Architektur und Programmierung von Grafik- und Koprozessoren

General Purpose Programmierung auf Grafikprozessoren

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Lernziele

1. **Charakteristiken von Grafikprozessoren** - die Studenten wenden das bisher erlernte Wissen über die Charakteristiken von Grafikprozessoren an und lernen, wie diese auf andere Probleme als auf Grafik abgebildet werden können.
2. **GPGPU Programmiermodell** - die Studenten verstehen das generelle GPGPU Programmiermodell und das dem Programmiermodell zugrunde liegende Speichermodell.
3. **Arten von Nebenläufigkeit** - die Studenten lernen CUDA kennen und können einfache Programme damit entwickeln.

Stream Computing und GPGPU

GPGPU

- Derzeit Parallelismus Antwort auf Skalierungsprobleme im Rahmen von Moore's Law (vgl. Vorlesungsteil 1).
- GPUs: hochparallele Many-Core Prozessoren.
- Andererseits: hohe Consumer Nachfrage nach GPUs, dadurch Preis für Chip *Herstellung* vergleichsweise gering
  - Chip Preis selber hängt von Nachfrage ab, z. B. EULAs von Nvidia, die Einsatz von Spielegrafikkarten in Rechenzentren für manche Anwendungen verbieten, erhöhte Nachfrage durch Crypo Currencies etc.

GPGPU

- Entwickler nutzen Grafikkarten für generelle Berechnungen wie Simulationen, Ray Tracing, DNA Sequencing, etc.
- Seit Mitte/Ende der 2000er Einzug von Grafikkarten und Koprozessoren in Rechenzentren.
- Grafikkarten: GPGPU Segment dominiert von Nvidia.
  - wesentlich der Popularität von Nvidia's CUDA API geschuldet, waren erster Hersteller, der entwicklerfreundliche Toolchain veröffentlicht hat.

TOP500 Supercomputer Koprozessoren

"A total of **102** *systems on the list are using* **accelerator / co-processor technology**, *up from 91 on the June 2017 list.* **86** *of these use* **NVIDIA chips**, **12** *systems with* **Intel Xeon Phi** *technology (as Co-Processors), and 5 are using PEZY technology. Two systems use a combination of Nvidia and Intel Xeon Phi accelerators / co-processors. An additional 14 Systems now use Xeon Phi as the main processing unit."*

(TOP500 November 2017 Highlights:
https://www.top500.org/lists/2017/11/highlights/)

