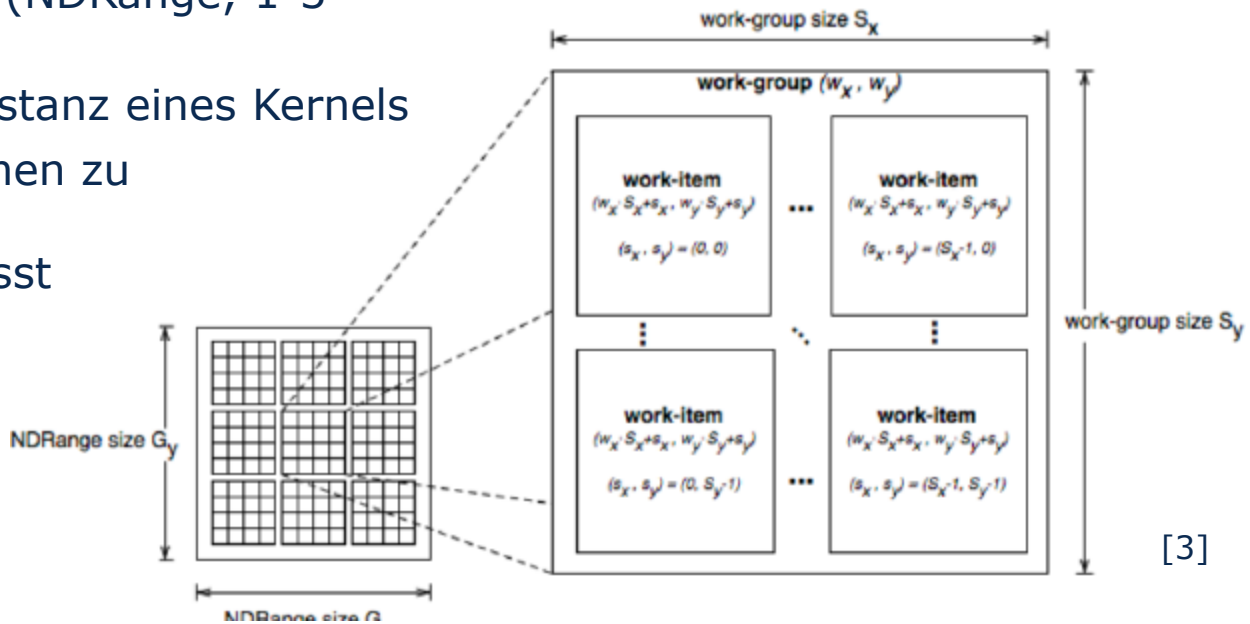


[3]

06 Programmierung

OpenCL – Execution Model

- Host-Anwendung enthält alle Kernel
- Host-Anwendung führt Kernel über Indexraum aus (NDRange, 1-3 Dimensionen)
- Workitem ist Instanz eines Kernels
- Workitems können zu Workgroups zusammengefasst werden

