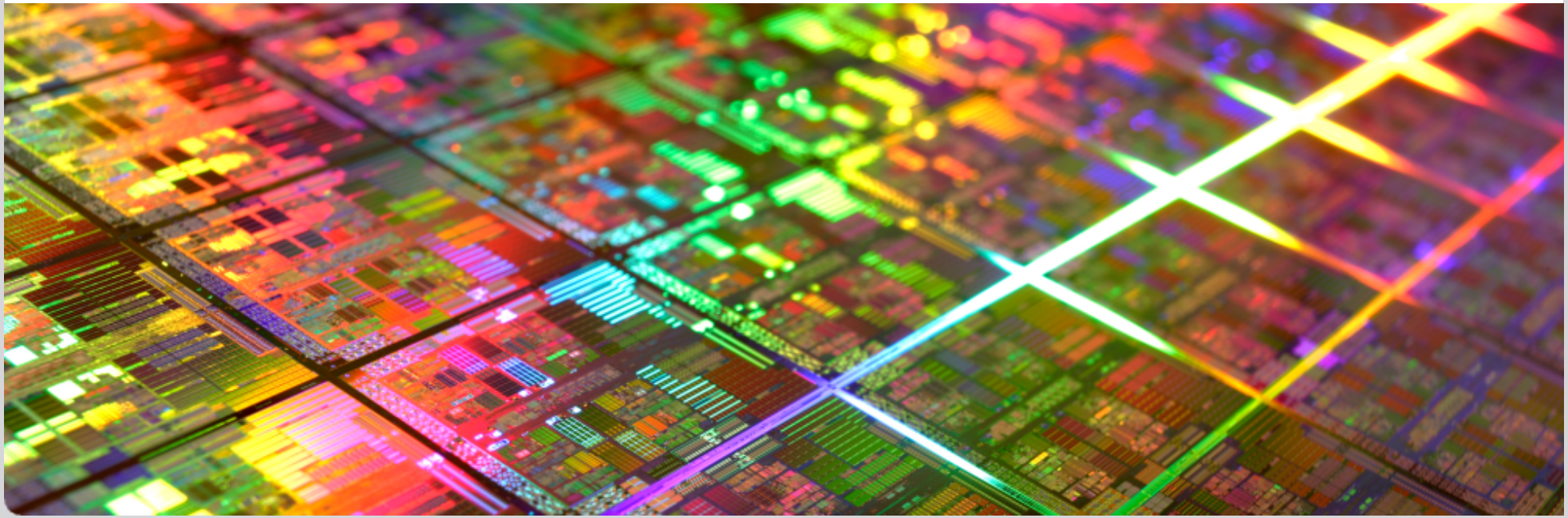
Karlsruhe Institute of Technology

Vorlesung Rechnerstrukturen
Heterogene Parallelsysteme

Mario Kicherer, Prof. Dr. Wolfgang Karl

Chair for Computer Architecture and Parallel Processing
Prof. Dr. Wolfgang Karl

Sommersemester 2015



Vorlesung Rechnerstrukturen

Karlsruhe Institute of Technology

Kapitel 5: Heterogene Parallelsysteme

- Einführung
- Hardware-Beschleuniger
  - Fallbeispiel: NVIDIA Fermi
  - GPGPU-Programmierung
- Forschung am Lehrstuhl

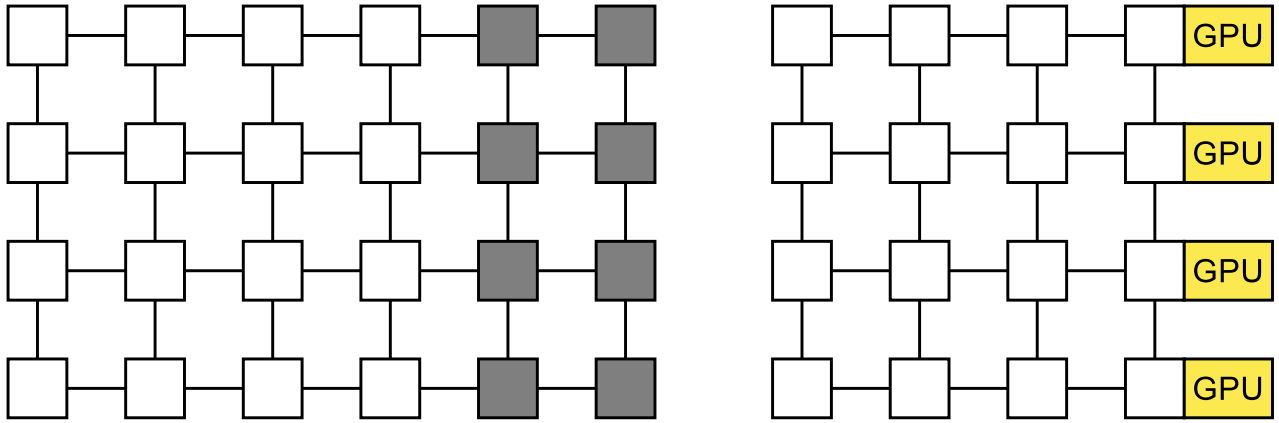
Einführung

Heterogene Systeme:

- Verbund von Einheiten mit unterschiedlichen Eigenschaften
- von Vorteil, wenn ein Einheittyp nicht für alle Aufgaben bzw. Anforderungen gleich gut geeignet ist
- u.a. zur Erhöhung der
  - Leistung (z.B. single-core vs. multi-core performance)
  - Robustheit (z.B. Fehlertoleranz mittels Redundanz)
- In dieser Vorlesung: Systeme mit unterschiedlichen Recheneinheiten
- Heterogenität auf verschiedenen Ebenen möglich

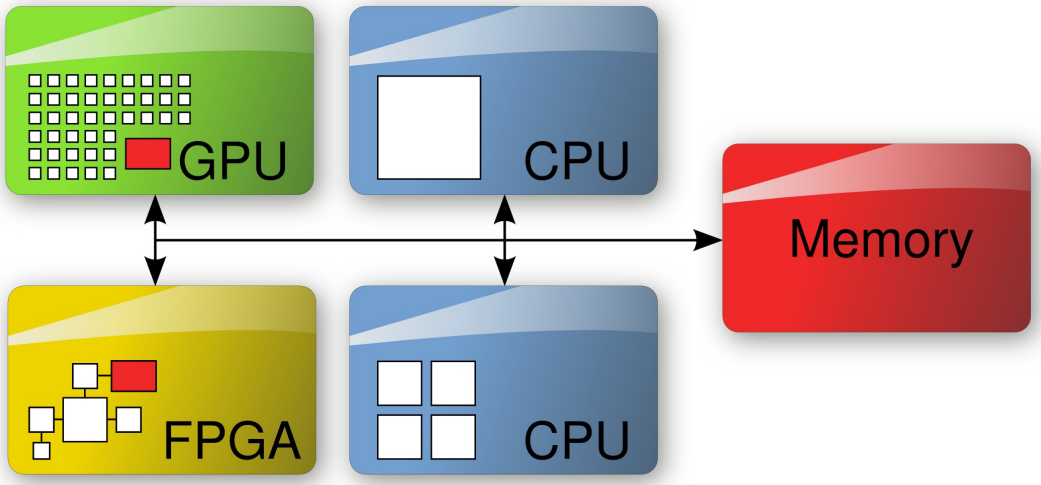
Heterogene Netzwerke

- Knoten mit z.B. unterschiedlicher Hardware oder Betriebssystem
- Beispiel:
  - Neue Aufbaustufe eines Clusters mit besseren Prozessoren
  - Einzelne Beschleunigerknoten (meist GPUs)



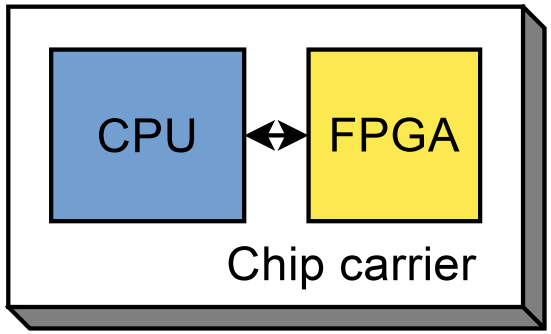
Heterogenität auf Knotenebene

- z.B. unterschiedliche CPUs oder Beschleuniger
- Verbunden z.B. über QuickPath/HyperTransport oder PCIe
- Meist: CPUs in Kombination mit dedizierten
  - Grafikkarten oder
  - FPGAs (z.B. Convey HC-1)



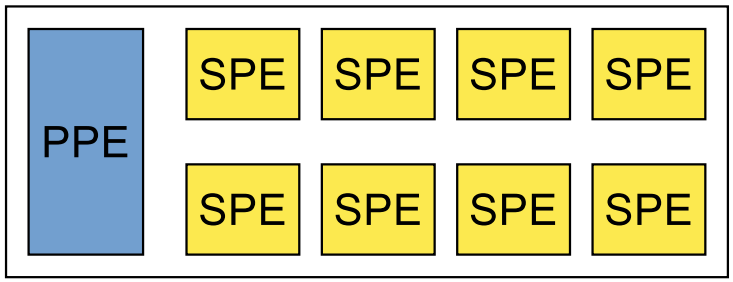
Heterogenität auf Carrier-Ebene

- Separate Dies auf selbem Chip carrier
- Beispiel:
  - Intel Atom E600C Serie mit CPU und FPGA, verbunden über on-chip PCIe

