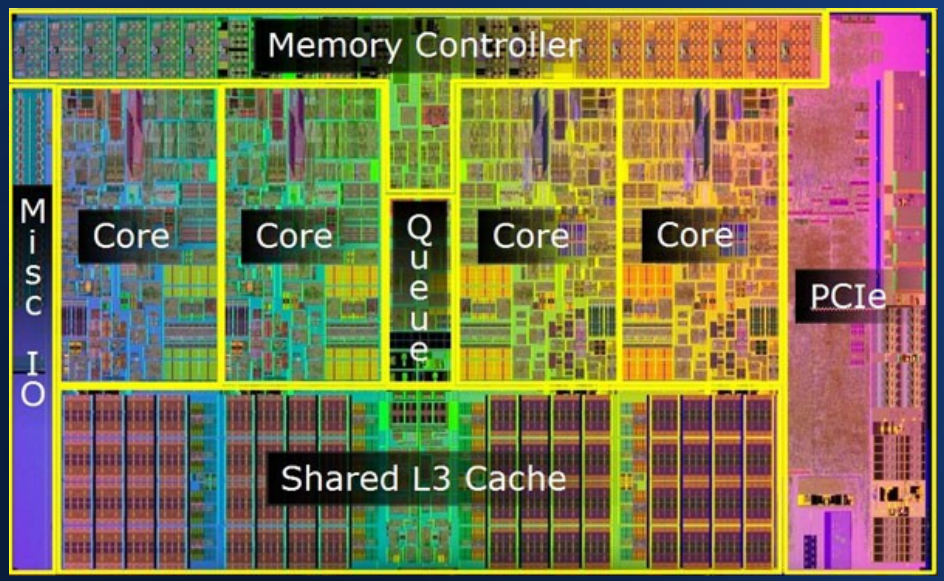
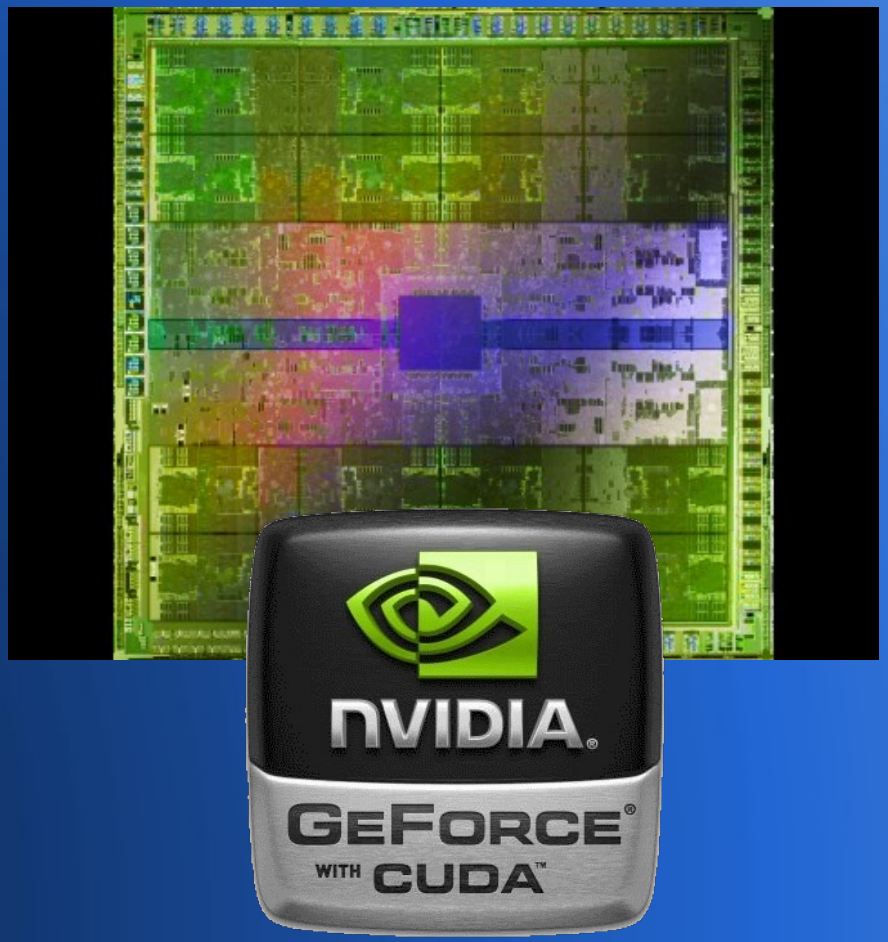
Parallelisierung am AIU

CPU – Cluster – GPU

OpenMP™





Kevin Marco Erler
AIU Jena

AIU (Jena) ↔ Home (Hermsdorf)

Kevin Marco Erler
AIU Jena

Inhalt

Parallelisierung am AIU

(1) Allgemeines zu Parallelisierung

(2) OpenMP

(3) CUDA

(4) Anwendungsbeispiele

Kevin Marco Erler
AIU Jena

Parallelisierung

CPU → Parallelisierung → Allgemeines

Nebenläufigkeit

Fast-parallel

Echt-parallel

Semi-parallel

Nicht-sequentiell

scheduled-Parallel

Multithreading

Multiprocessing

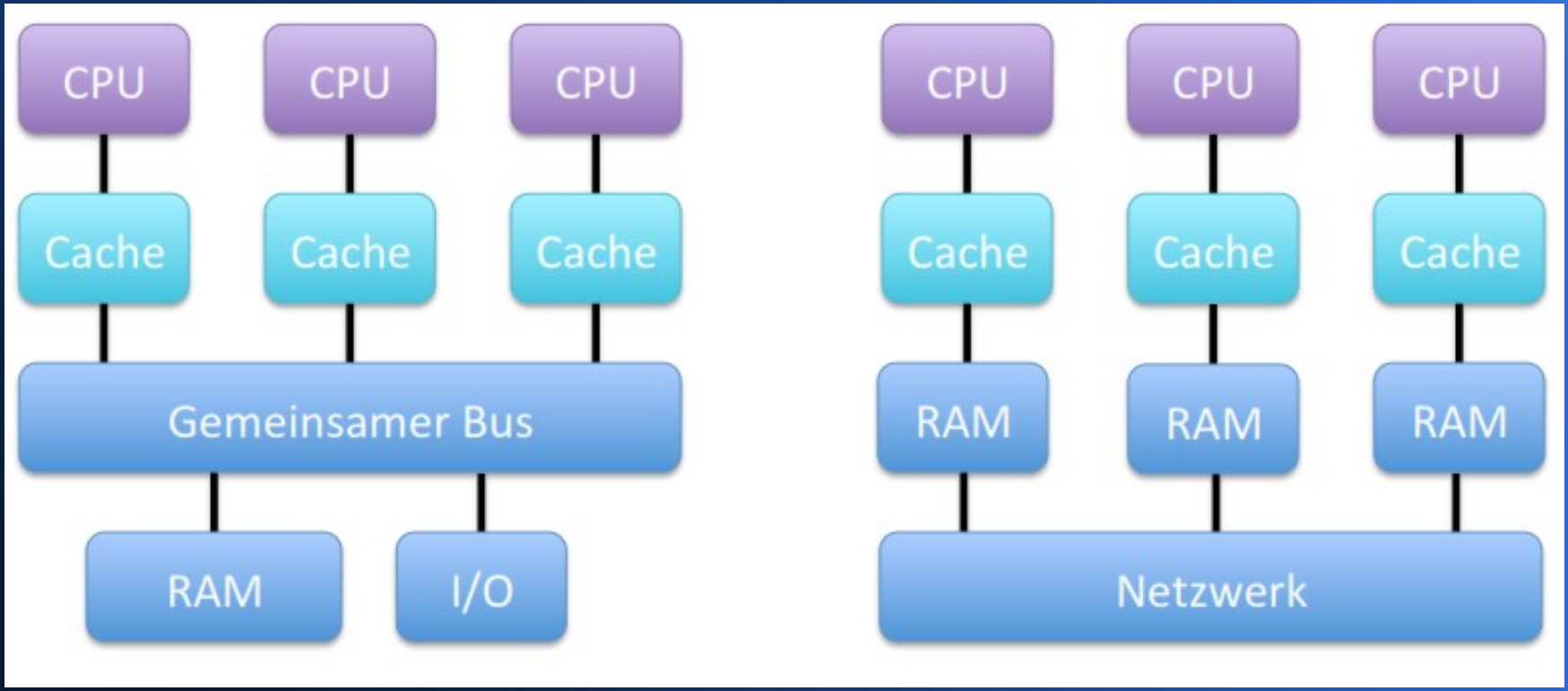
Kevin Marco Erler
AIU Jena

Parallele Plattformen

CPU → Parallelisierung → Allgemeines

Gemeinsamer Speicher (shared memory)

Verteilter Speicher (distributed memory)



Kevin Marco Erler
AIU Jena

Flynnsche Klassifikation

CPU → Parallelisierung → Allgemeines

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Kevin Marco Erler
AIU Jena

OpenMP

CPU → OMP → Allgemeines

- **Open Multi-Processing ([GNU = G]OMP)**
- Seit 1998 Standard **API** für SM-Parallelisierung → Multithreading:
  - Preprocessor (Compiler) Direktiven
  - Library Calls (Funktionen)
  - Environment Variables
- Erweiterung für existierende Programmiersprachen (C/C++, Fortran)

Kevin Marco Erler
AIU Jena

OpenMP ARB

CPU → OMP → Allgemeines

- **OpenMP Architecture Review Board**

„The OpenMP ARB (or just "ARB") is the non-profit corporation that owns the OpenMP brand, oversees the OpenMP specification and produces and approves new versions of the specification." –
http://openmp.org

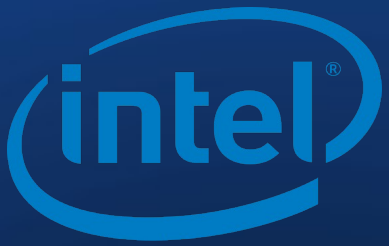
- Entwicklung eines herstellerunabhängigen Standard für Parallelprogrammierung
- Diverse Firmen:
  Permanente Mitglieder Zusätzl. Mitglieder

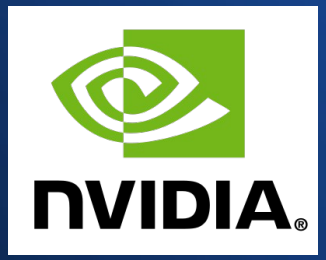
Kevin Marco Erler
AIU Jena

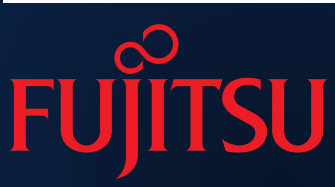
OpenMP ARB

CPU → OMP → Allgemeines

Permanente Mitglieder

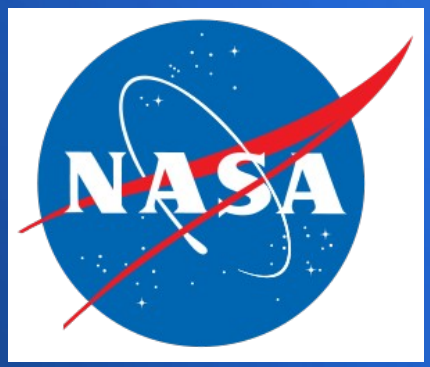






etc.

Zusätzl. Mitglieder

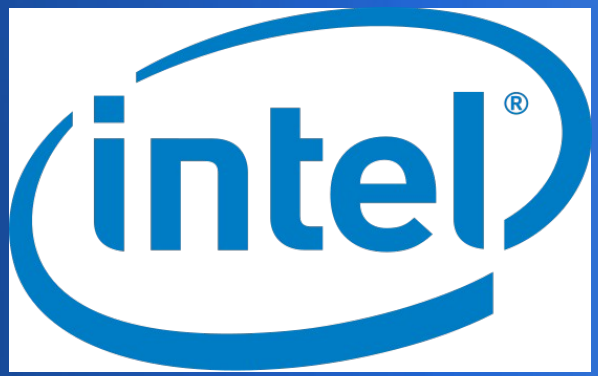


Kevin Marco Erler
AIU Jena

OpenMP – Compiler

CPU → OMP → Compiler

- Microsoft-Compiler (ab MS Visual Studio 2005 → nicht Express-Editions)
- Intel C/C++-Compiler (ICC; ab Version 8; OpenMP+Erweiterungen für Parallelisierung)
- GCC (C: gcc, C++: g++, Fortran: gfortran; ab Version 4.2)



Kevin Marco Erler
AIU Jena

Kompilierung

CPU → OMP → Compiler

- **Gültigkeit** nur bei OpenMP-Support:
  (1) OMP-Libs beim Kompilieren & Linken
  (2) Einbinden der OpenMP-Hauptbibliothek
  (3) aktivierter OpenMP-Compiler-Schalter
- kein OpenMP-Support:
  - OMP-Erweiterungen sind ohne Wirkung
  - Seriell-Parallel-Hybrid-Entwicklung möglich
  - Nicht-OMP-kompatible Compiler nutzbar

Kompilierung

CPU → OMP → Compiler

- **Gültigkeit** nur bei OpenMP-Support:
  - (1) OMP-Libs verwenden: compile & link
    - **libgomp** → „**-lgomp**“-Kompiler-Schalter
  - (2) Einbinden der OpenMP-Hauptbibliothek
    - **C/C++:** `#include <omp.h>`
    - **Fortran 90/95:** `!$ use omp_lib`
    - **Fortran 77:** `!$ include 'omp_lib.h'`
  - (3) aktivierter OpenMP-Kompiler-Schalter
    gcc / g++ / gfortran: **-fopenmp**

Kevin Marco Erler
AIU Jena

Kompilierung

CPU → OMP → Compiler

- **Compile-Aufrufe:** **C/C++** mit **gcc/g++**:
- **Release:**
  <gxx> [-std=c++0x] -m64 **[-fopenmp]** -Wall -Wextra -pedantic -pedantic-errors **[-lgomp]** -lm **-O3 -s <source.cxx> -o <dest>**
- **Debug:**
  <gxx> [-std=c++0x] -m64 **[-fopenmp]** -Wall -Wextra -pedantic -pedantic-errors **[-lgomp]** -lm **-g -ggdb3 <source.cxx> -o <dest>**

Kevin Marco Erler
AIU Jena

Kompilierung

CPU → OMP → Compiler

- **Compile-Aufrufe: Fortran** mit **gfortran**:
- **Release:**
  gfortran -m64 **[-fopenmp] [-lgomp]** -lm **-O3 -s <src.fxx> -o <dest>**
- **Debug:**
  gfortran -m64 **[-fopenmp] [-lgomp]** -lm **-g <src.f90> -o <dest>**
- **Auto-Parallelisierung ab GCC v4.7:**
  Compiler-Schalter: -parallel

Kevin Marco Erler
AIU Jena

Warum OpenMP?