TOP500 Supercomputer Koprozessoren

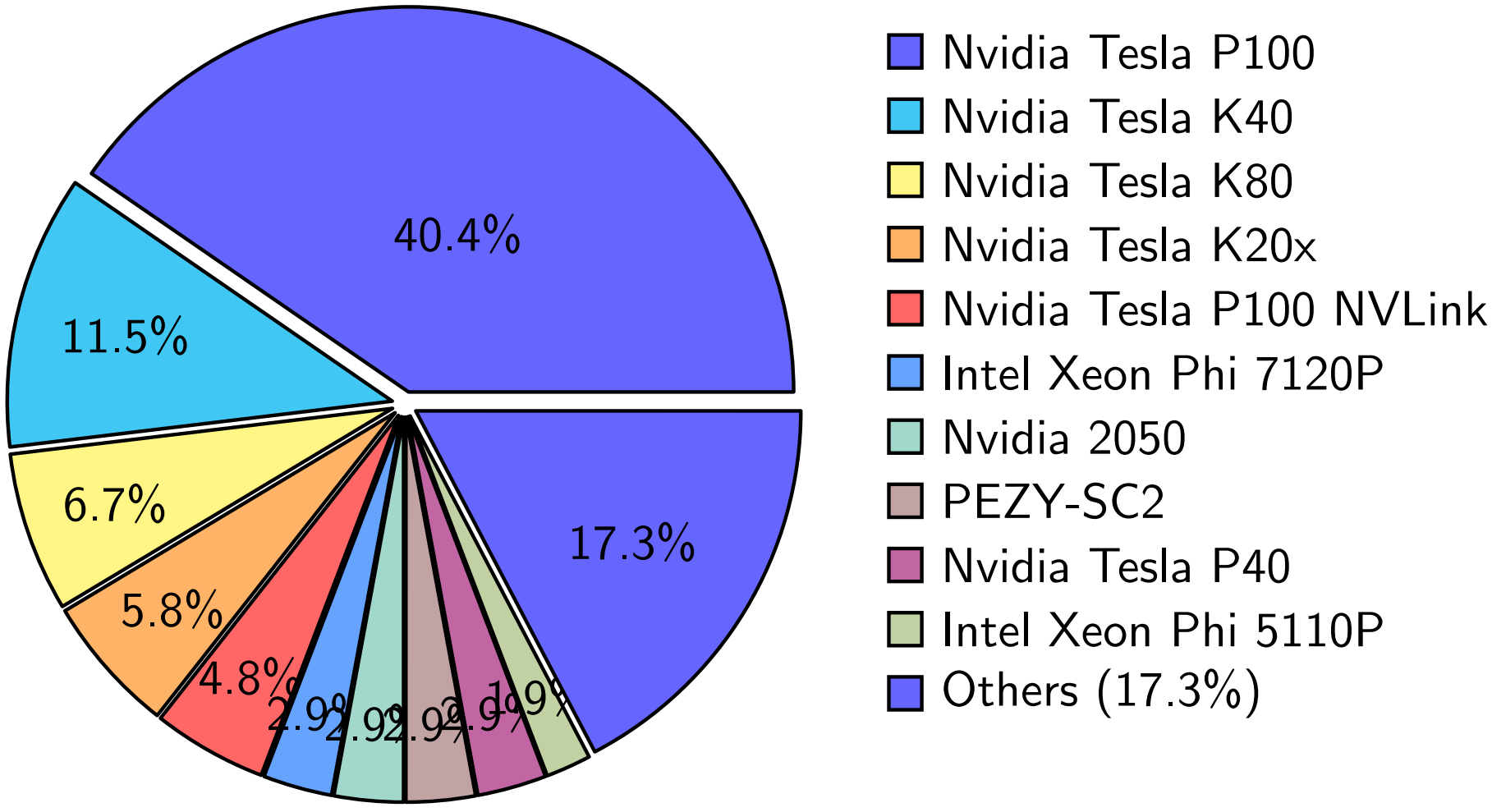
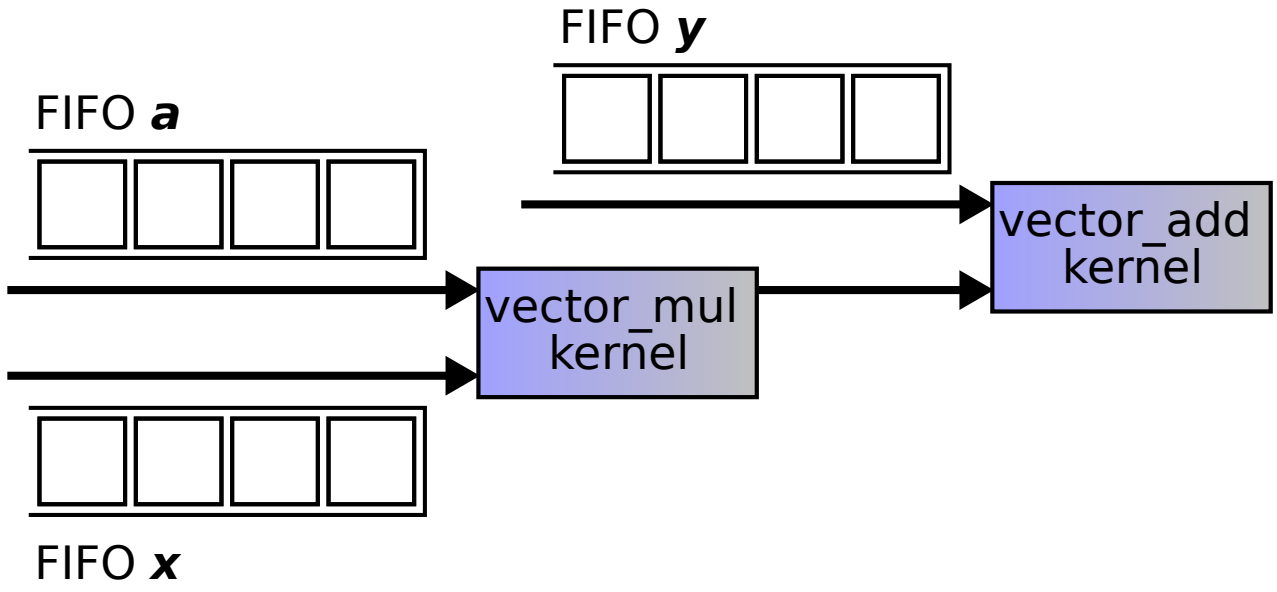


Abbildung: vgl. https://www.top500.org/statistics/list/,
Accelerator/Co-Processor Statistik für TOP500 Release November 2017.