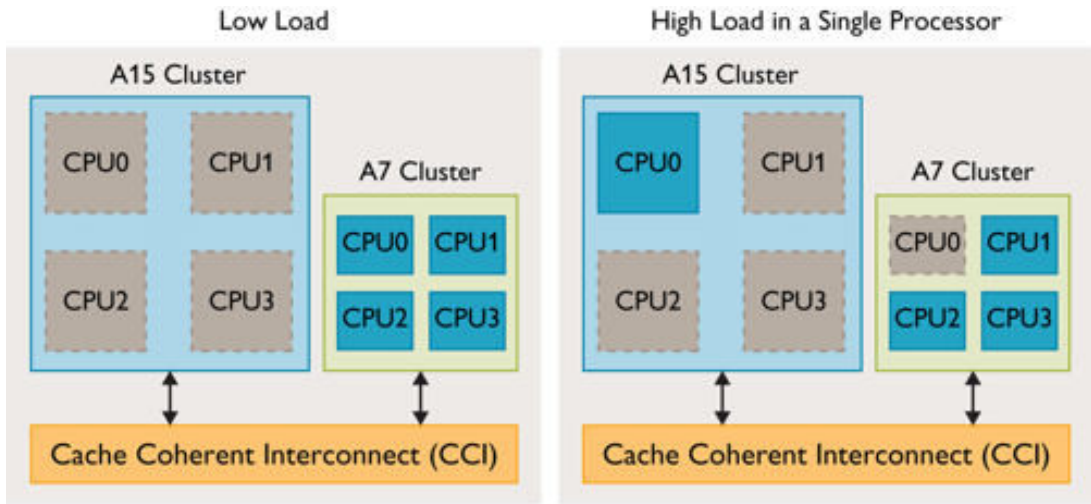


Heterogenität auf einem Die

- z.B. unterschiedliche CPUs oder Beschleuniger (GPUs, DSPs,...)
- Nutzen interne Verbindungsstrukturen
- Beispiel:
  - AMD APUs, Intel CPUs (s.a. Intel TurboBoost)
  - Cell BE (verbaut in Sony Playstation 3)
  - ARM big.LITTLE



Cell BE



Quelle: http://www.arm.com/products/processors/biglittleprocessing.php

Vergleich Einbindung von Beschleunigern

Karlsruhe Institute of Technology

Dedizierte Beschleuniger:

- \+ Beliebige Kombination mit CPU
- \+ Eigener Speicher mit hoher Bandbreite
- – Teure Speichertransfers notwendig
- – Datenkonsistenz Aufgabe des Programmierers

Integrierte Beschleuniger:

- \+ Gemeinsamer Speicher mit CPU
- – Nur begrenzt Platz, daher weniger leistungsfähig
  (Nvidia Kepler: 7 Mrd. vs. 2.4 Mrd. Transistoren für AMD Kaveri)
- – Fixe Kombination von CPU und GPU

Karlsruhe Institute of Technology

Beschleunigerarchitekturen

Beschleunigerarchitekturen

Field Programmable Gate Array (FPGA)

- Rekonfigurierbare Hardware
- Im Prinzip beliebige digitale Schaltungen realisierbar
- Anwendungsgebiete:
  - Anwendungsbeschleunigung (Verlagerung von Code „in Hardware“)
  - Funktionstest realer Hardware (z.B. von Prozessoren)
  - ...
- Programmierung meist mittels VHDL oder Verilog
- Hochsprachige Ansätze vorhanden, z.B. Maxeler, OpenCL

Bei Interesse: Basispraktikum am Lehrstuhl

Quelle: http://hothardware.com/news/nvidia-offers-peak-into-advanced-design-evaluation

Beschleunigerarchitekturen

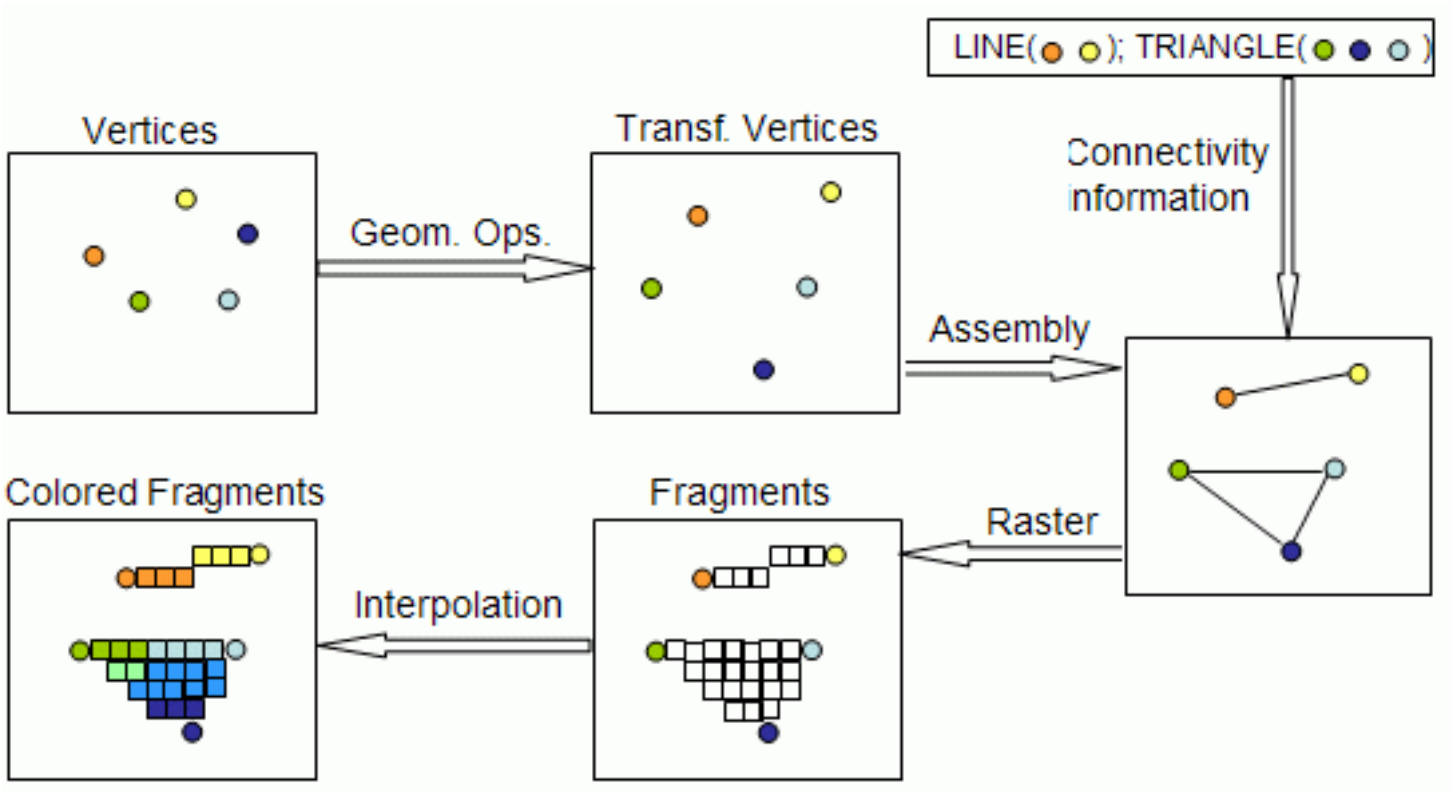
General-Purpose computation on Graphics Processing Units (GPGPU)

- Allzweckberechnungen auf Grafikkarten
- Nutzung der hohen Anzahl an Kernen für parallele Anwendungen
- Eingeführt $\sim$ 2002
- Erste Verwendung von GPUs für Allzweckberechnungen mittels Shader-Programmen

GPU shader

- Programmierbare Einheiten in der Grafik-Pipeline
- Frühere Einteilung:
  - Vector Shader
  - Geometry Shader
  - Pixel Shader
- Programmierung mittels spezieller Shader-Sprachen, z.B. GLSL
- Fokus auf Grafikberechnungen
- Einsatz schwierig für Allzweckberechnungen

GPU pipeline (vereinfacht)

