Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

GPU-Computing

Michael Vetter

Universität Hamburg

Scientific Visualization and Parallel Processing @ Informatik

Climate Visualization Laboratory @ Clisap/CEN

Übersicht

Hintergrund und Entwicklung von GPGPU

Programmierumgebungen & Werkzeuge (CUDA)

Programmierbeispiel (Matrix-Matrix Multiplikation)

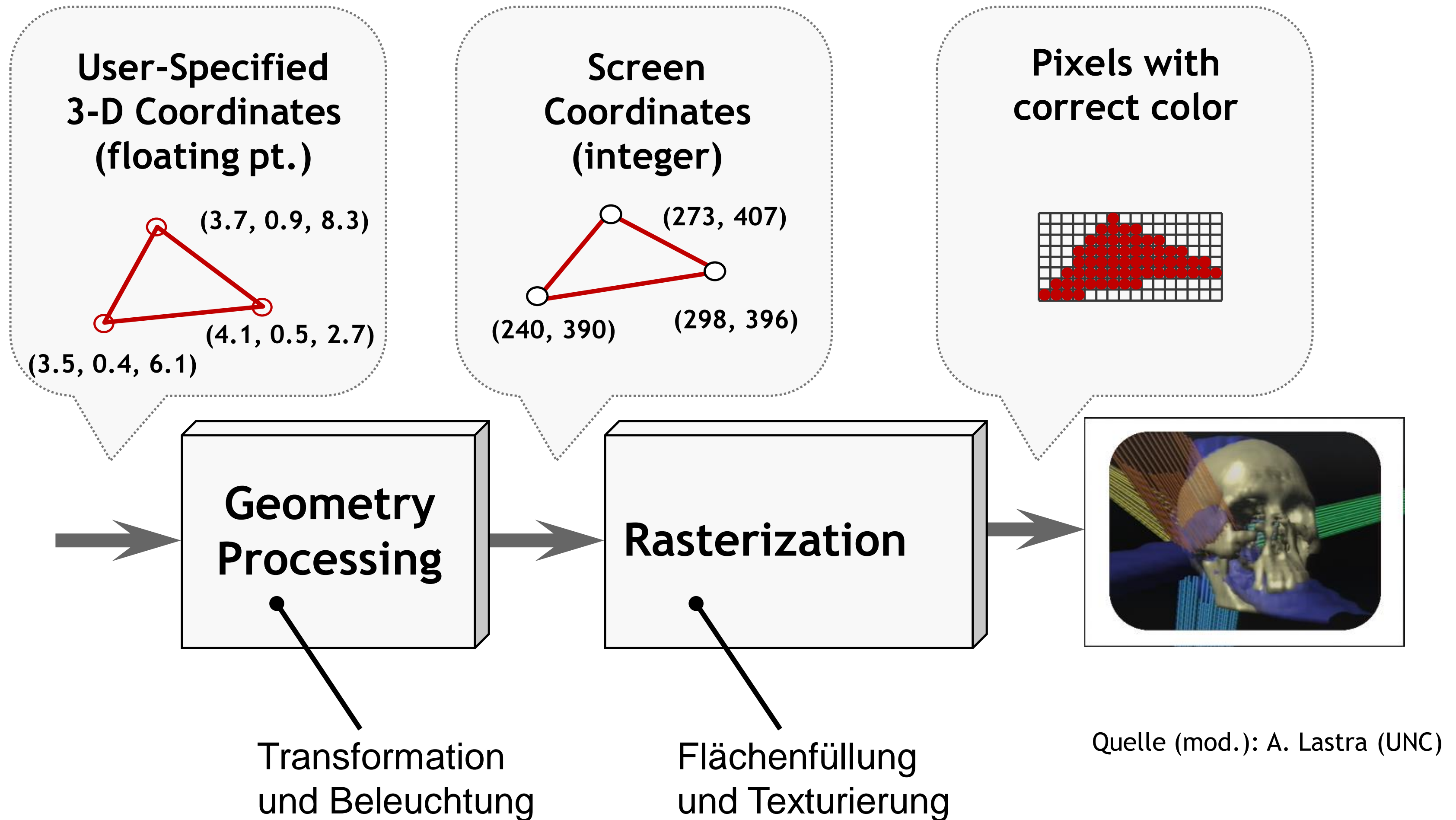
Einsatz von GPUs im Hochleistungsrechnen