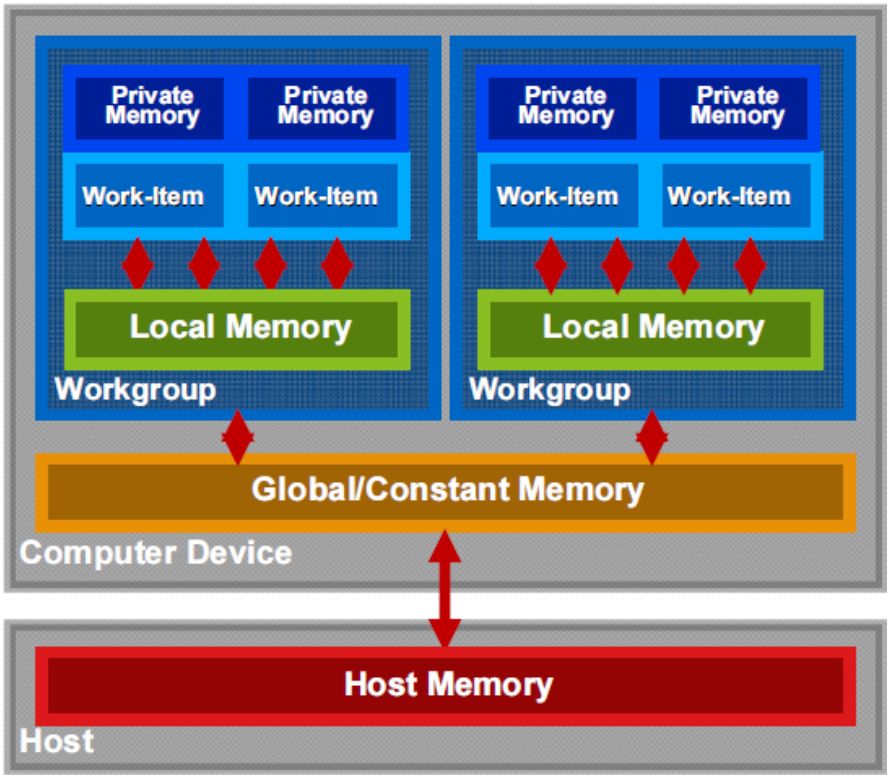


[3]

06 Programmierung

OpenCL – Memory Model

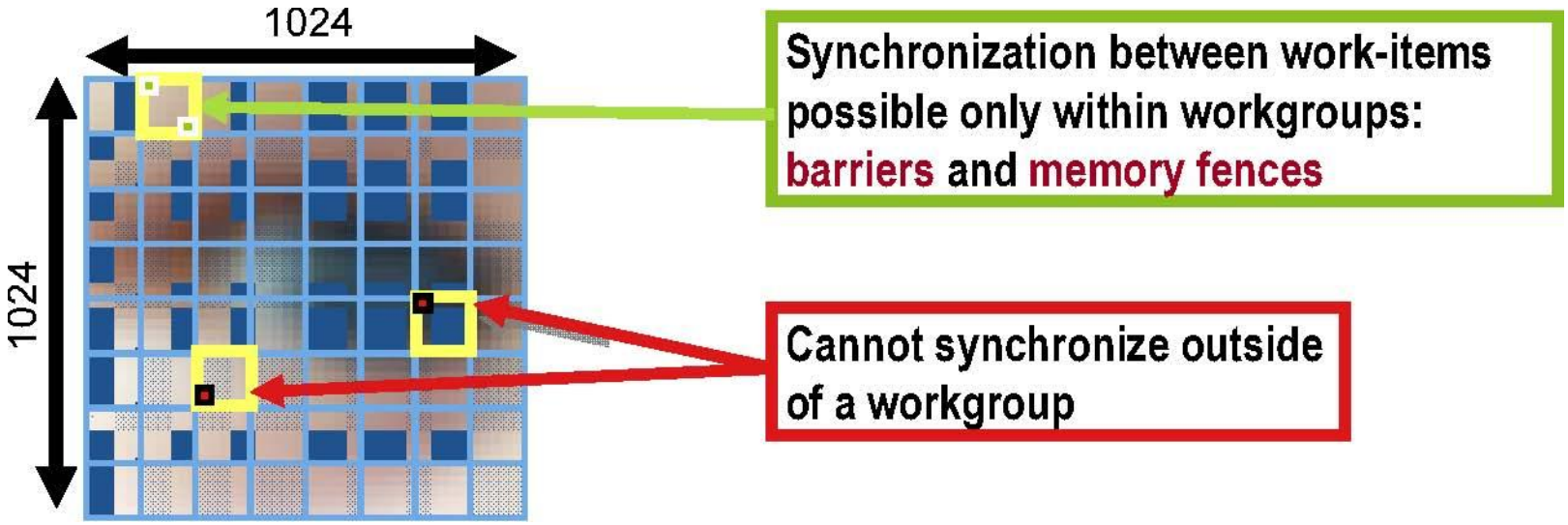
- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup (16Kb)
- **Local Global/Constant Memory**
  - Not synchronized
- **Host Memory**
  - On the CPU



[6]