Stream Computing

- Synonym: *Stream Processing*.
- Ähnlich wie SIMD, Datenstrom durchläuft Instruktionssequenz (Compute Kernels)
- SIMD Prozessoren: Structure of Array Datenströme $\Rightarrow$ Stream Processing Modell: FIFOs.
- Wesentlich entwickelt an Stanford University - StreamC, Imagine

Stream Computing



Stream Computing

- Stream Datenstrukturen mappen besonders gut für Datenströme, deren Länge a priori unbekannt ist.
  - FIFO $\Rightarrow$ es können immer Daten nachgeschoben werden.
- *Kernels* ähnlich wie Pipeline Stages, die *asynchron* ablaufen und dedizierte Aufgabe durchführen.
- Spezielle Algorithmen, designed um mit großen Datenmengen umzugehen.

Nvidia CUDA

CUDA Überblick

- CUDA: C++ Spracherweiterung von Nvidia für GPGPU.
- Bildet Stream Processing Modell auf GPU Architektur (vgl. Vorlesungsteil 3) ab.
  - Kernels für Rechenbefehle, FIFOs für Host Interface.
- Single-Source Paradigma: geteilte CPU und GPU Code-Basis, Funktionen werden *annotiert* für CPU (`__host__`), GPU (`__device__`) oder beide (`__host__ __device__`) kompiliert.
- Verschiedene Compiler für CPU Code (gewöhnliche CPU Toolchain) und GPU Code (CLANG basiert, spezielle Kompilation in Nvidias PTX Instruction Set Architecture).

CUDA Überblick

- Host Code: alloziert und verwaltet Ressourcen, implementiert Schnittstelle zur Anwendungslogik, ruft Device Code auf.
- Device Code: implementiert Logik auf GPU.
- Für Host Code stehen APIs zur Verfügung.
- Device Code wird in Subset von C++ mit wenigen Spracherweiterungen geschrieben.
- Device Code wird mittels Kernels organisiert und von vielen Threads gleichzeitig ausgeführt.

CUDA Überblick

- Zwei APIs:
  - *Driver API*: mehr Kontrolle, wie parallele GPU Funktion (*Kernel*, *Compute Kernel*) ausgeführt wird. Kernel wird vom Programmierer in Intermediär-Code (PTX) übersetzt, dieser wird explizit aufgerufen.
  - *Runtime API*: gebräuchlicher. Ausführung von GPU Funktionen als Teil der Programmiersprache.
- Für die meisten Anwendungen kein Unterschied bzgl. Performance.
- Wir betrachten nur das gebräuchlichere Runtime API.
- Dokumentation: "CUDA Toolkit Programming Guide".
  - Inhalte dieses Vorlesungsteils können im Programming Guide vertieft werden.

CUDA-fähige Devices

- CUDA wurde mit G80 Chipsatz eingeführt (2009, GeForce GTX 8800).
- Aktuelle CUDA Versionen unterstützen G80 nicht mehr.
- Devices werden aufgrund von *Compute Capability* kategorisiert (höher ist besser).
  - Niedrigste unterstützte Compute Capability ist 2.0 (Fermi Generation: GeForce GTX 480 & GTX 580 von 2010/11).
  - GTX 1070, GTX 1080: Compute Capability 6.1.
  - Titan V: Compute Capability 7.0.
- Manche Features sind an Compute Capability gekoppelt.