CPU → OMP → Allgemeines

- es gibt immer mehr Shared-Memory-Architekturen/-Systeme
- Gewinnung von mehr Leistung älterer Systeme
- Einfache Crosskompilierung von parallelen & seriellen Code möglich
- hersteller- & systemübergreifender Standard
- weitestgehende automatisierte Parallelisierungskonstrukte

Shared memory Parallelisierung

CPU → OMP → Grundlage

- **Shared-Memory Multiprozessor System**:
  - gemeinsamer I/O-(CPU)-Zugriff auf alle Speicher
- **Prozesskommunikation** erfolgt über Speicher & mittels Synchronisation
- **Parallelausführung** n. **FORK-JOIN-MODELL**
  - zu Beginn nur ein Master-Thread (TID: 0)
  - Thread-Teamerzeugung bei FORK-Point
  - Parallelausführung aller Thread-Teammitglieder bis JOIN-Point
  - Reduzierung des Thread-Teams zu Master-Thread

Kevin Marco Erler
AIU Jena

[G]OpenMP-API

CPU → OMP → API

Umgebungsvariablen

Library-calls (Funktionen)

(Preprocessor)Compilerdirektiven

Es wird die OpenMP-Hauptbibliothek benötigt!

→ INCLUDE!

Kevin Marco Erler
AIU Jena

[G]OpenMP-API

CPU → OMP → API

Umgebungsvariablen

Library-calls (Funktionen)

(Preprocessor)Compilerdirektiven

Kevin Marco Erler
AIU Jena

Umgebungsvariablen

CPU → OMP → API → Umgebungsvariablen

- in der OMP-Hauptbibliothek definiert
- Festlegung bestimmter OMP-Settings noch vor der Kompilierung (für die Programmumgebung → OS)
- Bsp.: → ***nächste Folie!***

Kevin Marco Erler
AIU Jena

Umgebungsvariablen

CPU → OMP → API → Umgebungsvariablen

Umgebungsvariable	Beschreibung
OMP_SCHEDULE **Format: =„Type,chunk“**	Festlegung des Scheduling-Typs Typen: Static,dynamic,guided chunk: Anzahl schedulter Aufgaben
OMP_NUM_THREADS **Format: =<int>**	Anzahl Threads pro Thread-Team Format: <int>
OMP_DYNAMIC **Format: =<bool>**	Dynamische Änderung der Thread-Anzahl
OMP_NESTED **Format: =<bool>**	Verschachtelte Parallelität möglich

Kevin Marco Erler
AIU Jena

[G]OpenMP-API

CPU → OMP → API

Umgebungsvariablen

Library-calls (Funktionen)

(Preprocessor)Compilerdirektiven

Kevin Marco Erler
AIU Jena

Library-calls (Funktionen)

CPU → OMP → API → Funktionen

- in der OMP-Hauptbibliothek definiert
- für die Arbeit mit OMP & (paralleler) SM-Programmierung
- Laufzeitsteuerung von & für OMP-Settings
- Funktionen haben Vorrang vor OMP-Umgebungsvariablen
- Empfehlung: Bedingte Kompilierung für Plattform-Interoperabilität!

Kevin Marco Erler
AIU Jena

Library-calls (Funktionen)

CPU → OMP → API → Funktionen

Funktion	Beschreibung
int omp_get_num_threads(void);	Anzahl Threads im Thread-Team
int omp_get_max_threads(void);	Max. von omp_get_num_threads();
void omp_set_num_threads(int);	Setzt Anzahl Threads pro Thread-Team
int omp_get_thread_num(void);	Liefert die aktuelle Thread-ID (TID)
int omp_get_num_procs(void);	Anzahl der CPU-Kernel
void omp_set_dynamic(bool);	Dynamische Thread-Anzahl-Manipulation zur Laufzeit
int omp_get_dynamic(void);	Liefert den Wert von OMP_DYNAMIC
void omp_set_nested(int k);	k = Nesting-Tiefe v. nested parallel regions
int omp_get_nested(void);	Liefert Nesting-Status (1 = nesting, 0 = no nesting)
double omp_get_wtime(void);	Liefert Zeitstempel (Zeit in Sec)

Kevin Marco Erler
AIU Jena

[G]OpenMP-API

CPU → OMP → API

Umgebungsvariablen

Library-calls (Funktionen)

(Preprocessor)Compilerdirektiven

Kevin Marco Erler
AIU Jena

(Preprocessor)Compilerdirektiven

CPU → OMP → API → Compilerdirektiven

- hauptsächliche Anwendung / Definition (die eigentlichen OMP-Erweiterungen)
- **Arten:**
  - Parallelization / Work Sharing directives (parallel region, par. for, par. sections, tasks)
  - Data environment directives (clauses; par. Settings)
  - Synchronization directives (barriers and locks)

Kevin Marco Erler
AIU Jena

(Preprocessor)Compilerdirektiven

- **Direktivenformat:**
  <directive>: Name der Direktive
  <clause list>: Klauseln für die Direktive
- **C/C++:**
  ```
  #pragma omp <directive> [<clause list>]
  ```
- **Fortran:**
  ```
  !$omp <directive> [<clause list>]
  ```

Kevin Marco Erler
AIU Jena

(Preprocessor)Compilerdirektiven

CPU → OMP → API → Compilerdirektiven

- **Direktivenformat:**
  <directive>: Name der Direktive
  <clause list>: Klauseln für die Direktive
  Beeinflussung der OMP-Settings
- **C/C++:**
  `#pragma omp <directive> [<clause list>]`
- **Fortran:**
  `!$omp <directive> [<clause list>]`

Kevin Marco Erler
AIU Jena

Bedingte Kompilierung I

CPU → OMP → API → Compilerdirektiven

- **Bedingte Kompilierung I: OpenMP-Compiler**
  - OpenMP-Schalter: **-fopenmp** (gcc/g++/gfortran)
  - aktiviert:
    - Kompilierung der OMP-Erweiterungen
    - OpenMP-parallele Ausführung möglich (Exec)
    - OMP-Makrodefinition: **_OPENMP**
  - deaktiviert:
    - OMP-Erweiterungen werden ignoriert
    - max. normale serielle Ausführung

Kevin Marco Erler
AIU Jena

Bedingte Kompilierung II

CPU → OMP → API → Compilerdirektiven

- **Bedingte Kompilierung II: OMP-Makrodef.**
  - OpenMP-Makrodefinition: **_OPENMP**
  - Nutzen von Standard-Preprocessordirektiven
  - **C/C++:**
```
#ifdef _OPENMP
  <Code with OMP-Elements>
#endif
```
  - **Fortran:** **!$ [oder C$ oder *$]** `<OMP-Code>`

Kevin Marco Erler
AIU Jena

OMP Parallel-Direktive

CPU → OMP → API → Compilerdirektiven → parallel

- Hauptdirektive → leitet parallelen Abschnitt ein:
  - Fork-Join-Ausführungsmodell
- Paralleler Abschnitt = Scope für div. OpenMP-Inline-Direktiven
- Parallelitätsgrad = Anzahl der Threads im Team des parallelen Abschnitts
- Reihenfolge der Thread-Ausführungen ist nicht vorhersagbar (ausgen. bei Thread-Synchr.)

Kevin Marco Erler
AIU Jena

Fork-Join Ausführungsmodell

CPU → OMP → Grundlage

(1) Master-Thread (TID: 0) erzeugt ein Team von Threads (sich eingeschlossen) → FORK

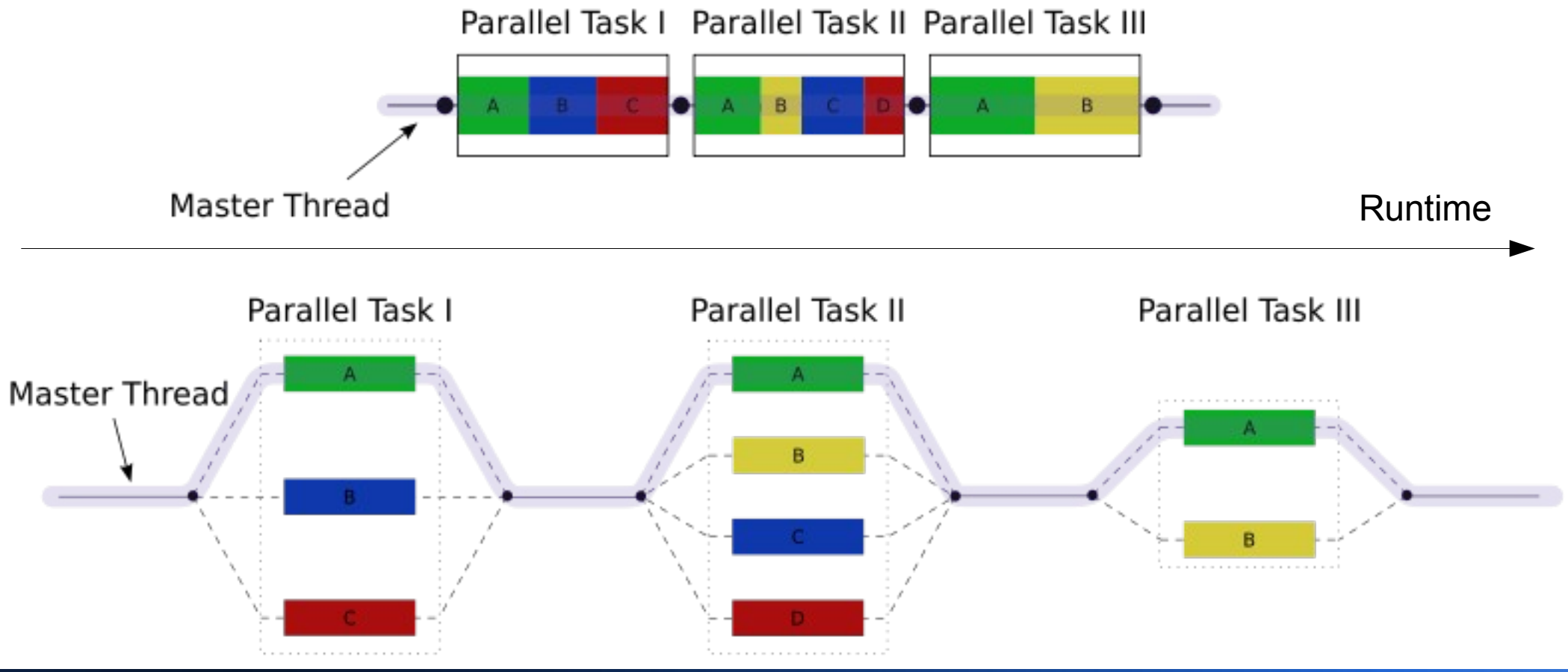
(2) jeder Thread führt das Gleiche aus

(3) Implizite Thread-Synchronisation und Reduzierung auf Master-Thread beim Austritt → JOIN

Kevin Marco Erler
AIU Jena

Fork-Join Ausführungsmodell

CPU → OMP → Grundlage



Kevin Marco Erler
AIU Jena

OMP Parallel-Direktive

CPU → OMP → API → Compilerdirektiven → parallel

- **Anwendungsformat:**
- **C/C++:**

```
#pragma omp parallel [clause list]
{
  <strukturierter Parallel-Code>;
}
```

- **Fortran:**

```
!$omp parallel [clause list]
  <strukt. Parallel-Code>
!$omp end parallel
```

Kevin Marco Erler
AIU Jena

OMP Parallel-Direktive

CPU → OMP → API → Compilerdirektiven → parallel

- **Anwendungsformat – Merkmale:**
  - Case sensitiv
  - orientiert an Konvention der Programmiersprache
  - OMP-Direktiven gelten immer nur für den nächsten strukturierten Code-Block
  - lange Direktiven-Zeilen können mittels „\“ umgebrochen werden

Kevin Marco Erler
AIU Jena

OMP Parallel-Direktive

CPU → OMP → API → Compilerdirektiven → parallel

- **Anwendungsformat:**
  - Klauseln für OMP-Settings-Manipulation
  - bedingte Parallelisierung möglich (→ gleich)
  - Kombination mit anderen OMP-Direktiven möglich (→ später)
  - Schachtelung (nesting) gestattet (seit OpenMP-Spezifikation 3.0 → später)