06 Programmierung

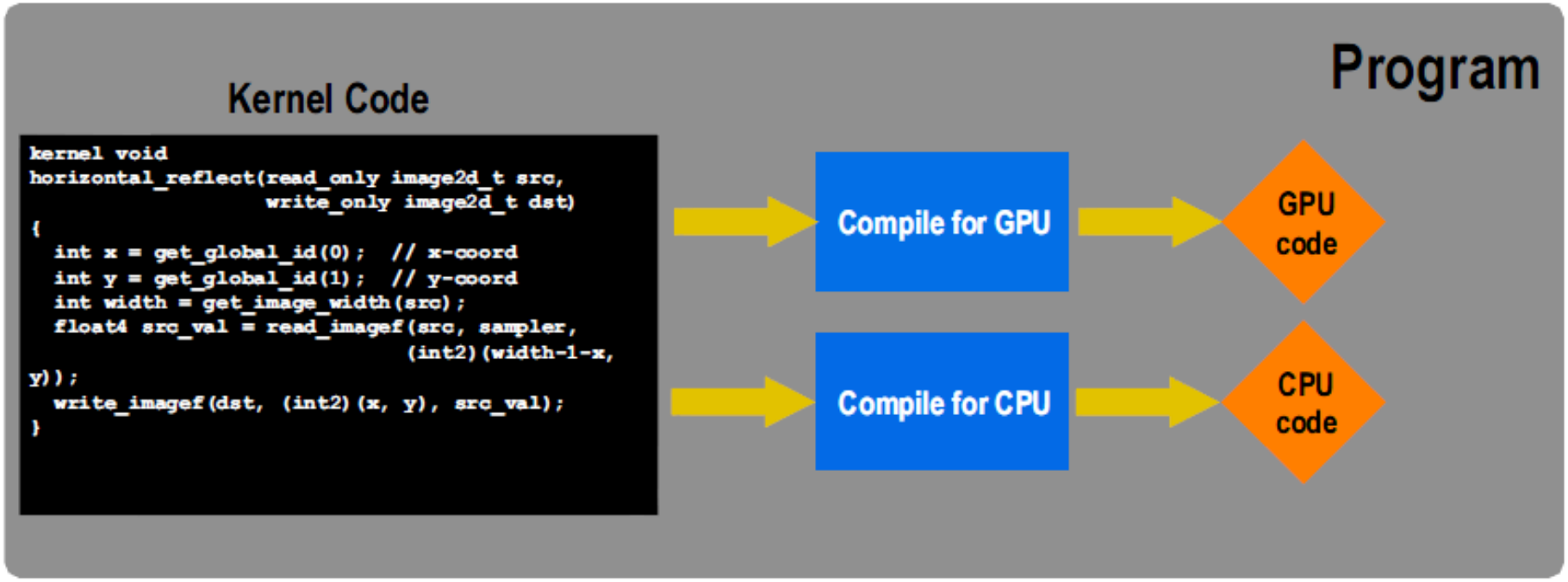
OpenCL - Synchronisation



[6]

06 Programmierung

OpenCL – Gleicher Code für unterschiedliche PUs



[6]

06 Programmierung

OpenCL – Datentypen

Supported Data Types

Built-in Scalar Data Types [6.1.1]

OpenCL Type	API Type	Description
bool	--	true (1) or false (0)
char	cl_char	8-bit signed
unsigned char, uchar	cl_uchar	8-bit unsigned
short	cl_short	16-bit signed
unsigned short, ushort	cl_ushort	16-bit unsigned
int	cl_int	32-bit signed
unsigned int, uint	cl_uint	32-bit unsigned
long	cl_long	64-bit signed
unsigned long, ulong	cl_ulong	64-bit unsigned
float	cl_float	32-bit float
half	cl_half	16-bit float (for storage only)
size_t	--	32- or 64-bit unsigned integer
ptrdiff_t	--	32- or 64-bit signed integer
intptr_t	--	signed integer
uintptr_t	--	unsigned integer
void	--	void

Built-in Vector Data Types [6.1.2]

OpenCL Type	API Type	Description
char*n*	cl_char*n*	8-bit signed
uchar*n*	cl_uchar*n*	8-bit unsigned
short*n*	cl_short*n*	16-bit signed
ushort*n*	cl_ushort*n*	16-bit unsigned
int*n*	cl_int*n*	32-bit signed
uint*n*	cl_uint*n*	32-bit unsigned
long*n*	cl_long*n*	64-bit signed
ulong*n*	cl_ulong*n*	64-bt unsigned
float*n*	cl_float*n*	32-bit float

Other Built-in Data Types [6.1.3]

OpenCL Type	Description
image2d_t	2D image handle
image3d_t	3D image handle
sampler_t	sampler handle
event_t	event handle

Reserved Data Types [6.1.4]