Referenzen

Michael Vetter
michael.vetter@uni-hamburg.de

Computergrafik

Was ist Rendering

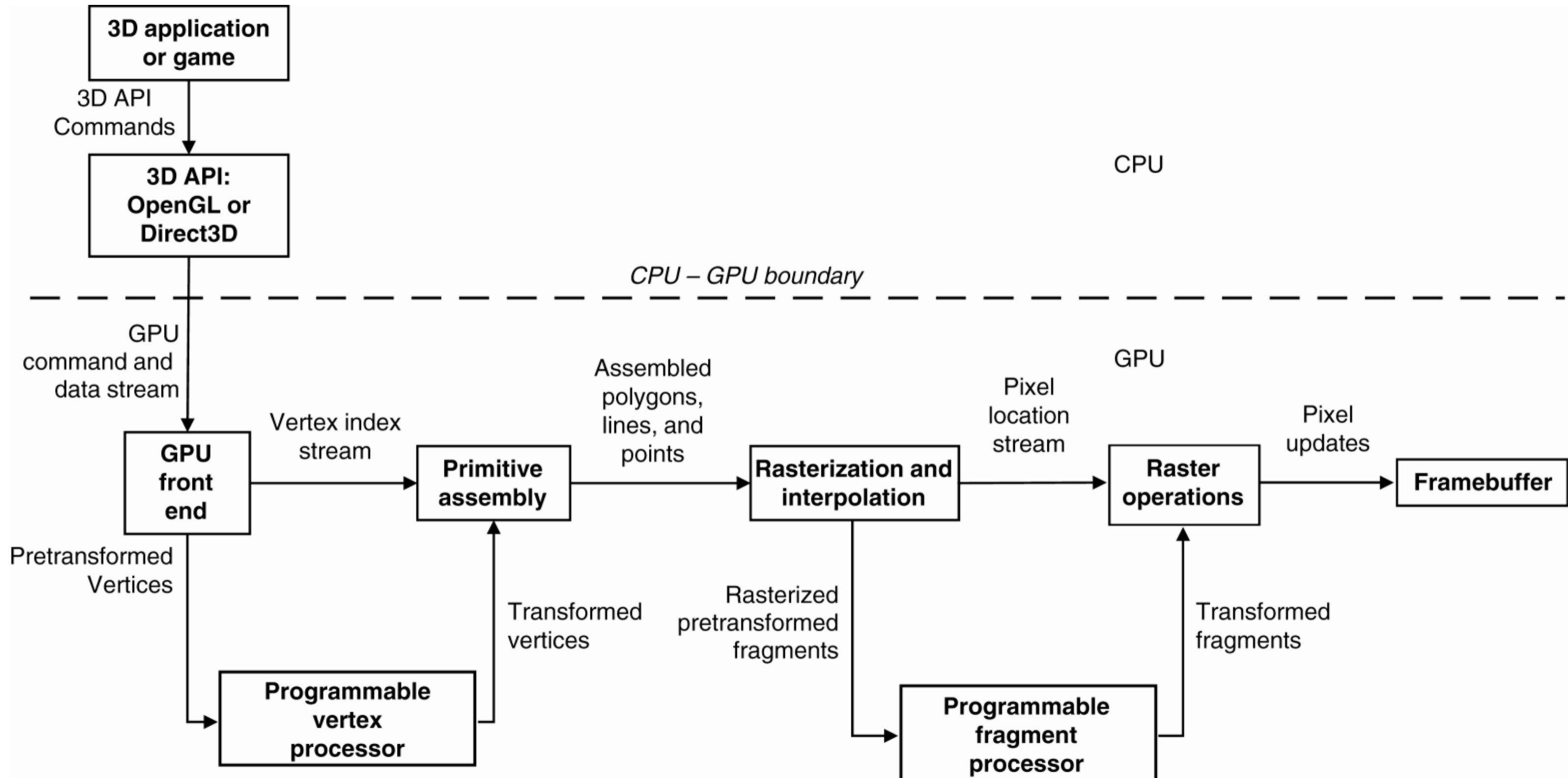


Quelle (mod.): A. Lastra (UNC)

Hintergrund

Bis 2000 bei Graphikkarten „fixed function pipeline“

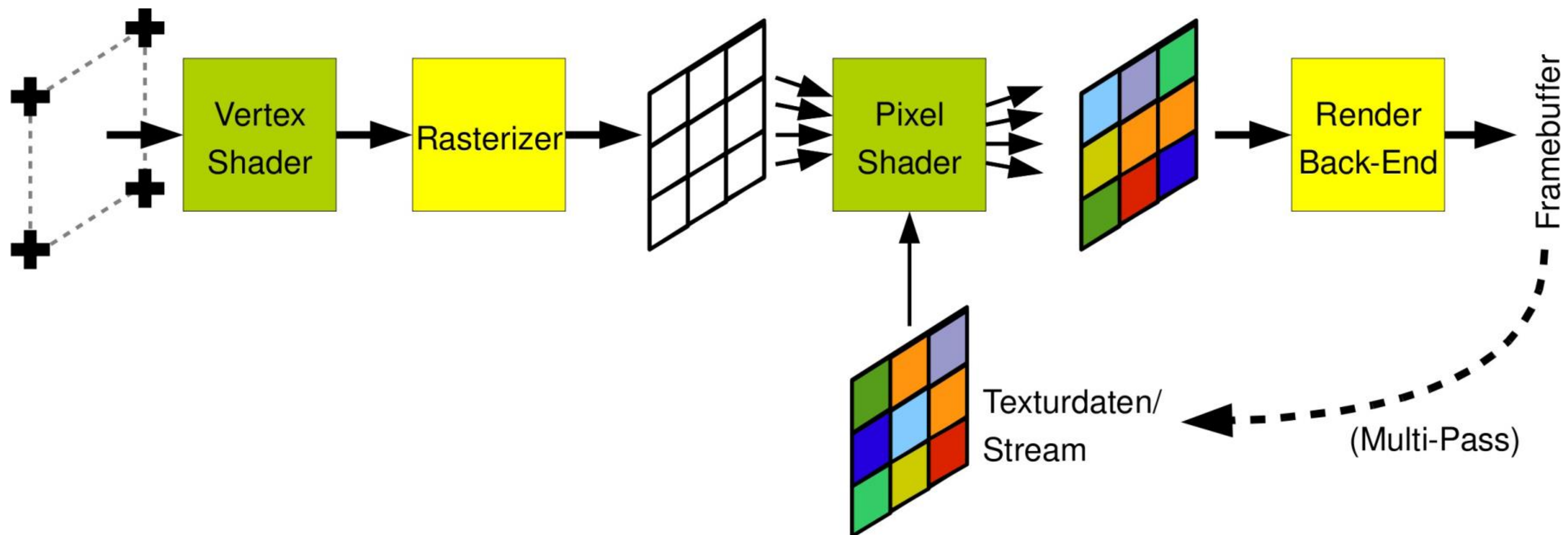
Transform & Lighting Engine (Geforce256, 99/00)



General Purpose Graphics Programming

Programmierung auf Basis der Graphik Pipeline

- Datenarrays in Texturen
- Filter als Fragment Shader
- Stream Programming



GPGPU Beispiel – DSP

CPU **GPU**

Weiterentwicklung

Nutzung der GPU als Parallelprozessor

Vorangetrieben durch Universitäten und Forschung

Weiterhin alles Graphik basiert (OpenGL)

Entwicklung von Hochsprachen:

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

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 … in GLSL (fragment shader)