Kevin Marco Erler
AIU Jena

Bedingte Parallelisierung IF

CPU → OMP → API → Compilerdirektiven → Klauseln → if

- **Klausel-Anwendungsformat:**

```
#pragma <direct.> [clause list] \
                  if(<Ausdruck>)
```

- Ausführung der Direktive nur wenn Ausdruck = TRUE → PARALLELE Ausführung
- keine Ausführung der Direkt., wenn Ausdruck = FALSE → SERIELLE Ausführung

Kevin Marco Erler
AIU Jena

OpenMP Memory Model (Scopes)

CPU → OMP → API → Compilerdirektiven → Klauseln → Scope-Access

- **Arten von Daten:**
  - globale Daten (alle Threads → SHARED)
    übergeordnete Scope[s]: Prozess (shared access)
  - lokale Daten (Thread-lokal → PRIVATE)
    Scopes: Thread (private access) +
    Prozess (shared access)
- **Klauseln** zur Regelung von R/W-Zugriffsmöglichkeiten:

Kevin Marco Erler
AIU Jena

OpenMP Memory Model (Scopes)

CPU → OMP → API → Compilerdirektiven → Klauseln → Scope-Access

- **Klauseln** zur Regelung von R/W-Zugriffsmöglichkeiten: `clause(<list of data>)`
  - **shared()**: shared R/W-access mit allen Threads möglich; Synchronisation oder Vektorisierung notwendig;
    → Default-Attribut: für Scope-globale Daten
  - **private()**: jeder Thread bekommt eigenes threadlokales nicht-init. Exemplar; nur threadlokaler R/W-access möglich;
    → Default-Attribut: nur Schleifen-Index-Variablen

Kevin Marco Erler
AIU Jena

OpenMP Memory Model (Scopes)

CPU → OMP → API → Compilerdirektiven → Klauseln → Scope-Access

- **Klauseln** zur Regelung von R/W-access
  - **default(<shared|private|none>)**:
    → legt den Default-Wert für alle Daten fest
    - keine default()-clause: → Default-Wert = shared
    - **shared**: siehe shared()
      → für Scope-globale Daten
    - **private**: siehe private()
      → für threadlokale Daten
    - **none**: explizite Spezifizierung aller Daten durch private() -& shared()-clauses erforderlich

Kevin Marco Erler
AIU Jena

OpenMP Memory Model (Scopes)

CPU → OMP → API → Compilerdirektiven → Klauseln → Scope-Access

- **Klauseln** zur Regelung von R/W-access
  - **firstprivate(<list>)**: wie private() + Initialisierung (globaler Wert = Init.-Wert)
  - **lastprivate(<list>)**: wie private(); Übertrag aus parallelen Abschnitt in sequent. Master-Programmfluss durch letzten Thread des parallelen Abschnitt
  - **firstprivate() & lastprivate() können kombiniert werden!**

Kevin Marco Erler
AIU Jena

OpenMP Memory Model (Scopes)

CPU → OMP → API → Compilerdirektiven → Klauseln → Scope-Access

- **Klauseln** zur Regelung von R/W-access
  - **`#pragma omp threadprivate(<list>)`**:
    - eigene OpenMP-Direktive
    - globale Daten, welche über mehrere parallele Abschnitte wie private()-spezifizierte Daten behandelt werden
    - Scope-Deklaration → globaler Wert bleibt erhalten
    - Scope-lokale Initialisierungen nur via **copyin()**-clause:
  - **copyin(<list of data>)**:
    - analog zu firstprivate(), aber nur für threadprivate()-data

Kevin Marco Erler
AIU Jena

OpenMP Memory Model (Scopes)

- **Klauseln** zur Regelung von R/W-access
  - **reduction(<op>:<data list>)**: **REDUZIERUNG**
    - wie private() + …
    - Operator-abhängige Initialisierung (z.B.: „+“: 0)
    - Reduzierung aller threadlokalen Daten (Teildaten) zur einen Scope-globalen Variable beim Austritt aus dem parallelen Abschnitt
    - **Bsp.:** Aufsummierung / Akkumulation mit Operator „+“:
      reduction(+: Var1) →
      Var1 = Var1_T1 + Var1_T2 + …+ Var1_Tn;

Kevin Marco Erler
AIU Jena

OpenMP Memory Model (Scopes)

CPU → OMP → API → Compilerdirektiven → Klauseln → Scope-Access

- **REDUCTION-Operatoren** & INIT-Werte:

Operator	Initialisierungswert
+ , - , | , ^	neutrales Element **0**
*****	neutrales Element **1**
&	neutrales Element **~0** (bei ~0 sind alle Bits gesetzt)
&&	neutrales Element **true**
||	neutrales Element **false**

Kevin Marco Erler
AIU Jena

Arbeitsaufteilung mit OMP

CPU → OMP → API → Compilerdirektiven → Arbeitsaufteilung

- **Work Sharing Directives** haben nur in parallelen Abschnitten Gültigkeit
- Arbeitsaufteilung auf die Mitglieder im TH-Team
- entweder **manuelle** oder **automatische** Arbeitsaufteilung
- Vorteil: (fast) autom. Parallelisierung von fein-körnigen Strukturen (z.B. Iterationen)
- Nachteil: Synchron. → impliziter Overhead

Kevin Marco Erler
AIU Jena

Manuelle Arbeitsaufteilung I

CPU → OMP → API → Compilerdirektiven → Arbeitsaufteilung

- **Aufteilung via Datenparallelität**
  - voneinander unabhängige / vektorisierte Daten
  - Aufteilungsfaktor = Thread-ID[´s] (TID)
- … und **Fallunterscheidungen** im parallelen Abschnitt (IF/ELSE, SWITCH-CASE)
- größte Kontrolle, aber auch größte Angriffsfläche für Fehler
- manuelles Thread-Management erforderlich

Manuelle Arbeitsaufteilung II

- **Aufteilung mittels Sektionen:**
- Sektions-Direktive: Format (C/C++ | Fortran)

```
#pragma omp [parallel] sections \
            [clause list]
{
  #pragma omp section
  {
    <Code for section>;
  }
  #pragma omp section
  {
    <Code for section>;
  }
}
```

Section 1

...

Section n

```
!$omp [parallel] sections \
   [clause list]

!$omp section

<Code for this section>

!$omp section

<Code for this section>

!$omp end [parallel] sections
```

Kevin Marco Erler
AIU Jena

Manuelle Arbeitsaufteilung II

- Sektionen – Merkmale:
  - nur 1 Thread pro Sektion
  - keine Abhängigkeiten zwischen Sektionen erlaubt
  - keine Garantie der Ausführungsreihenfolge
- Wo finden Sektionen Anwendung?
  - unterschiedliche Aufgaben mit mehreren Sektionen
  - gleiche Aufgaben (z.B. Iterationen) via manueller Datenpartitionierung

Automatische Arbeitsaufteilung

- **Loops: FOR (C/C++), DO (Fortran)**
- **OMP-FOR/DO-Direktive:** Format

C/C++

```
#pragma omp [parallel] for \
            [clause list]
for(Init;<Expr.>;inc/dec)
{
  <Code>;
}
```

Fortran

```
!$omp [parallel] do \
      [clause list]
do <Init>, <End/Expr.>
<Code>
end do
!$omp end [parallel] do
```

Automatische Arbeitsaufteilung

- **OMP-Loops – Merkmale:**
  - Iterationskopf (Arbeit) wird automatisiert unter den Threads aufgeteilt → par. Teilmengenausführung
  - Bedingung: Gesamtmenge aller Iterationen muss berechenbar sein & darf keine Abhängigkeiten untereinander aufweisen → **kanonische Form**
  - Scheduling-Manipulation durch Klauseln möglich
  - implizite Synchronisation nur am Ende, aber keine beim Eintritt

Kevin Marco Erler
AIU Jena

Automatische Arbeitsaufteilung

- **OMP-Loops – Einschränkungen:**
  - keine Abhängigkeiten zw. den Iterationen
  - erlaubte Abbruchanweisungen: `<,>,<=,>=`
  - erlaubter Inc/Dec: `++,+=,--,-=`
  - Init-Var. muss sich um die gleiche Menge verändern
  - Inline-Manipulation der Init-Var. ist unzulässig
  - Anzahl der max. Iterationen muss bekannt sein
  - break-Anweisungen sind unzulässig (continue OK!)
  - Exceptions müssen inline abgefangen werden

Kevin Marco Erler
AIU Jena

Arbeitsaufteilung – Scheduling

- **OMP Scheduling Options – Klauseln**
  - steuern die Verteilung der Aufg. auf die Threads
  - Scheduling-Klausel:
    `schedule(<scheduling option[, chunk]>)`
  - Scheduling: Aufgaben werden in n-chunks zerlegt und den Threads zugewiesen
  - Scheduling Options: Format

static	dynamic	guided	runtime

Kevin Marco Erler
AIU Jena

Arbeitsaufteilung – Scheduling

- **OMP Scheduling Option „static“:**
  - Format: `schedule(static[, chunk])`
  - Funktionsweise:
    - statische (festgelegte) Zuweisung der Thread-chunks
    - fest definierte Ausführungsreihenfolge der Aufgaben
    - jeder Thread macht nur so viele Aufgaben, wie es chunk-Stücke gibt
    - bei fehlender chunk-Angabe:
      Compiler versucht gleich große Stücke gleichmäßig auf alle Threads zu verteilen (chunk = 1)

Arbeitsaufteilung – Scheduling

- **OMP Scheduling Option „dynamic“:**
  - Format: `schedule(dynamic[, chunk])`
  - Funktionsweise:
    - dynamische Zuweisung der Thread-chunks
    - Threads holen sich dynamisch neue Aufgaben, wenn sie fertig sind
    - u.U. langsamer als „static“-Scheduling → Overhead
    - bei fehlender chunk-Angabe: chunk = 1
  - Anw.: bei schlecht vorhersagbaren Arbeitsaufwand

Kevin Marco Erler
AIU Jena

Arbeitsaufteilung – Scheduling

- **OMP Scheduling Option „guided“:**
  - Format: `schedule(guided[, chunk])`
  - Funktionsweise:
    - wie „dynamic“-Scheduling + …
    - Stückgröße nimmt exponentiell ab (max. bis chunk)
    - i.d.R. schneller als „dynamic“-Scheduling
      → exponentiell abnehmender Overhead
      → gute Kompromisslösung zw. „static“ & „dynamic“
    - bei fehlender chunk-Angabe: chunk = 1

Arbeitsaufteilung – Scheduling

- **OMP Scheduling Option „runtime“:**
  - Format: `schedule(runtime)`
  - Funktionsweise:
    - Scheduling wird zur Laufzeit durch Umgebungsvariable festgelegt:
      OS: OMP_SCHEDULE=„<Type>[,chunk]“
      OMP-Funktionen: **omp_set_schedule(<KType,Mod>);**
      omp_get_schedule():
  - keine chunk-Angabe erlaubt
  - Default-Wert: i.d.R.: Type: static, chunk = 1

Kevin Marco Erler
AIU Jena

Arbeitsaufteilung – Scheduling

- **OMP Scheduling-Ablaufpläne:**

+Overhead

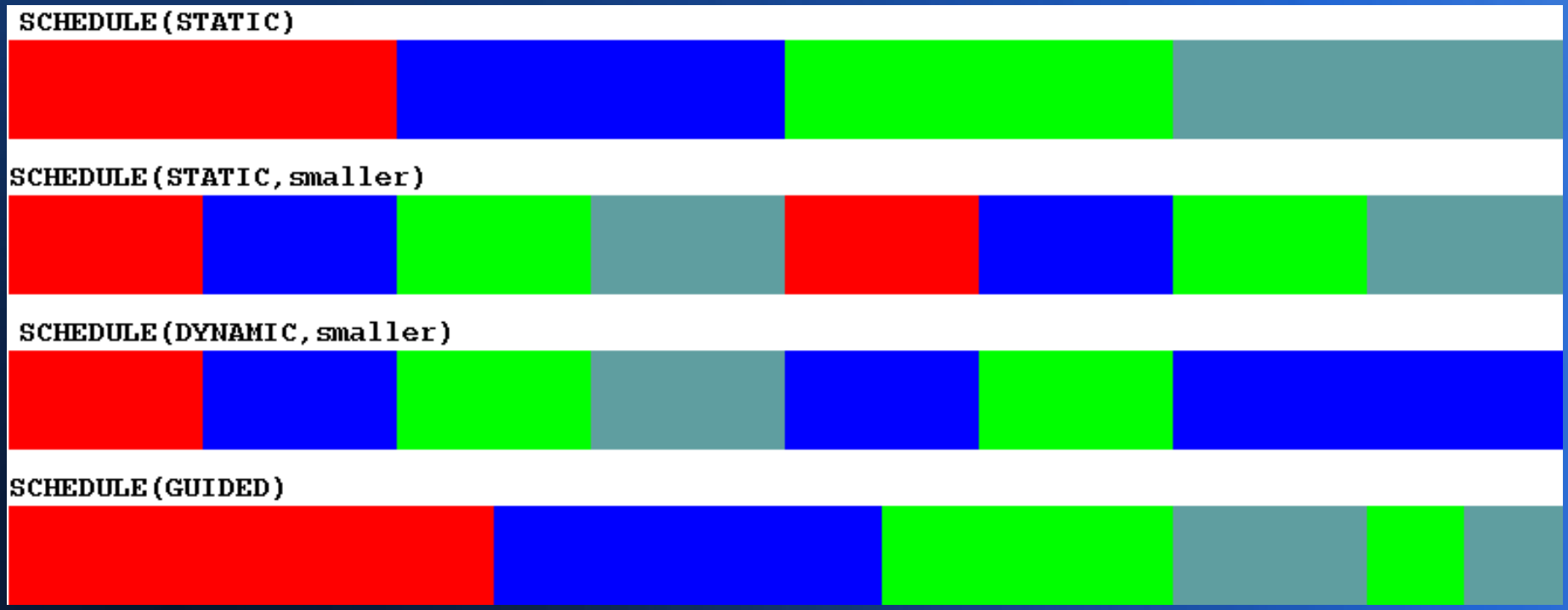
type	chunk?	Iterationen pro Stück	Stücke	Bezeichnung
`static`	nein	n/p	p	einfach statisch
`static`	ja	c	n/c	überlappend
`dynamic`	optional	c	n/c	einfach dynamisch
`guided`	optional	anfangs n/p, dann abnehmend	weniger als n/c	geführt
`runtime`	nein	unterschiedl.	unterschiedl.	unterschiedl.

n = Anzahl Iterationen p = Anzahl Threads c = Wert von chunk

Kevin Marco Erler
AIU Jena

Arbeitsaufteilung – Scheduling

- **OMP Scheduling-Ablaufpläne:**



Kevin Marco Erler
AIU Jena

Synchronisation

CPU → OMP → API → Compilerdirektiven → Synchronisation

- event. notwendig beim Einsatz von shared data
- bes. hohes Risiko bei gemeins. Write-Zugriffen auf ein & dieselben Speicherbereiche
  → Prävention und / durch Koordinierung
  → **Ziel: Threadsicherheit**
- **Möglichkeiten:**
  - explizite Synchronisation (barrier & flush, locks)
  - implizite Synchronisation (Compilerdirektiven)
  - explizite Serialisierung (master/single, critical, etc.)