OpenCL Type	Description
bool*n*	boolean vector
double, double*n* OPT	64-bit float, vector
half*n* OPT	16-bit float, vector
quad, quad*n*	128-bit float, vector
complex half, complex half*n* imaginary half, imaginary half*n*	16-bit complex, vector
complex float, complex float*n* imaginary float, imaginary float*n*	32-bit complex, vector
complex double, complex double*n* imaginary double, imaginary double*n*	64-bit complex, vector
complex quad, complex quad*n* imaginary quad, imaginary quad*n*	128-bit complex, vector
float*n*x*m*	*n***m* matrix of 32-bit floats
double*n*x*m*	*n***m* matrix of 64-bit floats
long double, long double*n*	64 - 128-bit float, vector
long long, long long*n*	128-bit signed
unsigned long long, ulong long, ulong long*n*	128-bit unsigned

[5]

06 Programmierung

OpenCL – Vektor-Komponenten-Adressierung

Vector Component Addressing [6.1.7]

The components of a vector may be addressed as shown below or as shown in the table of equivalencies.

Vector Components

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
float2 v;	v.x, v.s0	v.y, v.s1														
float4 v;	v.x, v.s0	v.y, v.s1	v.z, v.s2	v.w, v.s3												
float8 v;	v.s0	v.s1	v.s2	v.s3	v.s4	v.s5	v.s6	v.s7								
float16 v;	v.s0	v.s1	v.s2	v.s3	v.s4	v.s5	v.s6	v.s7	v.s8	v.s9	v.sa, v.sA	v.sb, v.sB	v.sc, v.sC	v.sd, v.sD	v.se, v.sE	v.sf, v.sF

Vector Addressing Equivalencies

	v.lo	v.hi	v.odd	v.even
float2	v.x, v.s0	v.y, v.s1	v.y, v.s1	v.x, v.s0
float4	v.s01, v.xy	v.s23, v.zw	v.s13, v.yw	v.s02, v.xz
float8	v.s0123	v.s4567	v.s1357	v.s0246
float16	v.s01234567	v.s89abcdef	v.s13579bdf	v.s02468ace

When addressing vector components by numeric indices, they must be preceded by the letter s or S, e.g.: s1.

Swizzling, duplication, and nesting are allowed, e.g.: v.yx, v.xx, v.lo.x

[5]

06 Programmierung

OpenCL – einfaches Code-Beispiel

Scalar

```
void
scalar_mul(int n,
           const float *a,
           const float *b,
           float *result)
{
  int i;
  for (i=0; i<n; i++)
    result[i] = a[i] * b[i];
}
```

Data Parallel

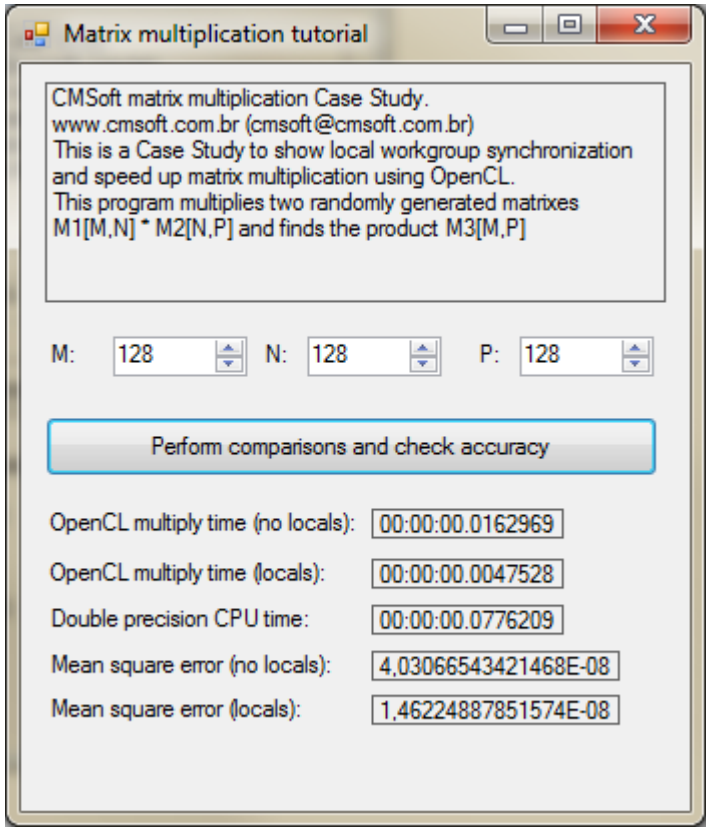
```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *result)
{
  int id = get_global_id(0);

  result[id] = a[id] * b[id];
}
// execute dp_mul over "n" work-items
```