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

Operationen erfordern Iteration:
for-Schleife (serielle Ausführung)

```
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

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)

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

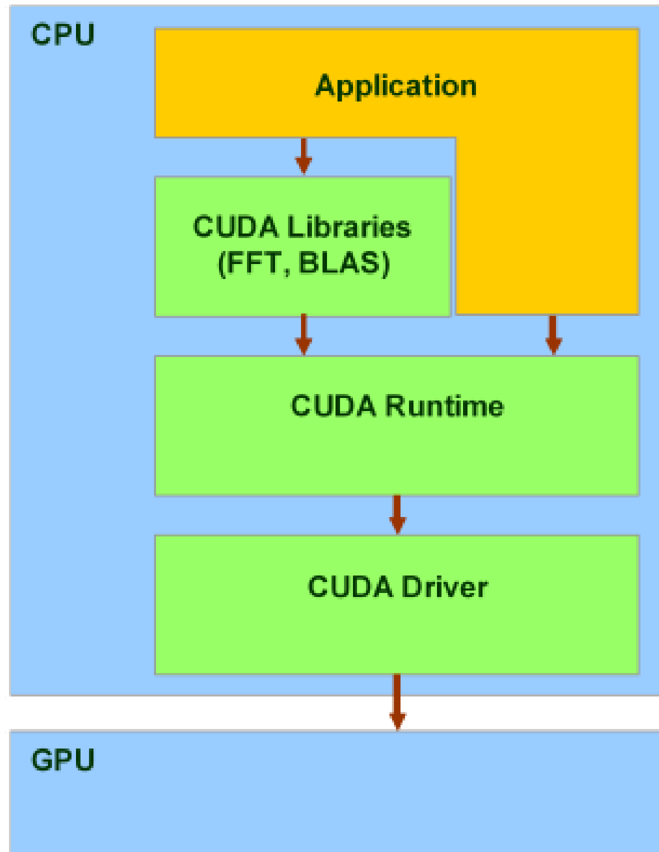
Highlevel Entwicklung in C/C++, Fortran, …

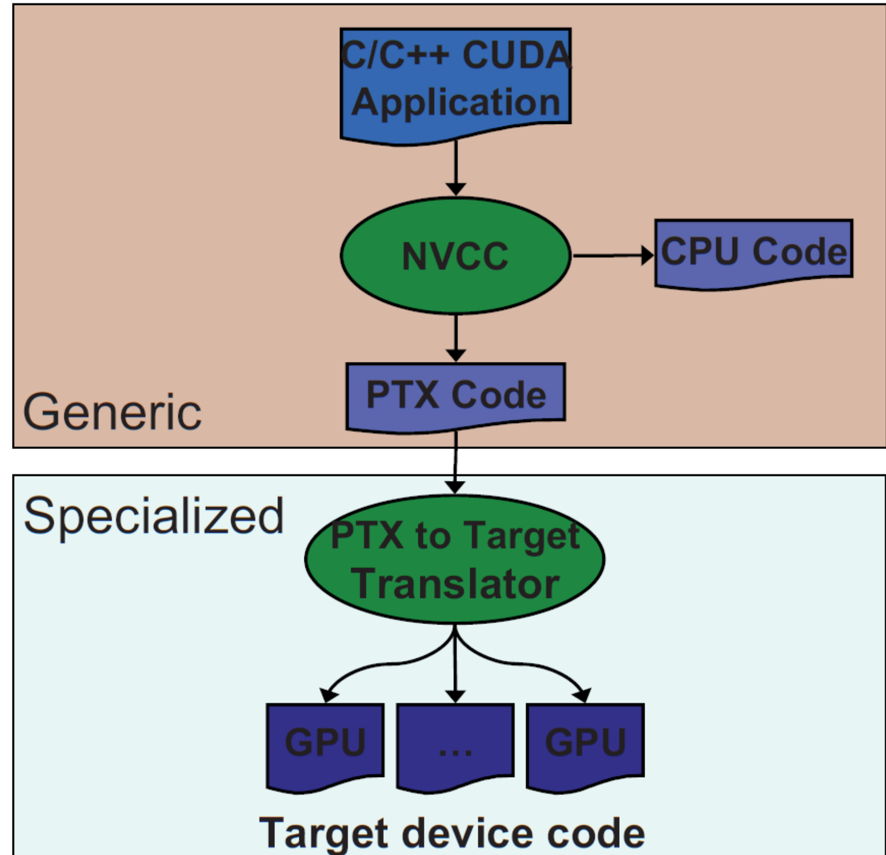
	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

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)

Codegenerierung





Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

GK110 Architektur (Keppler)

SMX-Processor

