GPU-Computing

im Rahmen der Vorlesung
Hochleistungsrechnen

Michael Vetter
Universität Hamburg
Scientific Visualization and Parallel Processing

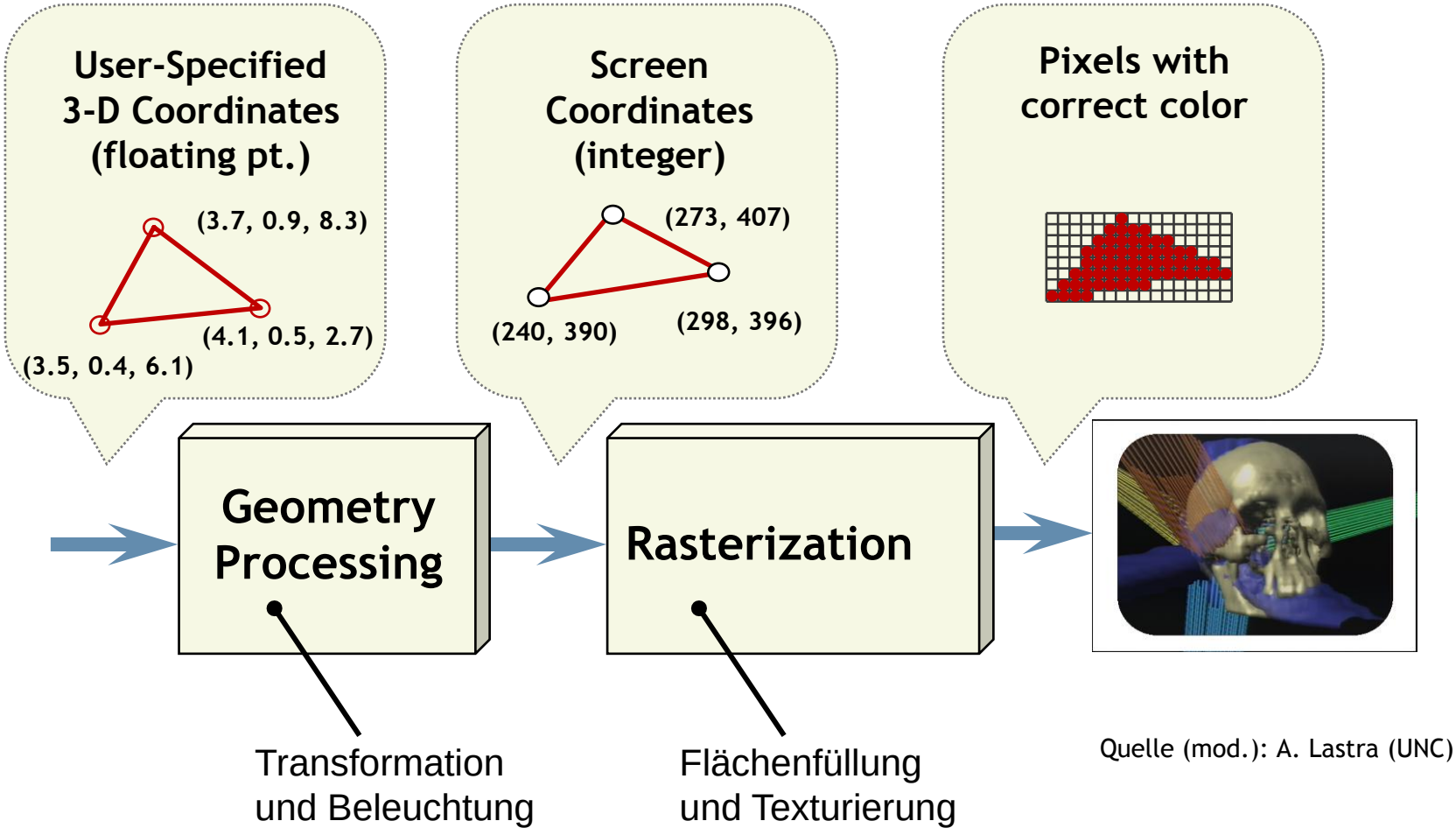
Übersicht

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

Foliensatz basiert auf einer Vorlage von Prof. Dr. Ludwig,
viele Beispiele und Graphiken aus [4] entnommen

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

Was ist Rendering

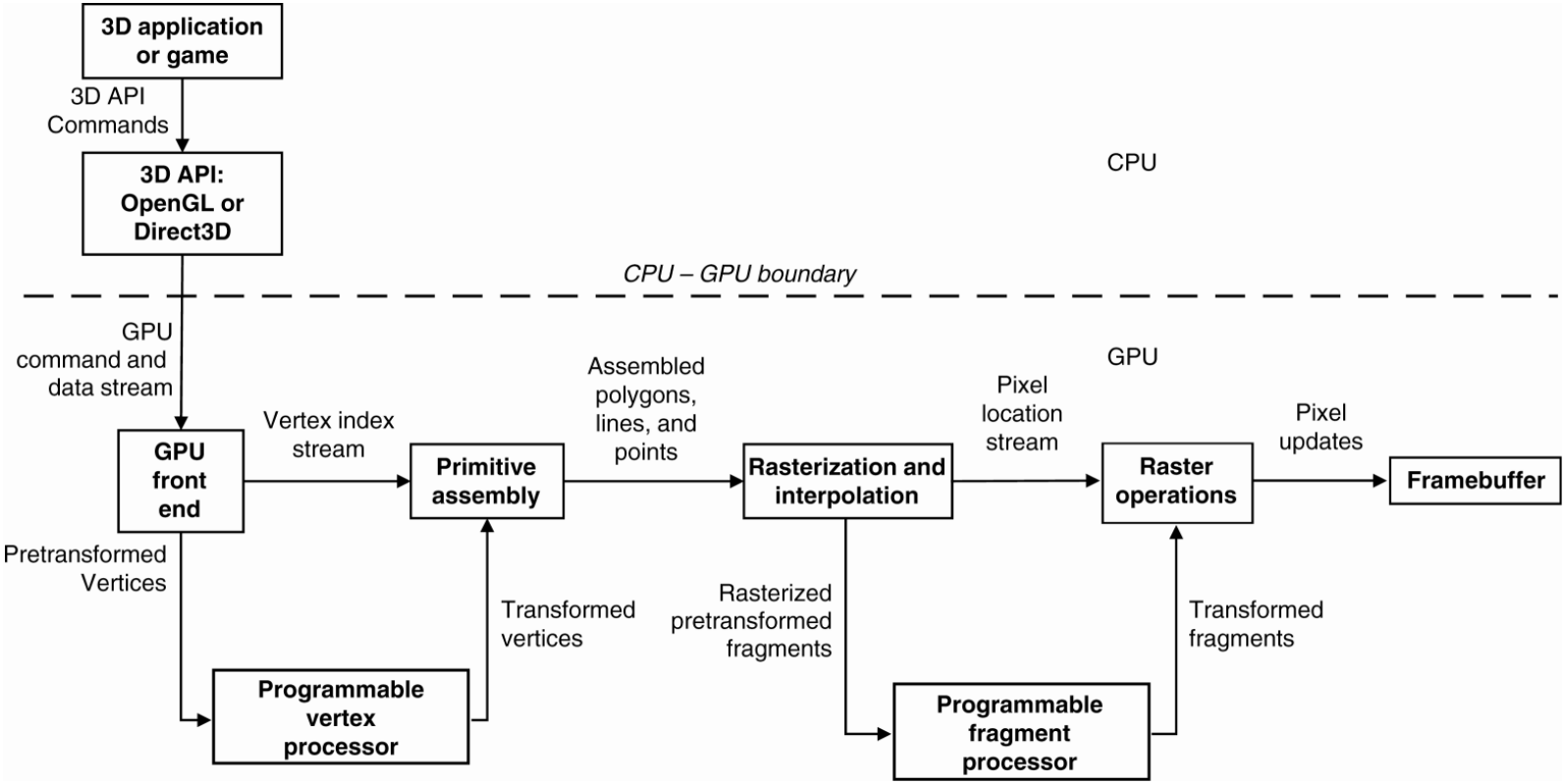


Quelle (mod.): A. Lastra (UNC)