[6]

06 Programmierung

OpenCL – Matrizen-Multiplikation[7]

- Matrizen-Multiplikation mit
  - C#-Funktion (CPU)
  - OpenCL–Kernel mit globalen IDs (GPU)
  - OpenCL–Kernel mit Unterteilung in Workgroups
- Implementiert mit
  - Visual Studio 2008
  - Windows Forms



06 Programmierung

OpenCL – Matrizen-Multiplikation ohne lokale IDs

```
#region OpenCL source for matrix multiplication
        /// <summary>Matrix multiplication. Dimensions { p, r }.
        /// </summary>
        public string matrixMultNoLocals = @„ __kernel void floatMatrixMult(   __global     float * MResp,
                                                                               __global     float * M1,
                                                                               __global     float * M2,
                                                                               __global     int * q)
{
   // Vector element index
   int i = get_global_id(0);
   int j = get_global_id(1);


   int p = get_global_size(0);
   int r = get_global_size(1);

   MResp[i + p * j] = 0;
   int QQ = q[0];
   for (int k = 0; k < QQ; k++)
   {
      MResp[i + p * j] += M1[i + p * k] * M2[k + QQ * j];
   }
}
";
```

06 Programmierung

OpenCL – Matrizen-Multiplikation mit lokalen IDs (1)

```
 public string matrixMultLocals = @"
#define BLOCK_SIZE 8

__kernel __attribute__((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE,
1))) void
floatMatrixMultLocals(__global      float * MResp,
                      __global      float * M1,
                      __global      float * M2,
                      __global      int * q)
{
  //Identification of this workgroup
  int i = get_group_id(0);
  int j = get_group_id(1);

  //Identification of work-item
  int idX = get_local_id(0);
  int idY = get_local_id(1);

  //matrixes dimensions
  int p = get_global_size(0);
  int r = get_global_size(1);
  int qq = q[0];
```