15 SMX-Prozessoren

- 192 CUDA Kerne
- Double-precission units
- 32 Load / Store Units
- 32 special functions units (sin, cos, sqr, …)
- 64k shared memory / L1 Cache
- Concurrent Thread Execution
- ECC Speicher

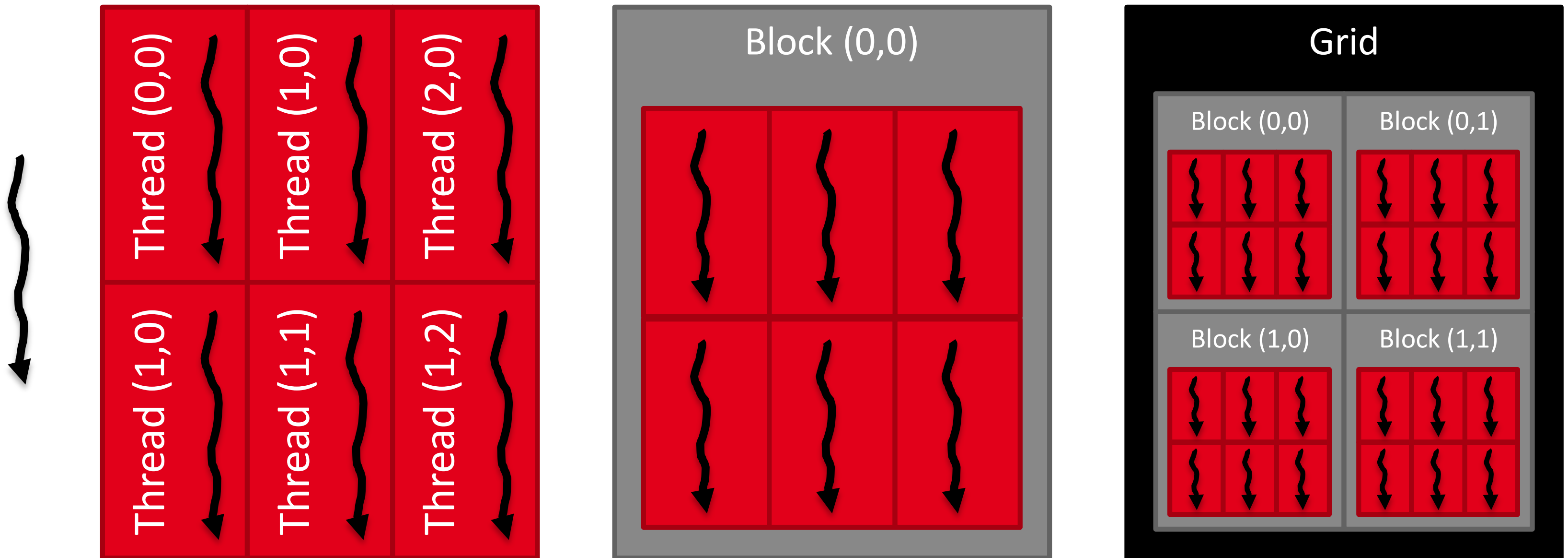
Vergleich Fermi / Keppler

Merkmale	Tesla M2090	Tesla K10	Tesla K20	Tesla K20x	Tesla K40	Tesla K80
Architektur	1 x Fermi	2 x Keppler GK104	1 x Keppler GK110		1 x Keppler GK110B	2x Keppler GK210
Peak Performance DP (Tflops)	0,665	0,19	1,17	1,31	1,66	2,91
Peak Performance SP (Tflops)	1,331	4,577	3,52	3,95	5	8,74
Speicherbandbreite	177 GB/s	320 GB/s	208 GB/s	250 GB/s	288 GB/s	480 GB/s