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

Hintergrund

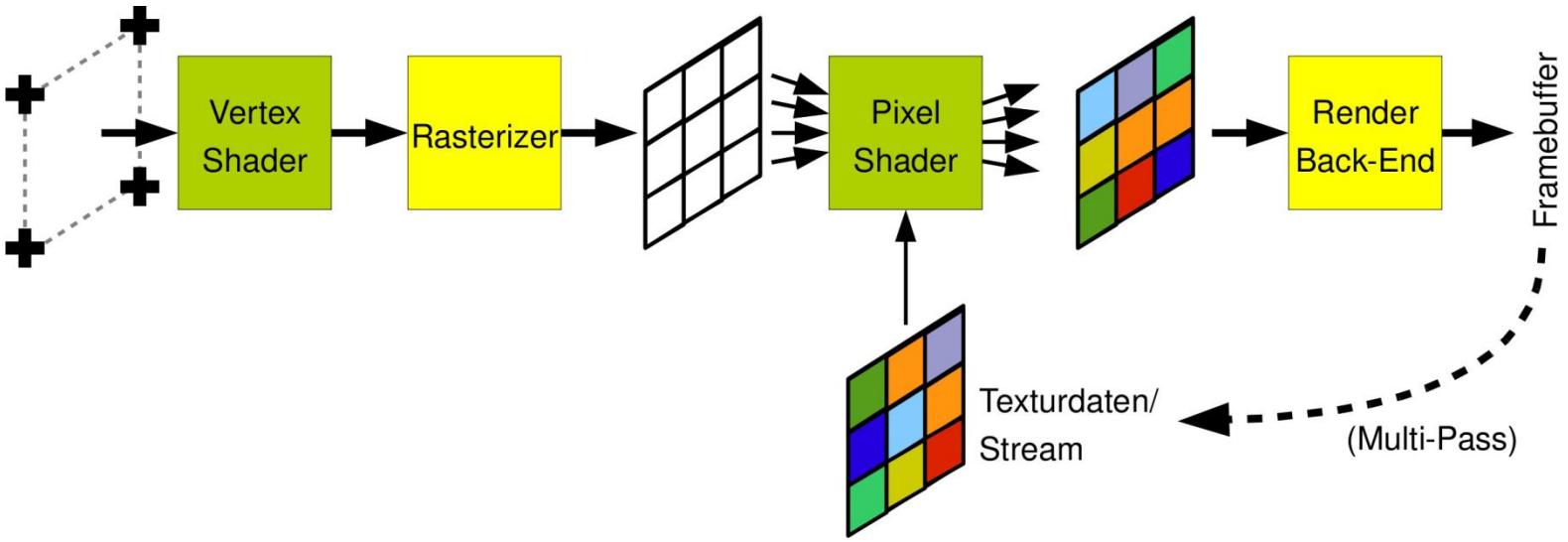
- Bis 2000 bei Graphikkarten „fixed function pipeline“
- Transform & Lighting Engine (Geforce256, 99/00)



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

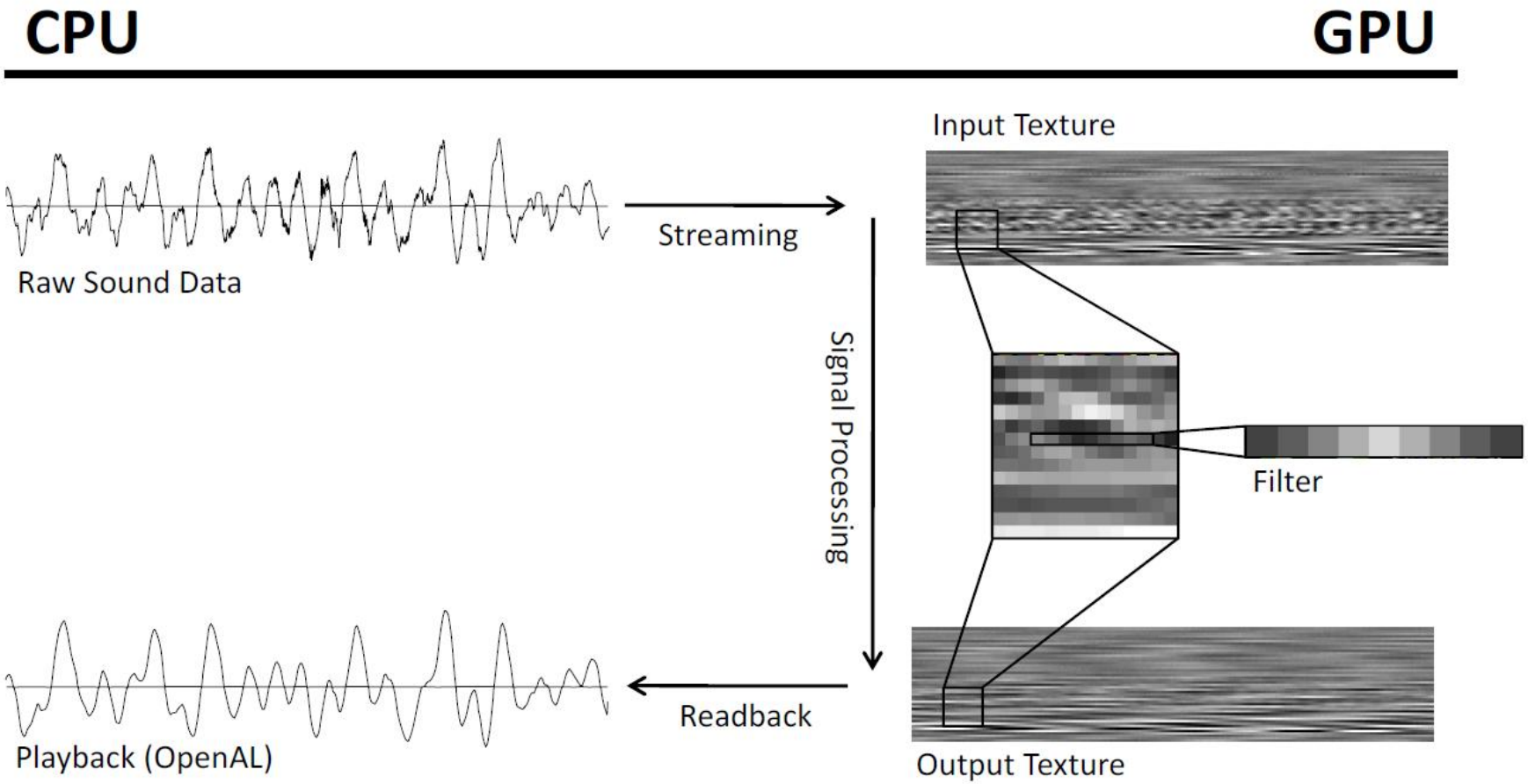
General Purpose Graphics Programming

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



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

GPGPU Beispiel – DSP



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

GLSL Beispiel

- Programmierbare Vertex-Shader und Pixel-Shader („fragment shader“) sind seit OpenGL 2.0 (2004) standardisiert: **GLSL (OpenGL Shading Language)**
- Literatur:
  - R. J. Rost, B. Licea-Kange: OpenGL Shading Language, Addison-Wesley, 2010