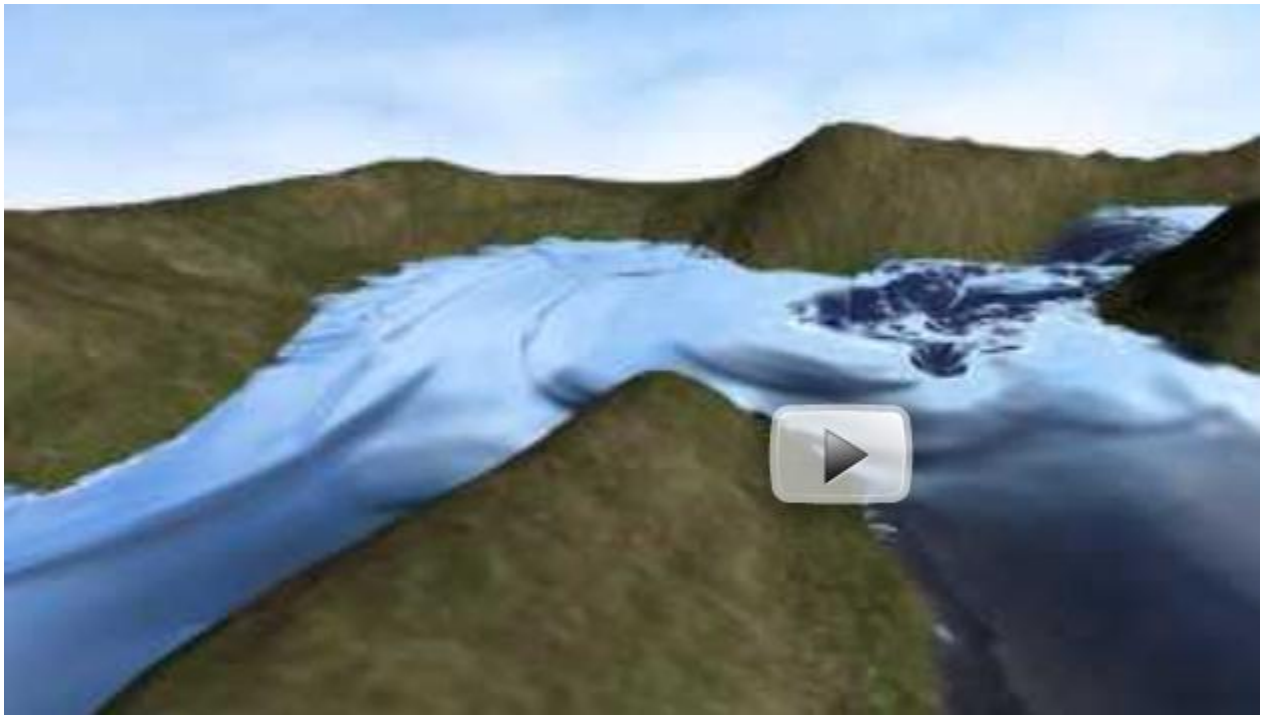
06 Programmierung

OpenCL – Matrizen-Multiplikation mit lokalen IDs (2)

```
//Number of submatrixes to be processed by each worker
  int numSubMat = qq/BLOCK_SIZE;

  float4 resp = (float4)(0,0,0,0);
  __local float A[BLOCK_SIZE][BLOCK_SIZE];
  __local float B[BLOCK_SIZE][BLOCK_SIZE];

  for (int k=0; k<numSubMat; k++)
  {
     //Copy submatrixes to local memory. Each worker copies one element
     //Notice that A[i,k] accesses elements starting from M[BLOCK_SIZE*i, BLOCK_SIZE*j]
     A[idX][idY] = M1[BLOCK_SIZE*i + idX + p*(BLOCK_SIZE*k+idY)];
     B[idX][idY] = M2[BLOCK_SIZE*k + idX + qq*(BLOCK_SIZE*j+idY)];
     barrier(CLK_LOCAL_MEM_FENCE);

     for (int k2 = 0; k2 < BLOCK_SIZE; k2+=4)
     {
        float4 temp1=(float4)(A[idX][k2],A[idX][k2+1],A[idX][k2+2],A[idX][k2+3]);
        float4 temp2=(float4)(B[k2][idY],B[k2+1][idY],B[k2+2][idY],B[k2+3][idY]);
        resp += temp1 * temp2;
     }
     barrier(CLK_LOCAL_MEM_FENCE);
  }

  MResp[BLOCK_SIZE*i + idX + p*(BLOCK_SIZE*j+idY)] = resp.x+resp.y+resp.z+resp.w;
}";
```

07 Anwendungsbeispiele (2)

Fluidsimulation



[4]

08 Zusammenfassung/Ausblick

- Kostengünstige Alternative zu Großrechnern
- Durch OpenCL und DirectX 11 wichtiger Schritt in Richtung Hardwareunabhängigkeit
- Problem: geringer Grafikkarten-Speicher

  -> spezielle GPGPU-Grafikkarten mit mehr Speicher, aber dafür ohne Bildschirmausgabe z. B.
  - Nvidia Tesla
  - AMD FireStream

  -> mit speziellen Fehlerkorrekturverfahren, da Speicherbausteine herkömmlicher Grafikkarten für Endverbraucher oft von schlechterer Qualität

Danke für Ihre Aufmerksamkeit!

»Wissen schafft Brücken.«

Literatur

[1] http://de.wikipedia.org/wiki/General_Purpose_Computation_on_Graphics_Processing_Unit

[2] http://de.wikipedia.org/wiki/Physikbeschleuniger

[3] GPU-Programmierung: OpenCL Markus Hauschild,
http://www.infosun.fim.uni-passau.de/cl/lehre/sem-ss09/HauschildHandout.pdf

[4] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie and J. R. Natvig: *"Simulation and Visualization of the Saint-Venant System using GPUs"*. In review, February 2010,
http://gpgpu.org/tag/fluid-simulation

[5] OpenCLTM API 1.0 Quick Reference Card,
http://www.khronos.org/files/opencl-quick-reference-card.pdf

[6] OpenCL PDF Overview, Khronos Group (Aug. 2009),
http://www.khronos.org/developers/library/overview/opencl_overview.pdf

[7] CMSoft: Case study: matrix multiplication
http://www.cmsoft.com.br/index.php?option=com_content&view=category&layout=blog&id=94&Itemid=145