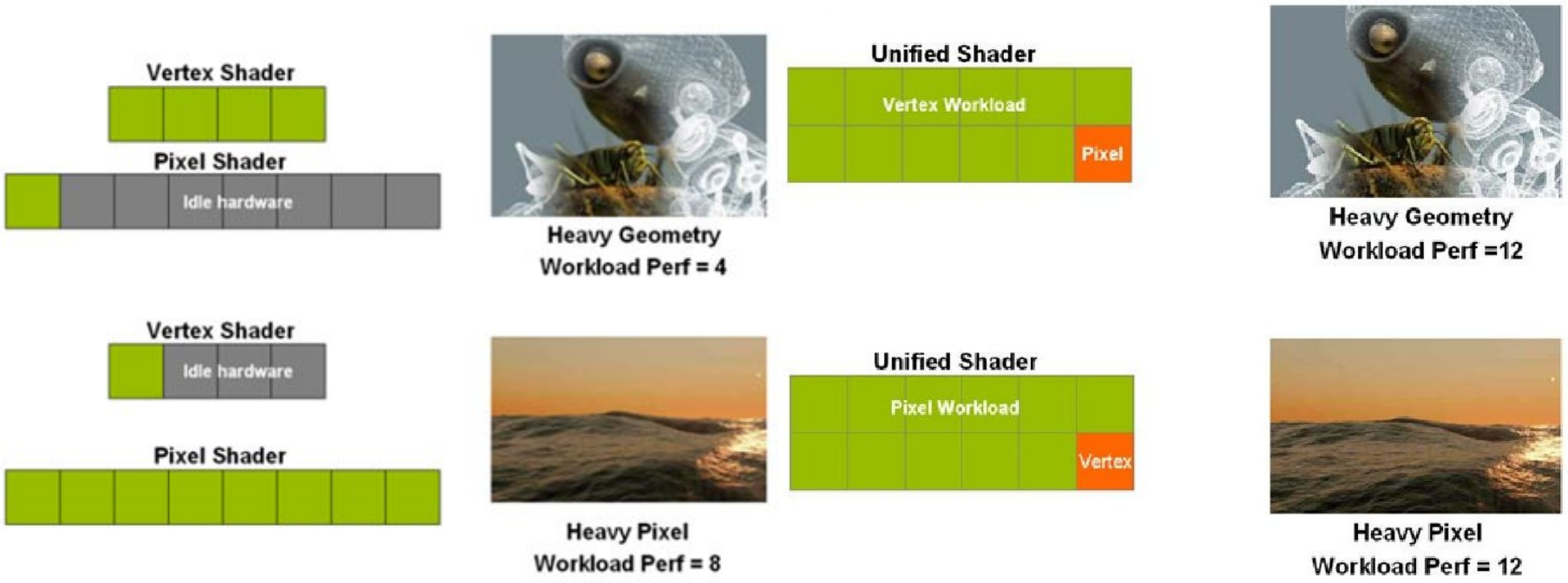


Quelle: http://www.viznet.ac.uk/reports/gpu/10

Heute: Unified Shader

- Flexibel einsetzbar
- Eingeführt mit Nvidia Geforce 8 (G80)
- Programmiersprachen für Allzweckberechnungen, z.B.:
  - Brook+ (AMD)
  - CUDA (Nvidia)
  - OpenCL (Khronos Group)

Quelle: NVIDIA GeForce 8800 GPU Architecture Overview



Wieso GPGPU?

GPUs bieten:

- Massiv-parallele Ausführung durch große Anzahl Cores
- Schnelle Anbindung von dediziertem, hierarchischem Speicher
- Maskierung von Speicherlatenzen durch schnelle Thread-Wechsel

Nachteile GPUs:

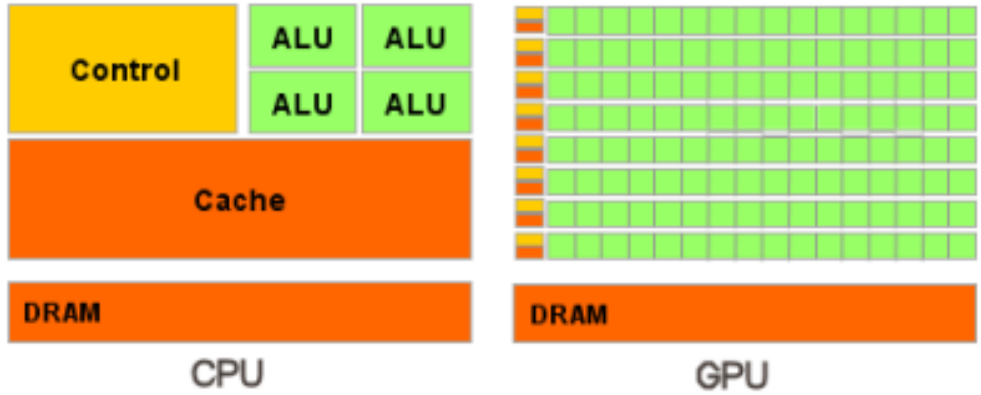
- Hoher Zeitverbrauch für Datentransfer und Kernelinitialisierung
- Cores haben begrenzte Funktionalität

Einsatzbereich GPGPU

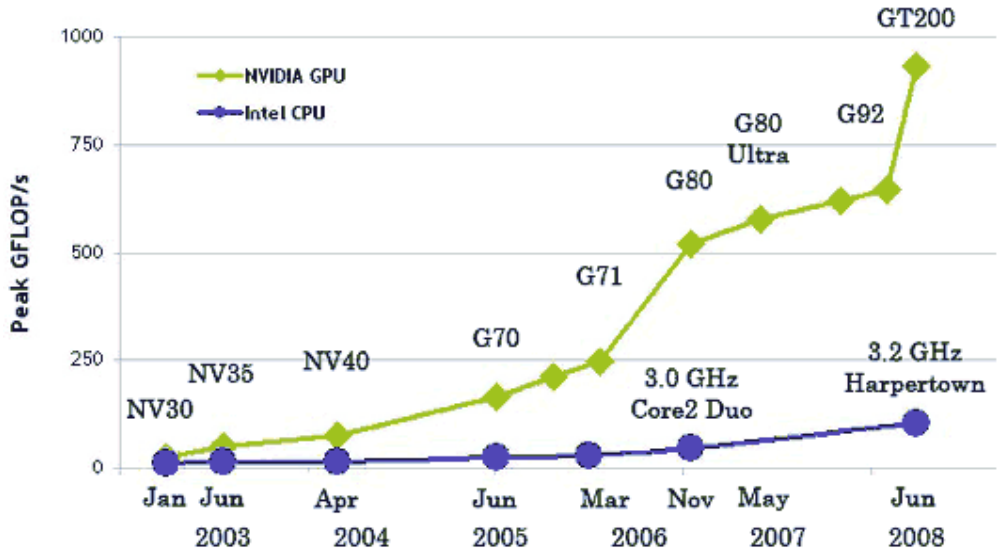
GPGPU eignet sich besonders für Probleme mit:

- unabhängigen Einzeldaten ($\rightarrow$ massiv-parallele Ausführung)
- hohem Berechnungsanteil ($\rightarrow$ begrenzte Funktionalität)
- großen Problemgrößen ($\rightarrow$ Datentransfer)

Prinzipieller Vergleich CPU vs. GPU



- CPU: kleine Anzahl Allzweckkerne, große Caches
- GPU: viele einfache Kerne
- Peak-Performance der CPU leichter zu erreichen



Quelle: http://ixbtlabs.com/articles3/video/cuda-1-p1.html

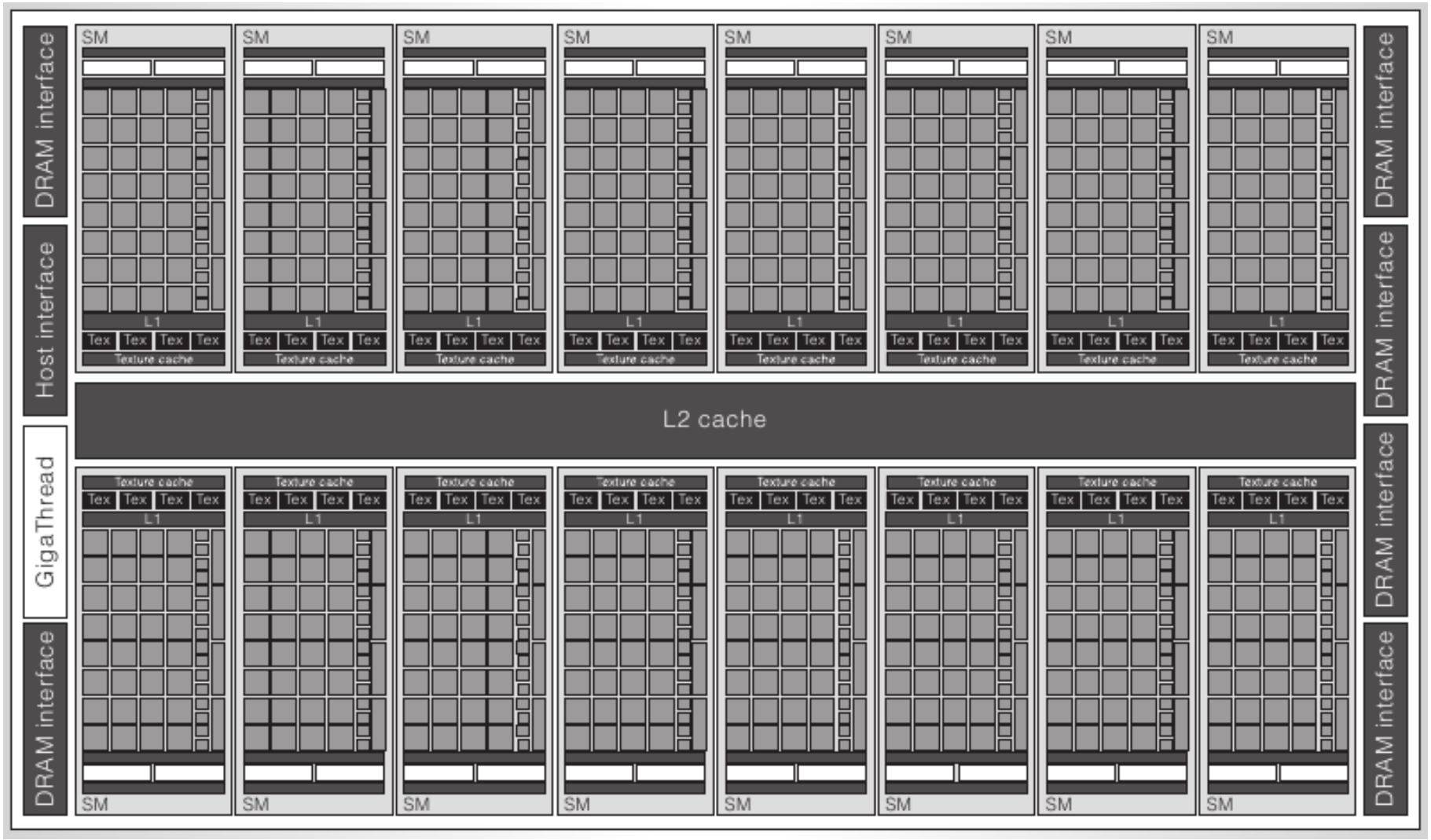
Karlsruhe Institute of Technology

Hardware: Nvidia Fermi

Eigenschaften [5]

