Hochleistungsrechnen
Grafikkartenprogrammierung

Prof. Dr. Thomas Ludwig
Universität Hamburg – Informatik – Wissenschaftliches Rechnen

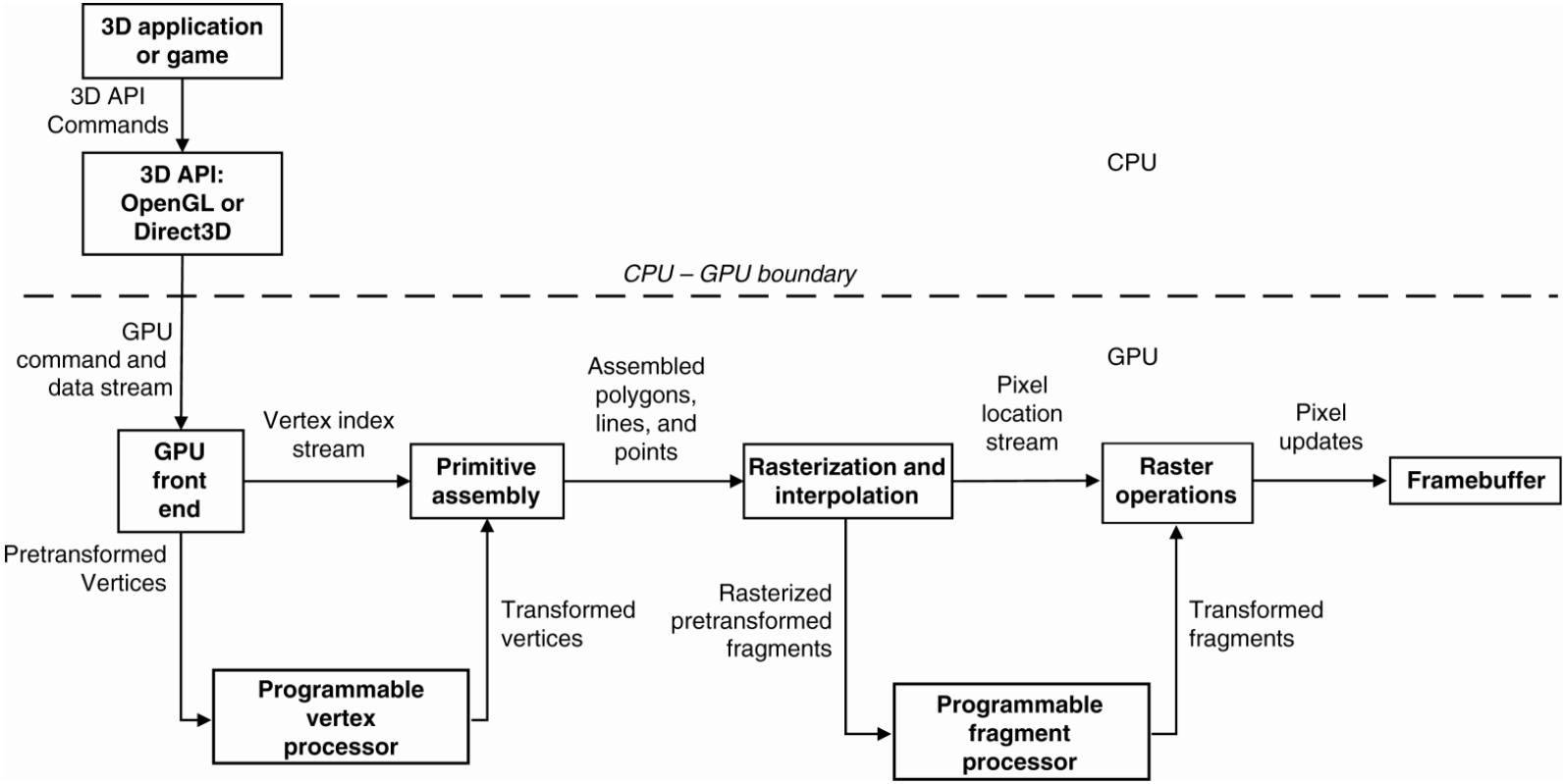
Übersicht

- Hintergrund und Entwicklung von GPGPU
- Programmierumgebungen & Werkzeuge (CUDA)
- Programmierbeispiel (Matrix-Matrix Multiplikation)
- Einsatz von GPUs im Hochleistungsrechnen
- Referenzen

viele Beispiele und Graphiken aus [4] entnommen

Hintergrund

- Bis 2000 bei Graphikkarten „fixed function pipeline“
- Transform & Lighting Engine (Geforce256, 99/00)



Anfänge von GPGPU

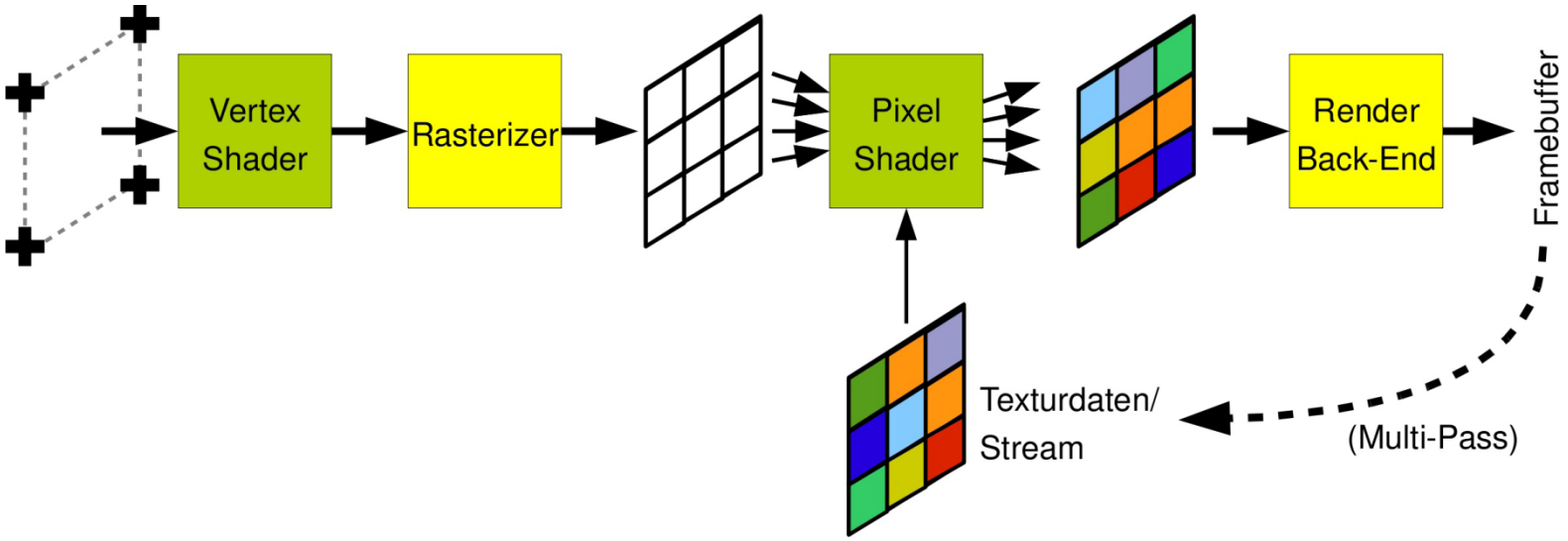
- Assembler und Register Combiner:

```
glTexEnvi(GL_Texture_Env, GL_Texture_Env_MODE, GL_Combine_Ext);
glTexEnvi(GL_Texture_Env, GL_Combine_Alpha_Ext, GL_Replace);
glTexEnvi(GL_Texture_Env, GL_Combine_RGB_Ext, GL_DOT3_RGB_Ext);
glTexEnvi(GL_Texture_Env, GL_Source0_RGB_Ext, GL_Primary_Color_Ext);
glTexEnvi(GL_Texture_Env, GL_Operand0_RGB_Ext, GL_SRC_Color);
glTexEnvi(GL_Texture_Env, GL_Source1_RGB_Ext, GL_Texture);
glTexEnvi(GL_Texture_Env, GL_Operand1_RGB_Ext, GL_SRC_Color);
```

Beispiel: Einfaches Volume Shading

General Purpose Graphics Programming

- Programmierung auf Basis der Graphik Pipeline
  - Datenarrays in Texturen
  - Filter als Fragment Shader
  - Stream Programming