Kevin Marco Erler
AIU Jena

Explizite Synchronisation

CPU → OMP → API → Compilerdirektiven → Synchronisation

- **Barrierensynchronisation:**
  → Barriere-Direktive: Format (C/C++ | Fortran)
  `#pragma omp barrier` | `!$omp barrier`
  → jeder Thread wartet an diesem Punkt, bis alle Threads ihn erreicht haben
- **Konsistente Speichersicht:**
  → Flush-Direktive: Format (C/C++ | Fortran)
  `#pragma omp flush [datal.]` | `$omp flush [datal.]`
  → jeder Thread bekommt kons. Sicht auf data

Kevin Marco Erler
AIU Jena

Implizite Synchronisation

CPU → OMP → API → Compilerdirektiven → Synchronisation

- einige OMP-Direktiven haben Eigenschaften von impliziter Synchronisation beim Eintritt und / oder Austritt:
  - **omp parallel – Direktive**
  - **omp for – Direktive**
  - **Omp single – Direktive**
- Verzicht auf impliziter Synchronisation mittels **nowait**-Klausel: Format

```
<direktive> [clause list] nowait
```

Kevin Marco Erler
AIU Jena

Explizite Serialisierung

CPU → OMP → API → Compilerdirektiven → Synchronisation

- nur **ein Thread** zur gleichen Zeit
- **ACHTUNG:** Overhead steigt oder kein Nutzen von Parallelität!
- **Methoden:**
  - One-Thread-Ausführung
  - kritischer Abschnitt
  - atomare Operationen
  - Ordering

Kevin Marco Erler
AIU Jena

Serialisierung - One Thread-Exec.

- Code-Ausführung nur eines bestimmten Threads
- Möglichkeiten:
  - Fallentscheidungen (IF/ELSE, SWITCH-CASE): IF/ELSE-Konstrukte im parallelen Abschnitt unter Auswertung der Thread-ID mittels:
    `omp_get_thread_num();`
  - mittels OMP-Direktiven:
    OMP Master -& Single

Kevin Marco Erler
AIU Jena

Serialisierung - One Thread-Exec.

- expl. Serialisierung mittels OMP-Direktiven:
  - OMP-Master-Direktive: Format
    `#pragma omp master [<clause list>] <EOL> {}`
    - nur der Master-Thread im Th.-Team führt den Code aus
  - OMP-Single-Direktive: Format
    `#pragma omp single [<clause list>] <EOL> {}`
    - irgendein Thread, aber nur EIN Thread im Thread-Team, führt den Code aus
  - eventuelle explizite Synchronisation notwendig!

Kevin Marco Erler
AIU Jena

Serialisierung – kritischer Absch.

- Anwendungsformat:
  `#pragma omp critical[(name)] <EOL> {}`
- alle Threads warten zu Beginn und immer nur ein Thread darf den kritischen Abschnitt zur gleichen Zeit ausführen
- Schutz ganzer Speicherbereiche / Codeblöcke
- mehrere kritische Bereiche:
  - definiert durch expl. Namen-Spezifizierung
  - impl. Namen-Spezifizierung bei fehlender Angabe!

Kevin Marco Erler
AIU Jena

Serialisierung – atomic Ops

- **Erweiterungen des Befehlssatz einer CPU**
  - Schutz einer Speicherzelle während einer atomaren (skalaren) Operation
  - Garantie von Cachekonsistenz (z.B. durch Bussperre)
  - weniger Overhead als bei kritischen Bereichen
  - nicht flexibel anwendbar → Verw. von Locks!
- Anwendungsformat:
  `#pragma omp atomic <EOL> <atom. Op>`

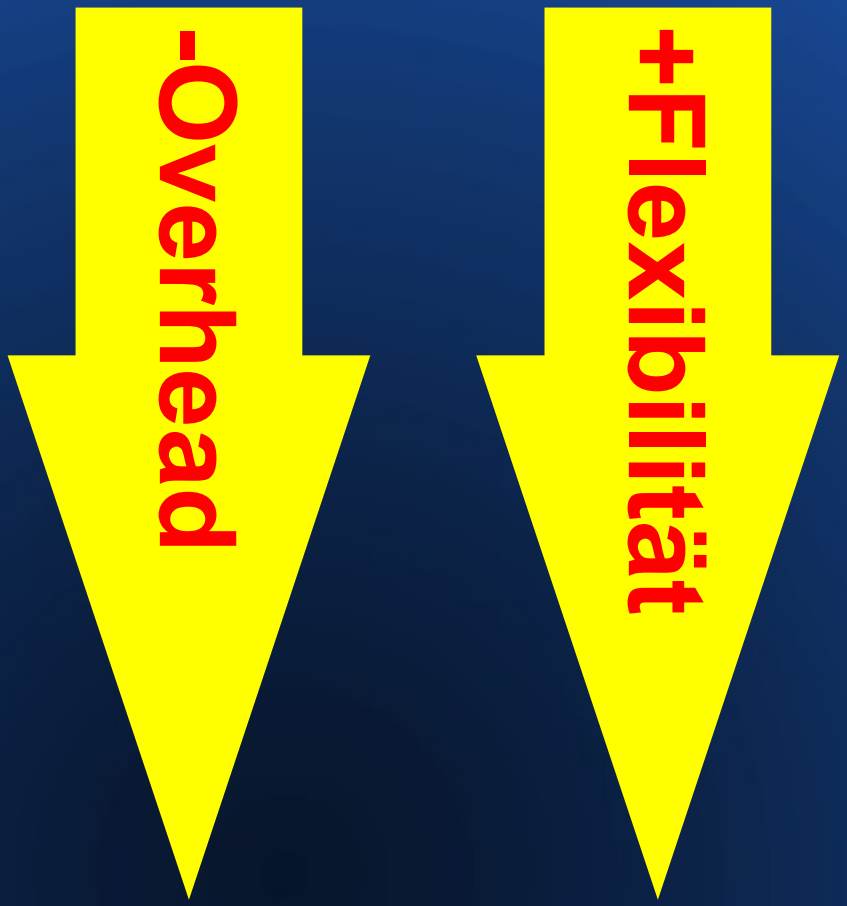
Serialisierung – Ordering

- **ORDERED-Klausel** nur zulässig bei OpenMP-FOR/DO-LOOP! → Format:
  `<OMP-FOR/DO> [clause list] ordered`
- nur ein Thread kann die Schleife ausführen
- Folge: Reihenfolge der Ausführung der parallelen Iterationen
  =
  Reihenfolge wie bei serieller Programm-ausführung

Kevin Marco Erler
AIU Jena

Synchronisation – Overhead

CPU → OMP → API → Compilerdirektiven → Synchronisation



F=

- critical Region
- ordered Loop
- Single-Thread clause
- Master-Thread clause
- Atomic-Operations
- Locks

Kevin Marco Erler
AIU Jena

Direktives ↔ Clauses

CPU → OMP → API → Compilerdirektiven → Klauseln

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Kevin Marco Erler
AIU Jena

Nested Parallelisierung

CPU → OMP → API → Compilerdirektiven → OMP 3.0 → Nesting

- **Nested parallel Regions:**
  - jeder Thread im Thread-Team kann selbst wieder jeweils ein Thread-Team erzeugen
  - Bedingung: Nesting-Support aktivieren!
    - `OMP_NESTED <TRUE | FALSE>`
    - `omp_set_nested(<bool>);`
- **Collapse-Klausel bei OMP-FOR/DO-Direktive:**
  `<OMP-FOR/DO> [<clause list>] collapse(<k>)`
  - kombinierter Indexraum von k Ebenen

Kevin Marco Erler
AIU Jena

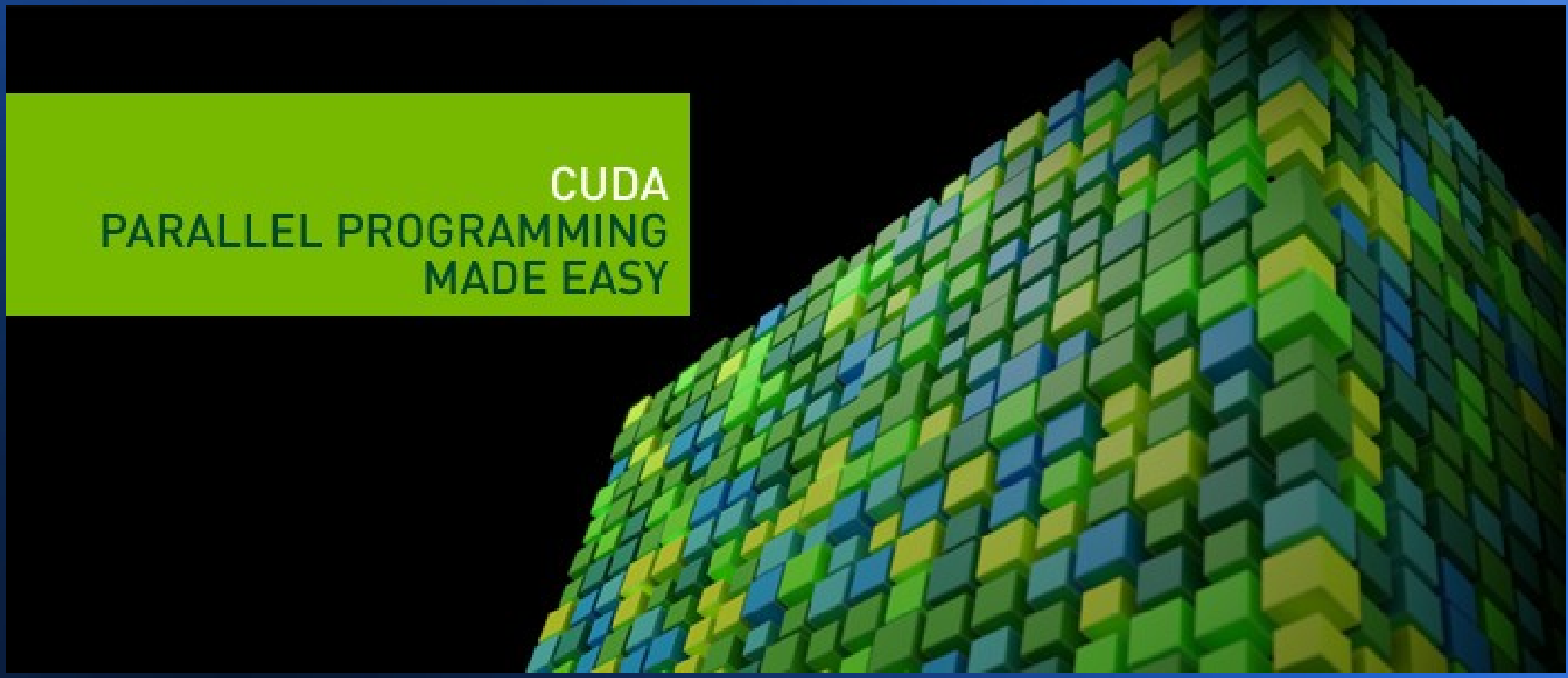
Zusammenfassung - OpenMP

- optimale Ausnutzung aktueller Multicore-CPU´s
- einfacher Ansatz zur Shared-Memory-Parallelisierung / Multithreading
- Wann OpenMP verwenden?
  - bei feingranularen Strukturen (z.B. Schleifen)
  - wenn kaum noch serielle Leistungssteigerung
- Hinweise:
  → Thread-Management kostet Taktzyklen
  → es gibt keine Fehlerbehandlungen
  → nicht jedes Problem eignet sich für SMP