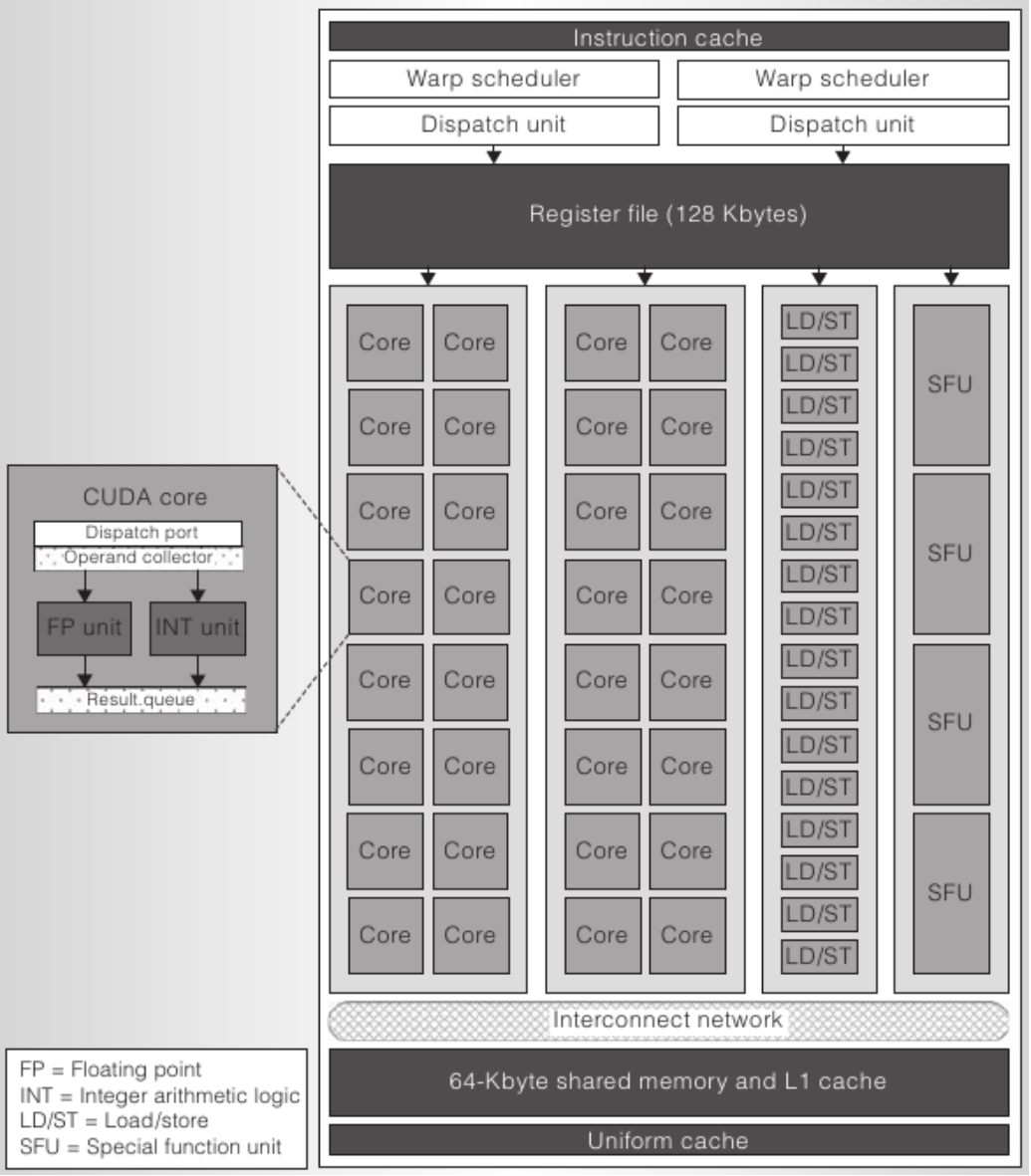
- Vorstellung $\sim$ 2009
- 16 Streaming Multiprocessors (SM)
- Jeder SM besitzt 32 CUDA cores $\rightarrow$ 512 cores
- Bis zu 1536 Threads pro SM $\rightarrow$ 24576 threads
- 6 GDDR5 DRAM interfaces
- 768 Kbyte L2 cache
- Optionale ECC-Unterstützung

Chip-Architektur



Quelle: The GPU computing Era, Nickolls & Dally, Nvidia

SM-Architektur



- 32 cores
- 16 Load/store Units
- 4 Special Function Units
- 64 Kbyte L1 cache

Quelle: The GPU computing Era, Nickolls & Dally, Nvidia

Karlsruhe Institute of Technology

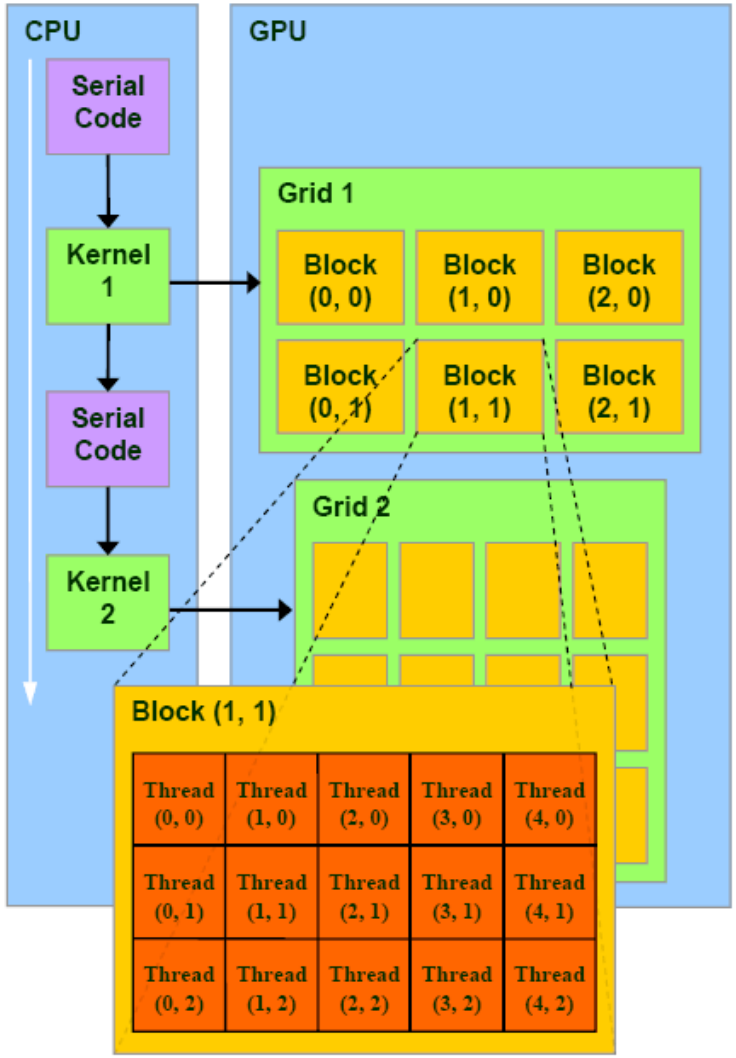
Programmierung: Nvidia CUDA

CUDA [1]

- Compute Unified Device Architecture
- Hauseigene GPGPU-Lösung von NVIDIA
- Veröffentlicht Feb. 2007
- Nutzbar mit Geforce 8 und neuer
- Programmierung in leicht verändertem C