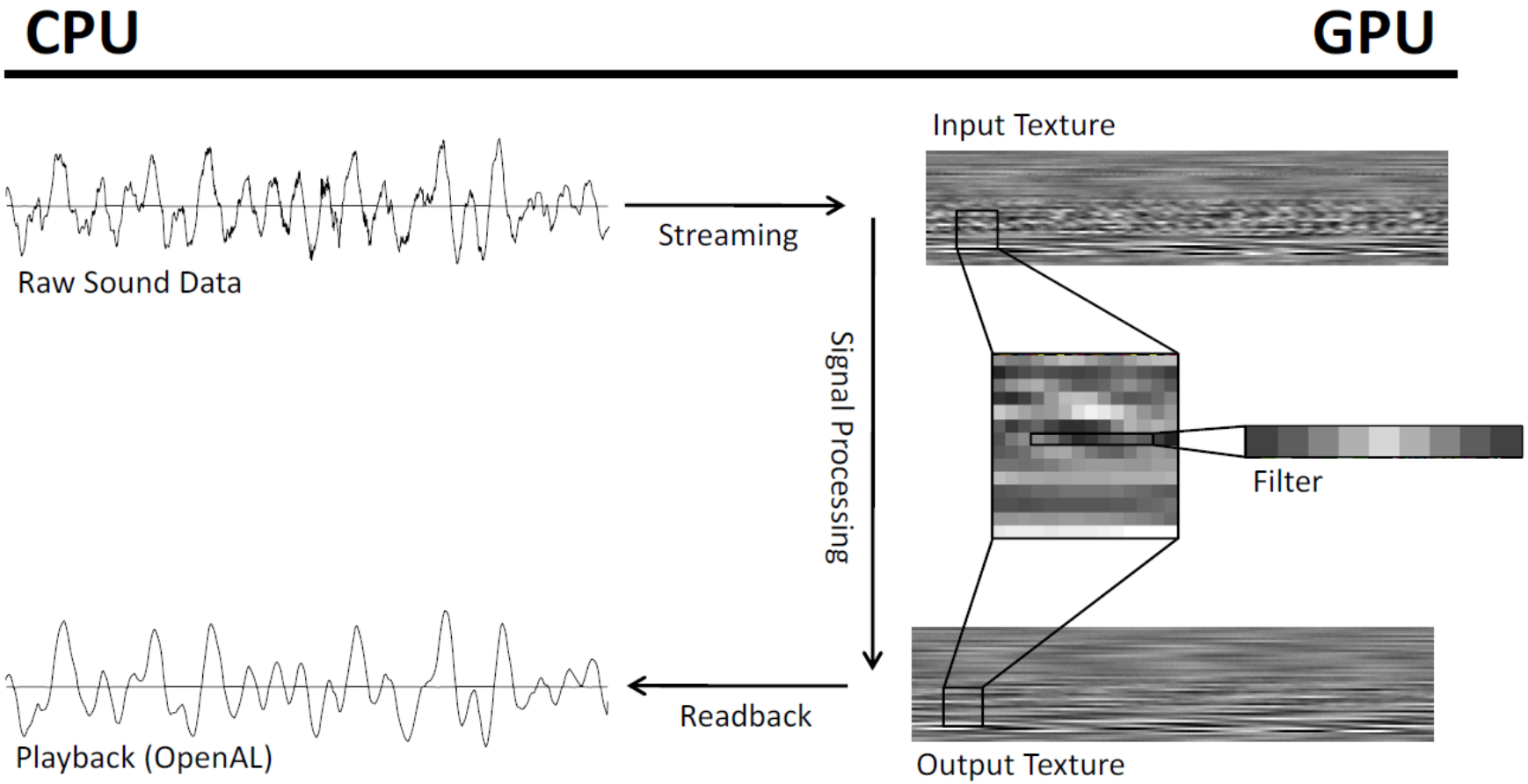


Weiterentwicklung

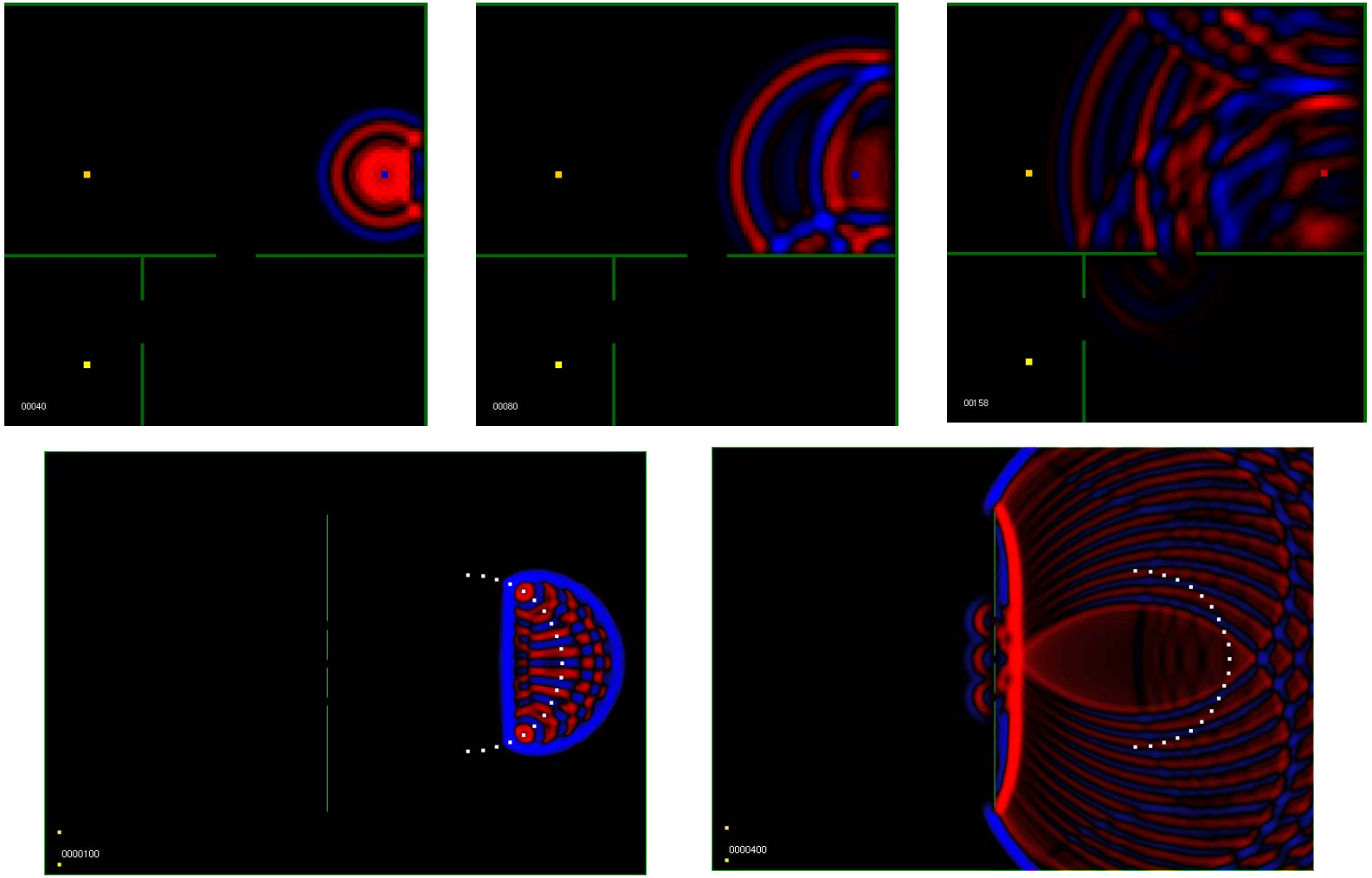
- Nutzung der GPU als Parallelprozessor
- Vorangetrieben durch Universitäten und Forschung
- Weiterhin alles Graphik basiert (OpenGL)

- Entwicklung von Hochsprachen:
  - Cg – C for Graphics
  - HLSL – High Level Shading Language
  - GLSL – OpenGL Shading Language
  - BrookGPU – Stream Programming

GPGPU Beispiel – DSP



GPGPU Beispiel – 3D Waveguides



GPU Programmierung Heute

- Verschiedene Entwicklungsumgebungen
  - Nvidia CUDA
  - AMD Stream
  - Brook GPU / Brook+
  - Rapid Mind Platform
  - PGI Accelerator Compiler Suite
- Middleware und Support Bibliotheken
- Mathematik Bibliotheken (lineare Algebra)