GPU Ausführungsmodell

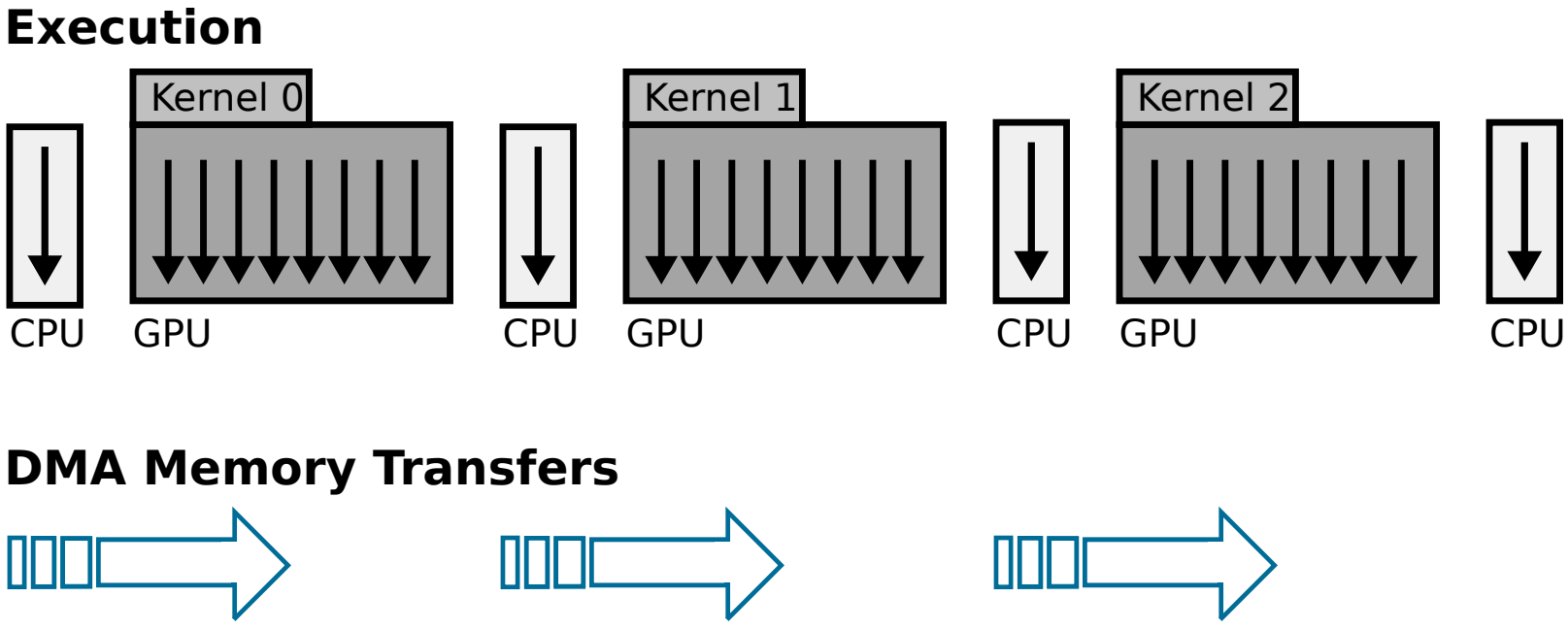


Abbildung: vgl. CUDA Toolkit Programming Guide.

Sequentielle Programmausführung auf CPU ("Host"), parallele Programmausführung auf GPU ("Device"), bidirektionaler Datenaustausch über DMA.