Speichergröße	6 GB	8 GB	5 GB	6 GB	12 GB	24 GB
CUDA-Cores	512	3072	2496	2688	2880	4992

Michael Vetter
michael.vetter@uni-hamburg.de

CUDA Programming Model

Threads, Blocks and Grids



CUDA Programming Model

Typical CUDA workflow

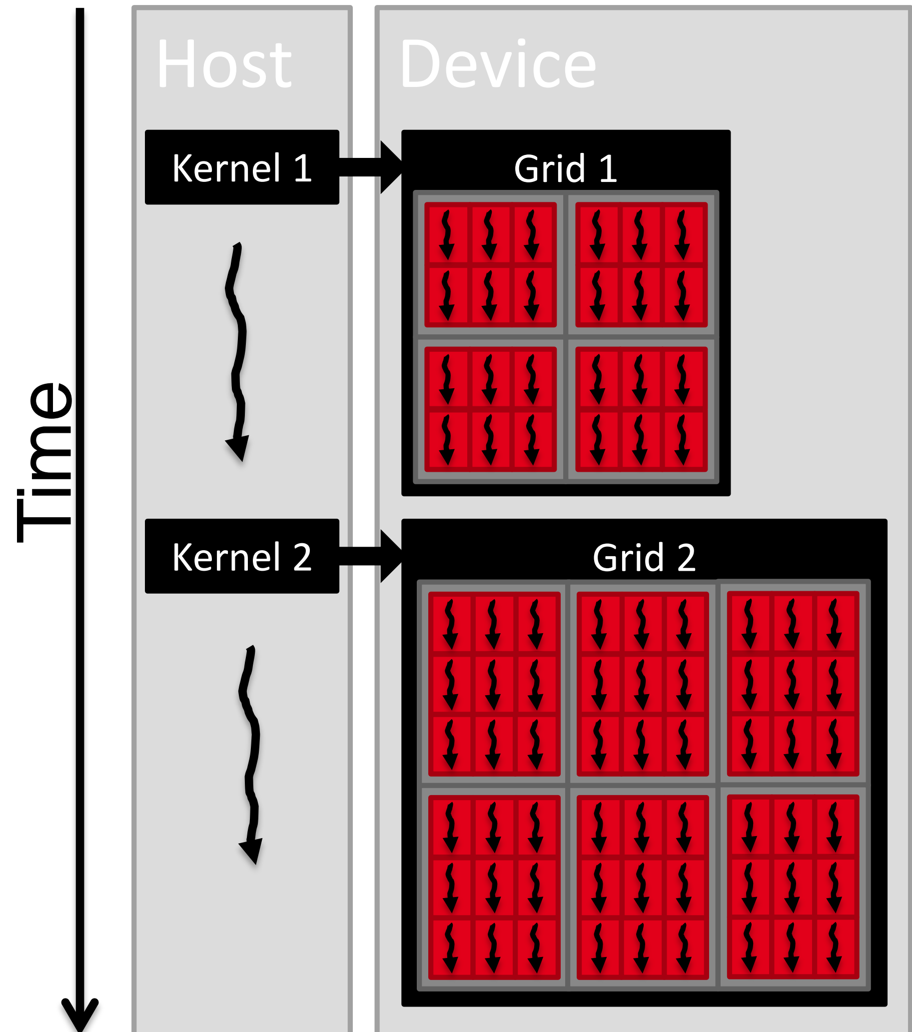
Allocate memory in device DRAM

Copy initial data to device

Process data by one or more kernel calls

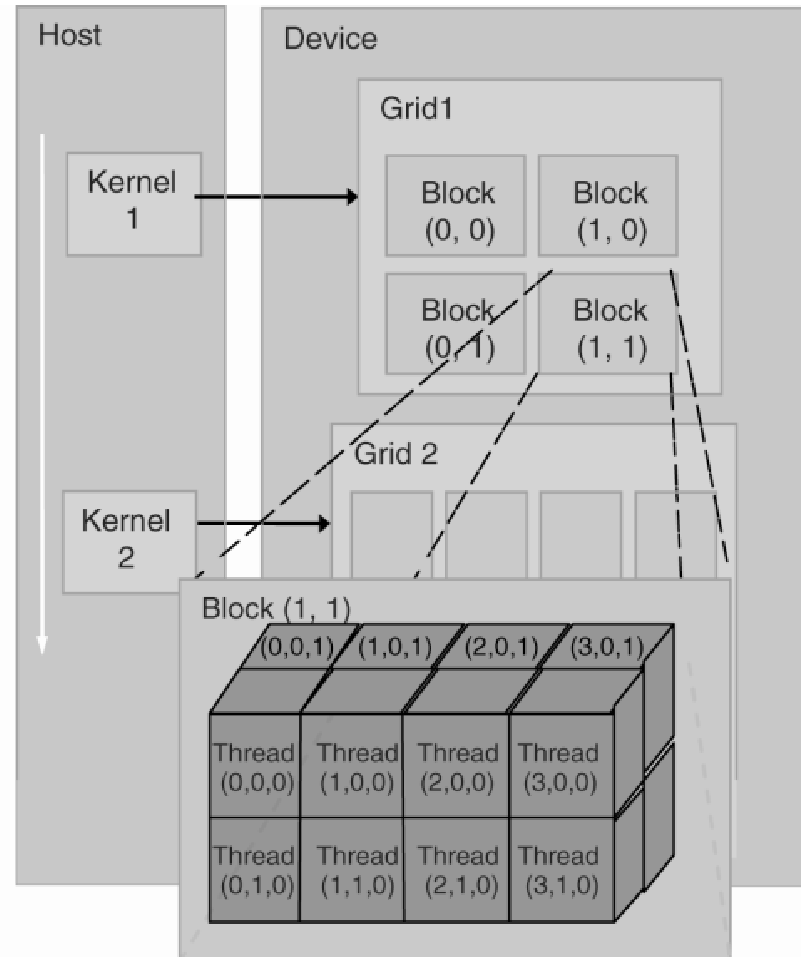
Copy back results to host DRAM

Free allocated device DRAM



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



CUDA Programming Model

Memory Model

Private local memory

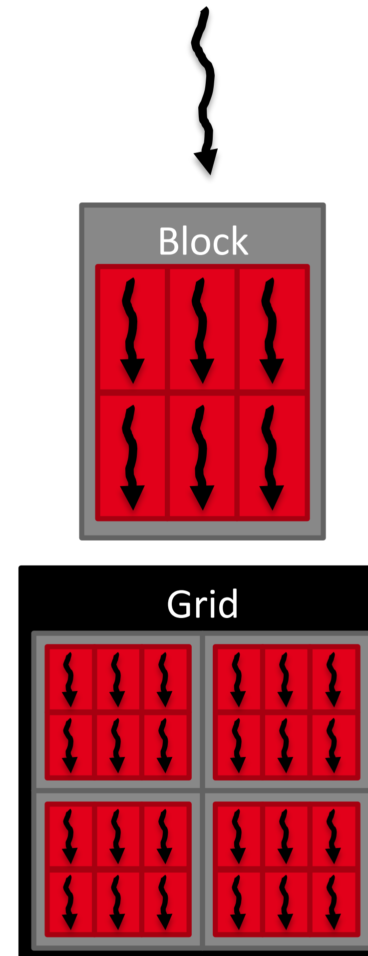
- Per thread