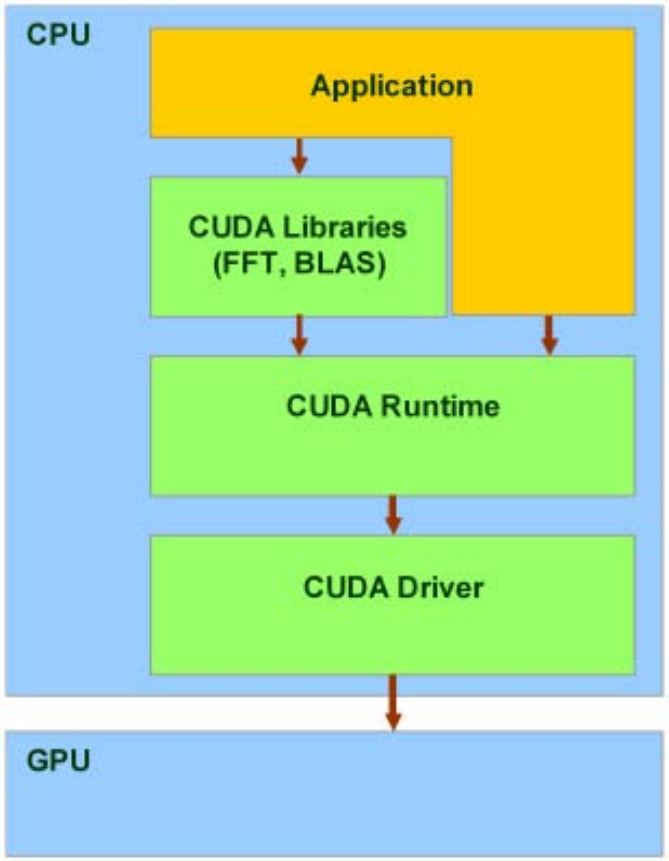
CUDA Konzept

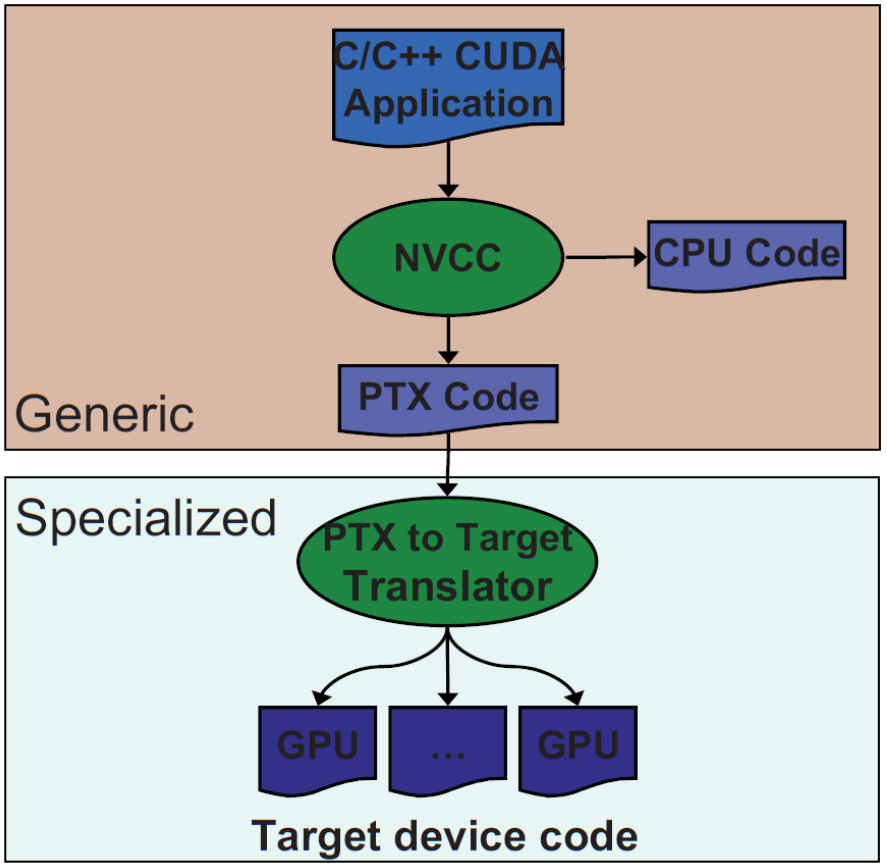
- Compute Unified Device Architecture
  - Unified Hardware (Processors) and Software
- Dedicated Many-Core Co-Prozessor
- Programmier Model:
  - Nutzer startet batches of threads (SIMT)
- Keine Graphik API mehr
- Highlevel Entwicklung in C/C++, Fortran, ...

CUDA Werkzeuge

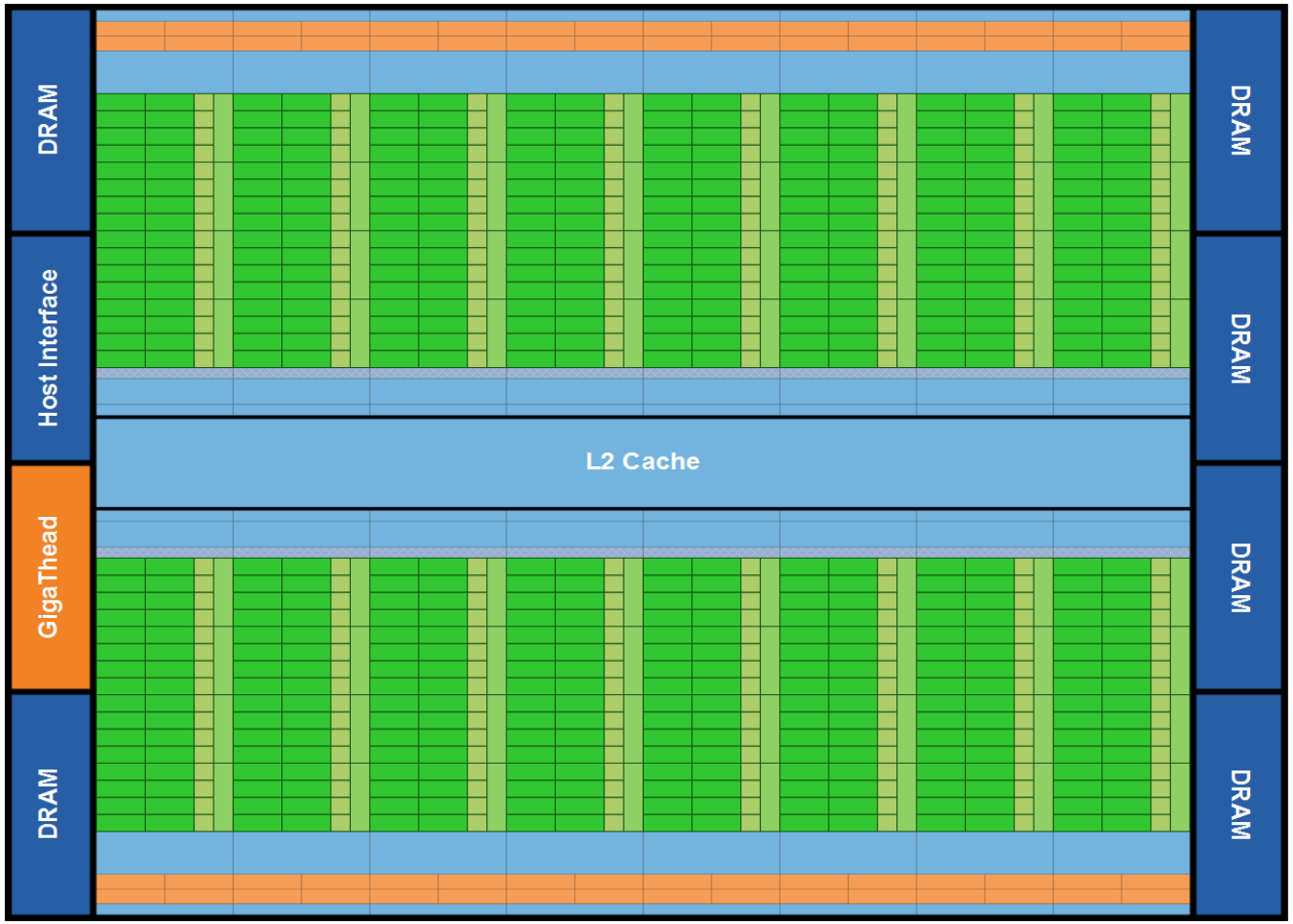
- CUDA Toolkit / SDK
  - Treiber, SDK, Compiler, Beispiele
  - Profiler, Occupancy Calculator, Debugger
  - Unterstützt werden: C/C++, FORTRAN, OpenCL, DirectCompute
- Bibliotheken
  - CUBLAS (Basic Linear Algebra Subprograms)
  - CUFFT (Fourier Transformation, Basis: fftw)
  - CUDPP (Data Parallel Primitives)
- Entwicklungsumgebungen
  - Visual Studio + NEXUS (Parallel Nsight)