SAXPY (Single-precision Alpha **X** Plus **Y**) in C

```
float x = x_orig[i];
float y = y_orig[i];
y_new[i] = y + alpha * x;
```

Operationen erfordern Iteration:
for-Schleife (serielle Ausführung)

... in GLSL (fragment shader)

```
float saxpy (
  float2 coords : TEXCOORD0,
  uniform sampler2D textureY,
  uniform sampler2D textureX,
  uniform float alpha ) : COLOR
{
  float x = tex2D(textureX,coords);
  float y = tex2D(textureY,coords);
  float result = y + alpha * x;
  return result;
}
```

Operationen werden beim Zeichnen eines texturierten Rechtecks parallel ausgeführt

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

GPU Programmierung Heute

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

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

CUDA Konzept

- Compute Unified Device Architecture
  - Unified Hardware (Processors) and Software
  - Erste Grafikkarte verfügbar seit Ende 2006
- Dedicated Many-Core Co-Prozessor
- Programmier Model:
  - SIMD ➔ SIMT
    (Single-Instruction Multiple-Thread)
- Keine Graphik API mehr

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

- Highlevel Entwicklung in C/C++, Fortran, ...

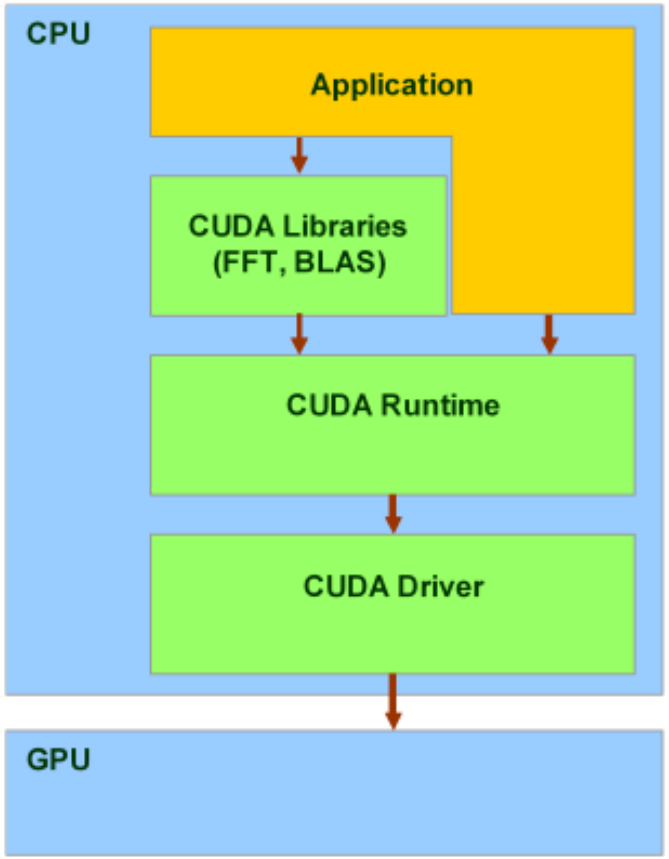
Michael Vetter
michael.vetter@rrz.uni-hamburg.de

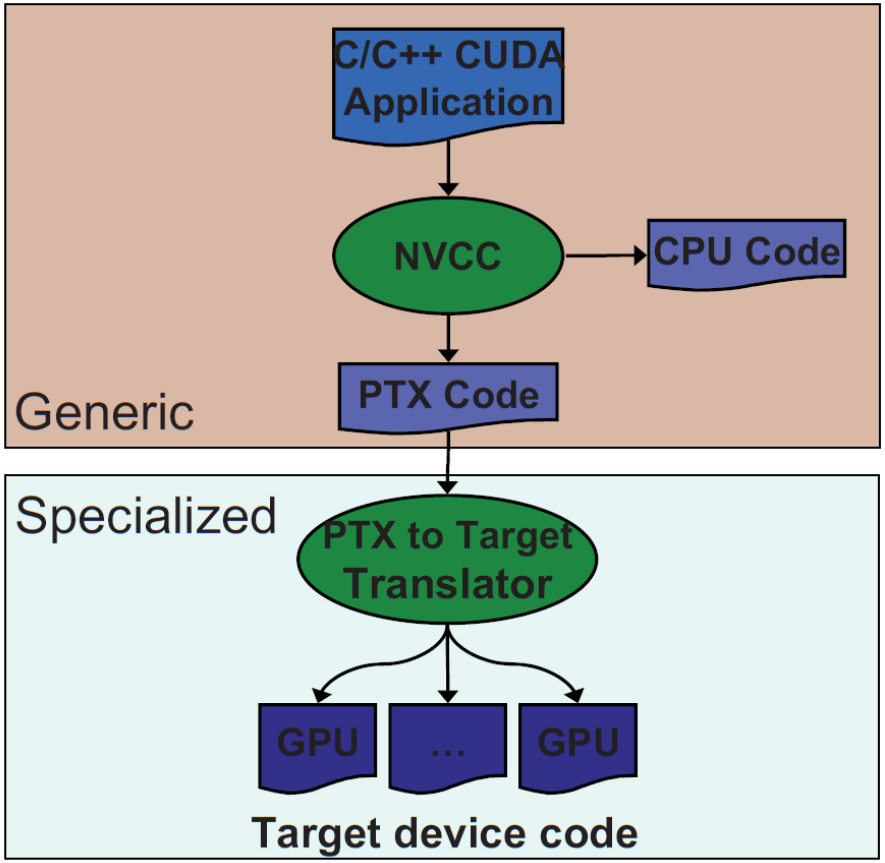
CUDA Werkzeuge

- 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), THRUST (STL)
- Entwicklungsumgebungen
  - Visual Studio + NEXUS (Parallel Nsight)