- Registers or in global memory

Shared memory

- Shared by all threads within a block
- On-Chip

Global memory

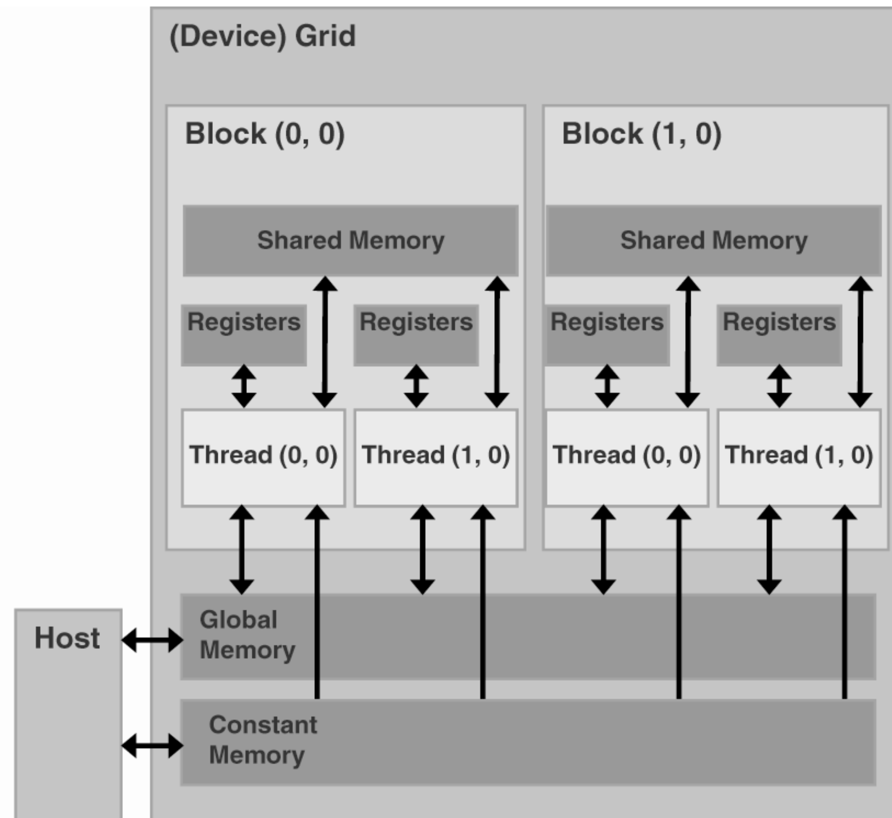
- Accessible by all threads
- Persistent across kernel calls
- Special memory:
  - Constant memory (cached)
  - Texture memory (cached)



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

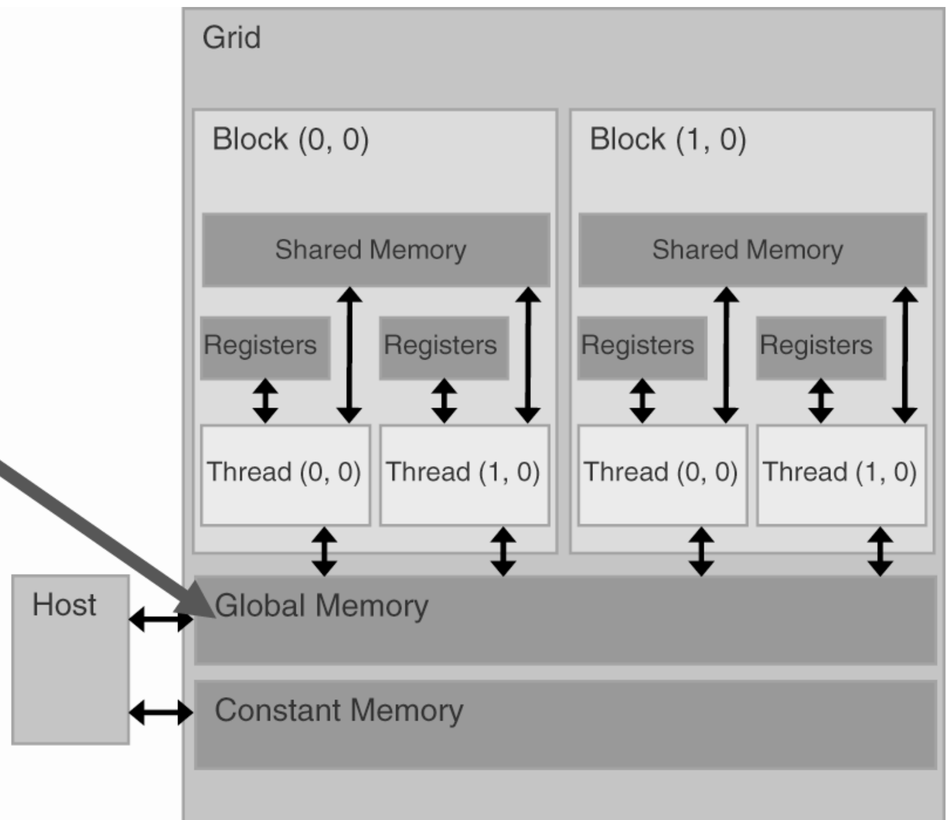
Exkurs: CUDA Device Memory Model

- 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



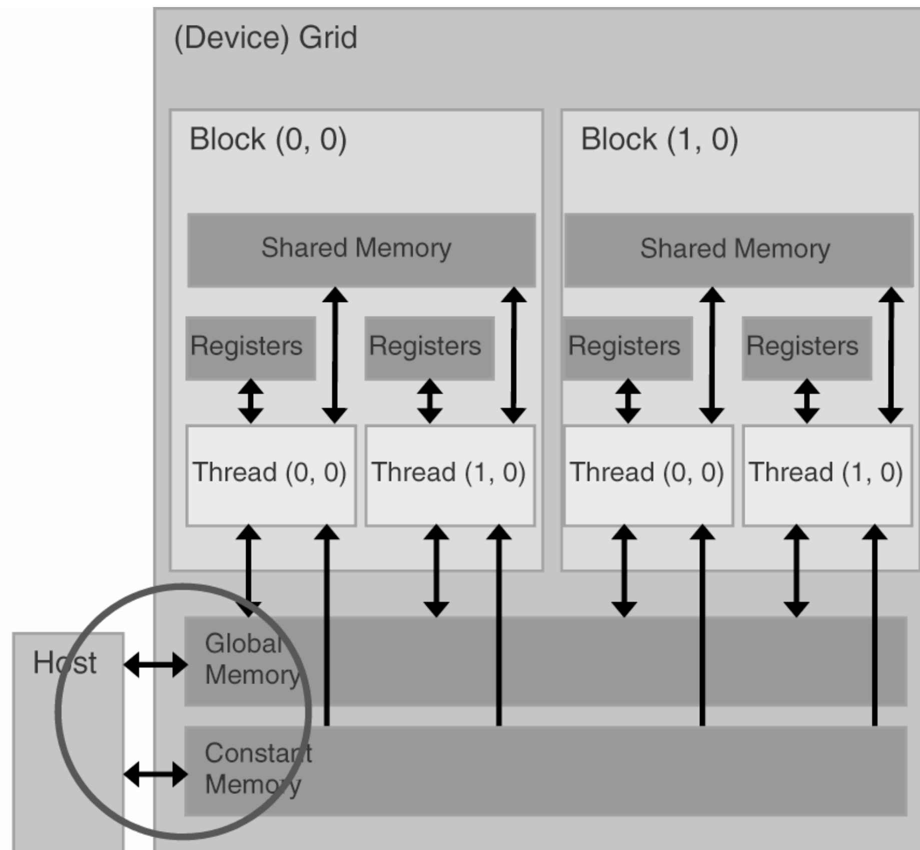
Exkurs: Speicher Management

- 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 Hardware Architecture

Warps