Codegenerierung



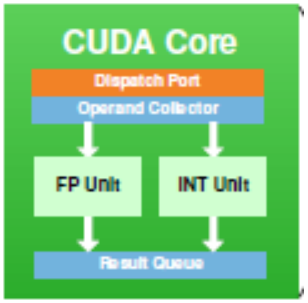


GF100 Architektur (Fermi)



Streaming Multiprozessor

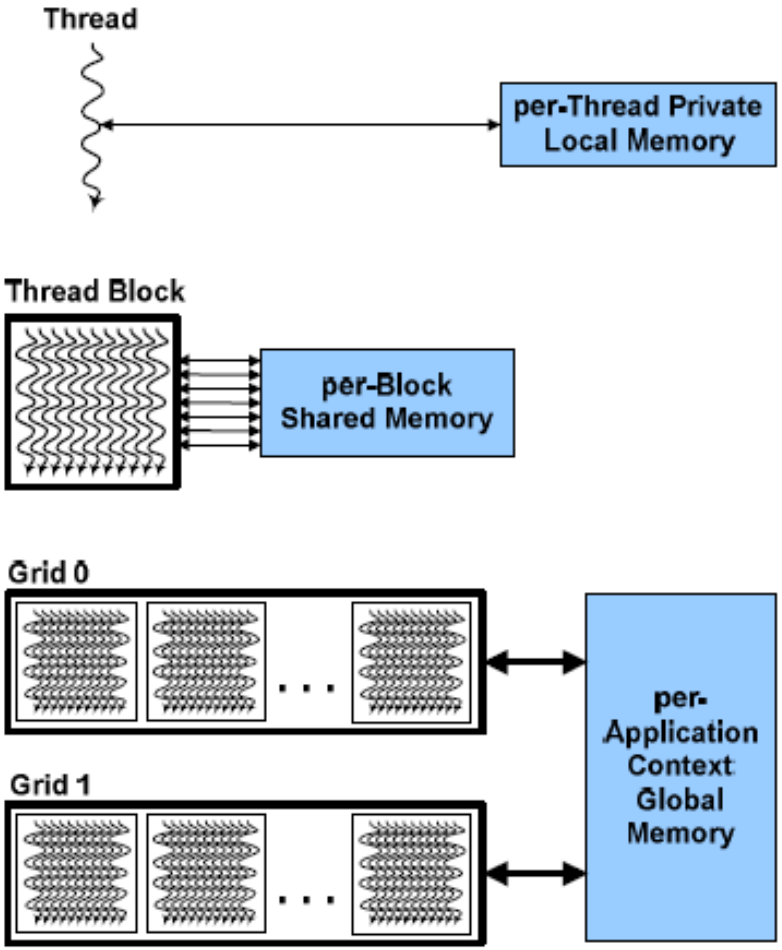
- 16 Streaming Multiprozessoren
  - 32 CUDA Kerne
  - FP / INT Unit
  - 16 Load / Store Units
  - 64k shared memory / L1 Cache
  - 4 Special Function Units (SFU)
    (sin, cos, sqr, …)
  - Concurrent Thread Execution
  - IEEE 754-2008 (FMA)
  - ECC Speicher