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

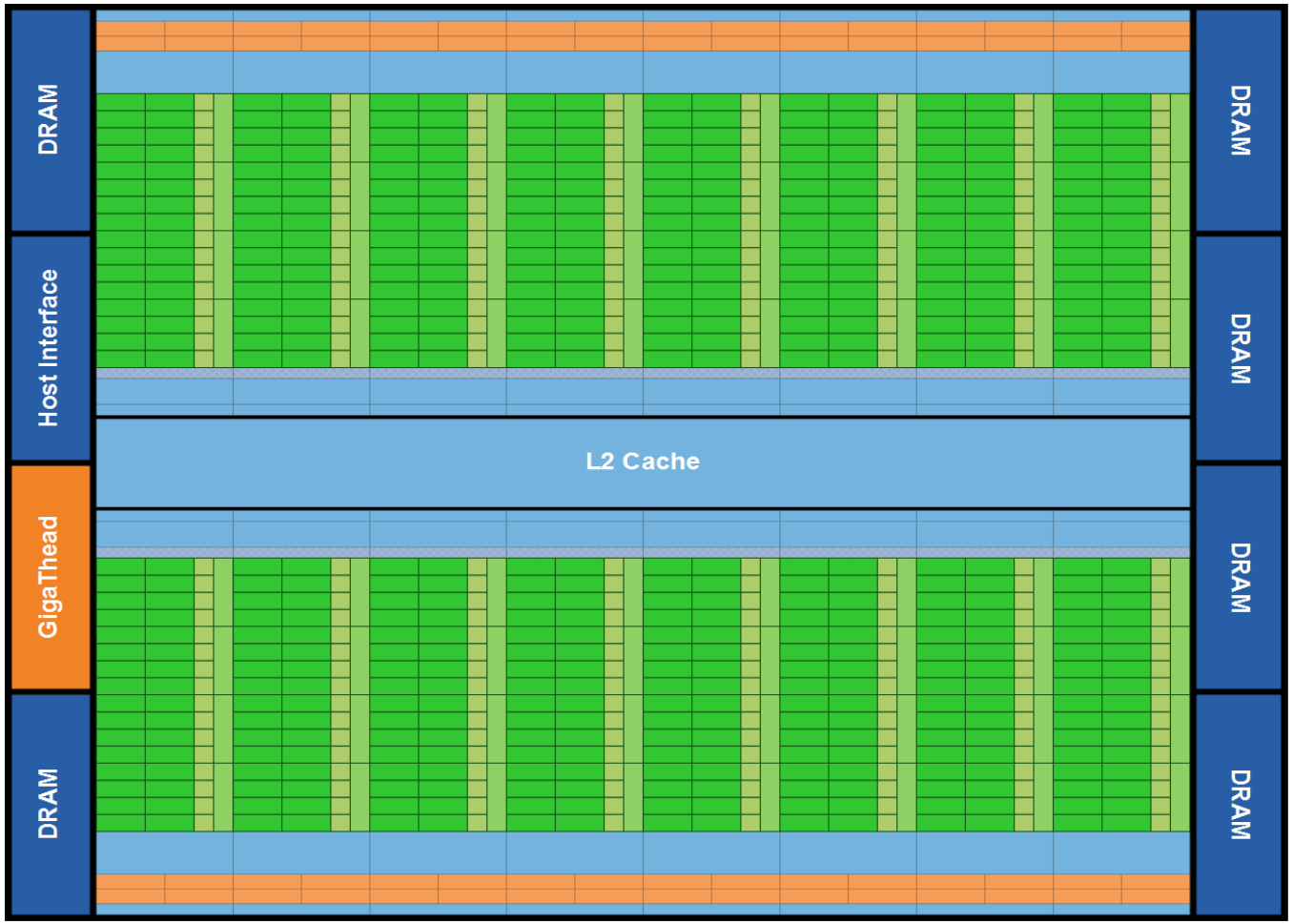
Codegenerierung





Michael Vetter
michael.vetter@rrz.uni-hamburg.de

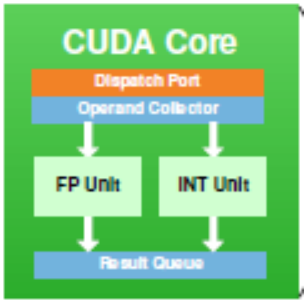
GF100 Architektur (Fermi)



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

Vergleich Fermi / Keppler

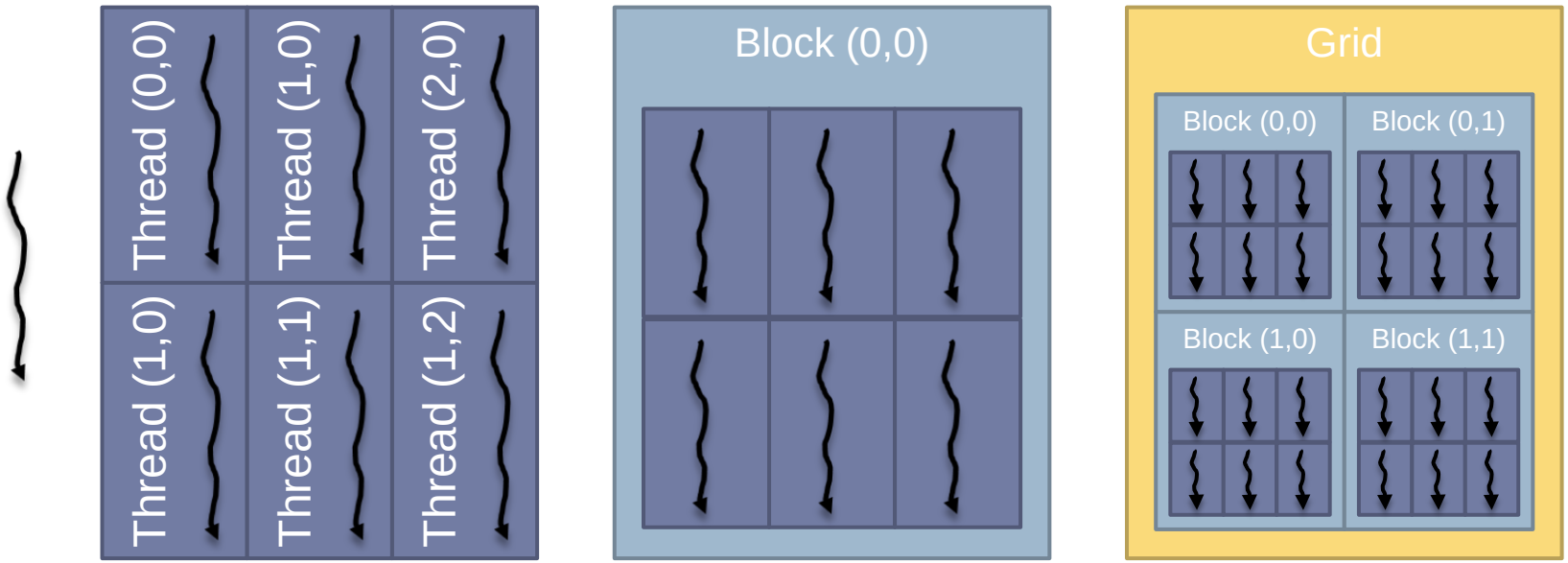
HAUPTMERKMALE	TESLA K20X	TESLA K20	TESLA K10	TESLA M2090	TESLA M2075
GPGPU Leistung	1 Kepler GK110		2 Kepler GK104s	1 Fermi GPU	1 Fermi GPU
GPU Computing Anwendungen	Seismische Berechnungen, CFD, CAE, Finanzberechnungen, Chemoinformatik und computergestützte Physik, Datenanalysen, Satellitenbildgebung, Wettermodellierung		Seismische Datenverarbeitung, Signal- und Bildverarbeitung, Videoanalyse	Seismische Datenverarbeitung, CFD, CAE, Finanzmathematik, Computerchemie und physik, Datenanalyse, Satellitenbildgebung, Wettermodelle	
Peak double precision floating point performance	1.31 Tflops	1.17 Tflops	190 Gigaflops (95 Gflops pro Grafikprozessor)	665 Gigaflops	515 Gigaflops
Maximale Single-Precision Gleitkommaleistung	3.95 Tflops	3.52 Tflops	4577 Gigaflops (2288 Gflops pro Grafikprozessor)	1331 Gigaflops	1030 Gigaflops
Speicherbandbreite (ECC deaktiviert)	250 GB/sec	208 GB/sec	320 GB/s (160 GB/s pro Grafikprozessor)	177 GB/s	150 GB/s
Speichergröße (GDDR5)	6 GB	5 GB	8 GB (4 GB pro Grafikprozessor)	6 GB	6 GB
CUDA Recheneinheiten	2688	2496	3072 (1536 pro Grafikprozessor)	512	448

* Hinweis: Ist ECC aktiviert, werden 12,5 % des Grafikprozessorspeichers für ECC-Bits genutzt. Beispiel: Von 3 GB Gesamtspeicher sind 2,625 GB für den Benutzer verfügbar, wenn ECC aktiviert ist.