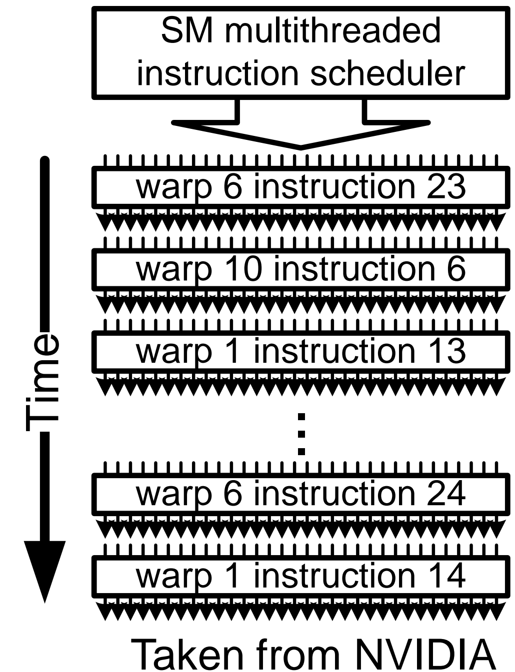
Threads are organised into warps

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

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

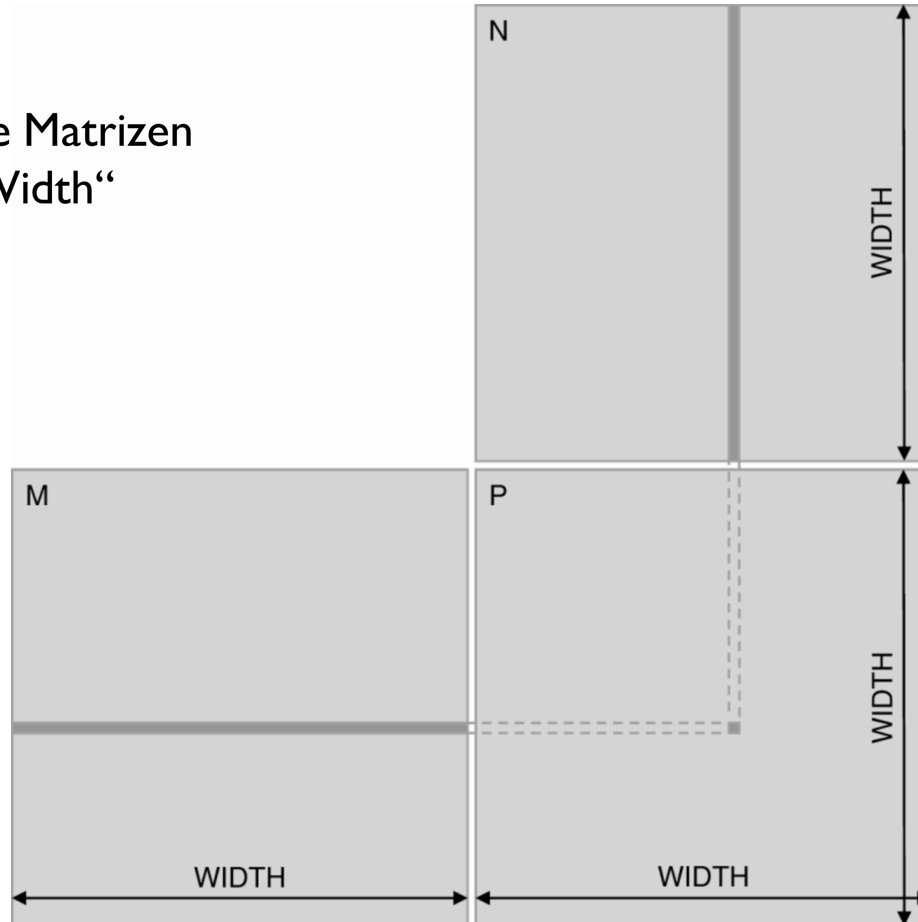
Fine tuning is done on a very low programming level

- Need to understand hardware
- GPU hardware is simpler than CPU
- Hard to maintain

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Beispiel: Matrix-Matrix Multiplikation

2 quadratische Matrizen
der Größe „Width“



Main Function

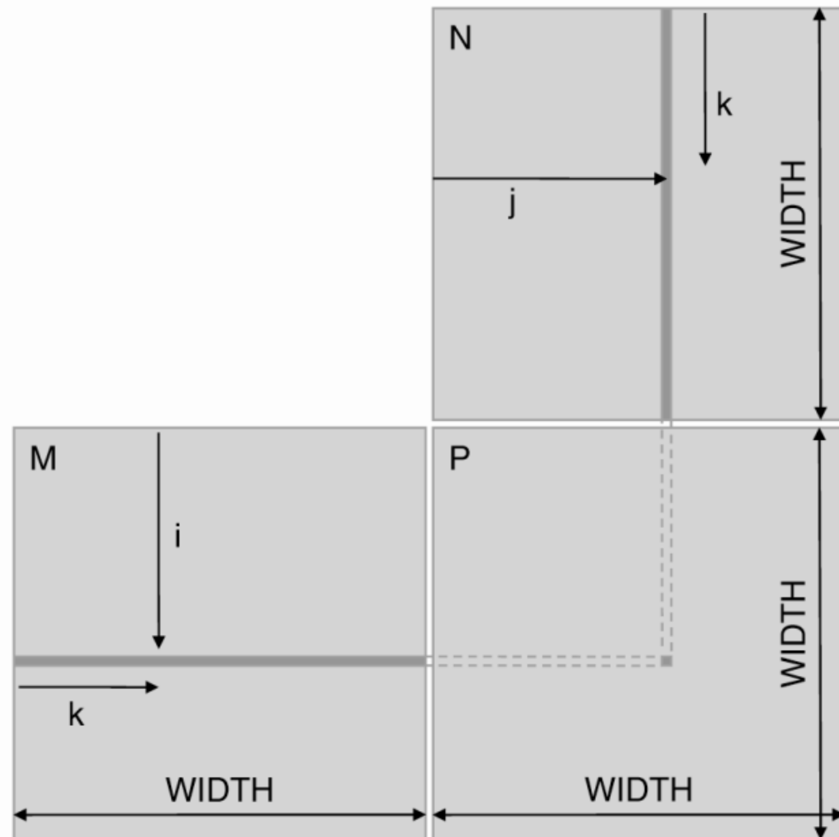
```
int main(void) {
1.  // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N
....