Vergleich G80 / GT200 / GF100

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

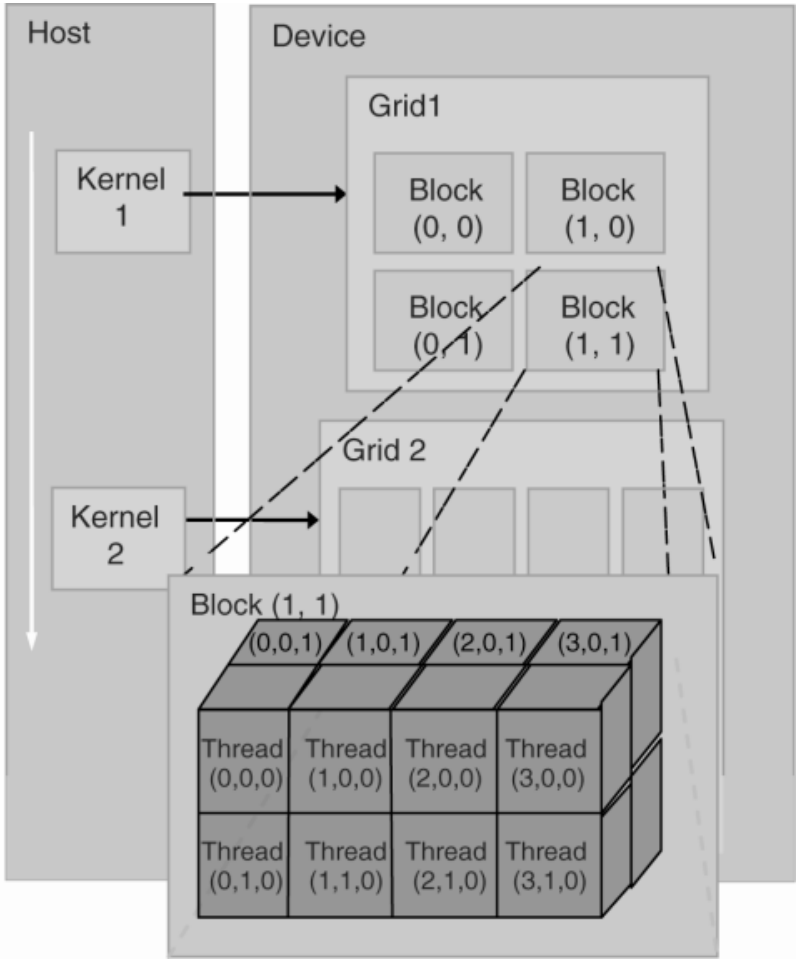
CUDA Hierarchie



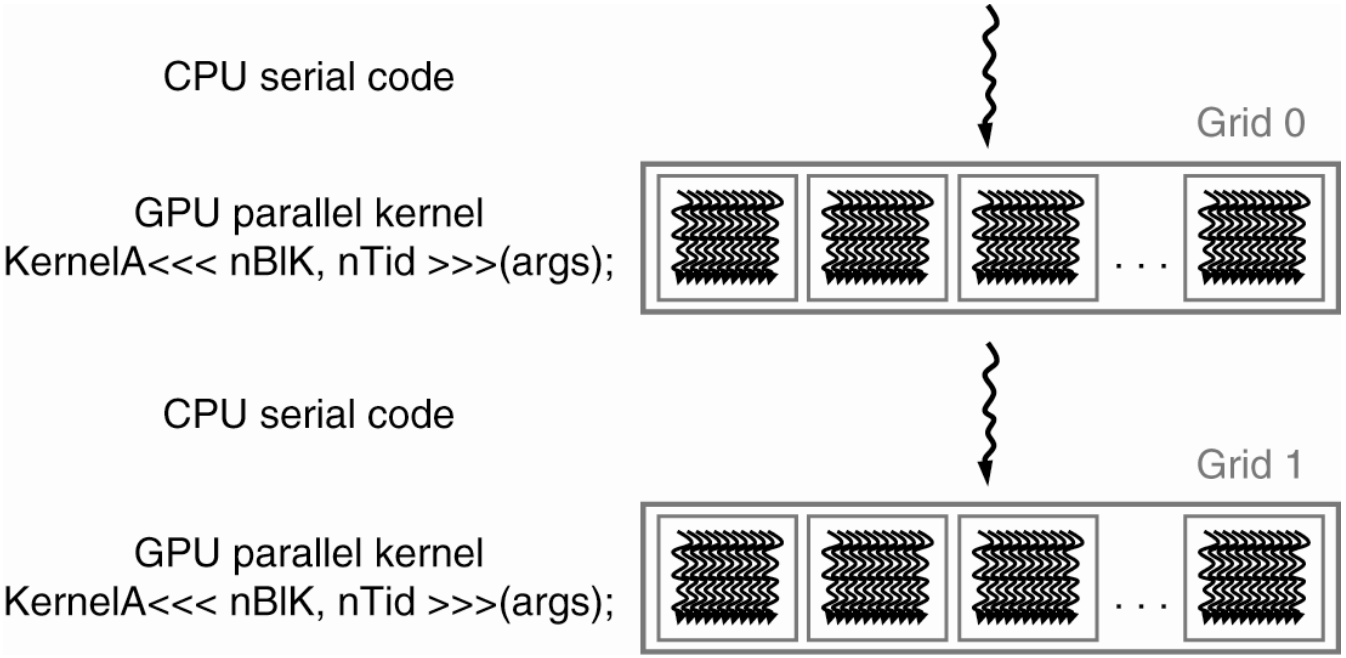
- Threads
- Blocks
- Grids

CUDA Threads

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate

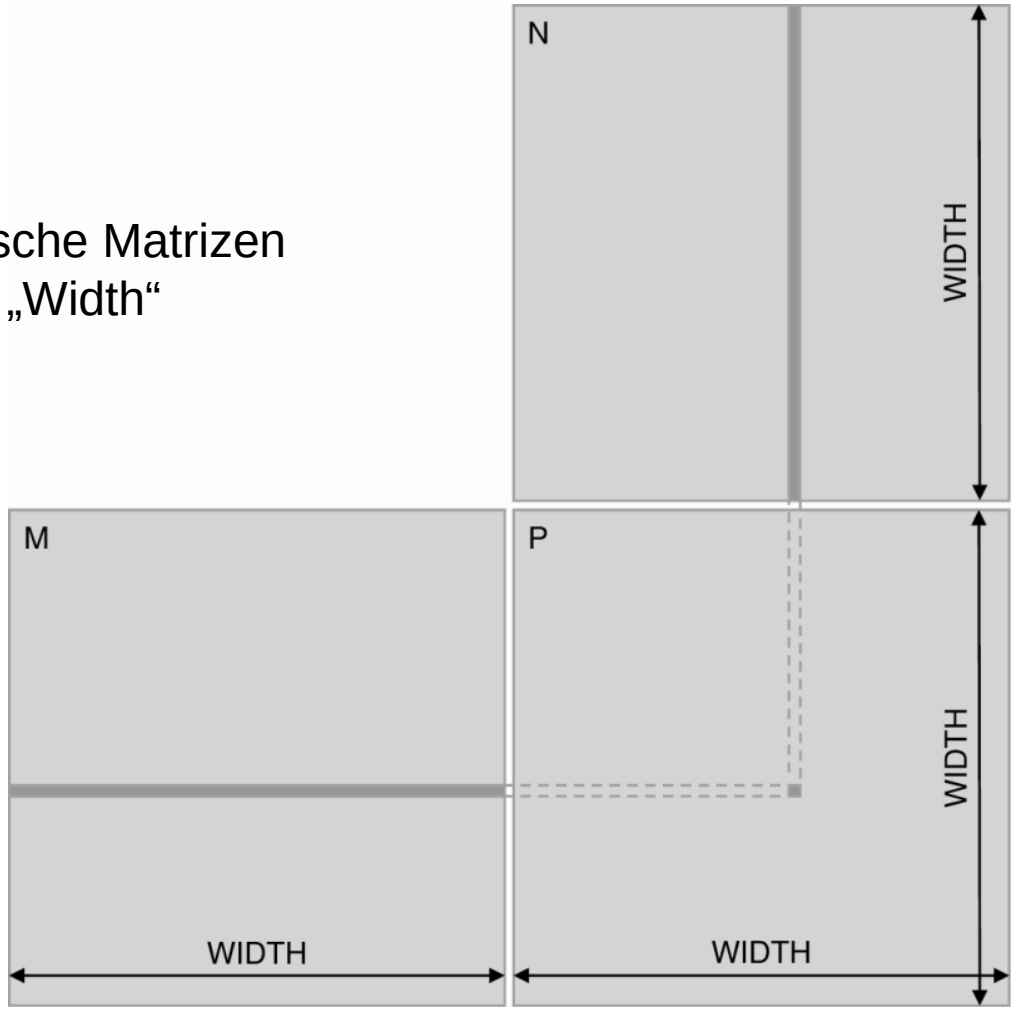


Ausführung eines CUDA Programms



Beispiel: Matrix-Matrix Multiplikation

2 quadratische Matrizen
der Größe „Width“



Main Function

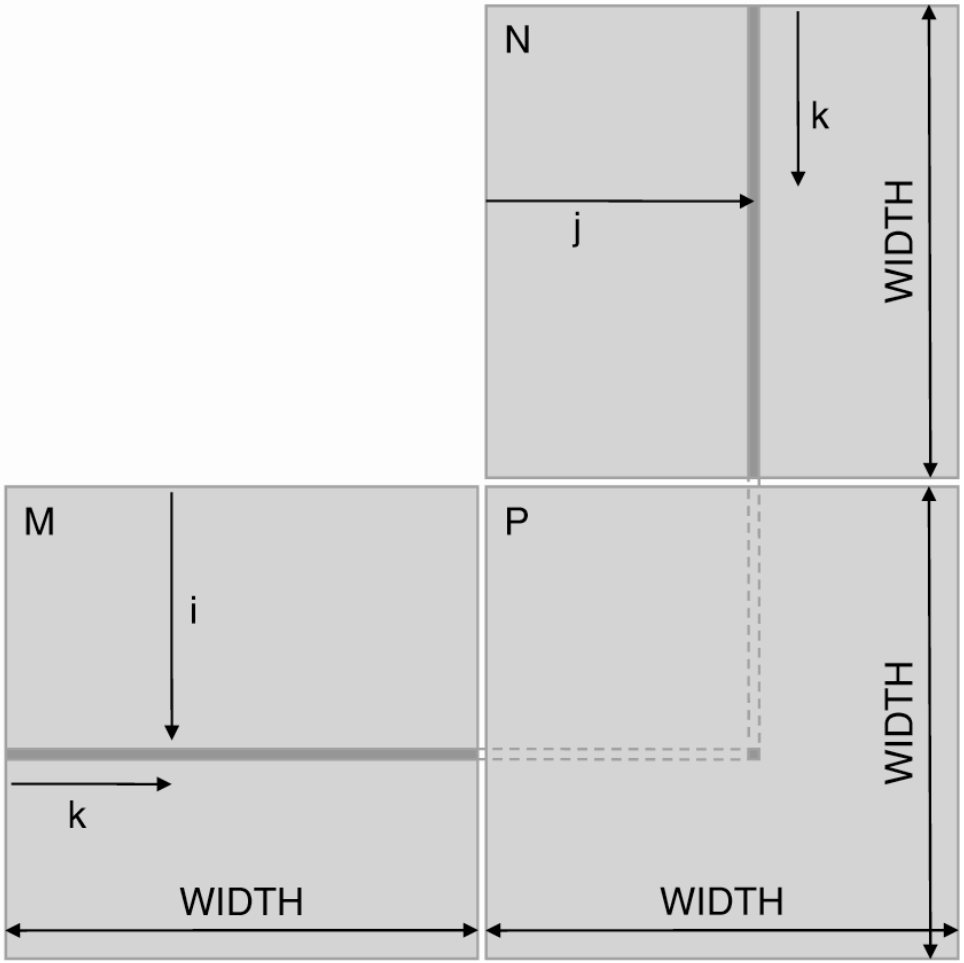
```
int main(void) {
1.  // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N
....

2.  // M * N on the device
    MatrixMultiplication(M, N, P, Width);


3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

Matrix-Matrix Multiplikation

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k  = 0; k < Width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



CUDA Implementierung

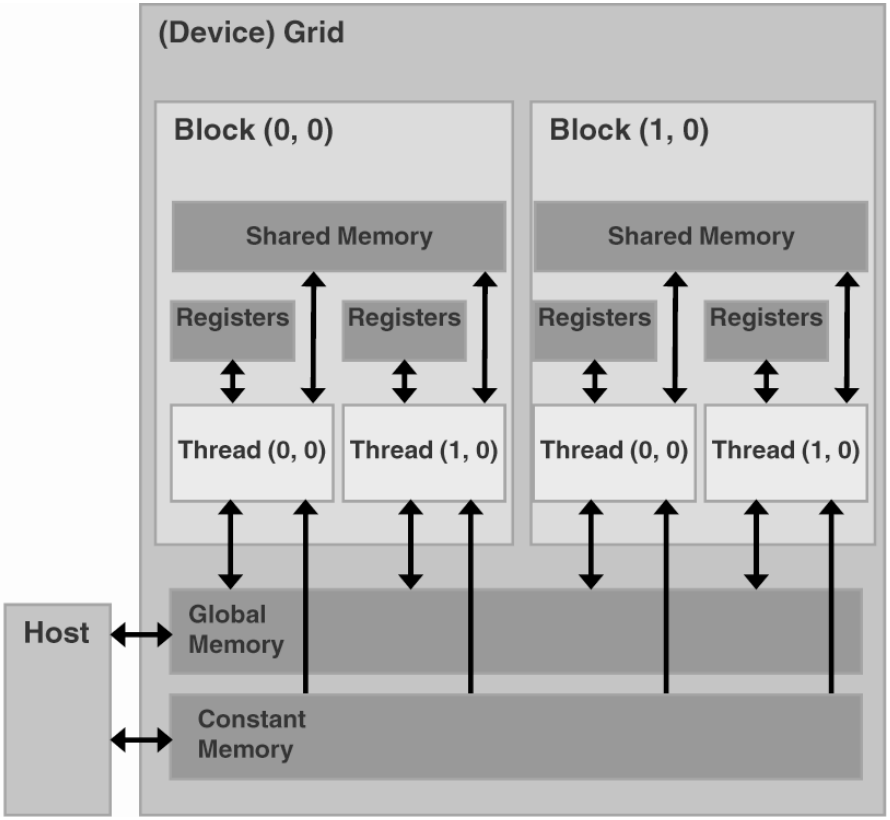
```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Allocate device memory for M, N, and P
   // copy M and N to allocated device memory locations