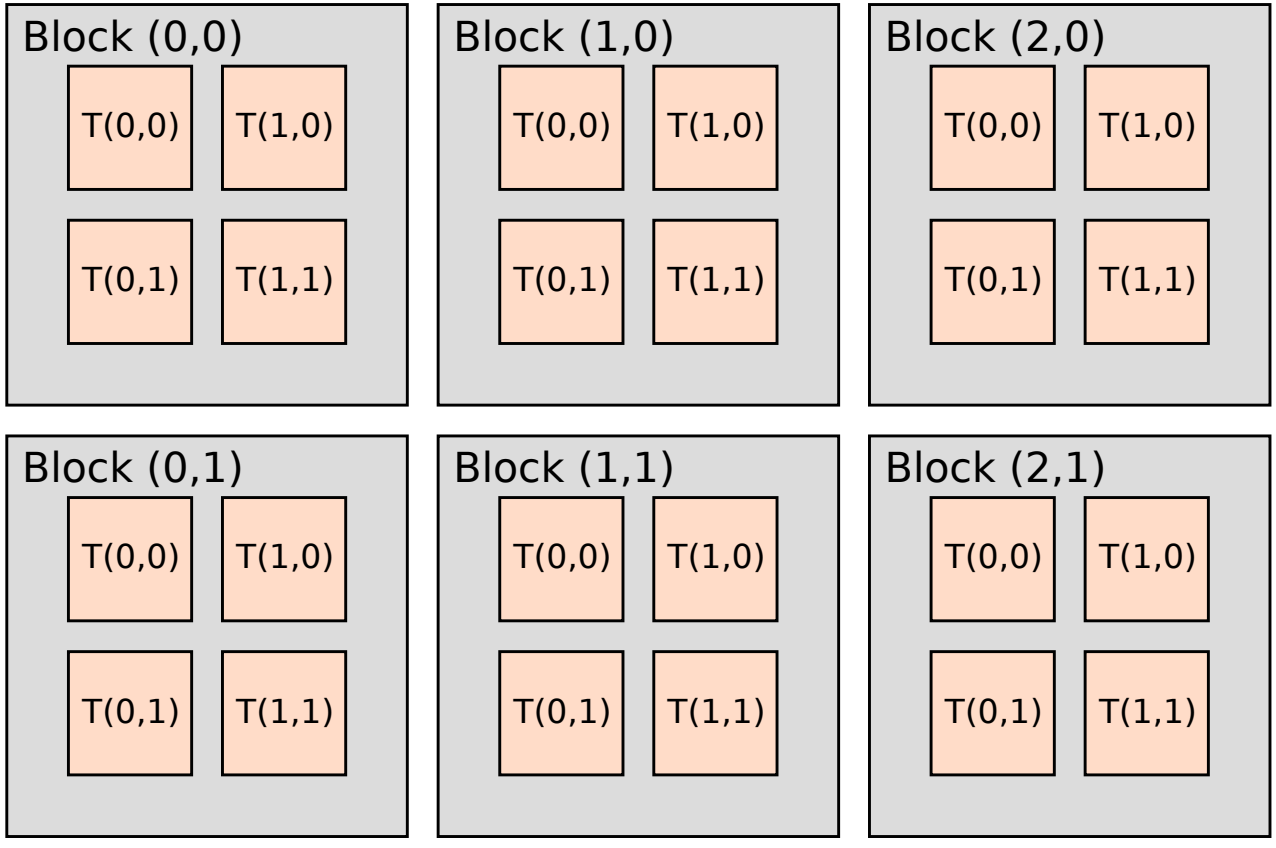
GPU Many-Core Skalierung

Thread-Blocks

- CUDA bildet parallele Programme auf *uniforme Gitter* ab.
- **1D**: Wir *sortieren* N Zahlen. Dazu unterteilen wir die N Zahlen in *Blöcke* (Streifen) der Größe $B \Rightarrow$ Gittergröße: $G = \lceil \frac{N}{B} \rceil$ Blöcke.
- **2D**: Wir rendern ein Bild mit $W \times H$ Pixeln. Dieses unterteilen wir in Blöcke (Kacheln) der Größe $B_x \times B_y$ Threads $\Rightarrow$ Gittergröße: $G_x = \lceil \frac{W}{B_x} \rceil$, $G_y = \lceil \frac{H}{B_y} \rceil$.
- **3D**: Wir führen eine Berechnung auf einem CT Scan durch, dieser besteht aus Z Schichten mit Auflösung $X \times Y$. Wir unterteilen in Blöcke der Größe $B_x \times B_y \times B_z$ und erhalten Gitter der Größe $G_x = \lceil \frac{X}{B_x} \rceil$, $G_y = \lceil \frac{Y}{B_y} \rceil$, $G_z = \lceil \frac{Z}{B_z} \rceil$.

GPU Many-Core Skalierung

Thread-Blocks



2D Gitter der Größe 3×2 mit Blöcken von 2×2 Threads.