Thread-Organisation

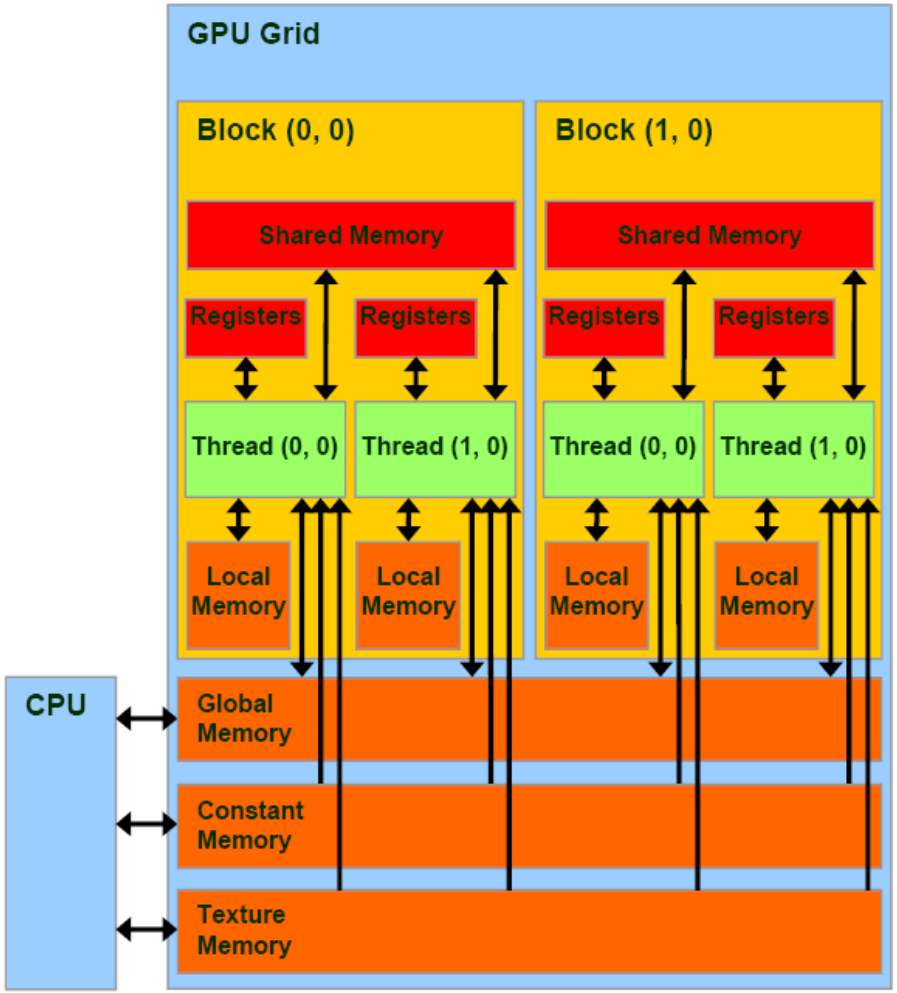
- Threads organisiert in ein- oder mehrdimensionalen Blöcken
- Blöcke organisiert in ein- oder mehrdimensionalen Grids
- Einteilung Threads/Block/Grid wichtig für effiziente Kernelausführung



Quelle: http://ixbtlabs.com/articles3/video/cuda-1-p5.html

Speicherhierarchie

- **Shared Memory** - Schneller gemeinsamer Speicher für den Block
- **Global memory** - Großer Speicher, rel. langsamer Zugriff
- **Local Memory** - Rel. langsamer Speicher für jeden Core
- **Constant & Texture Memory** - Rel. langsamer read-only Speicher



Quelle: http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

Speicherzugriff

- CPU kann nur auf Global, Constant und Texture Memory zugreifen
- GPU-Cores sollten nur auf GPU-Speicher zugreifen
- CPU sollte Speichertransfers initiieren
- Anderes Vorgehen meist ineffizient
- Speichertransfers vom Haupt- in den Gerätespeicher notwendig

→ Zusätzlicher Overhead durch Transport über relativ langsame PCIe-Schnittstelle (PCIe 3.0 x16: ∼15GB/s vs. Fermi: 192GB/s)

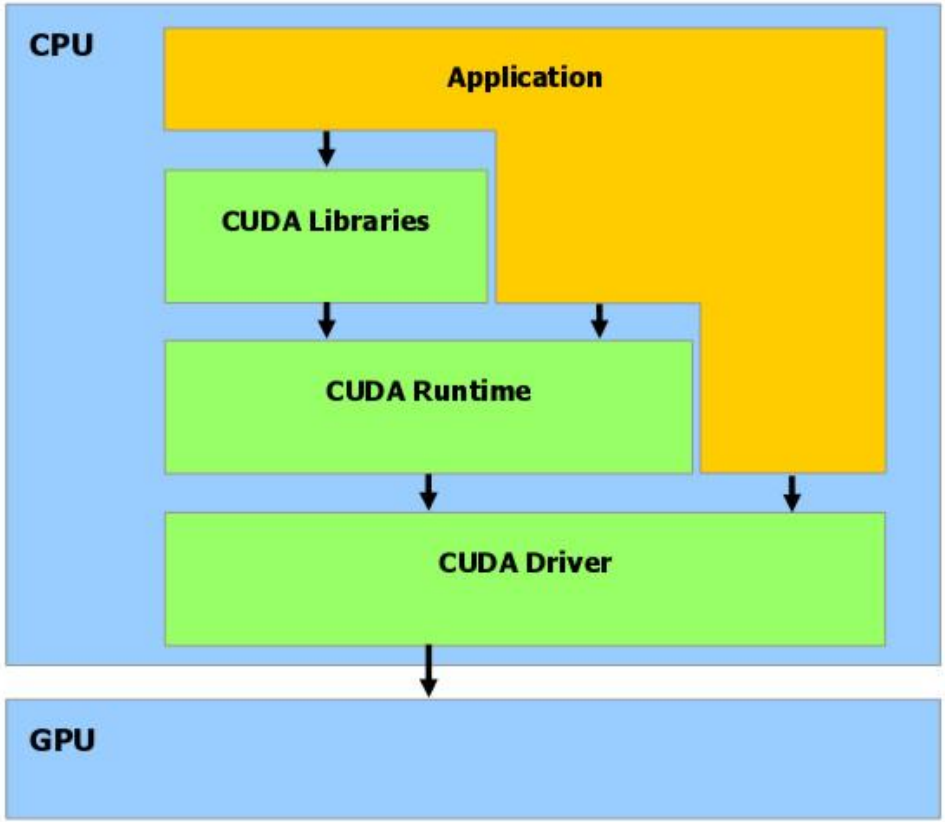
Anwendungsausführung

Typischer Ablauf der GPU-Nutzung:

1. Ausführung normaler CPU-Code
2. Speicherallokation auf der GPU
3. Transfer notwendiger Daten zur GPU
4. Start Kernel (z.B. mit Angabe der Anzahl Blöcke und Threads)
5. Transfer der Ergebnisse von GPU in den Host-RAM
6. Verarbeitung der Ergebnisse auf CPU

Abstraktionslevel

- **CUDA Libs** - z.B. cuBLAS, cuFFT
- **CUDA Runtime** - API für den normalen Gebrauch
- **CUDA Driver** - Low-level Programmierung



Quelle: http://www.viznet.ac.uk/reports/gpu/7

CUDA Runtime API

Karlsruhe Institute of Technology

Allgemeine Informationen

Anzahl CUDA-Karten:

```
cudaError_t     cudaGetDeviceCount (int *count)
```

Eigenschaften einer Karte (z.B. Speicher, Cores, usw.):

```
cudaError_t     cudaGetDeviceProperties (struct
    cudaDeviceProp *prop, int device)
```

CUDA Runtime API

Karlsruhe Institute of Technology

Speichermanagement

Speicher allokieren:

```
cudaError_t    cudaMalloc (void **devPtr, size_t size)
```

Speicher freigeben:

```
cudaError_t    cudaFree (void *devPtr)
```

Speichertransfer:

```
cudaError_t    cudaMemcpy (void *dst, const void *src,
    size_t count, enum cudaMemcpyKind kind)
```

Arten von Speichertransfers (`cudaMemcpyKind`):

```
cudaMemcpyHostToDevice, cudaMemcpyHostToHost,
cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice
```

CUDA Runtime API

Karlsruhe Institute of Technology

Kernels

Funktionssignatur:

```
__global__ void my_GPU_kernel(int arg1, ...) {}
```

Kernel-Aufruf:

```
my_GPU_kernel <<< n_blocks, block_size >>> (arg1, ...)
```

Thread-Organisation I

Karlsruhe Institute of Technology

Vordefinierte Strukturen: `blockDim`, `blockIdx`, `threadIdx`

Eindimensional

- Bestimmung des Thread-Index:

```
__global__ void my_kernel(int * array) {
   int idx = blockIdx.x * blockDim.x
                        + threadIdx.x;
   array[idx] = 5;
}
```

- Kernel-Aufruf:

```
int n_blocks, block_size;
my_kernel <<< n_blocks,block_size >>> (array);
```

Thread-Organisation II

Karlsruhe Institute of Technology

Zweidimensional

- Bestimmung des Thread-Index:

```
__global__ void my_kernel(int ** array) {
   int col = blockIdx.x * blockDim.x;
                    + threadIdx.x;
   int row = blockIdx.y * blockDim.y;
                    + threadIdx.y;
   array[row][col] = 5;
}
```

- Kernel-Aufruf:

```
dim3 dimGrid = (dimGridX, dimGridY, 1);
dim3 dimBlock = (dimBlockX, dimBlockY, 1);
my_kernel <<< dimGrid, dimBlock >>>(array);
```

Programmbeispiel

```
__global__ void increment(char *array, int length) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    array[idx] = array[idx] + 1;
}

int increment_on_gpu(char *array_host, int length) {
    cudaMalloc((void **) &array_gpu, length);
    cudaMemcpy(array_gpu, array_host, length,
            cudaMemcpyHostToDevice);

    increment <<< n_blocks, block_size >>>
            (array_gpu, length);

    cudaMemcpy(array_host, array_gpu, length,
            cudaMemcpyDeviceToHost);
}
```

Threads, Blocks, Grids, ...??

Karlsruhe Institute of Technology

Problem

Beispiel funktioniert nur wenn `length` == `n_blocks*block_size` gilt.

Begründung

- Problemaufteilung auf Threads nicht automatisch wie bei OpenMP
- **Unser Bsp.:** jeder Thread bearbeitet **eine** Stelle im Array

→ Anzahl Threads = `n_blocks*block_size` $\stackrel{!}{=}$ length

Programmbeispiel - Korrektur 1

Karlsruhe Institute of Technology

```
__global__ void increment(char *array, int length) {                    1
    int idx = blockIdx.x*blockDim.x + threadIdx.x;                      2
    array[idx] = array[idx] + 1;                                        3
}                                                                       4

int increment_on_gpu(char *array_host, int length) {                    6
    /* ... */                                                           7
    int block_size = 16;                                                8
    int n_blocks = length/block_size;                                   9

    increment <<< n_blocks, block_size >>>                              11
          (array_gpu, length);                                          12

    /* ... */                                                           14
}                                                                       15
```

Threads, Blocks, Grids, ...?? (forts.)