2. // Kernel invocation code - to have the device to perform
   // the actual matrix multiplication

3. // copy P from the device memory
   // Free device matrices
}
```

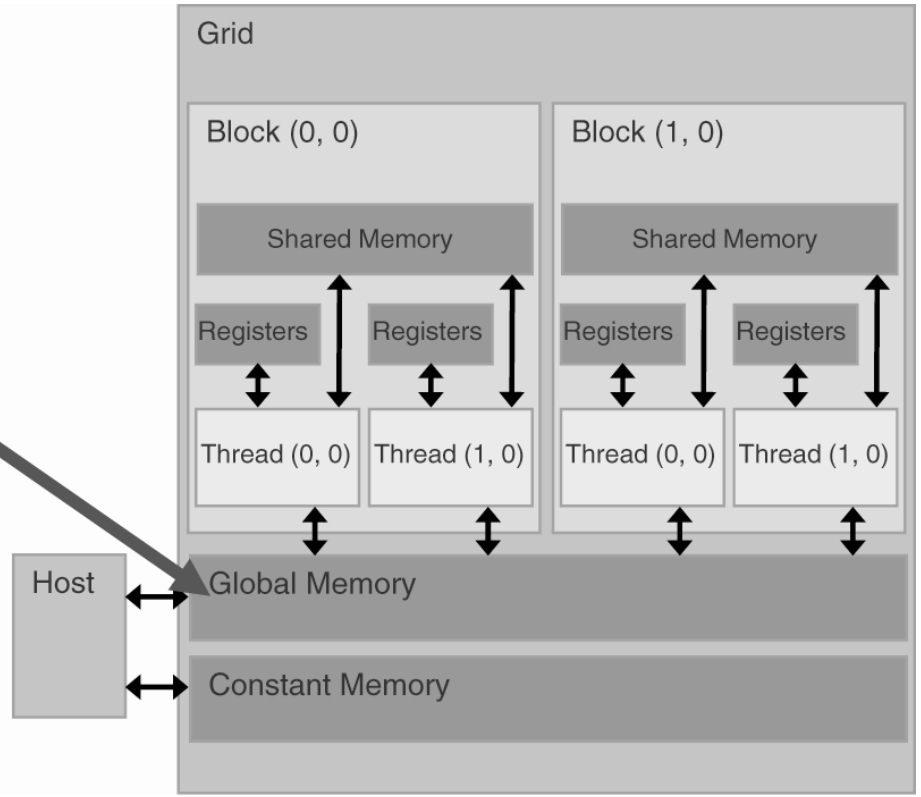
Exkurs: CUDA Device Memory Model

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories



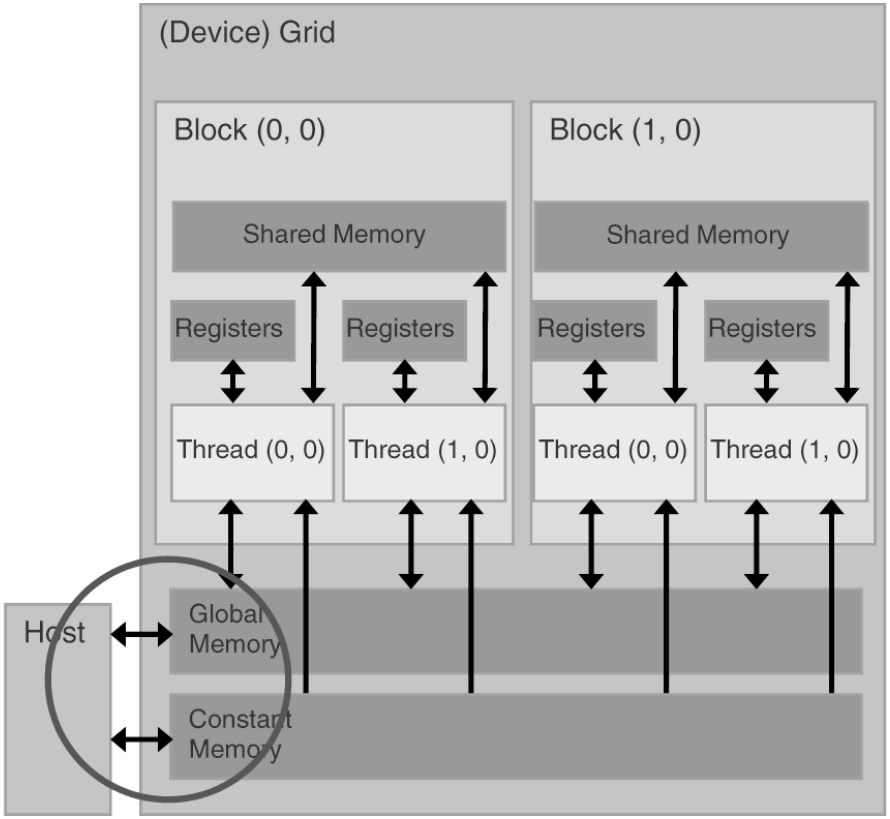
Exkurs: Speicher Management

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object



Exkurs: Datentransfer Host – Device

- cudaMemcpy()
  - **Memory** data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Transfer is asynchronous



CUDA Implementierung

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;

1. // Transfer M and N to device memory
   cudaMalloc((void**) &Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc((void**) &Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

   // Allocate P on the device
   cudaMalloc((void**) &Pd, size);

2. // Kernel invocation code - to be shown later
   ...
3. // Transfer P from device to host
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

CUDA Kernel

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue stores the Pd element that is computed by the thread
   float Pvalue = 0;

   for (int k = 0; k < Width; ++k)
   {
      float Mdelement = Md[ty * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```

Aufruf des Kernels

```
// Setup the execution configuration
dim3 dimBlock(Width, Width);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Optimierung

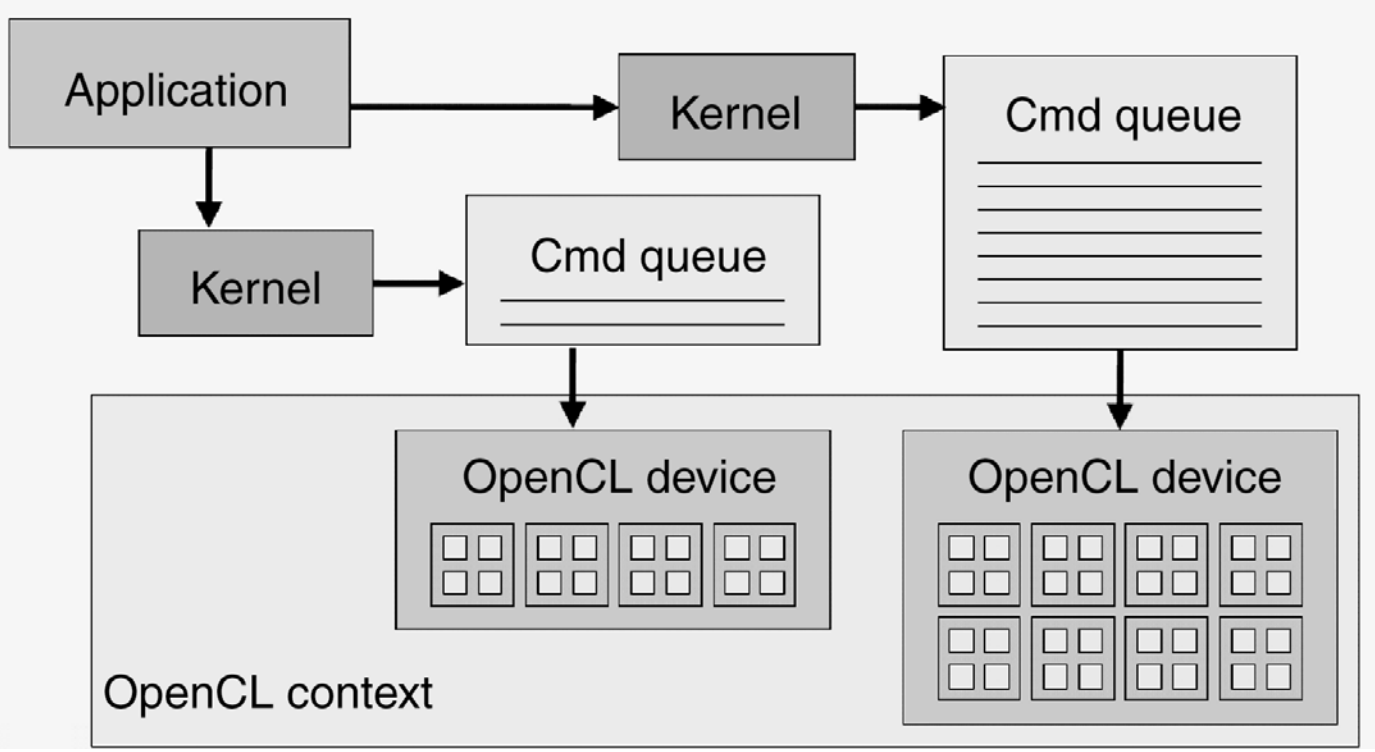
- **Optimierung ist enorm wichtig!**
- Maximierung der Auslastung (Occupancy):
  - Speicherdurchsatz (Bandbreite)
  - Anweisungsdurchsatz
  - Auslastung