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;
        }
}
```



Michael Vetter
michael.vetter@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
}
```

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

Entwicklung Fermi / Keppler

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

OpenCL

Compute Language für CPUs und GPUs

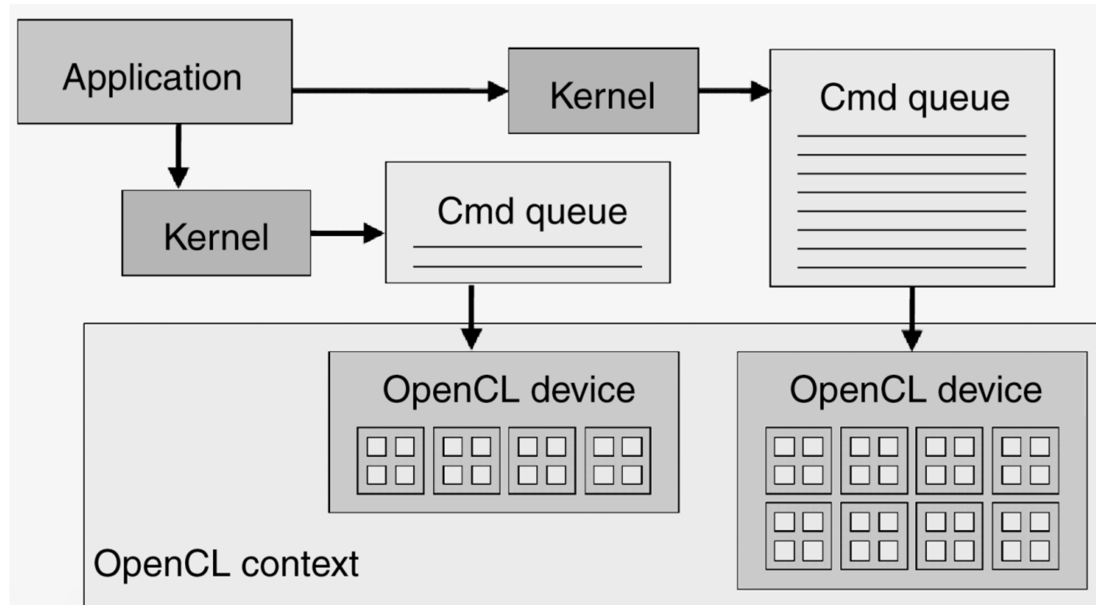
Offener Standard für heterogene Umgebungen

- 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



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

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);
```

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

Top500 (11/2012)

Accelerator/CP Family	Count	System Share (%)
N/A	438	87.6
Nvidia Fermi	48	9.6
Intel Xeon Phi	7	1.4
ATI Radeon	3	0.6
Nvidia Kepler	2	0.4
IBM Cell	2	0.4

Top500 (11/2013)

Accelerator/CP Family	Count	System Share (%)
N/A	447	89.4
Nvidia Fermi	22	4.4
Nvidia Kepler	15	3
Intel Xeon Phi	12	2.4
Hybrid	2	0.4
ATI Radeon	2	0.4

Top500 (11/2014)

Accelerator/CP Family	Count	System Share (%)
N/A	425	85.0
Nvidia Kepler	28	5.6
Intel Xeon Phi	21	4.2
Nvidia Fermi	18	3.6
Hybrid	4	0.8
ATI Radeon	3	0.6
PEZY-SC	1	0.2

Einsatzgebiete:

- Astronomie und Astrophysik
- Biologie, Chemie
- Finanzwirtschaft

GPU Cluster in der Top500 (11/2014)

Titan (Top500 Platz 2)

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

Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x

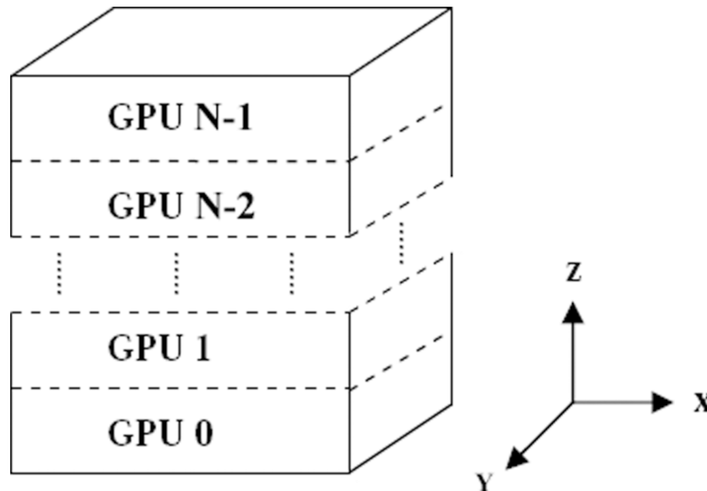
Piz Daint(Top500 Platz 6)

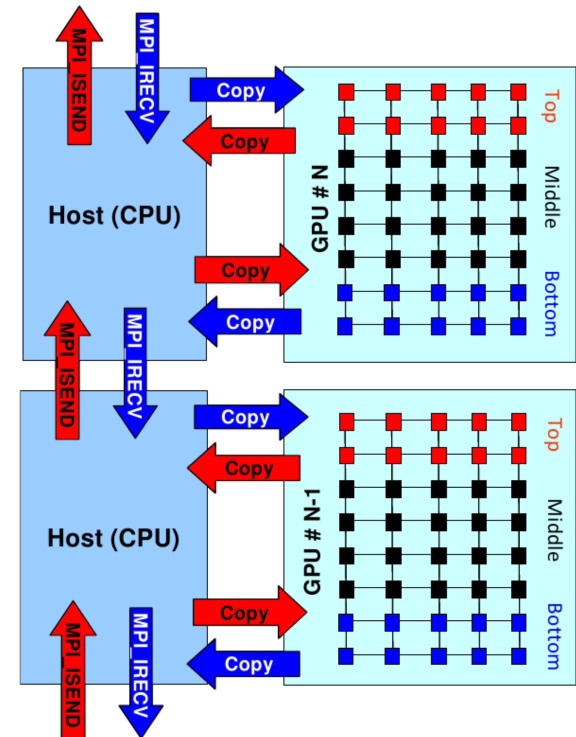
R_{max} 6.271 R_{peak} 7.789 (TFlops)

CRAY XC30, XEON E5-2670 8C 2.600GHZ, ARIES INTERCONNECT, NVIDIA K20X

GPU Cluster

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





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
```

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

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