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

CUDA Programming Model

Threads, Blocks and Grids

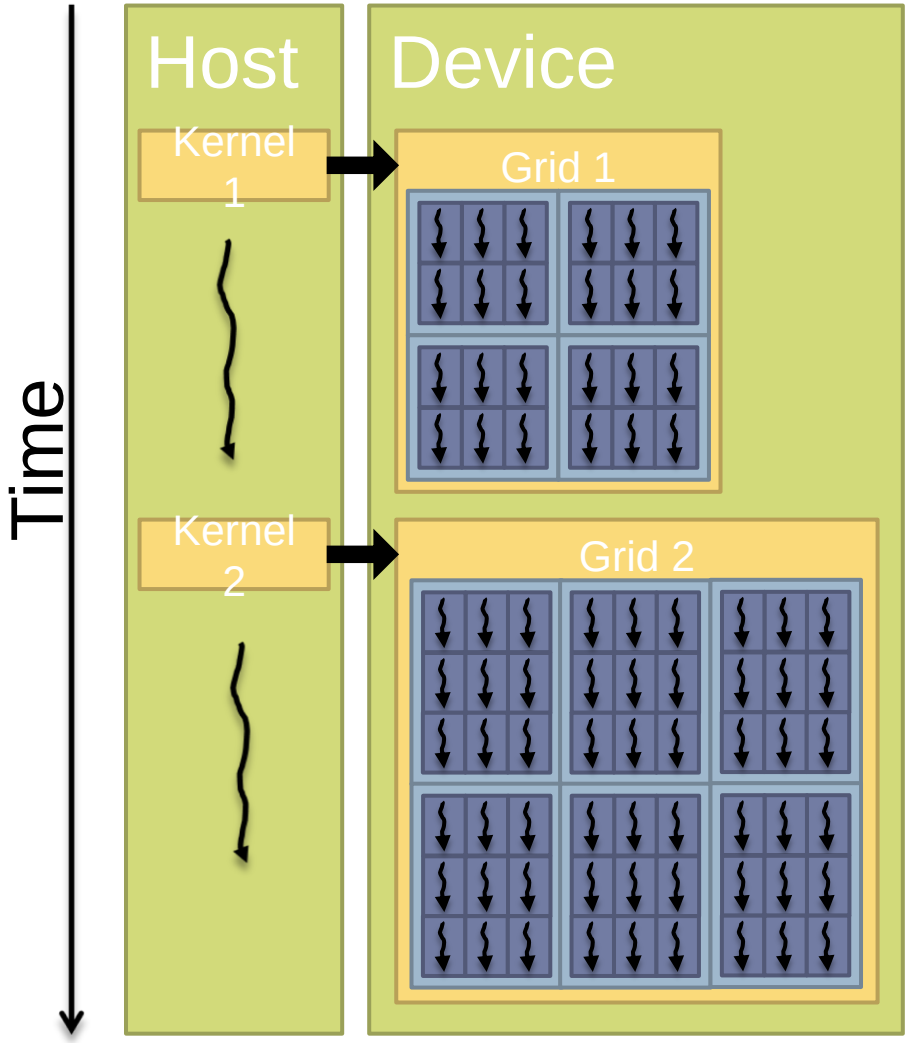


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

CUDA Programming Model

Typical CUDA workflow

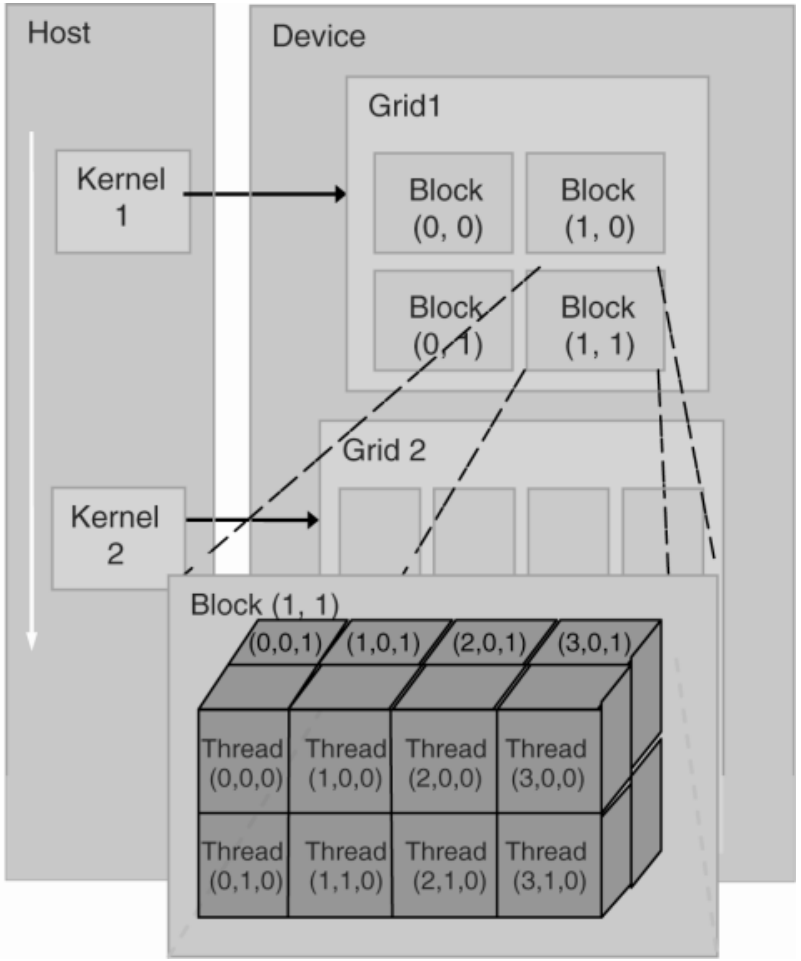
- Allocate memory in device DRAM
- Copy initial data to device
- Process data by one or more kernel calls
- Copy back results to host DRAM
- Free allocated device DRAM



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

CUDA Threads

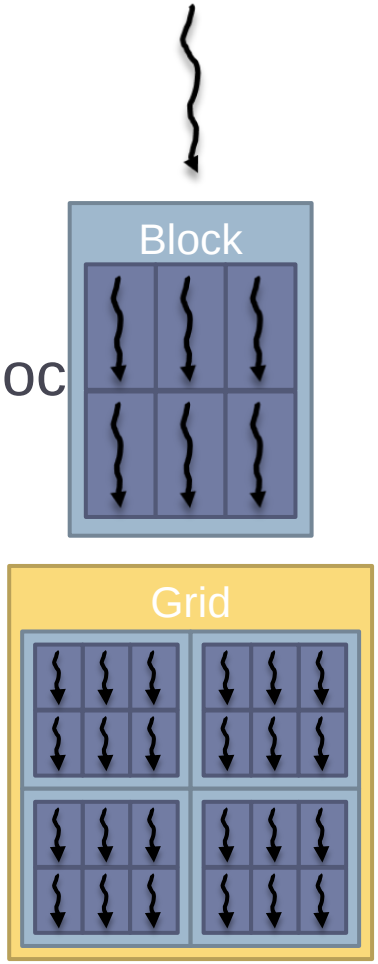
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

CUDA Programming Model
Memory Model

- Private local memory
  - Per thread
  - Registers or in global memory
- Shared memory
  - Shared by all threads within a bloc
  - On-Chip
- Global memory
  - Accessible by all threads
  - Persistent across kernel calls
  - Special memory:
    - Constant memory (cached)
    - Texture memory (cached)

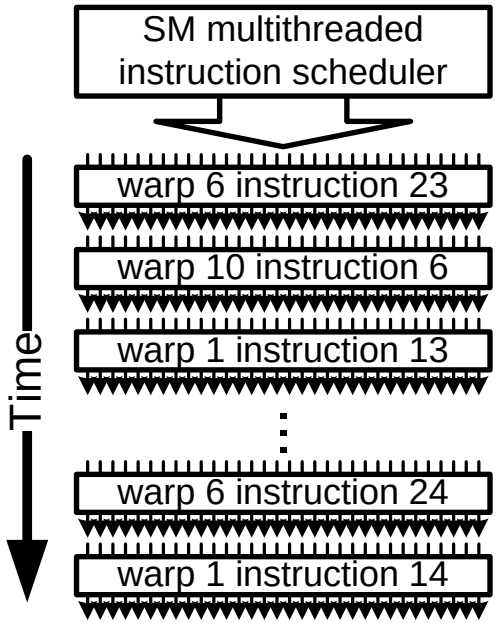


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

CUDA Hardware Architecture

Warps

- Threads are organised into warps
- Warp consists of 32 Threads (on current hardware)
- Divergent threads only occur within warps
- Switching between warps comes at no cost, because all threads have their own set of registers
- Care needs to be taken when accessing shared or global memory by threads of a warp