Kevin Marco Erler
AIU Jena

GPGPU-Computing & CUDA

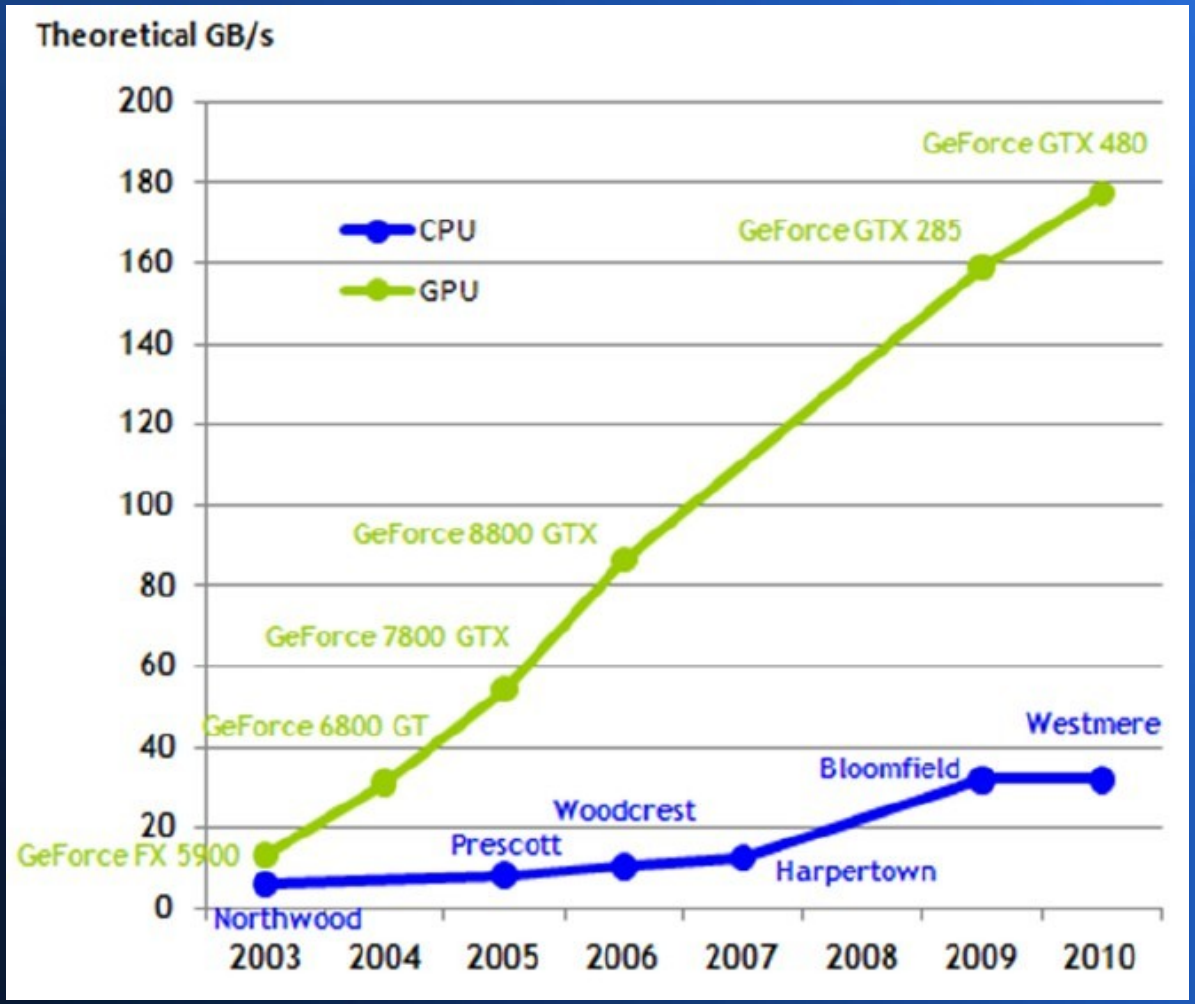
GPU → GPGPU



Kevin Marco Erler
AIU Jena

GPGPU-Computing - Motivation

GPU → GPGPU



Kevin Marco Erler
AIU Jena

GPGPU

GPU → GPGPU → Allgemeines

- **General Purpose Graphics Processing Unit**
- GPU[´s] flexibler programmierbar
- allgemeine Operationen für / auf GPU´s
- Verwendung von GPU´s für allgemeine – auch nicht-grafikspezifische – Aufgaben
- Streamprocessing:
  - weniger I/O, mehr Rechenlast
  - Datenparallelität (SIMD / SIMT) → Massive Parallel.

Kevin Marco Erler
AIU Jena

Eigenschaften von GPGPU´s

GPU → GPGPU → Allgemeines

- **General Purpose Graphics Processing Unit**
- GPU[´s] flexibler programmierbar
- Allgemeine Operationen für / auf GPU´s
- Streamprocessing:
  - weniger I/O, mehr Rechenlast
  - Datenparallelität (SIMD / SIMT)

Kevin Marco Erler
AIU Jena

GPU-API´s

GPU → GPGPU → Allgemeines

- **Abstraktion von Hardware & Treiber**
- **Direkt3D → DirextX**
  - Microsoft
  - proprietär
  - nur für Windows-Systeme
- **OpenGL**
  - entwickelt durch die Khronos Group
  - Open & plattformunabhängig

Kevin Marco Erler
AIU Jena

GPU-API´s

GPU → GPGPU → Allgemeines

- **speziell für einfaches GPGPU-Computing**
- **CUDA**
  - NVIDIA
  - proprietär (teilweise OpenSource)
  - für Windows, Linux und MacOSX
- **OpenCL**
  - Khronos Group
  - Open & plattformunabhängig

Kevin Marco Erler
AIU Jena

Unified Shader [Design]

GPU → GPGPU → Allgemeines

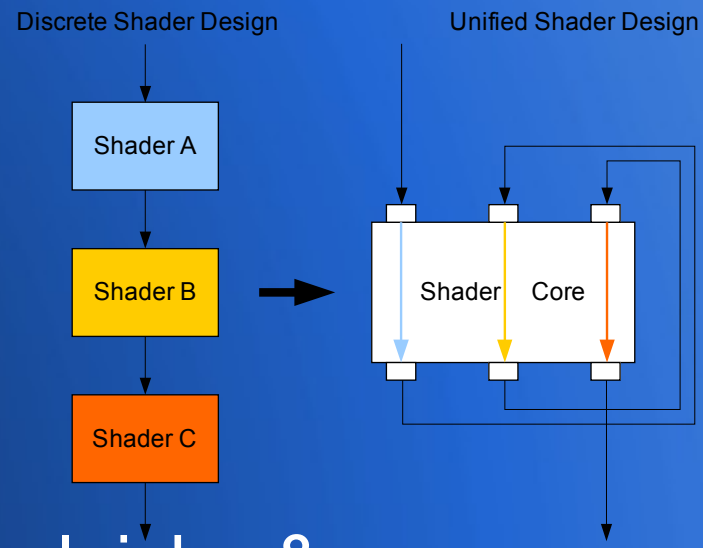
- Früher: viele, dafür wenige Schader-Arten für verschiedene Aufgaben
- mind. drei Typen:
  Vertex Shader|Geometry Shader|Pixel Shader
- Nachteil: →weniger Ressourcen
  →eher für grafikspezifische Aufgaben (Bsp.: Bild rendern)
  → oftmals ungleichmäßige Auslastung

Kevin Marco Erler
AIU Jena

Unified Shader [Design]

GPU → GPGPU → Allgemeines

- Heute: viele einheitliche & universell einsetzbare Schader → Unified Shader
- Harmonisierung aller Schader



- Vorteil: → mehr Ressourcen für gleiche & versch. Aufgaben → universell
  → gleichmäßige Auslastung möglich

Kevin Marco Erler
AIU Jena

GPU´s

GPU → GPGPU → Allgemeines

- **Arten:**
  - dedizierte GPU´s (Cards)
  - integrierte GPU´s (Onboard, CPU´s)
- **Systemanbindung (CPU ↔ GPU):**
  - PCI → AGP → PCI Express (x16 Lanes)
  - Bandbreite: 500 MB/s pro Lane
  - aktuell: PCIe v2 (8 GB/s) → PCIe v3 (16 GB/s)
  - häufiger Flaschenhals für GPGPU-Computing!

Kevin Marco Erler
AIU Jena

GPGPU´s & GPU-Computing

GPU → GPGPU → Allgemeines

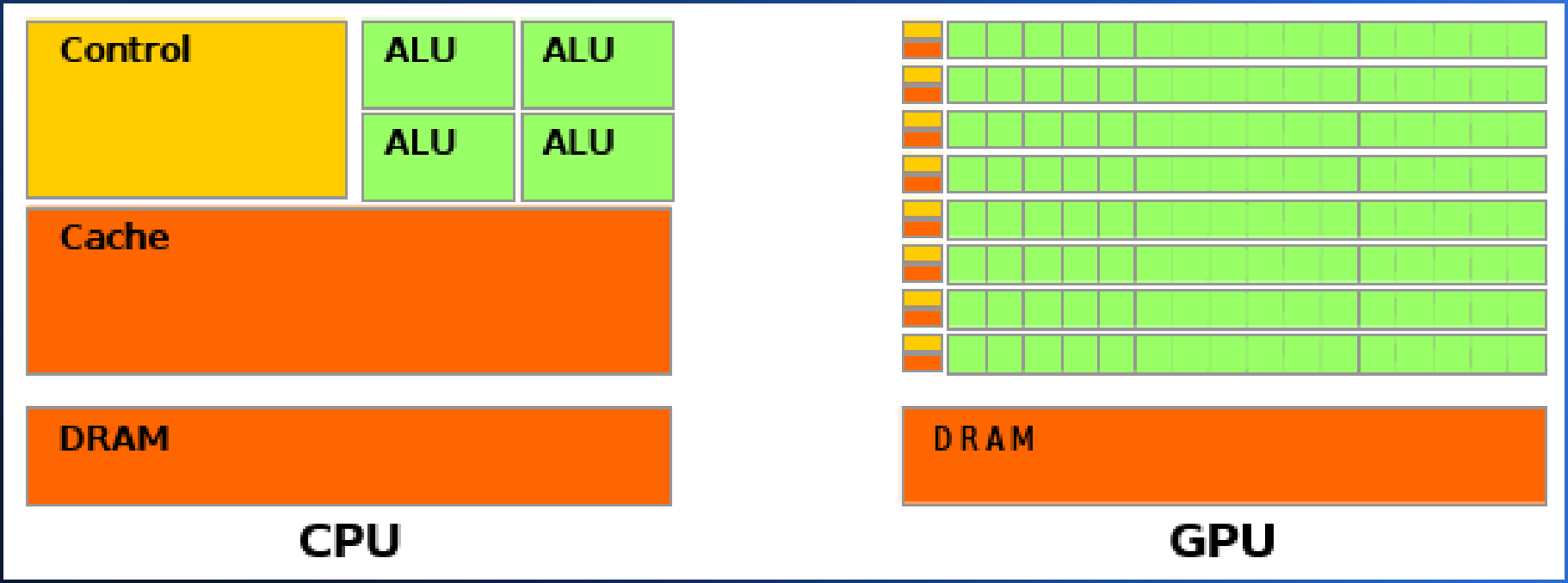
- **Stream Processing:**
  - einfaches paralleles Programmiermodell
  - Instruction-Streams (Kernel) → [Data-]Streams
  - Processing-Art: **SIMD** oder MIMD
  - keine explizite Thread-Synchronisation -& Kommunikation
- geeignet für:
  - hohe arithmetische Komplexität
  - Datenparallelität

Kevin Marco Erler
AIU Jena

GPGPU´s & GPU-Computing

GPU → GPGPU → Allgemeines

- **<u>Vergleich CPU- ↔ GPGPU-Architektur:</u>**
  - mehr Hauptfunktionseinheiten (Stream Processors)



Kevin Marco Erler
AIU Jena

GPGPU´s & GPU-Computing

GPU → GPGPU → Allgemeines

- **Vergleich GPGPU ↔ CUDA-GPGPU:**
- **GPGPU-Computing mittels Grafik-API´s:**
  - sehr komplex
  - teils eigene Programmiersprachen- & Konzepte
  - CPU-untypische Datentypen (12-, 96-, 127-Bit, etc.)
  - homogene Ausführung (GPU-spezifischer Code)
- **NVIDIA CUDA** → *nächste Folie*