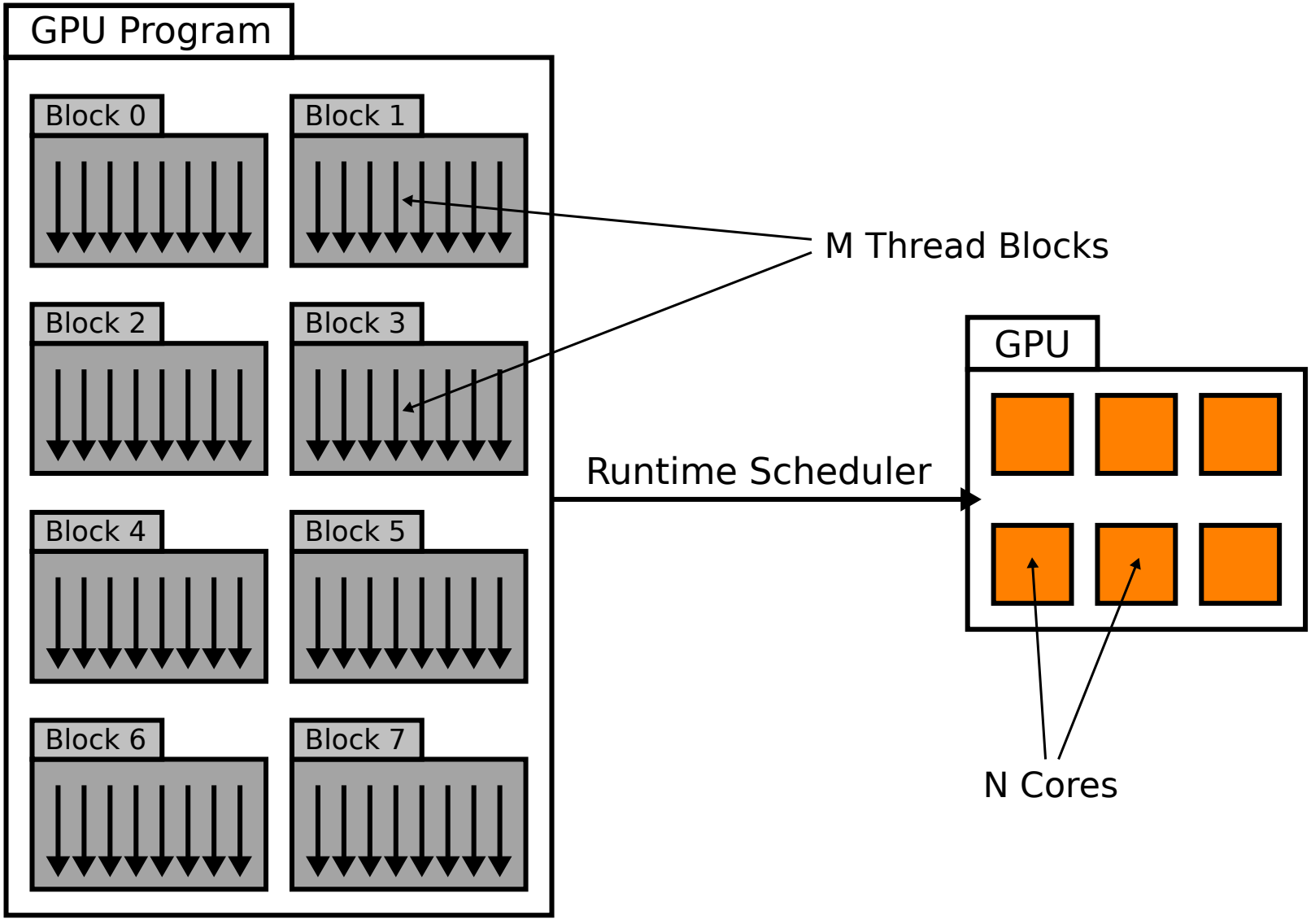
GPU Many-Core Skalierung

- GPGPU Programm bestehend aus M Blöcken.
- N Cores (Nvidia: *Streaming Multiprocessors* (SM)) führen M Blöcke aus.
  - Runtime Scheduler: Verteilung von Blöcken auf Cores, evtl. Lastverteilung (nicht näher spezifiziert).
  - Cores führen potentiell mehrere Blöcke nacheinander aus.
  - Andererseits: Applikation muss genügend Blöcke bereitstellen, um Starvation zu vermeiden.

GPU Many-Core Skalierung

- Je nach Architektur: 16/32/.. Threads pro Block bilden eine *Warp*.
- Warps werden ähnlich wie in SIMD Modell zusammen ausgeführt. Dynamisches Branching / Divergenz: Threads warten aufeinander, bis alle Threads alle Branches abgearbeitet haben.

GPU Many-Core Skalierung



CUDA Speichermodell

- CUDA Speichermodell exponiert schnellen on-chip *shared memory* pro SM.
  - Ähnlich als könnte man mit CPU direkt auf L1-Cache zugreifen.
  - Threads / Warps auf einem SM müssen bei Zugriffen synchronisiert werden.
  - Je nach Architektur z. B. 16 kb, 64 kb o. ä.
- Alle Threads haben Zugriff auf *globalen Speicher* (DDR3). Je nach Architektur gecached oder nicht.
- Threads haben *stark limitierte* Anzahl an Registern (nicht explizit). Sind diese aufgebraucht $\Rightarrow$ Daten in lokalen Speicher (DDR3).
- Spezieller Speicherbereich für Konstanten (*constant memory*).
- Texturspeicher, gecached, schnelle lesende Zugriffe, nicht direkt adressierbar!