Taken from NVIDIA

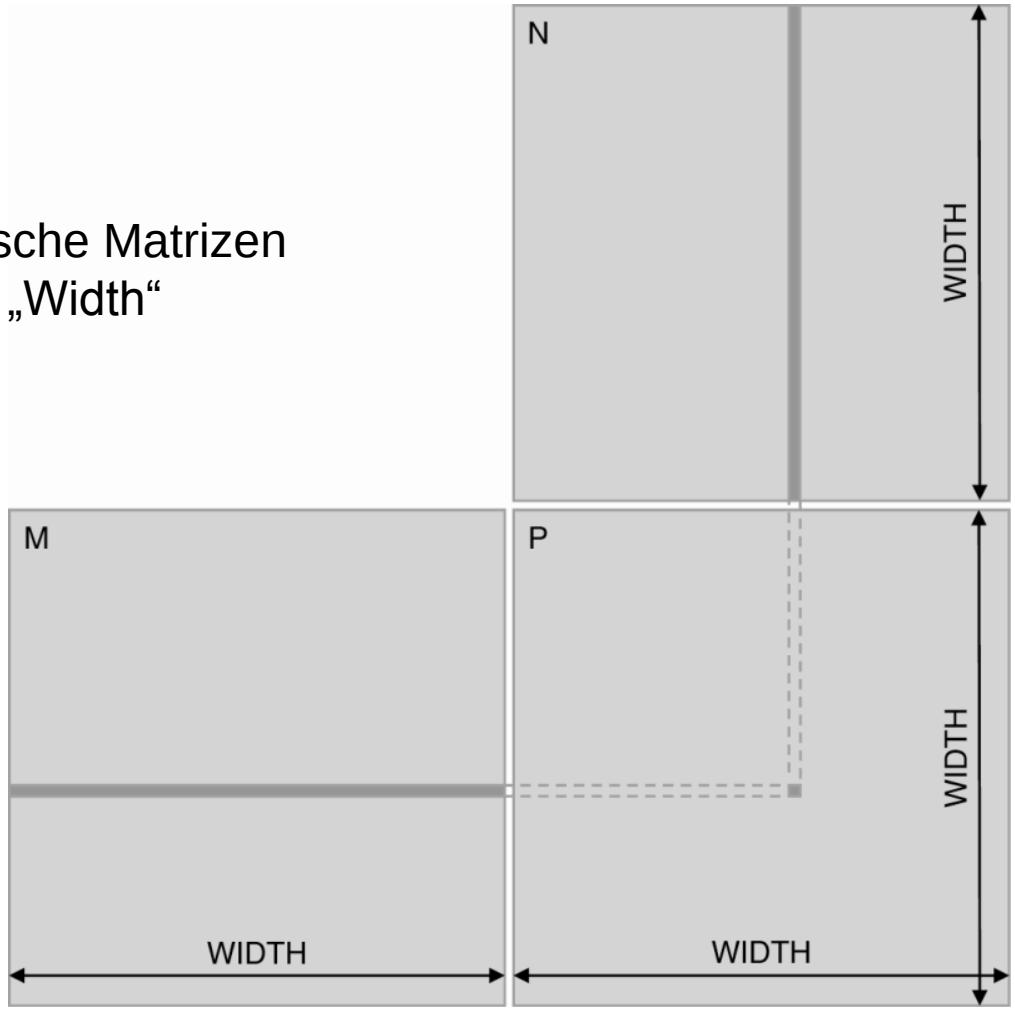
Michael Vetter
michael.vetter@rrz.uni-hamburg.de

Limitations of CUDA

- Requires specific problem type:
  - Enough data parallelism
  - Enough data to work on
  - Enough computation per data element
- Often data transfer between host and device DRAM is limiting factor
  - Keep data on GPU
- Fine tuning is done on a very low programming level
  - Need to understand hardware
  - GPU hardware is simpler than CPU
  - Hard to maintain

Beispiel: Matrix-Matrix Multiplikation

2 quadratische Matrizen
der Größe „Width“



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

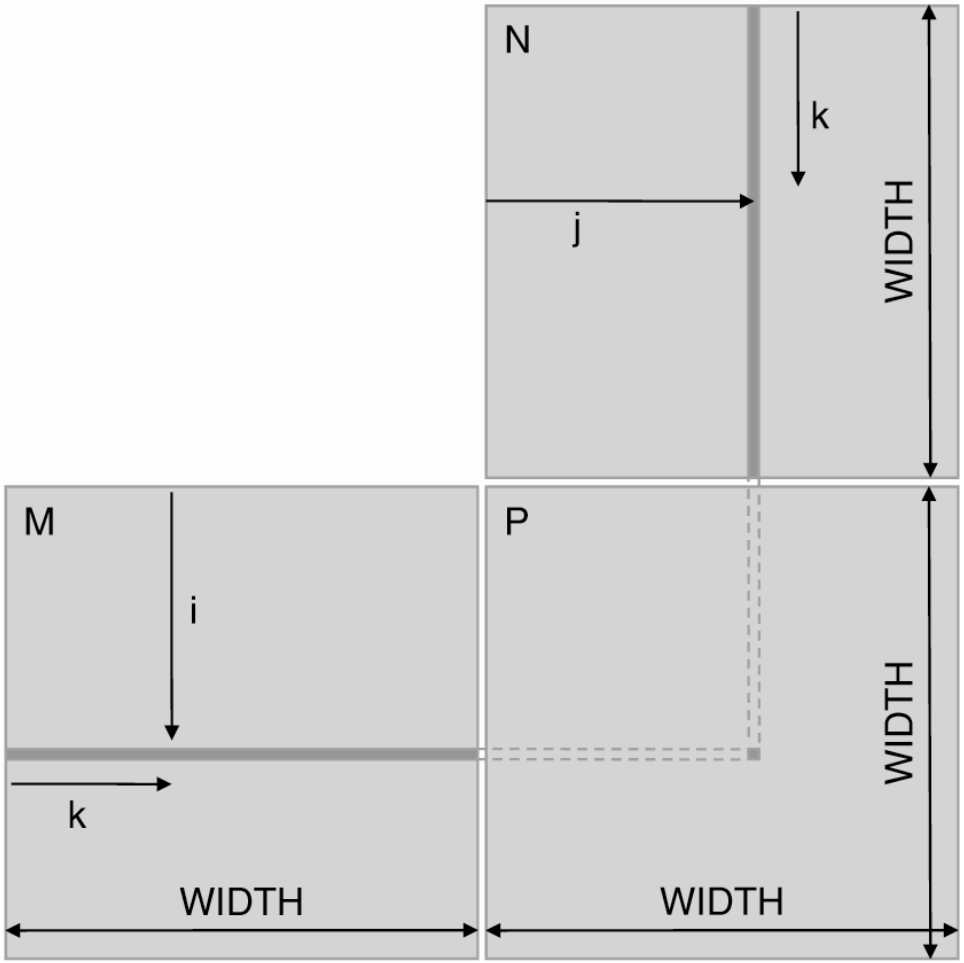
Main Function

```
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;
}
```