Kevin Marco Erler
AIU Jena

GPGPU-Computing – CUDA

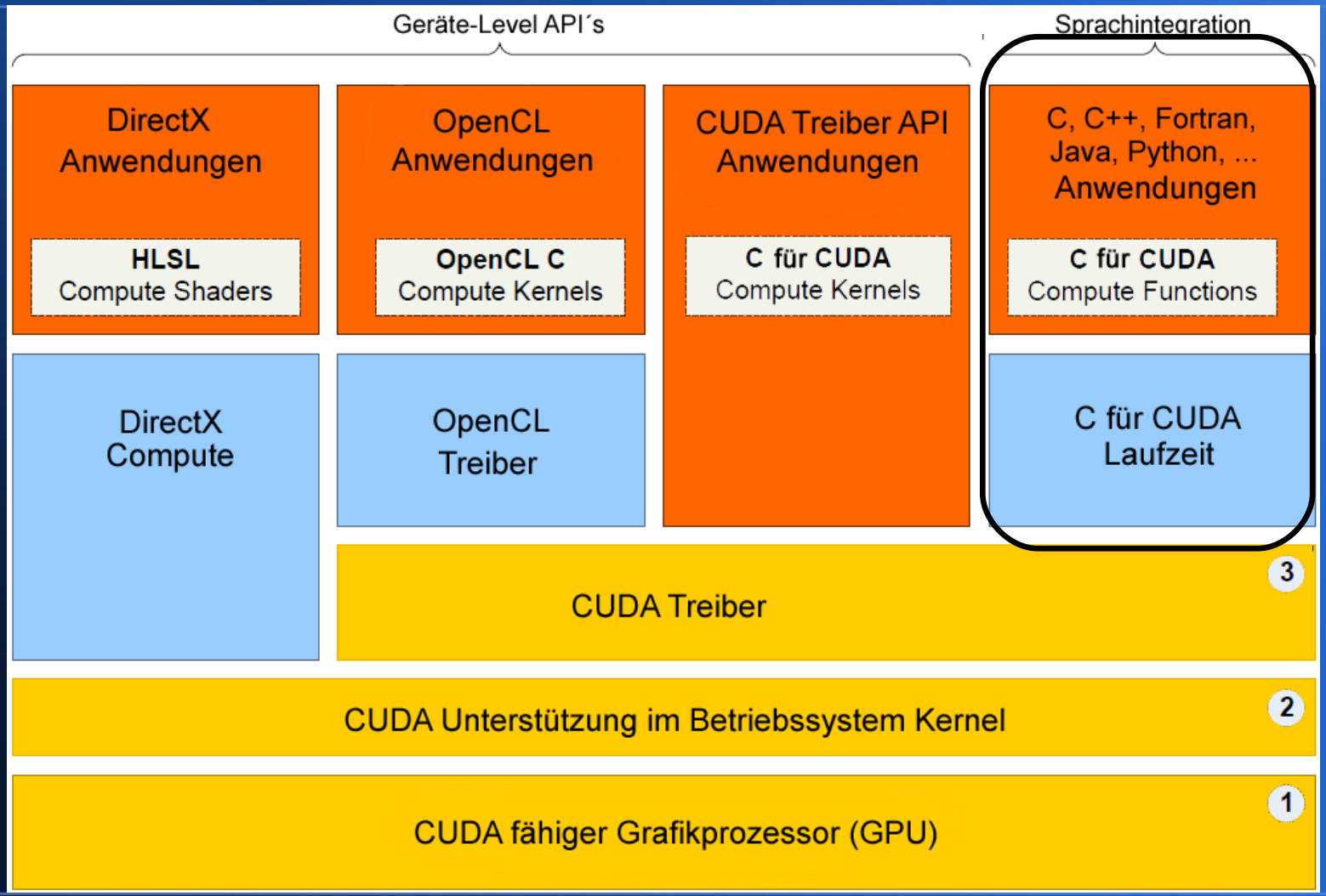
GPU → GPGPU → CUDA

- **CUDA (Compute Unified Device Architecture)**
- **Verbesserung des GPGPU-Computing:**
  - stark vereinfacht
  - Erweiterung etablierter Programmiersprachen (C/C++ offiziell & Fortran, Python, uvm. mittels Bindings)
  - Unterstützung für CPU-typische Datentypen (16-, 32-, 64-, 128-Bit; INT & REAL/FLOATP.)
  - heterogene Ausführung (CPU- & GPU-Code)

Kevin Marco Erler
AIU Jena

GPGPU-Computing – CUDA

GPU → GPGPU → CUDA



Kevin Marco Erler
AIU Jena

GPGPU-Computing – CUDA

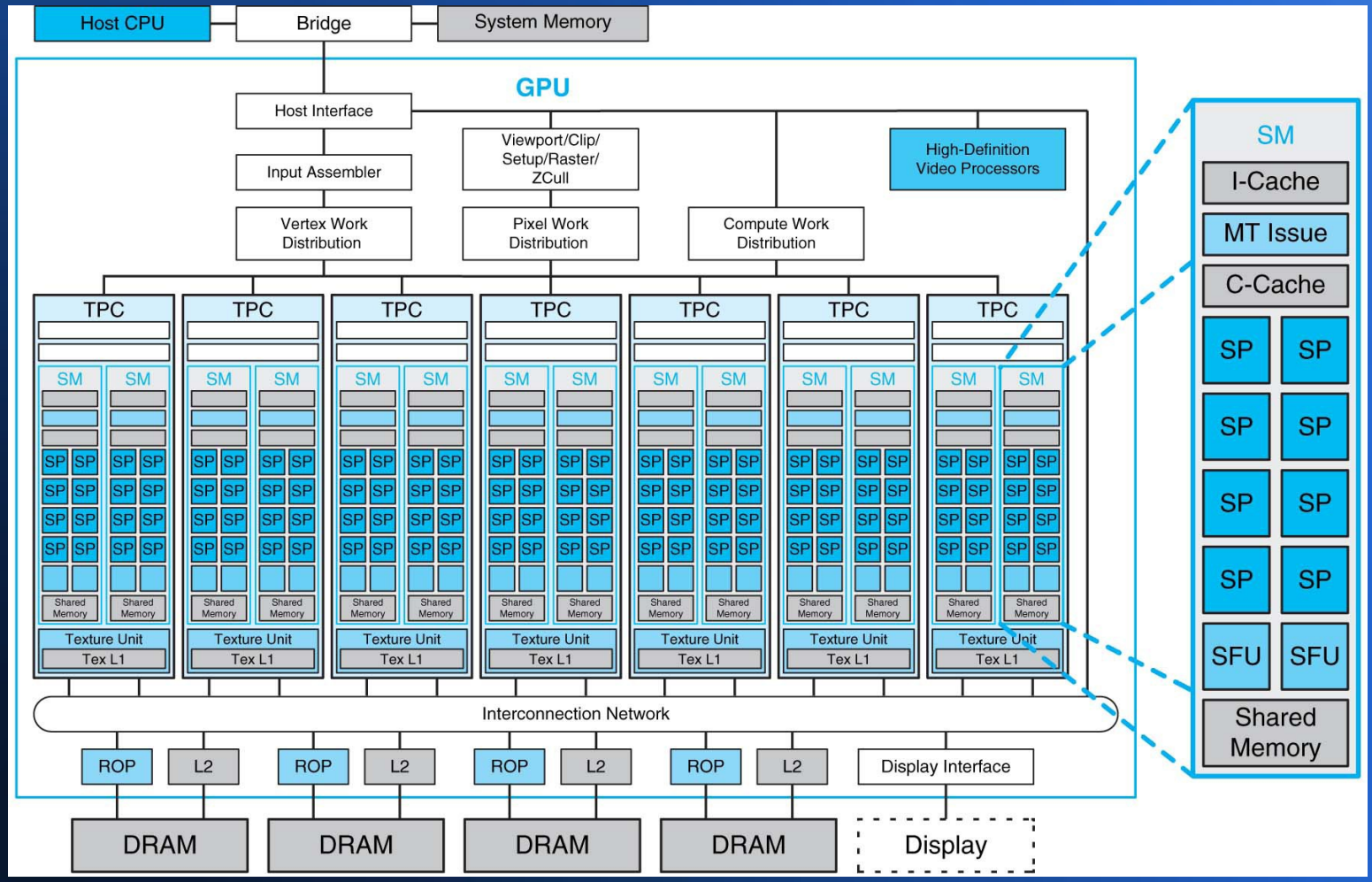
GPU → GPGPU → CUDA

- **<u>weitere CUDA-Komponenten:</u>**
- Funktionsbibliotheken
  - CUBLAS (lineare Algebra)
  - CUFFT (Fourier-Transformations (FFT))
- Compiler: nvcc (Wrapper-Compiler)
- Debugger: cudagdb
- Top-Tools für GPU: nvidia-smi

Kevin Marco Erler
AIU Jena

NVIDIA GPU: G80 (1G)

GPU → GPGPU → NVIDIA → Architekturen → G80



Kevin Marco Erler
AIU Jena

NVIDIA GPU: GF100 (Fermi; 3G)

GPU → GPGPU → NVIDIA → Architekturen → GF100 (Fermi)

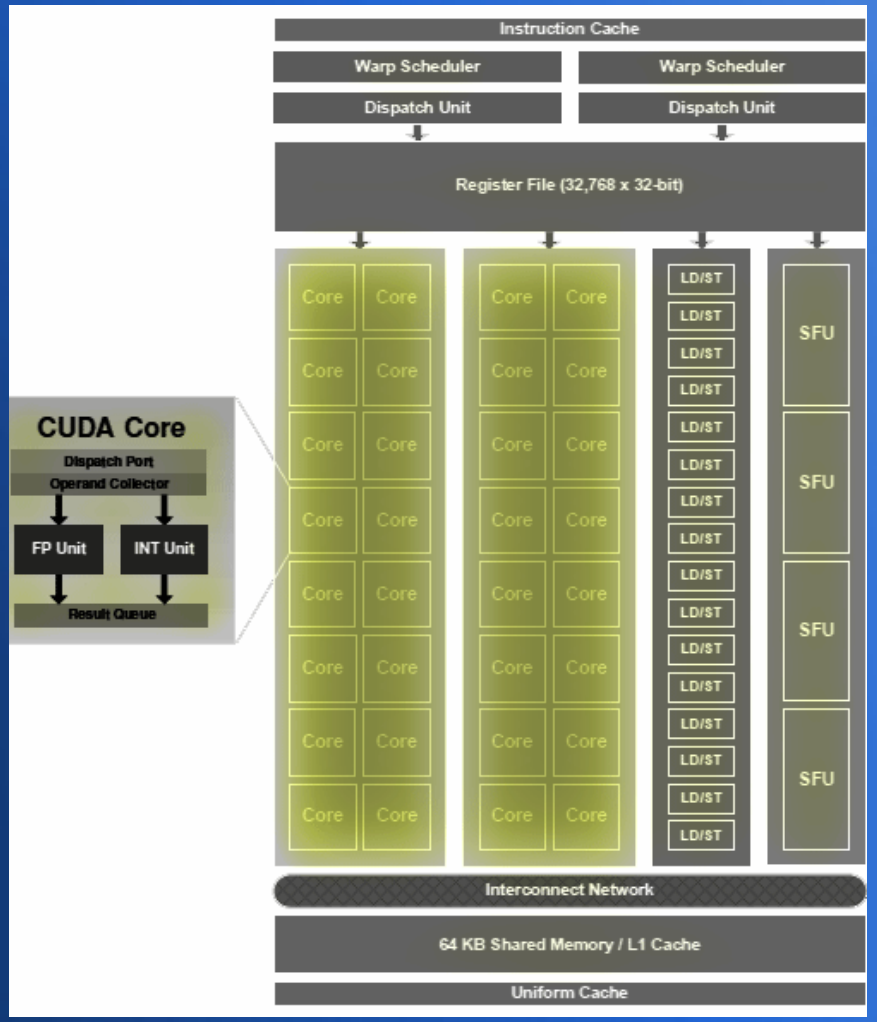
Kevin Marco Erler
AIU Jena

Streaming Multiprocessors (3G)

GPU → GPGPU → NVIDIA → Architekturen → GF100 (Fermi)

- **16 Streaming Multi-processors (SM´s)**
- 1 SM:
  - 32 SP
  - 8 Thread-Blöcke
  - 1536 Threads
  - 48 Warps

gleichzeitig.

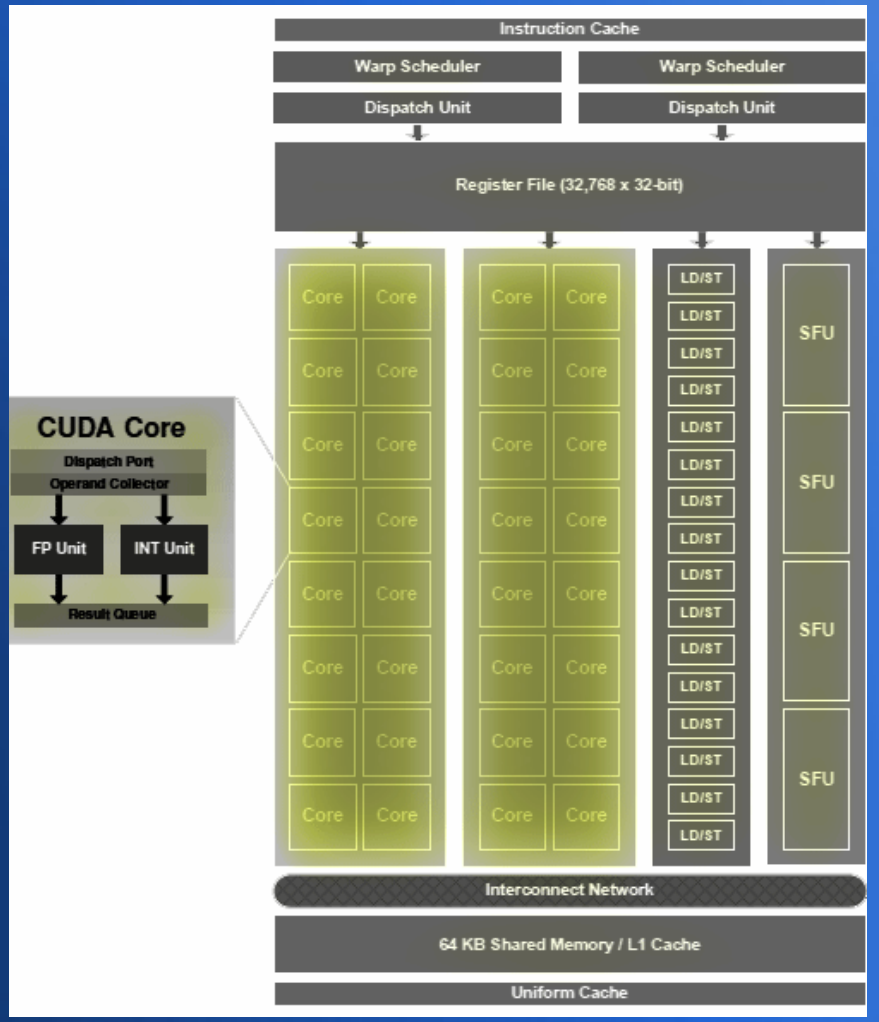


Kevin Marco Erler
AIU Jena

Streaming Multiprocessors (3G)

- **16 Streaming Multi-processors (SM´s)**
  - 512 SP
  - 128 Thread-Blöcke
  - 24576 Threads
  - 768 Warps

gleichzeitig.