CUDA Speichermodell

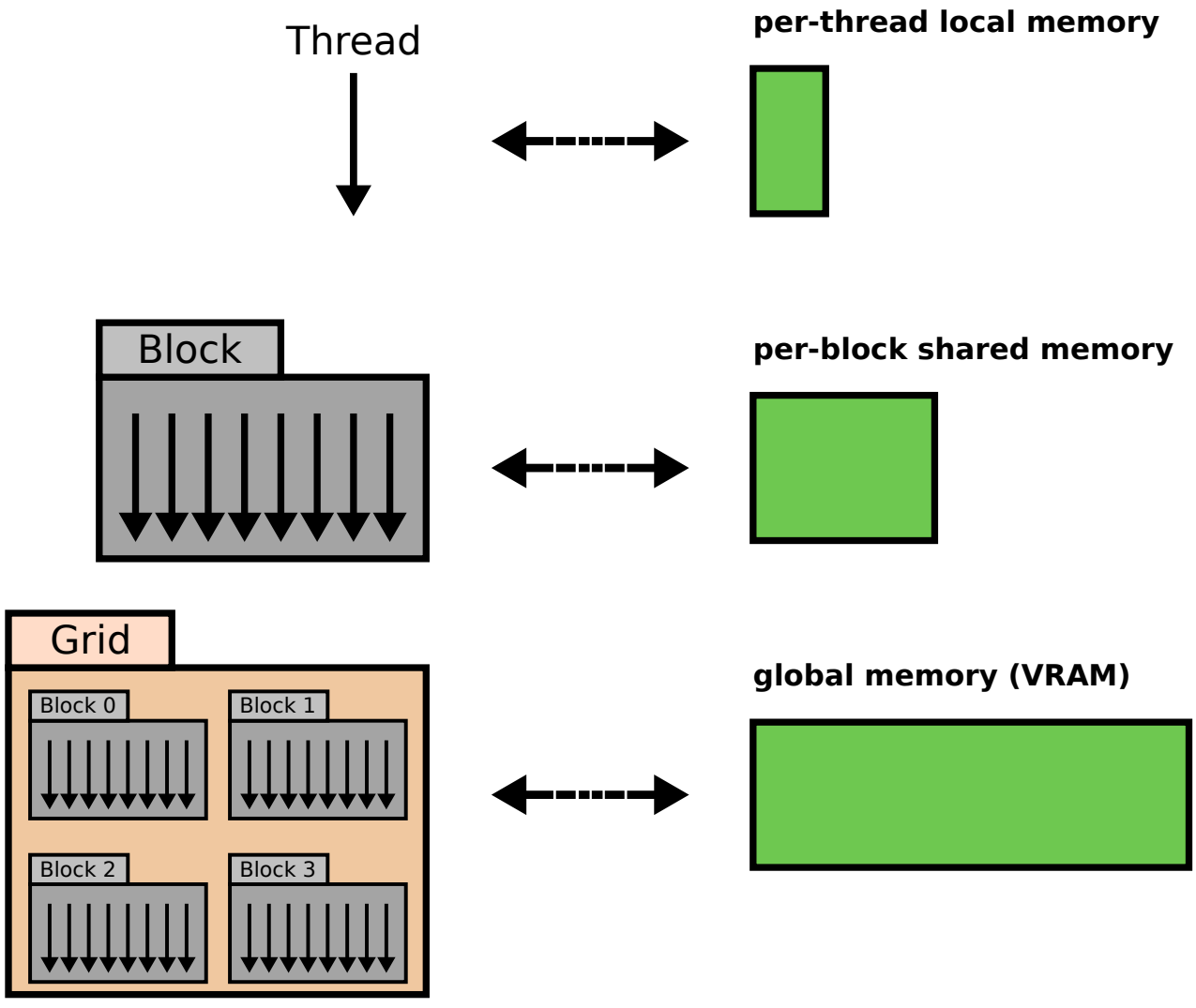


Abbildung: vgl. CUDA Toolkit Programming Guide.

CUDA Speichermodell

Registerspeicher

- vgl. Vorlesungseinheit 3: GPUs haben "riesige" Register Files. Zum Vergleich:
  - Tesla P100: 256 KB Register File pro Core/SM (64 K 32-bit Register) $\Rightarrow$ **14,3 MB Registerspeicher** (gesamte **GPU**) (!)
  - Intel Skylake: 180 Integer Register, 168 Floating Point Vektor Register. Größe nicht ganz klar, vermutlich 256-bit oder 512-bit[1]. Obere Schranke, **28-Core** Skylake Prozessor: (348 $\times$ 512-bit) $\Rightarrow$ *höchstens* **22 KB Registerspeicher** (gesamte **CPU**).
- Warp Scheduler planen mehrere Warps auf SM. Zustand aller aktiven Warps verbleiben in Registern. Wartet eine Warp (z. B. wegen DDR Speicherzugriff), kann andere Warp geplant werden, die z. B. Arithmetik macht $\Rightarrow$ wenig Kosten für Kontext Switch, da Zustand der Warps in Registern.

[1] http://www.agner.org/optimize/blog/read.php?i=962

CUDA Speichermodell

Registerspeicher

Da Zustand der aktiven Warps in Registern verbleibt, ist Umschalten zwischen Warps ausgesprochen schnell.

Nun wird aber Registerallokation zum Optimierungsproblem.

Compiler alloziert Register basierend auf Instruktionen in Compute Kernel. Zu große Kernels $\Rightarrow$ Register Spilling in DDR Speicher ("Höchststrafe").

"Große" Kernel $\Rightarrow$ weniger Kernel können gleichzeitig ausgeführt werden.

Tool, um optimale *Auslastung* basierend auf Register Count zu berechnen: "Occupancy_Calculator.xls".

CUDA Speichermodell

Lokaler Speicher (Shared Memory)

- Schneller on-chip Speicher, ähnlich wie L1 Cache.
- Alle Threads, die gemeinsam auf SM ausgeführt werden, teilen sich Shared Memory.
- Speicher muss dediziert von Host alloziert werden (keine Speicherallokation aus GPU Programm selbst heraus).
- Zugriffe müssen synchronisiert werden (eingebaute Funktion `__syncthreads()`, s. u.).
- Niedrige Zugriffslatenz (ca. 30-90 Taktzyklen).
  - Zum Vergleich: L1 Cache Hit auf CPU ca. 4-5 Taktzyklen.