Karlsruhe Institute of Technology

Problem

Annahme: `length = 170, block_size = 16`

- `n_blocks = length/block_size = 10` (C: int/int=int)
  $\rightarrow$ 160 Threads
- `n_blocks = length/block_size + 1 = 11`
  $\rightarrow$ 176 Threads

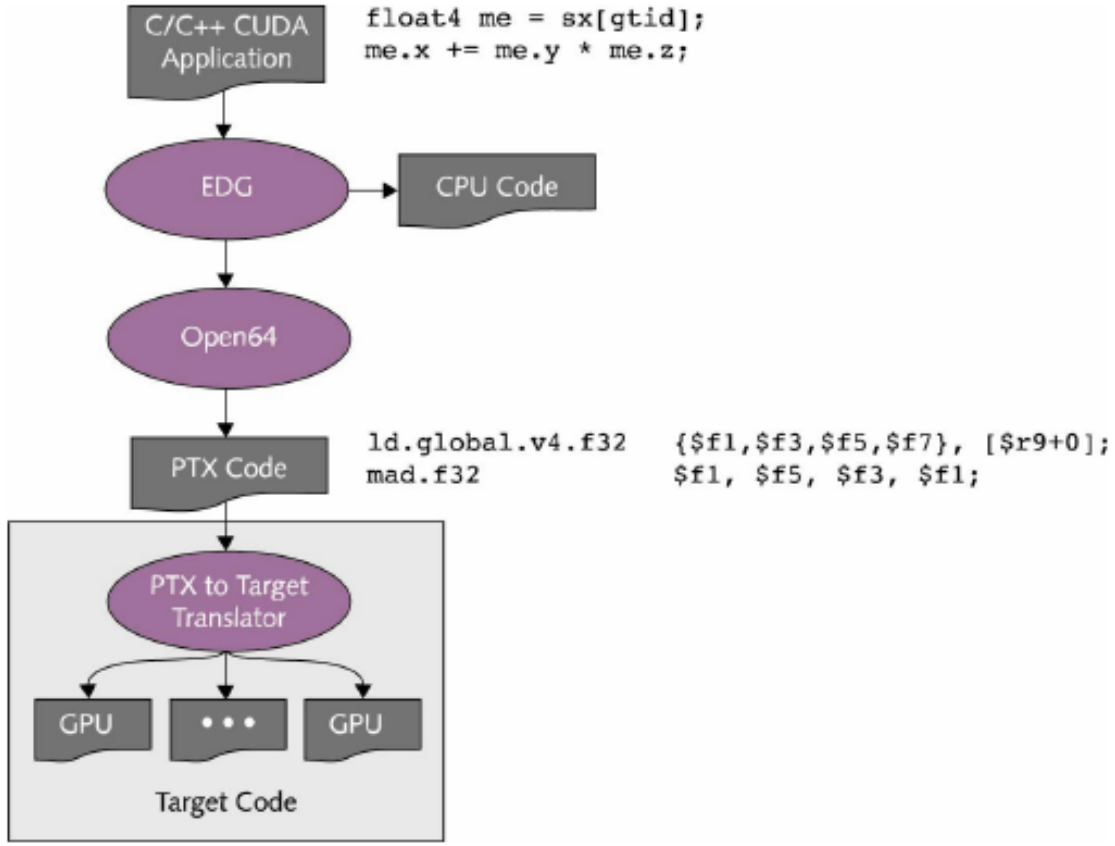
$\rightarrow$ Unvollständiges Ergebnis oder unberechtigter Speicherzugriff

Programmbeispiel - Korrektur 2

```
__global__ void increment(char *array, int length) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < length) array[idx] = array[idx] + 1;
}

int increment_on_gpu(char *array_host, int length) {
    /* ... */
    int block_size = 16;
    int n_blocks = length/block_size + (length%block_size)?1:0;

    increment <<< n_blocks, block_size >>>
          (array_gpu, length);

    /* ... */
}
```

Kompiliervorgang (CUDA $<$ v4.1)

- Preprocessor (EDG) trennt CPU- und GPU-Code
- Modifizierter Open64-Compiler generiert Byte-Code (PTX)
- Treiber generiert Binärcode aus generischem Byte-Code
- Ab v4.1: LLVM



Quelle: Parallel processing with cuda, Tom R. Halfhill

Karlsruhe Institute of Technology

Programmierung: OpenCL

OpenCL [3]

- Hersteller- und Architekturübergreifendes Programmiermodell
- Veröffentlicht Dezember 2008
- Verfügbar unter anderem für:
  - CPUs von Intel und AMD
  - GPUs von Intel, AMD und Nvidia
  - Cell BE Prozessor
  - Intel MIC und Altera FPGAs

Eigenschaften

- OpenCL-Kernels werden zur Laufzeit für eine lokal verfügbare Recheneinheit übersetzt.
- Bibliotheksbasierter Ansatz, daher
  - Kernel-Kompilierung zur Laufzeit
  - Host-Teil der Anwendung unabhängig von (C-)Compiler
- Programmierung ähnlich zu CUDA, u.a. andere Terminologie (z.B. thread block = work group)

Vergleich zu CUDA

Karlsruhe Institute of Technology

+ Herstellerunabhängiger Standard
+ Ein Kernel nutzbar für verschiedene Typen von Recheneinheiten
- (Noch) langsamer im Vergleich zu „nativem" Programmiermodell
- Abstraktion erschwert Hardware-spezifische Optimierung
- Aufwendige Initialisierung

Beispiel - Anwendungsteil I

Karlsruhe Institute of Technology

```
// create a compute context with GPU device                                    1
context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);  2

// create a command queue                                                      4
queue = clCreateCommandQueue(context, NULL, 0, NULL);                          5

// allocate the buffer memory objects                                          7
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, \ 8
    sizeof(float)*2*num_entries, srcA, NULL);                                  9
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*2*num_entries, \ 10
    NULL, NULL);                                                               11

// create the compute program                                                  13
program = clCreateProgramWithSource(context, 1, &fft1D_1024_kernel_src, NULL, NULL); 14

// build the compute program executable                                        16
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);                            17

// create the compute kernel                                                   19
kernel = clCreateKernel(program, "fft1D_1024", NULL);                          20
```

Quelle: Wikipedia

Beispiel - Anwendungsteil II

Karlsruhe Institute of Technology

```
// set the args values                                                    21
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);           22
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);           23
clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0]+1)*16, NULL); 24
clSetKernelArg(kernel, 3, sizeof(float)*(local_work_size[0]+1)*16, NULL); 25

// create N-D range object with work-item dimensions and execute kernel   27
global_work_size[0] = num_entries;                                         28
local_work_size[0] = 64;                                                   29
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, local_work_size, \  30
    0, NULL, NULL);                                                        31
```

Quelle: Wikipedia

Beispiel - Kernel

Karlsruhe Institute of Technology

```
char * fft1D_1024_kernel_src = "                                                    1
  __kernel void fft1D_1024 (__global float2 *in, __global float2 *out,              2
                            __local float *sMemx, __local float *sMemy) {           3
    int tid = get_local_id(0);                                                      4
    int blockIdx = get_group_id(0) * 1024 + tid;                                    5
    float2 data[16];                                                                6

    // starting index of data to/from global memory                                 8
    in = in + blockIdx;  out = out + blockIdx;                                      9
    globalLoads(data, in, 64); // coalesced global reads                            10
    fftRadix16Pass(data);      // in-place radix-16 pass                            11
    twiddleFactorMul(data, tid, 1024, 0);                                           12

    localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));        14
    fftRadix16Pass(data);                // in-place radix-16 pass                  15
    twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication           16
    localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));        17

    fftRadix4Pass(data);      // radix-4 function number 1                         19
    fftRadix4Pass(data + 4);  // radix-4 function number 2                         20
    fftRadix4Pass(data + 8);  // radix-4 function number 3                         21
    fftRadix4Pass(data + 12); // radix-4 function number 4                         22

    globalStores(data, out, 64);                                                    24
  }";                                                                               25
```

Quelle: Wikipedia

OpenCL auf unterschiedlichen Architekturen

Karlsruhe Institute of Technology

GPUs: analog zu CUDA

CPUs:

- Ein Thread berechnet mehrere „work items/groups”
- Wichtig: Cache-Nutzung
- Ähnlich zu OpenMP

FPGAs (Altera):

- Operationen werden in Verilog übersetzt
- Generierung möglichst vieler Pipelines mit SIMD-Eigenschaften
- Limitierung durch begrenzten Platz auf FPGA

OpenACC

- Markierung von Code mittels Pragmas (analog zu OpenMP)
- Vereinfacht Nutzung von Beschleuniger
- Mehr Aufwand für Compiler
- Noch keine breite Unterstützung der Hersteller

```
#pragma acc kernels copyin(a,b) copy(c)                1
for (i = 0; i < SIZE; ++i) {                           2
  for (j = 0; j < SIZE; ++j) {                         3
    for (k = 0; k < SIZE; ++k) {                       4
      c[i][j] += a[i][k] * b[k][j];                    5
    }                                                  6
  }                                                    7
}                                                      8
```

Karlsruhe Institute of Technology

Forschung am Lehrstuhl

Einleitung

Vorgestellte Sprachen bzw. Erweiterungen erlauben Programmierung verschiedener Recheneinheiten.