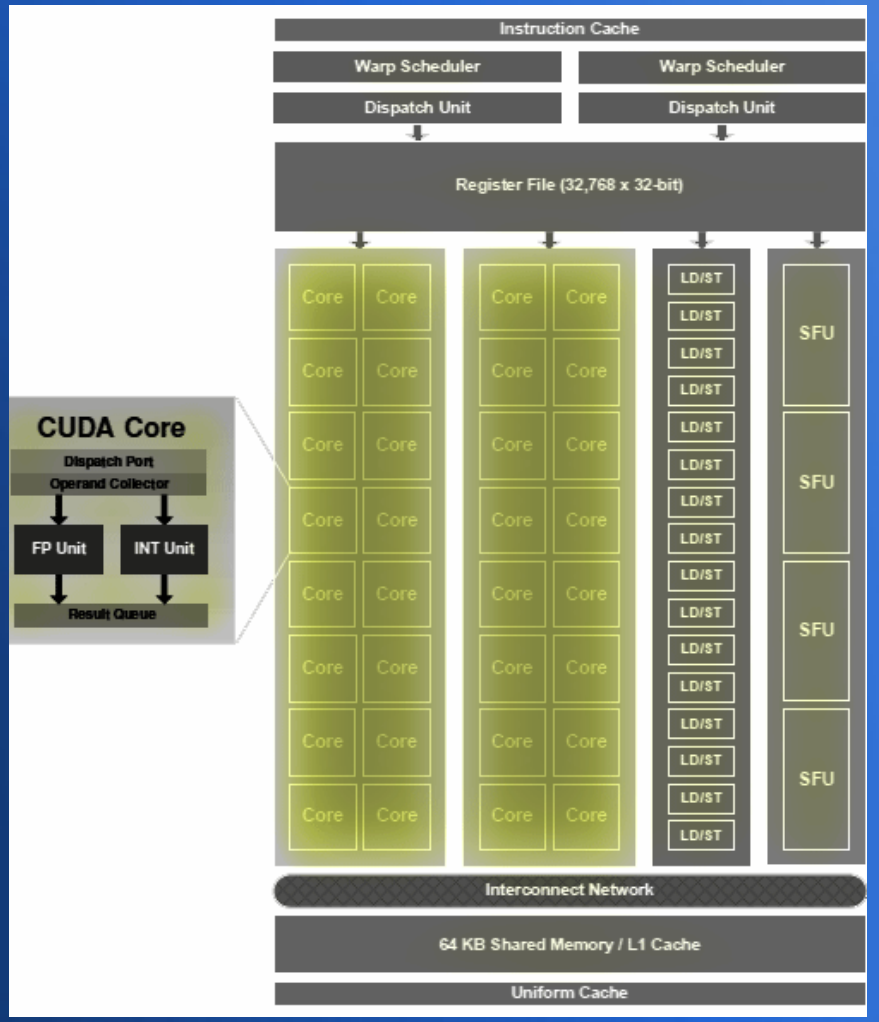


Kevin Marco Erler
AIU Jena

NVIDIA Tesla C2070 (3G)

- **14 Streaming Multi-processors (SM´s)**
  - 448 SP
  - 112 Thread-Blöcke
  - 21504 Threads
  - 672 Warps

gleichzeitig.



Kevin Marco Erler
AIU Jena

NVIDIA: G80 → GT200 → Fermi

GPU → GPGPU → NVIDIA → Architekturen → GF100 (Fermi)

	G80	GT200	Fermi
Release Year	2006	2008	2010
Fabrication Process	90 nm	55 nm	40 nm
Number of Transistors	681 million	1.4 billion	3.0 billion
Streaming Multiprocessors (SM)	16	30	16
Streaming Processors (per SM)	8	8	32
Streaming Processors (total)	128	240	512
Single Precision FP Capability	128 MAD ops/clock	240 MAD ops/clock	512 FMA ops/clock
Double Precision FP Capability	None	30 FMA ops/clock	256 FMA ops/clock
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

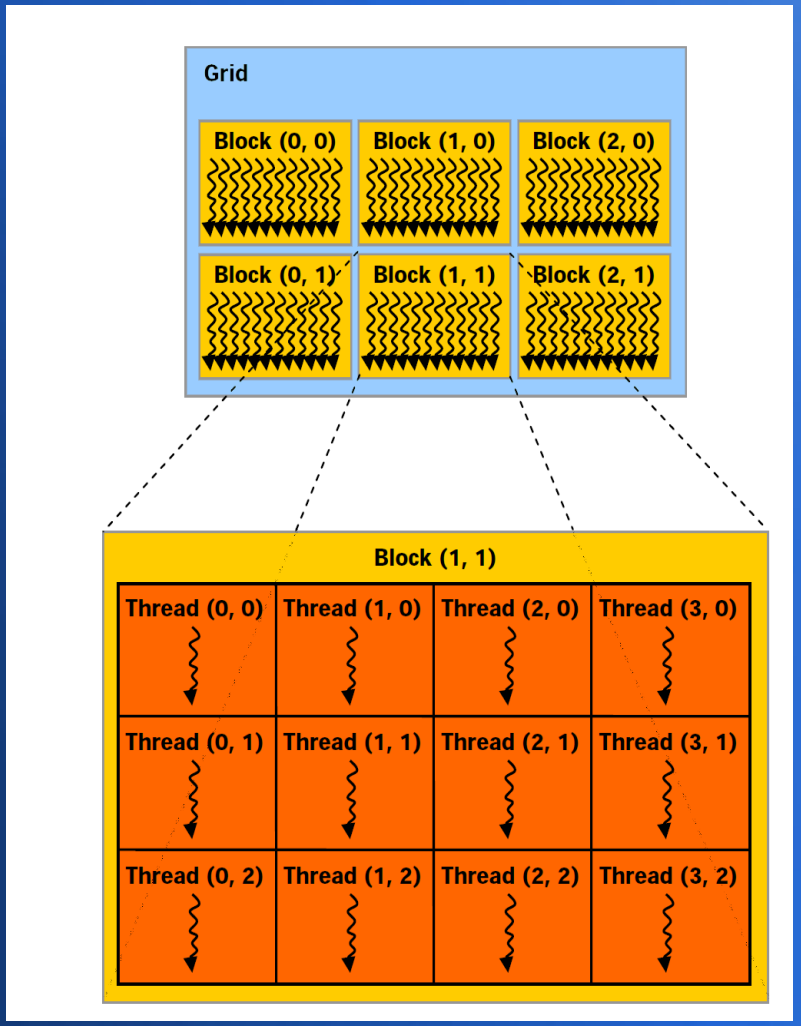
Tab. 1: Comparison of the three major CUDA GPU architecture specifications [NVI09f p. 11] Note that the number of processors are maximum values.

Kevin Marco Erler
AIU Jena

CUDA-GPU – Virtueller Kontext

GPU → GPGPU → CUDA

- **Thread-Hierarchie**
  - Grids (2D → 3D) (Gruppe von Blocks)
  - Thread blocks (3D) (Gruppe von Threads)
  - Threads (1D)



Kevin Marco Erler
AIU Jena

CUDA-GPU – Virtueller Kontext

GPU → GPGPU → CUDA

- **Beschränkungen:**
  - ~5 GB globaler Speicher
  - sonstige Speicherbegrenzungen
  - max. Threads per Block: 1024
  - max. Threads per blockdim: 1024x1024x64 (x*y*z)
  - max. Blocks per Griddim: 65535x65535x65535 (x*y*z)

CUDA-GPU – Virtueller Kontext

GPU → GPGPU → CUDA

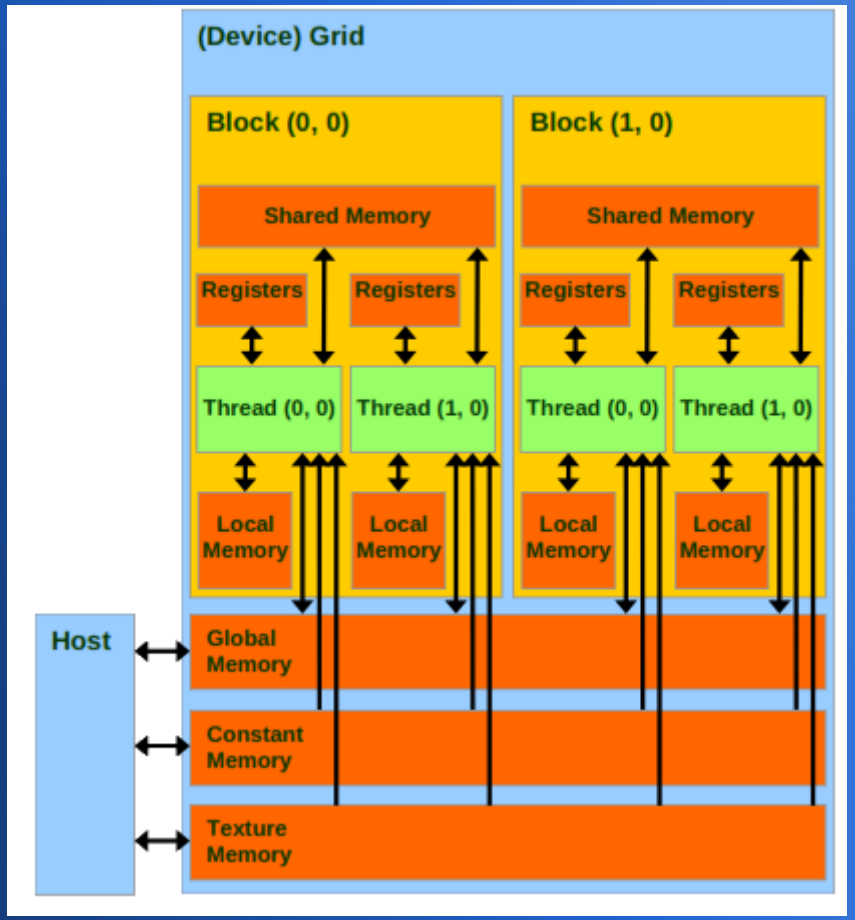
- **ID-Lokale Objekte** (implizit definiert in **cuda.h**):
  - **threadIdx.[x|y|z]** block-lokale Thread-ID
  - **blockIdx.[x|y|z]** grid-lokale Block-ID
  - **blockDim.[x|y|z]** Anzahl Threads pro Block-Dim.
  - **gridDim.[x|y|z]** Anzahl Blocks pro Grid-Dim.
- **Thread-Identifizierung:**
  - nur via Linearisierung:
    TID = threadIdx.x + blockDim.x*blockIdx.x

Kevin Marco Erler
AIU Jena

CUDA-GPU – Speichertypen

GPU → GPGPU → CUDA

- **Speicherhierarchie**
  - Pro Thread:
    - lokaler Speicher (gl. Mem.)
    - Register (on-Chip)
  - Pro Thread-Block:
    - Gemeinsame Speicher (Shared Memory; on-Chip)
  - Pro Device-Anwendung:
    - Globaler Speicher

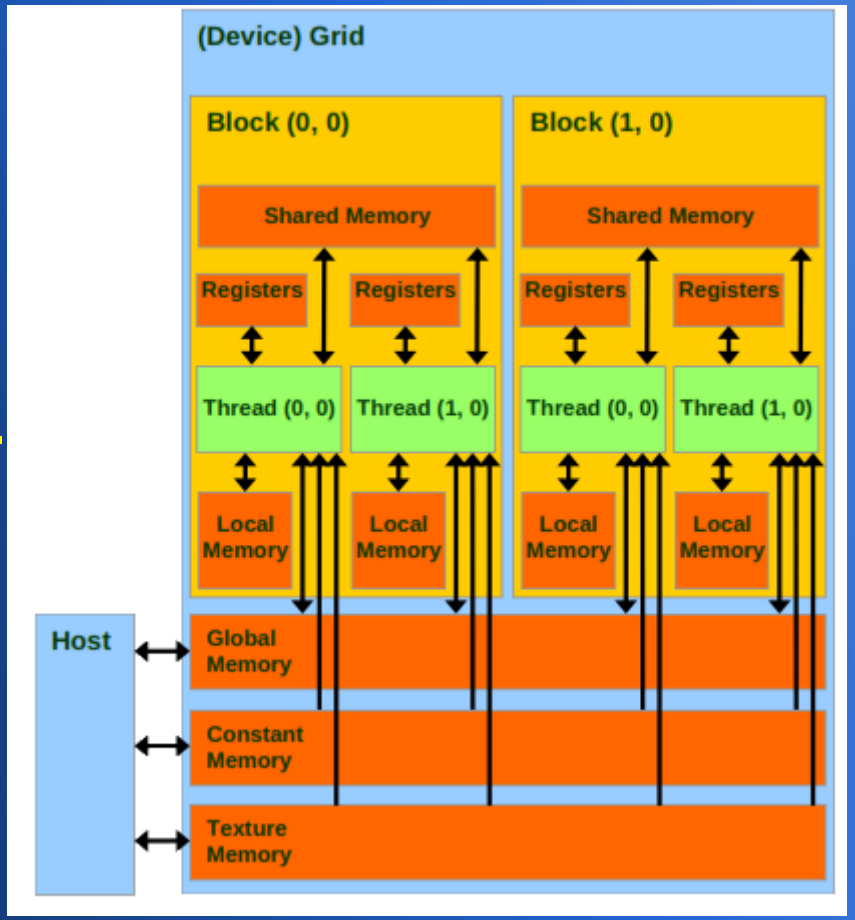


Kevin Marco Erler
AIU Jena

CUDA-GPU – Speichertypen

GPU → GPGPU → CUDA

- **DEVICE kann:**
  - R/W per-Thread register
  - R/W per-Thread local mem.
  - R/W per-Block shared mem.
  - R/W per-Grid global Mem.
  - weitere Speicherzugriffe
- **HOST kann:**
  - Daten zw. HOST-RAM & DEVICE global Memory austauschen



Kevin Marco Erler
AIU Jena

CUDA-GPU – Speichertypen

GPU → GPGPU → CUDA

- **Variablenqualifier:**

Var. Deklar.	Speicher	Scope	Lebenszeit
auto (keine arrays)	Register	Thread	Kernel
auto (arrays)	Local	Thread	Kernel
__shared__ int var;	Shared	Block	Kernel
__device__ int var	Global	Grid	App.
__constant__ int constVar	Constant	Grid	App.