CUDA Speichermodell

Globaler Speicher (DDR)

- Enorm hohe Bandbreite (Nvidia P100 z. B. bis zu 720 GB/s[2])
- Dafür enorm hohe Latenz (200-800 Taktzyklen).
- Auf GPUs ist es wichtig, immer "genügend" Compute Instruktionen zu haben, um diese Latenz zu verstecken.
- Zugriffe auf globalen Speicher müssen koaleszierend sein (benachbarte Threads lesen nicht aus und schreiben nicht in gleiche Speicherzelle).
  - z. B. durch 16-Byte alignierte Datenstrukturen.

  Sonst Bankkonflikte $\Rightarrow$ noch höhere Latenz.

[2] https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/

CUDA Speichermodell

Texturspeicher

- Spezieller, gecachter Speicher optimiert für lokale Speicherzugriffe (Implementierungsdetails unbekannt).
- 1D, 2D und 3D Texturen.
- Genau wie Grafik-APIs: bounds checking (Wrap, Clamp, Mirror etc.), Hardware Support für lineare Interpolation.
- Vor Kepler Architektur (GTX 680): fixe Anzahl an Texturen, sehr unflexibel. Seit Kepler: "Texture Objects" (a.k.a. "Bindless Textures").
  - Vorher: Texture Atlas, um mehrere Texturen in eine zu packen.
  - Heute: variable Anzahl GPU Texturen.

CUDA Speichermodell

Konstanter Speicher

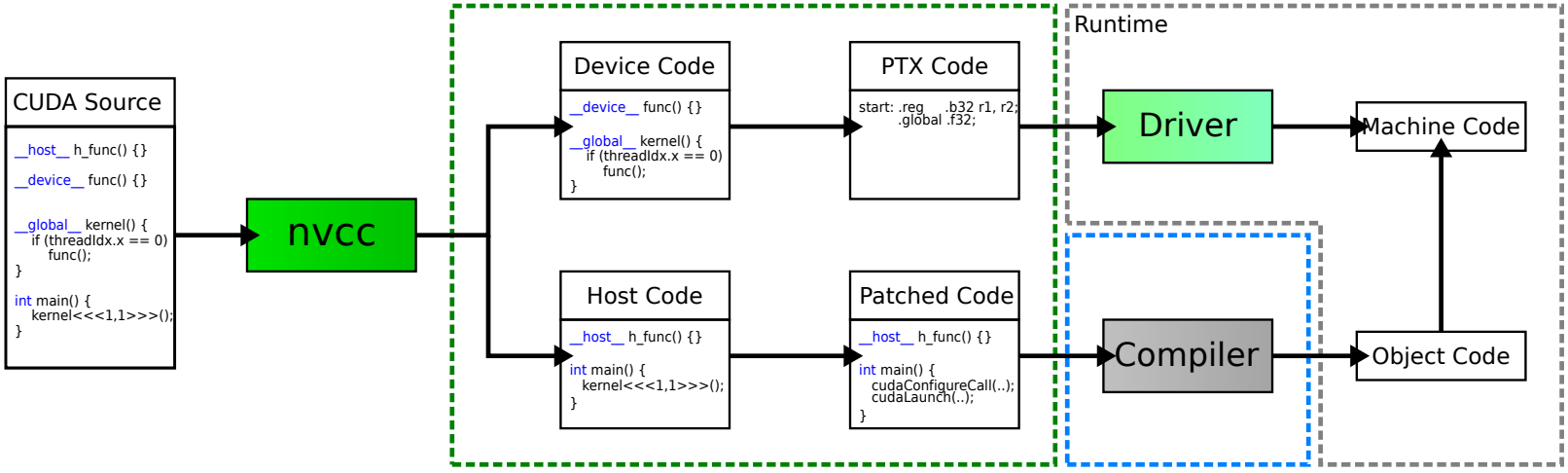
Sehr kleiner (je nach Architektur etwa 64 kb *insgesamt*), gecachter Speicher für Konstanten.

Konstanter Speicher in DDR3, je nach Architektur etwa 8 kb Cache pro SM.

Ein Thread liest von Speicheradresse, Broadcast an alle anderen Threads. Broadcast 4 Taktzyklen.

$\Rightarrow$ verwende konstanten Speicher, wenn alle Threads in einem Block von der gleichen Speicheradresse lesen. Sonst Cache Misses (sehr teuer).

Kompilieren mit nvcc und CUDA Runtime API



1. nvcc verarbeitet `.cu` Dateien mit Host- und Device Code
   - Device Code $\Rightarrow$ PTX (Nvidias GPU ISA).
   - Host Code $\Rightarrow$ gepatchter Host Code, Routinen zum Laden von PTX und Aufruf von GPU Kernels.
2. Runtime: Treiber kompiliert PTX in Chip-spezifischen Maschinencode
3. Host Programm ruft GPU Maschinencode auf.