Aber, z. B.:

- Welche Recheneinheit(en) in einem System am besten geeignet für Problem?
- Beste Recheneinheit auf allen Systemen vorhanden?

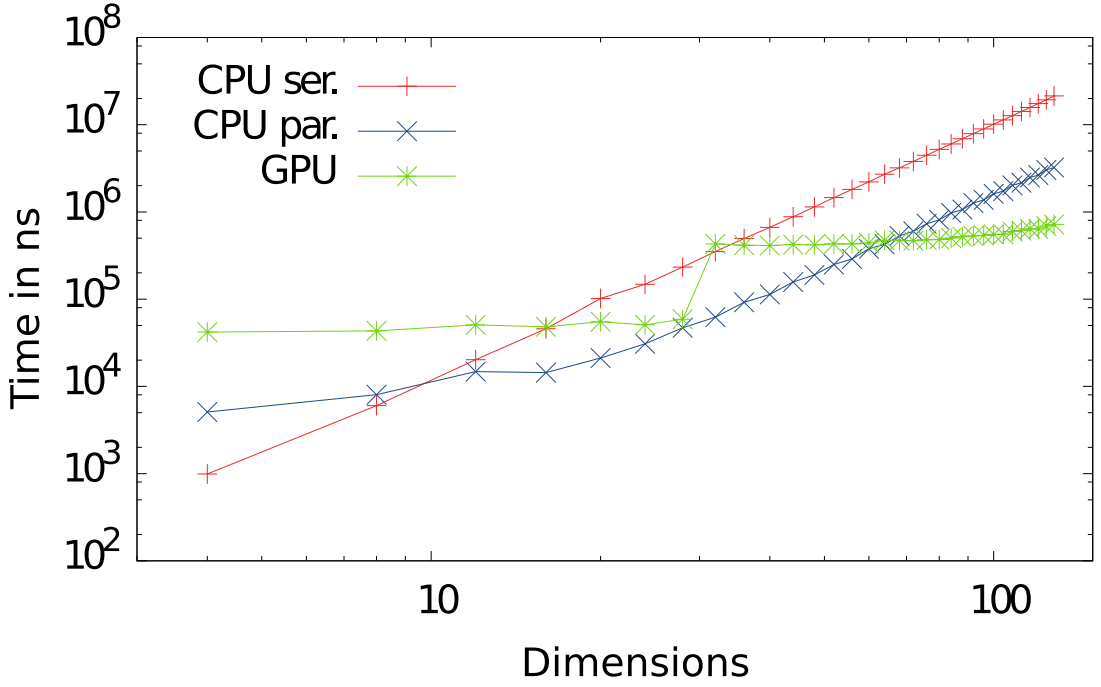
Einleitung

Wahl der schnellsten Recheneinheit(en):

- Unterschiedliche Anzahl und Typen
- Ausführungszeit auch abhängig von
  - Problemgröße
  - Systemzustand (z. B. konkur. Anwendungen)

→ Statische Wahl nur in bestimmten Fällen sinnvoll

Quadratische Matrix-Matrix-Multiplikation:



Einleitung

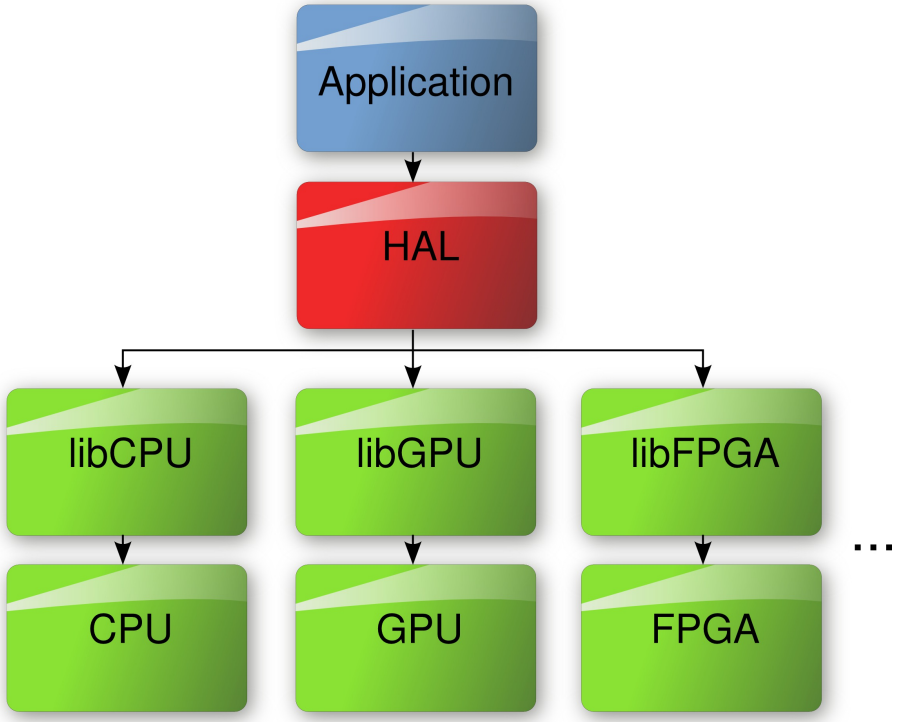
Herausforderungen für Programmierer:

- Verschiedene Programmiermodelle
- Hardware-Anpassungen (Caches, Instruktionssatzerweiterungen, ...)
- Effiziente Datenhaltung (z. B. Transfers notwendig?)
- Verschiedene Optimierungsziele (z. B. Zeit, Energie)
- Fehler in Software/Hardware möglich
- ...

→ Hoher Aufwand für Entwickler, steigende Komplexität

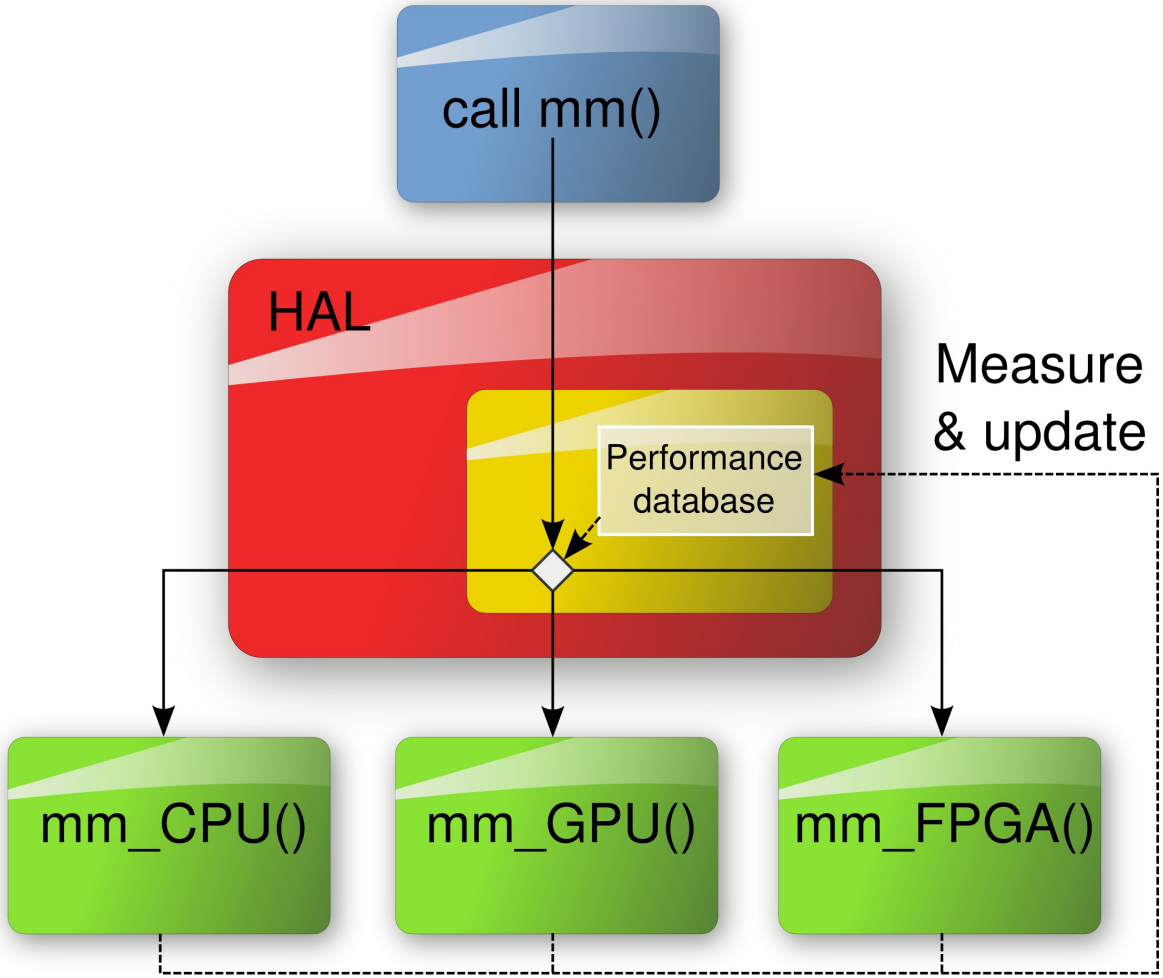
Hardware abstraction layer (HAL)