Kevin Marco Erler
AIU Jena

CUDA - Programmstruktur

GPU → GPGPU → CUDA

- **Host-Teil (CPU)**
  - herkömmlicher Code in C/C++
  - i.d.R. sequentielle Abschnitte
  - normale Compiler → gcc/g++ → OBJ1
- **Device-Teil (GPU)** → Co-Prozessor für HOST
  - C/C++ plus CUDA-Erweiterungen
  - Compiler: nvcc → GPU-Code (PTX) → OBJ2
  - Kernel (Func) werden gleichzeitig von vielen Threads ausgeführt

Kevin Marco Erler
AIU Jena

CUDA - Programmstruktur

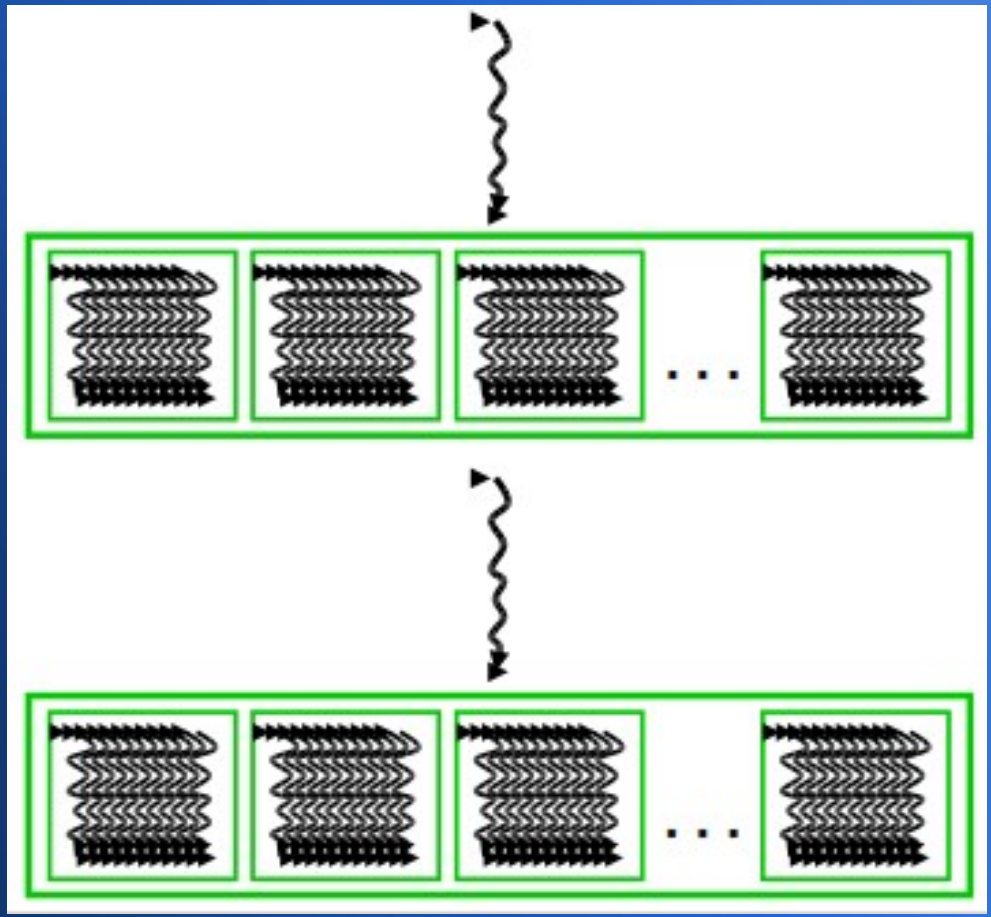
GPU → GPGPU → CUDA

Serial Code (HOST)

Parallel Kernel (DEVICE)

Serial Code (HOST)

Parallel Kernel (DEVICE)



Kevin Marco Erler
AIU Jena

CUDA – Globaler Speicher

GPU → GPGPU → CUDA

- Schnittstelle zw. HOST und DEVICE
- für alle GPU-Threads sichtbar
- hohe Latenz (500-600 Taktzyklen)
- Speichermanagement & Ausführungsmanagement übernimmt der HOST (Allozierung, Datentransfer, CALL, Freigabe)
  → GPU ist reine Ausführungseinheit!

Kevin Marco Erler
AIU Jena

HOST-DEVICE-Programmfluss

GPU → GPGPU → CUDA

(1) HOST alloziert Speicher auf DEVICE:
cudaMalloc()

(2) HOST kopiert HOST-data auf DEVICE-Mem.
cudaMemcpy(dest,src,sizeof,H↔D|D↔D)

(3) HOST löst CUDA-Func.-Call aus

(4) DEVICE führt DEVICE-Kernel´s aus

(5) HOST kopiert DEVICE-data zurück in H-Mem.

(6) HOST gibt D.-data wieder frei: **cudaFree()**

Kevin Marco Erler
AIU Jena

CUDA – Funktionsqualifier

GPU → GPGPU → CUDA

- HOST- & DEVICE-Funktionen werden durch CUDA-Erweiterungen näher spezifiziert:

	Ausgeführt auf:	Aufrufbar von:
__device__ T DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ T HostFunc()	host	host

Kevin Marco Erler
AIU Jena

CUDA – Kernelaufruf

GPU → GPGPU → CUDA

- nur der HOST kann DEVICE-Kernel-Calls ausführen
- Anwendungsformat für Kernel-Aufruf:
  `foo(parameter);`

CUDA – Kernelaufruf

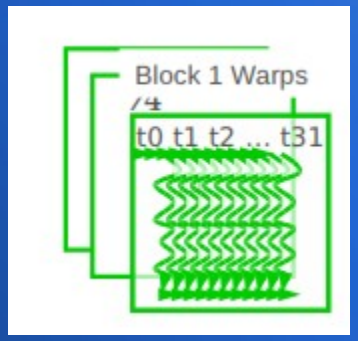
GPU → GPGPU → CUDA

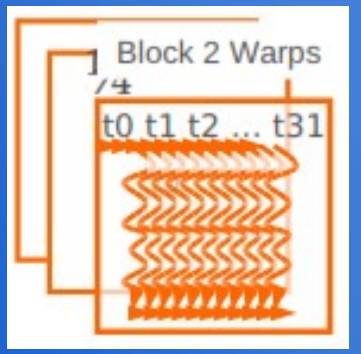
- nur der HOST kann DEVICE-Kernel-Calls ausführen
- Anwendungsformat für Kernel-Aufruf:
  `foo<<<gridDim,blockDim>>>(parameter);`
  - foo = Funktions-/Kernel-Bezeichner
  - GridDim = Anzahl Blocks
  - BlockDim = Anzahl Threads per Block
- HINWEIS: Beschränkungen beachten!

Kevin Marco Erler
AIU Jena

CUDA – Threadausführung

GPU → GPGPU → CUDA





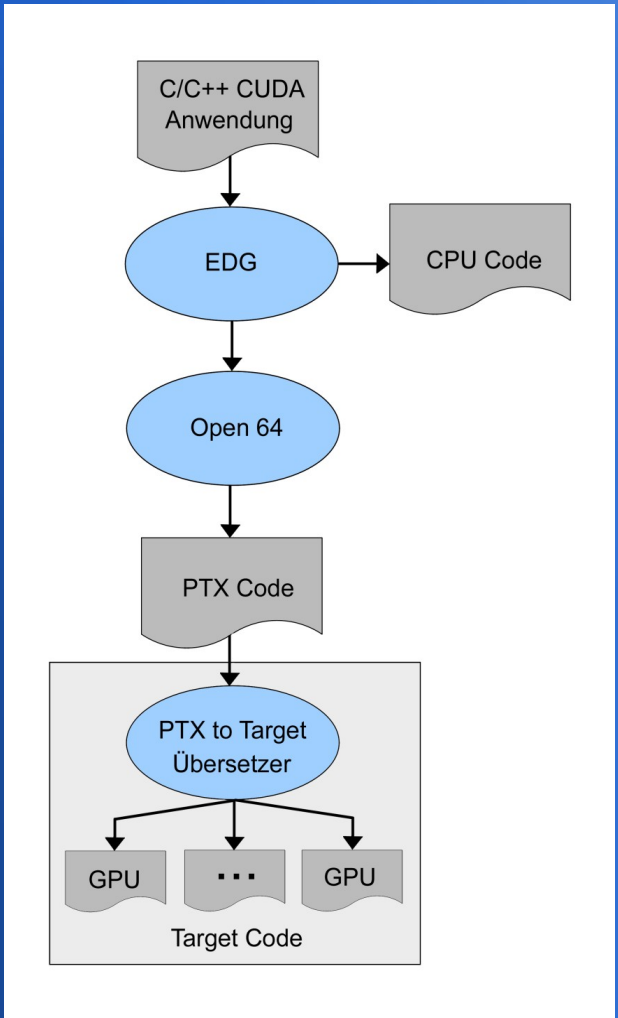
- Threads sind in Blocks organisiert
- jeder Thread führt grundsätzlich eine Kernel-Kopie parallel aus
- GPU fast dabei jeweils 32 Threads zu einem Warp zusammen → max. 48 Warps bei Fermi-GPU´s
- alle Threads in einem Warp führen zum selben Zeitpunkt die selbe Operation[en] aus → SIMD / SIMT
- Aufteilung in Warps ist für die Korrektheit des Programm egal
  → aber wichtig für Leistungsoptimierung (Mem.-Acc.)

Kevin Marco Erler
AIU Jena

CUDA – Compiler

GPU → GPGPU → CUDA → Compiler

- Trennung von HOST- & DEVICE-spezifischen Code
- HOST-Code:
  → standard-Compiler
  → OBJ1-Erzeugung
- DEVICE-Code:
  → nvcc → Open64
  → Generierung v. PTX-BIN-Code
  → OBJ2-Erzeugung
- anschließendes Linken beider OBJ-Files durch Std.-Compiler zu Exec.



Kevin Marco Erler
AIU Jena

CUDA – Compiler

GPU → GPGPU → CUDA → Compiler

- **Compile-Aufrufe:** **C/C++** mit **nvcc + gcc/g++**:
- **Release: .cu-File → .cpp-File**

```
nvcc -ccbin /usr/bin/g++ -Xcompiler "Std.-Compiler-
Options" -m64 -gencode=arch=compute_10,code=sm_10
-gencode=arch=compute_10,code=compute_10
-gencode=arch=compute_20,code=sm_20
-gencode=arch=compute_20,code=compute_20 <LIBS>
-lcuda -lcudart -L/usr/local/cuda/lib -L/usr/local/cuda/lib64
-L/usr/local/cuda/include/ -I/usr/local/cuda/lib
-I/usr/local/cuda/lib64 -I /usr/local/cuda/include/ -cuda
src.cu -o src.cpp
```

Kevin Marco Erler
AIU Jena

CUDA – Compiler

GPU → GPGPU → CUDA → Compiler

- **Compile-Aufrufe:** **C/C++** mit **nvcc + gcc/g++**:
- **Release:** **.cpp-File → Exec-File**

```
nvcc -x c++ -ccbin /usr/bin/g++ -Xcompiler "Std.-Compiler-
Options>“ -m64 -gencode=arch=compute_10,code=sm_10
-gencode=arch=compute_10,code=compute_10
-gencode=arch=compute_20,code=sm_20
-gencode=arch=compute_20,code=compute_20 <LIBS>
-lcudart -L/usr/local/cuda/lib -L/usr/local/cuda/lib64
-L/usr/local/cuda/include/ -I/usr/local/cuda/lib
-I/usr/local/cuda/lib64 -I /usr/local/cuda/include/ src.cpp -o
destexec
```

Kevin Marco Erler
AIU Jena

Parallelisierung am AIU

Ende

ENDE

– vielen Dank fürs Zuhören –

Kevin Marco Erler
AIU Jena