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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;
        }
}
```



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

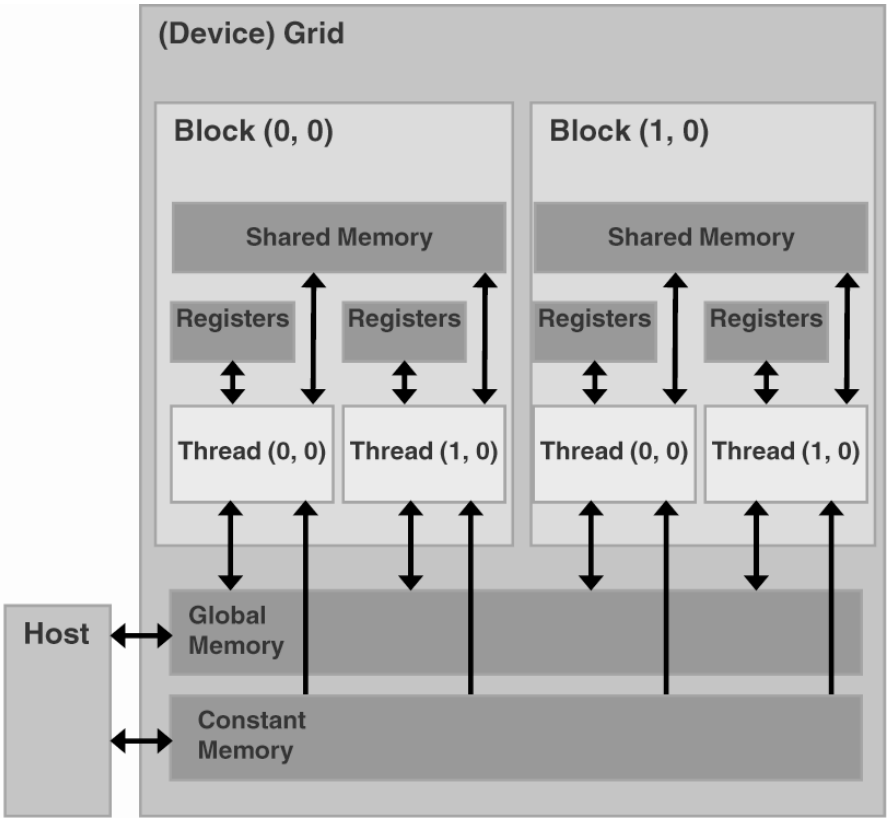
CUDA Implementierung

```
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
}
```


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

Exkurs: CUDA Device Memory Model

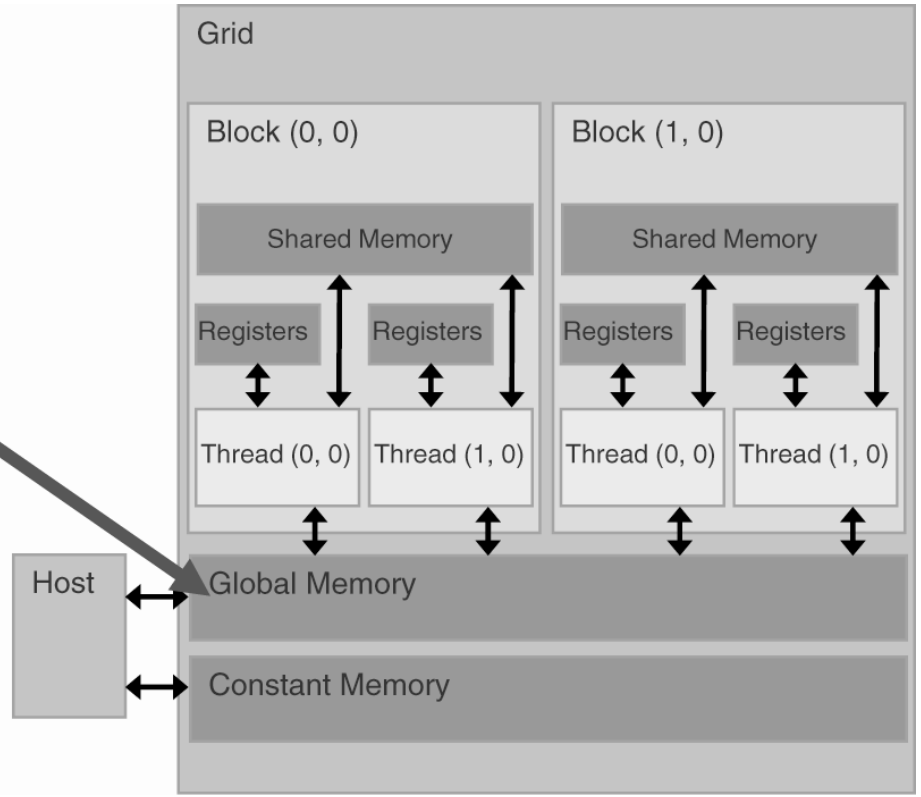
- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

Exkurs: Speicher Management

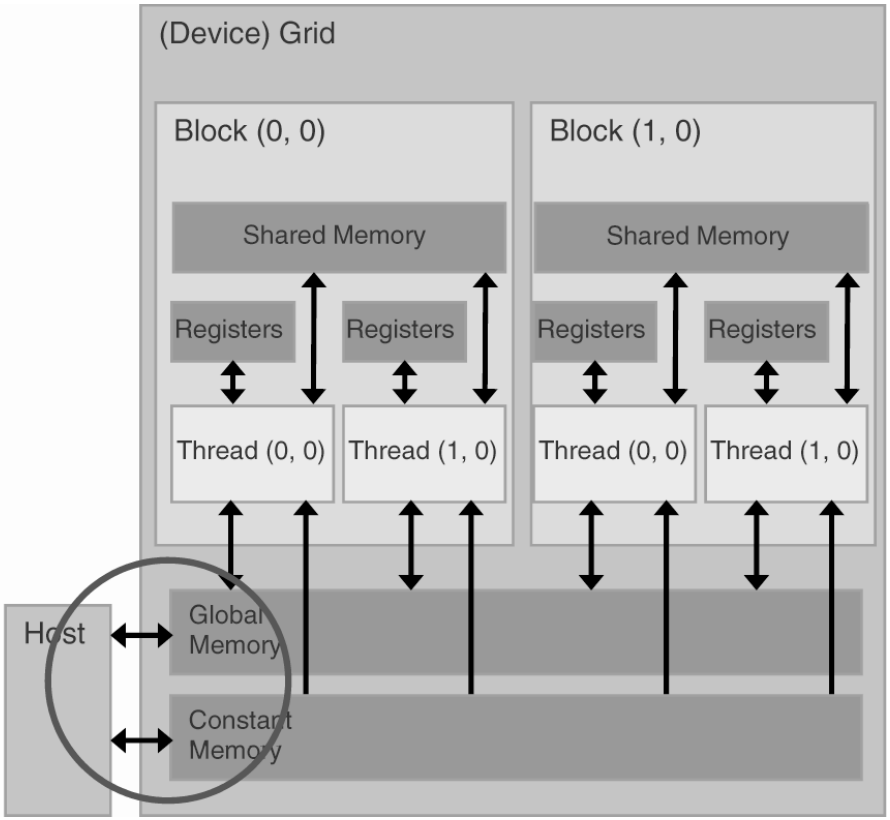
- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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);
   }
```


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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;
}
```


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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);
```


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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