- Beispiel: Speicher Optimierung
  - Transfer zwischen Host – Device
  - Speichertypen (global, shared, constant)
  - Coalesced vs. non-coalesced Zugriff

OpenCL

- Compute Language für CPUs und GPUs
- Offener Standard für heterogene Umgebungen
  - Khronos Group (Apple)
  - OpenCL 1.0 (8.12.2008)
- OpenGL and OpenCL share Resources
  - OpenCL is designed to efficiently share with OpenGL
  - Textures, Buffer Objects and Renderbuffers
  - Data is shared, not copied

OpenCL Context



OpenCL Beispiel

```
// OpenCL Objects
  cl_device_id device;
  cl_context context;
  cl_command_queue queue;
  cl_program program;
  cl_kernel kernel;
  cl_mem buffer;

// Setup OpenCL
  clGetDeviceIDs(NULL, CL_DEVICE_TYPE_DEFAULT, 1, &device, NULL);
  context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
  queue = clCreateCommandQueue(context, device,
  (cl_command_queue_properties)0, NULL);

// Setup the input
  buffer = clCreateBuffer(context, CL_MEM_COPY_HOST_PTR,
  sizeof(cl_float)*10240, data, NULL);
```

OpenCL Beispiel cont.

```
// Build the kernel
   program = clCreateProgramWithSource(context, 1, (const
   char**)&source, NULL, NULL);
   clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
   kernel = clCreateKernel(program, "calcSin", NULL);

// Execute the kernel
   clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
   size_t global_dimensions[] = {LENGTH,0,0};
   clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
   global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
   clEnqueueReadBuffer(queue, buffer, CL_TRUE, 0,
   sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Clean up
```

Einsatz von GPUs im HPC

- Top100 (11/2010)
  - 3 Systeme mit Nvidia/Tesla (1, 3, 4, 28, 72, 88)
  - 1 System mit ATI (22)
- Viele Systeme in Entwicklung
  - zB ORNL, Tesla-basiert, >2PF
- Einsatzgebiete:
  - Astronomie und Astrophysik
  - Biologie, Chemie
  - Finanzwirtschaft

GPU Cluster in der Top100 (11/2010)

Tianhe-1a (Top500 Platz 1)

R_{max} 2.566 R_{peak} 4.701 (TFlops)

NUDT TH MPP, X5670 2.93Ghz 6C, Nvidia GF104

TSUBAME 2.0 (Top500 Platz 4)

R_{max} 1.192 R_{peak} 2.287 (TFlops)

HP ProLiant SL390s G7,Xeon 6C X5670, NVidia Tesla C2050 GPU

FORTRAN und CUDA

- FORTRAN noch weit verbreitet (z.B. Klimaforschung)
- CUBLAS, CUFFT (NVIDIA)
  - CUDA implementation of BLAS routines with Fortran API
- F2C-ACC (NOAA Earth System Research Laboratory)
  - Generates C or CUDA output from Fortran95 input
- HMPP Workbench (CAPS Enterprise)
  - Directive-based source-to-source compiler
- PGI Compiler Suite
  - Directive-based
  - Compiler

Fortran CUDA Beispiel

- PGI Accelerator Compiler
  - OpenMP-like implicit programming model for X64+GPU-systems

```
!$acc region
  do k = 1,n1
    do i = 1,n3
      c(i,k) = 0.0
      do j = 1,n2
        c(i,k) = c(i,k) + a(i,j) * b(j,k)
      enddo
    enddo
  enddo
!$acc end region
```

Example http://www.pgroup.com

Fortran CUDA Beispiel

- PGI CUDA Fortran Compiler

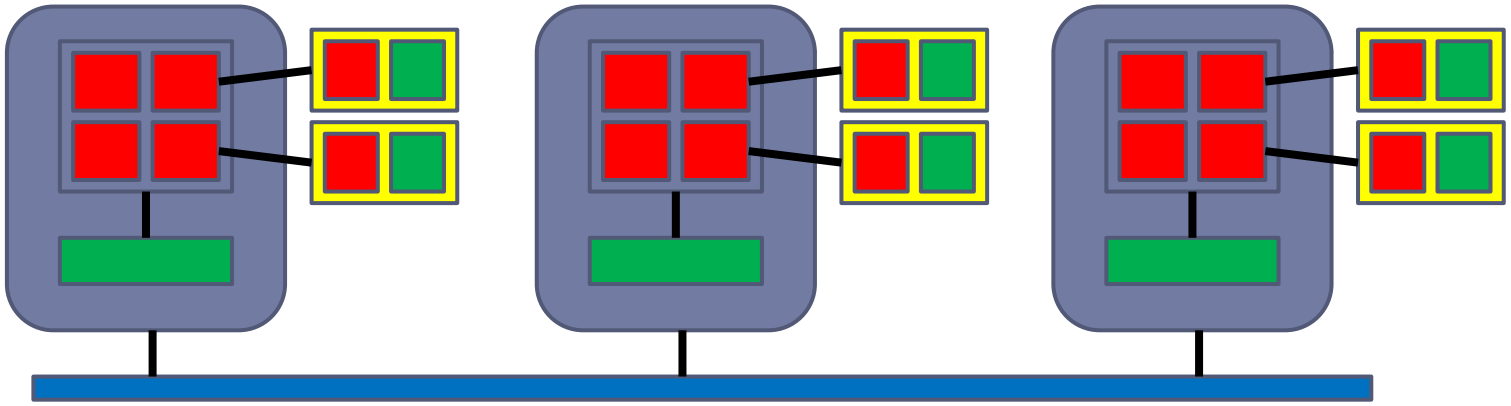
```
! Kernel definition
attributes(global) subroutine ksaxpy( n, a, x, y )
   real, dimension(*) :: x,y
   real, value :: a
   integer, value :: n, i
   i = (blockidx%x-1) * blockdim%x + threadidx%x
   if( i <= n ) y(i) = a * x(i) + y(i)
end subroutine
! Host subroutine
subroutine solve( n, a, x, y )
   real, device, dimension(*) :: x, y
   real :: a
   integer :: n
! call the kernel
   call ksaxpy<<<n/64, 64>>>( n, a, x, y )
end subroutine
```

Example http://www.pgroup.com

Hybride Systeme

- Berechnungen auf GPU Clustern
  - MPI zur Kommunikation der Knoten untereinander
  - OpenMP im Knoten (SMP)
  - CUDA für Beschleunigerkarten

  - Sehr gute Auslastung und Effizienz/Strom Verhältnis
  - Schwierig: Umsetzung und Optimierung