- Wählt selbstständig beste Recheneinheit
- Vereinfacht Anwendungsprogrammierung



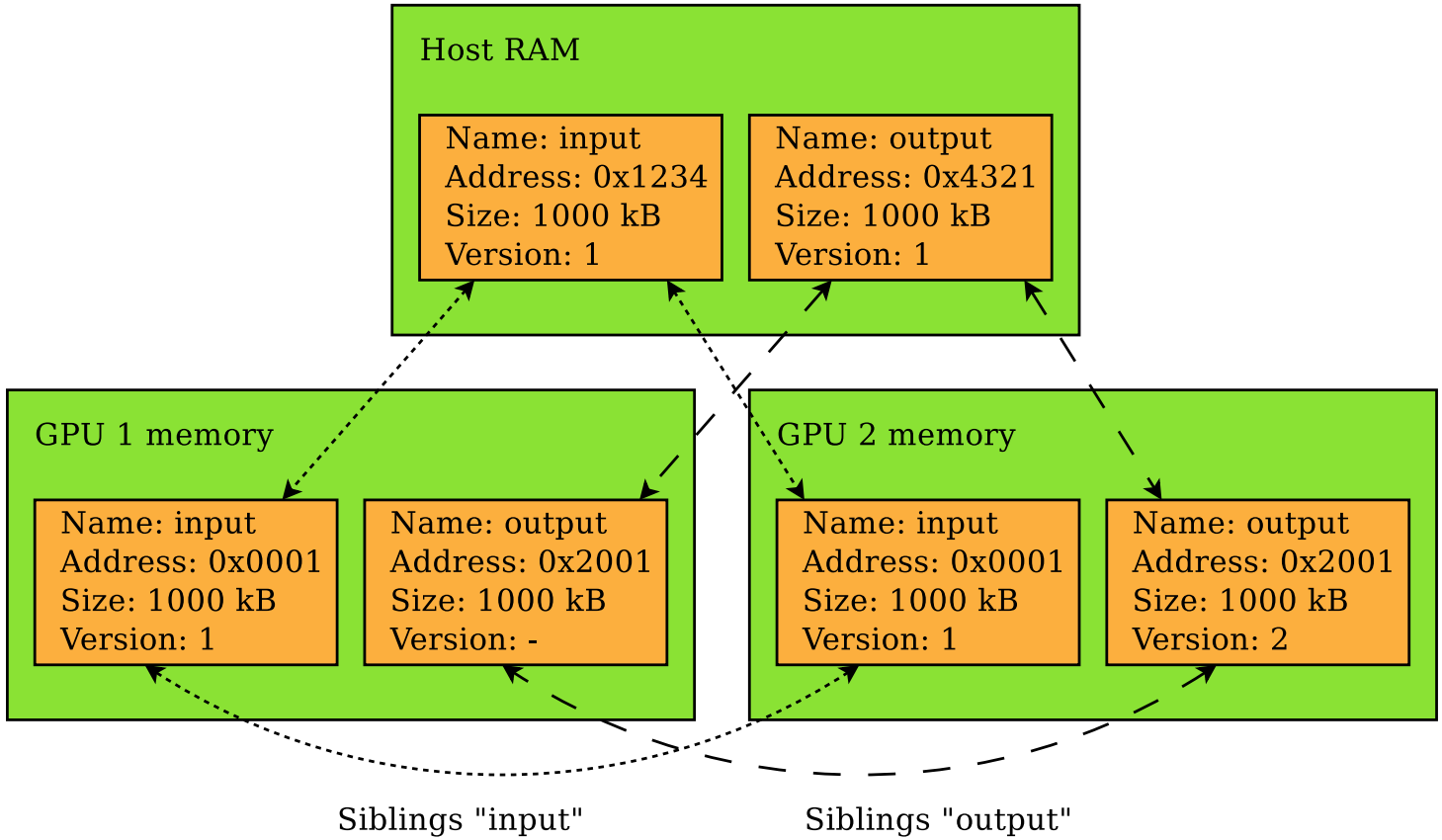
Dynamische Laufzeitvorhersage

- Kernels werden zur Laufzeit evaluiert
- Rechendauer wird in Datenbank gespeichert
- Vorhersage für zukünftige Berechnungen, z.B. in Abhängigkeit von Problemgröße



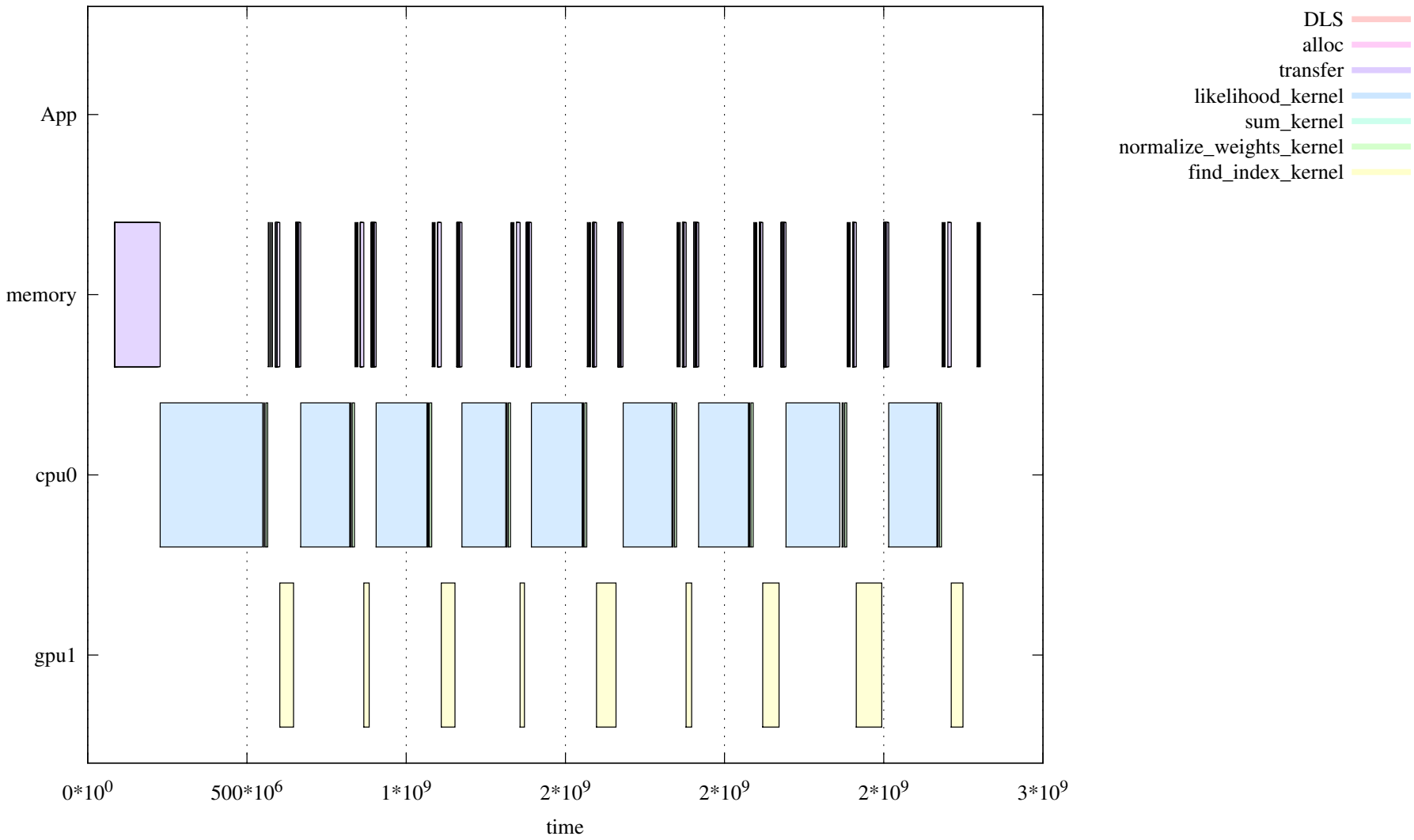
Automatische Datenhaltung

- Versionskontrolle
- Automatische Datentransfers



Tools - Gantt

Rodinia Benchmarks, Particle filter



Einbindung Quellcode

```
#include "dls.h"

dls_decdef(matmul, void, int *a, int *b, int *c, \
    int size);

void matmul_CPU(int *a, int *b, int *c, int size) {
    /* ... */
}
void matmul_GPU(int *a, int *b, int *c, int size) {
    /* ... */
}

void main() {
    /* ... */
    matmul(a,b,c, size);
}
```

Einbindung Quellcode II

Weitere Möglichkeiten

- OpenCL wrapper
- Matlab script wrapper

Quellen

1. *Parallel processing with CUDA*, Tom R. Halfhill
2. *The GPU computing Era*, Nickolls & Dally, Nvidia
3. *Heterogeneous Computing With OpenCL*, Gaster & Kaeli & Howes
4. GPGPU.org
5. Fermi Whitepaper
6. *Cost-Aware Function Migration in Heterogeneous Systems*, Mario Kicherer, Rainer Buchty, Wolfgang Karl [HiPEAC'11]
7. *Seamlessly portable applications: Managing the diversity of modern heterogeneous systems*, Mario Kicherer, Fabian Nowak, Rainer Buchty, Wolfang Karl [ACM TACO]

Karlsruhe Institute of Technology

Vorlesung Rechnerstrukturen
Heterogene Parallelsysteme

Mario Kicherer, Prof. Dr. Wolfgang Karl

Chair for Computer Architecture and Parallel Processing
Prof. Dr. Wolfgang Karl

Sommersemester 2015