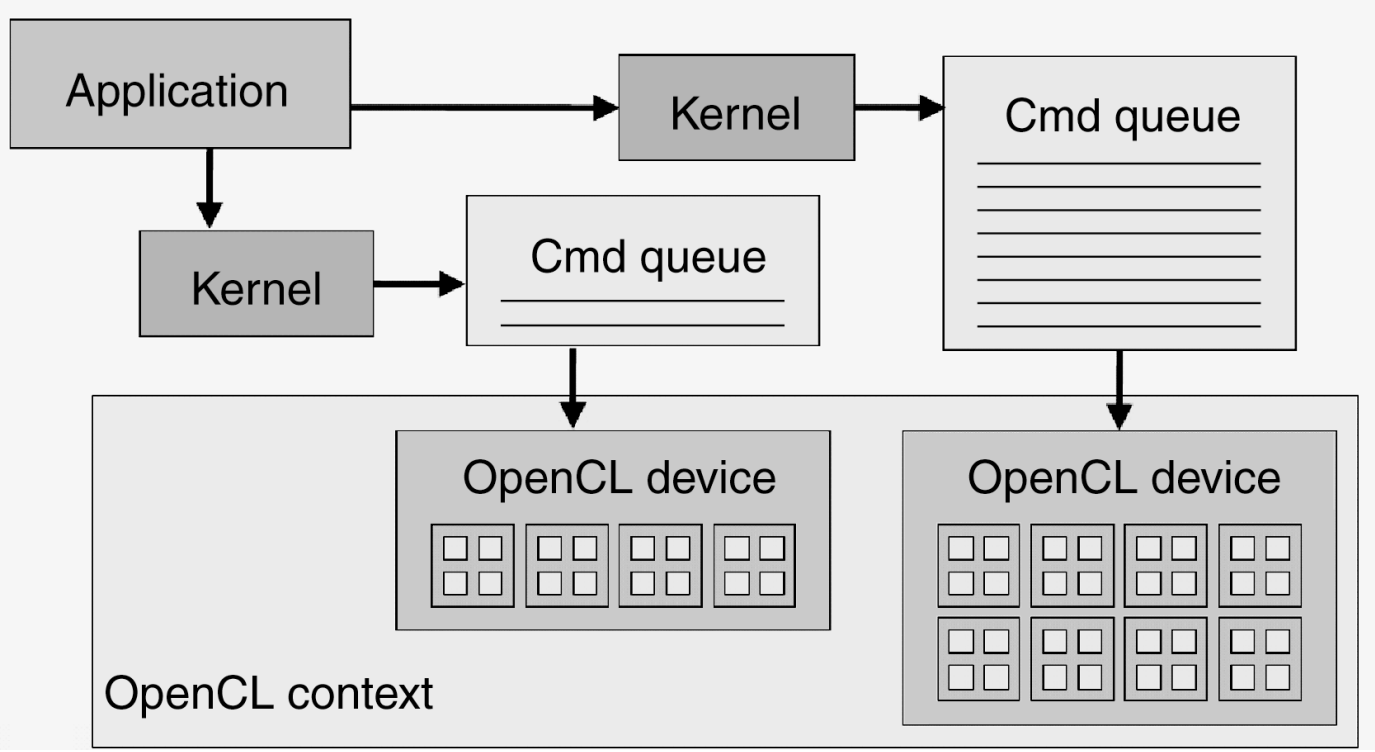
Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

OpenCL Context



Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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 Buffer
  buffer = clCreateBuffer(context, CL_MEM_COPY_HOST_PTR,
  sizeof(cl_float)*10240, data, NULL);
```


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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
```


Michael Vetter
michael.vetter@rrz.uni-hamburg.de

Einsatz von GPUs im HPC

- Top100 (11/2012)
  - 50 Systeme mit Nvidia/Tesla
  - 3 System mit ATI (22)

- Einsatzgebiete:
  - Astronomie und Astrophysik
  - Biologie, Chemie
  - Finanzwirtschaft

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

GPU Cluster in der Top100 (11/2012)

Titan (Top500 Platz 1)

R_{max} 17.590 R_{peak} 27.112 (TFlops)

Tianhe-1a (Top500 Platz 8)

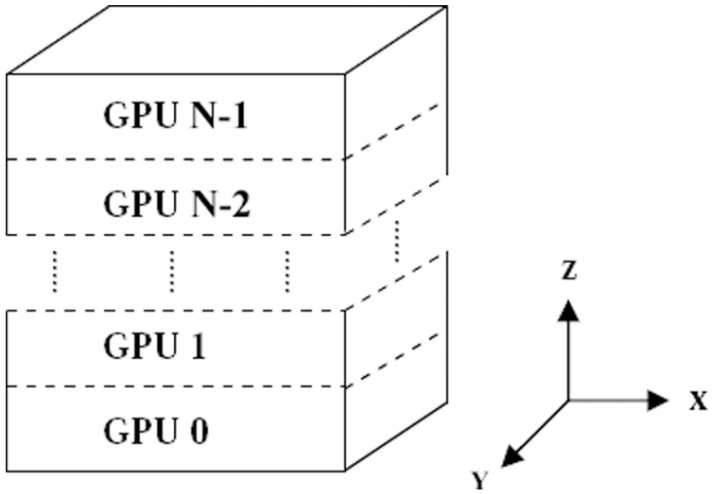
R_{max} 2.566 R_{peak} 4.701 (TFlops)

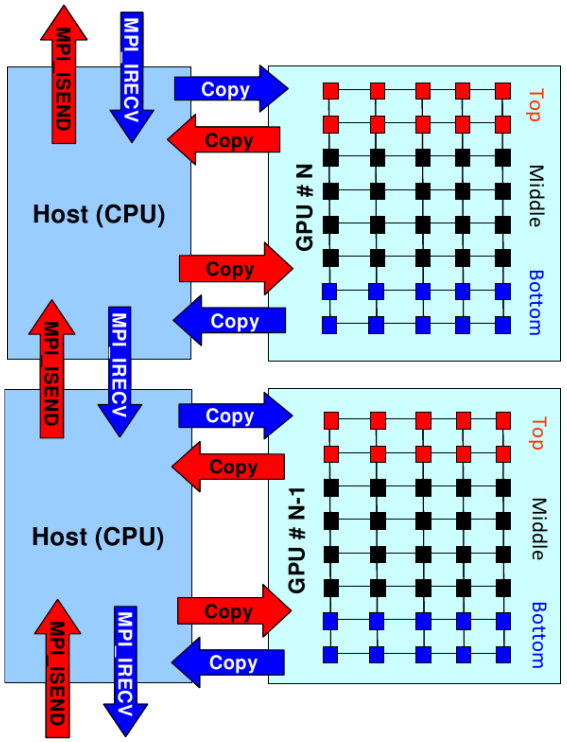
Cray XK7 , Opteron 6274 16C 2.200GHz, NUDT TH MPP, X5670 2.93Ghz 6C,
Cray Gemini interconnect, NVIDIA K20x Nvidia GF104

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

GPU Cluster

- MPI zur Kommunikation der Knoten untereinande
- Sehr gute Auslastung und Effizienz/Strom Verhältnis
- Schwierig: Umsetzung und Optimierung





Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

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

Michael Vetter
michael.vetter@rrz.uni-hamburg.de

GPU-Computing am RRZ/SVPP

- Arbeitsgruppe:
  - Scientific Visualization and Parallel Processing der Informatik
- GPU-Cluster:
  - 96 CPU-Cores und 24 Nvidia Tesla M2070Q verteilt auf 8 Knoten
  - DDN Storage mit 60 TB RAW Kapazität
  - ausschließlich zur Entwicklung und Erprobung, kein produktiver Betrieb
- Forschung:
  - Entwicklung von Verfahren zur Visualisierung wissenschaftlicher Daten
- Lehre:
  - Vorlesung „Datenvisualisierung und GPU-Computing“ von Prof. Dr.-Ing. Olbrich im Sommersemester

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
  - OpenACC
  - C++ AMP

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