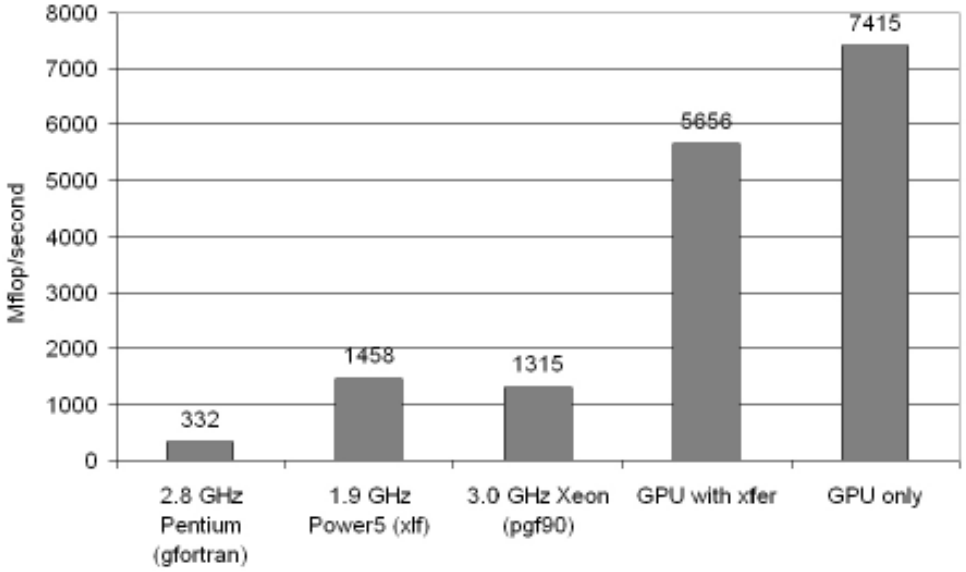


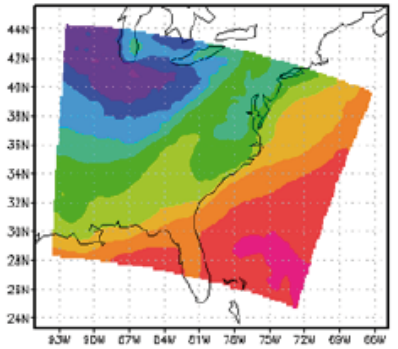
Beispiele aus der Klimaforschung

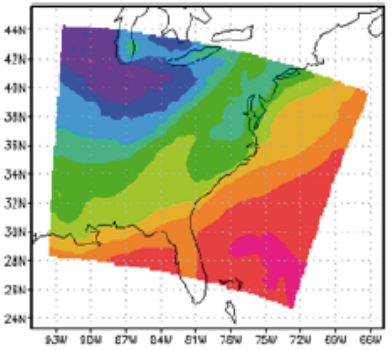
- GPU-unterstützte Klimasimulationen
- GPU-basierter CFD Solver
- Hybride Systeme (MPI-CUDA Implementierung)

Wettervorhersage (WRF)

Michalakes, Vachharajani, *"GPU Acceleration of Numerical Weather Prediction"*, Parallel Processing Letters Vol.18 No.4, 2008







WMS5 Potential Temperature
Original (oben), GPU (unten)

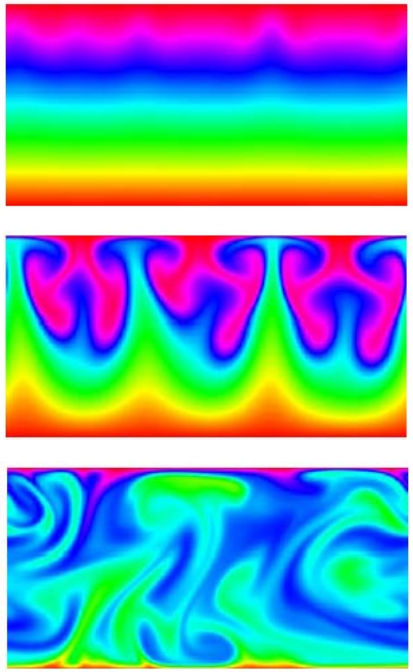
GPU-basierte CFD Solver

Cohen, Molemaker, *"A Fast Double Precision CFD Code using CUDA"*, Proceedings of Parallel CFD, 2009

CPU Fortran Code			
Resolution	ms/step	ms/step/node	Scaling
$64^2 \times 32$	47	37.0e-5	-
$128^2 \times 64$	327	31.2e-5	0.84x
$256^2 \times 128$	4070	48.5e-5	1.55x
$384^2 \times 192$	13670	48.3e-5	1.00x

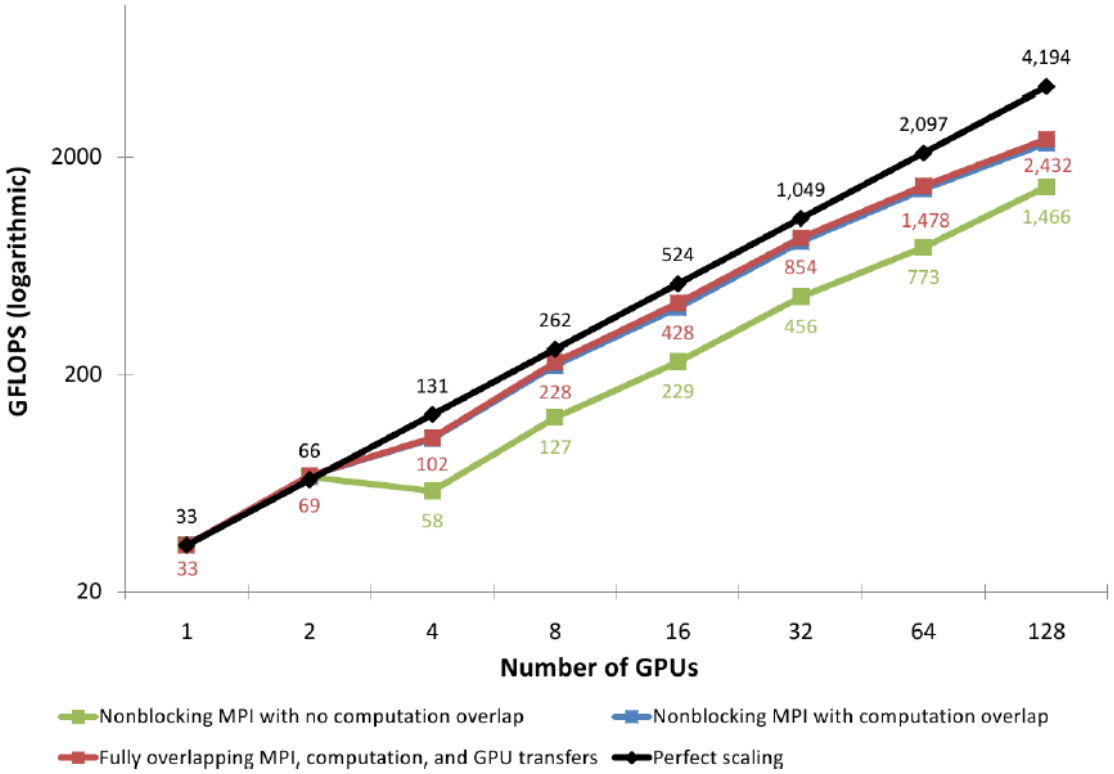
GPU CUDA Code				**GPU Speedup**
Resolution	ms/step	ms/step/node	Scaling	
$64^2 \times 32$	24	18.3e-5	-	**2.0x**
$128^2 \times 64$	79	7.5e-5	0.41x	**5.3x**
$256^2 \times 128$	498	5.9e-5	0.79x	**8.2x**
$384^2 \times 192$	1616	5.7e-5	0.97x	**8.5x**

8 Core Xeon vs. GT200



Hybride Systeme

Jacobsen, Thibault, Senocak, *"An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters"*, AIAA Meeting, 2010



Sustained performance of 2.4 TeraFLOPS with 128 GPUs on 64 compute nodes.

Ausblick

- FERMI nicht so stark wie erwartet, dennoch ein Meilenstein (Vorsicht bei Geforce Karten!)
- GPUDirect (Nvidia und Mellanox)

- Intels Knight Familiy (Ex-Larrabee)
- Multi- und Many Core Systeme
- Entwicklung heterogener Systeme (Verschmelzung CPU/GPU)

Zusammenfassung

- Stetig steigende Entwicklung seit 2000
- Beschleunigt seit Einführung von CUDA (2007)
- IEEE 754-2008 Unterstützung / ECC Speicher
- Für FORTRAN Source2Source Compiler
- Optimierung ist enorm wichtig

- Alternativen zu CUDA:
  - OpenCL
  - Intel‘s Knights Family

Referenzen

- [1] http://developer.nvidia.com/object/gpucomputing.html
- [2] http://www.gpucomputing.net
- [3] http://www.gpgpu.org/developer

- [4] “Programming Massively Parallel Processors: A Hands-On Approach”, Kirk & Hwu
- [5] “Cuda by Example: An Introduction to General-Purpose GPU Programming”, Sanders & Kandrot
- [6] “GPU Computing Gems”, Hwu (Erscheint im März 2011)

- Bei Fragen: Niklas Röber

  DKRZ – Raum 211 (roeber@dkrz.de)