# Architektur und Programmierung von Grafik- und Koprozessoren

## General Purpose Programmierung auf Grafikprozessoren

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

# Lernziele

1. **Charakteristiken von Grafikprozessoren** - die Studierenden 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 Studierenden verstehen das generelle GPGPU Programmiermodell und das dem Programmiermodell zugrunde liegende Speichermodell.
3. **Arten von Nebenläufigkeit** - die Studierenden 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 niedrig.
  - 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 Crypto 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

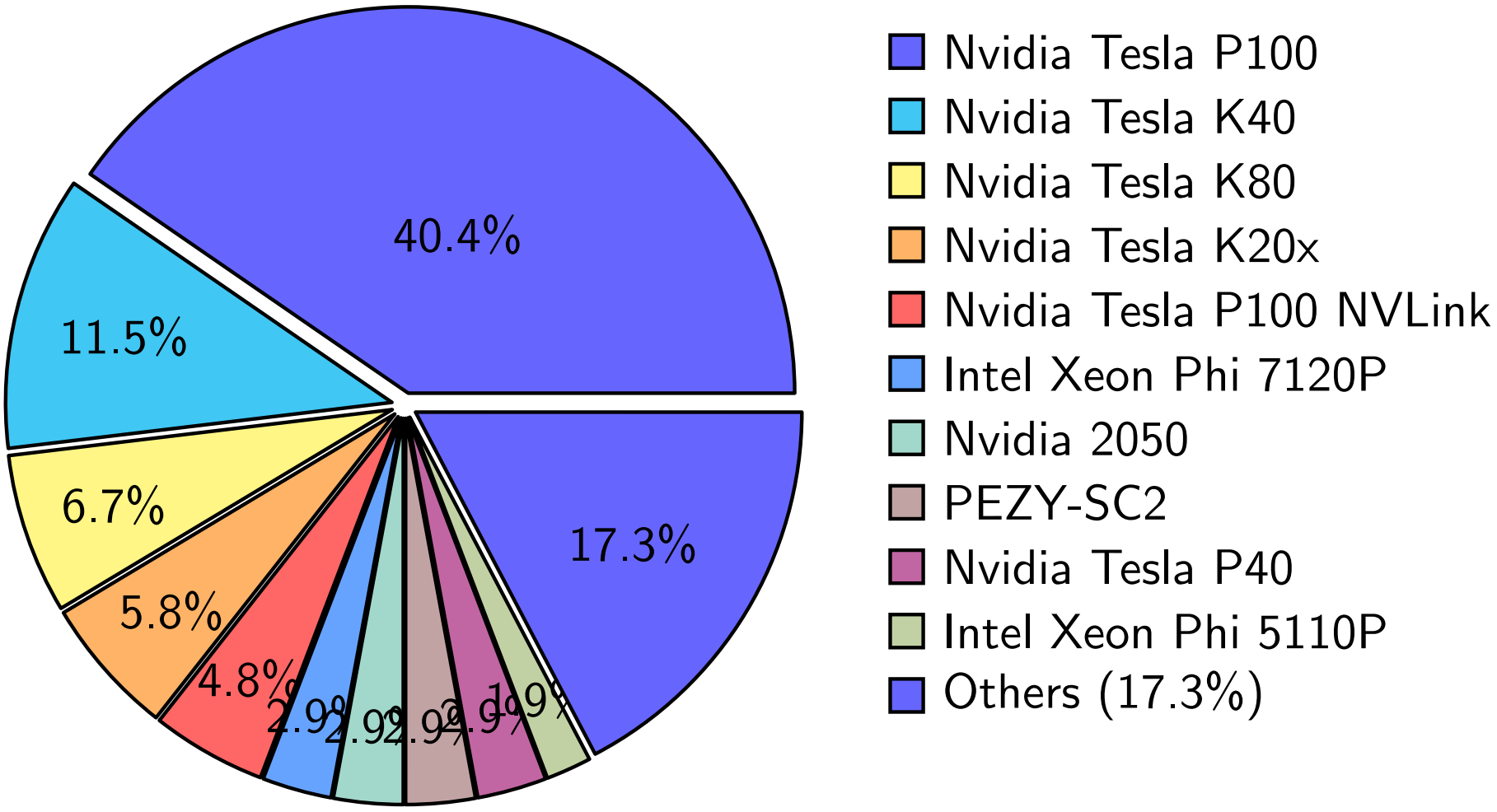
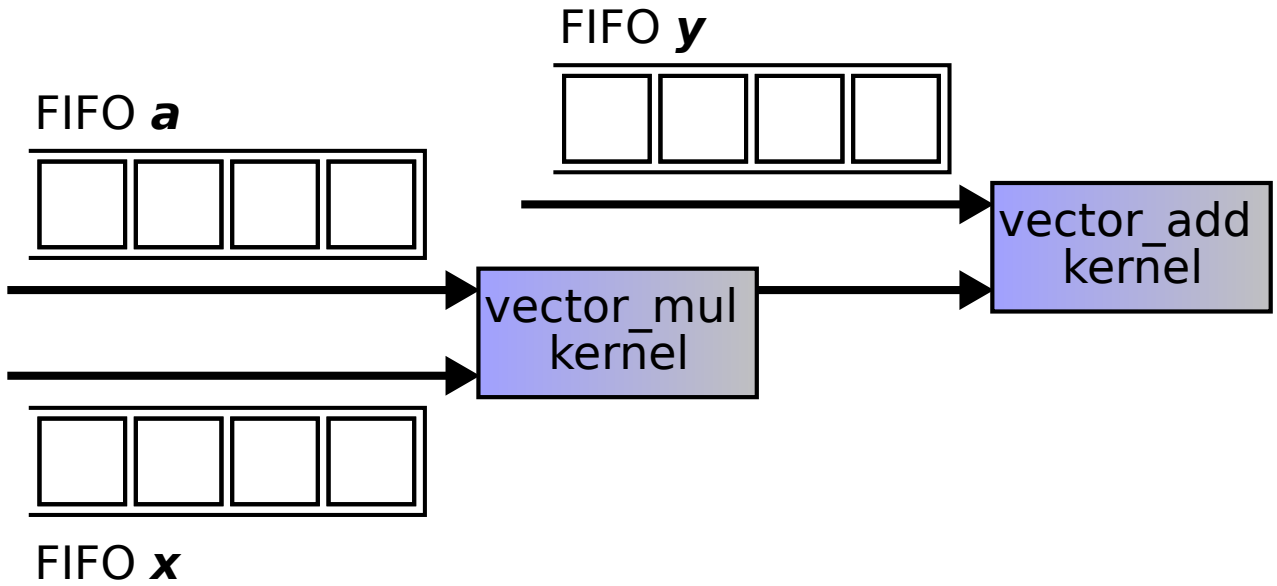



Abbildung: vgl. https://www.top500.org/statistics/list/, Accelerator/Co-Processor Statistik für TOP500 Release November 2017.

# 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

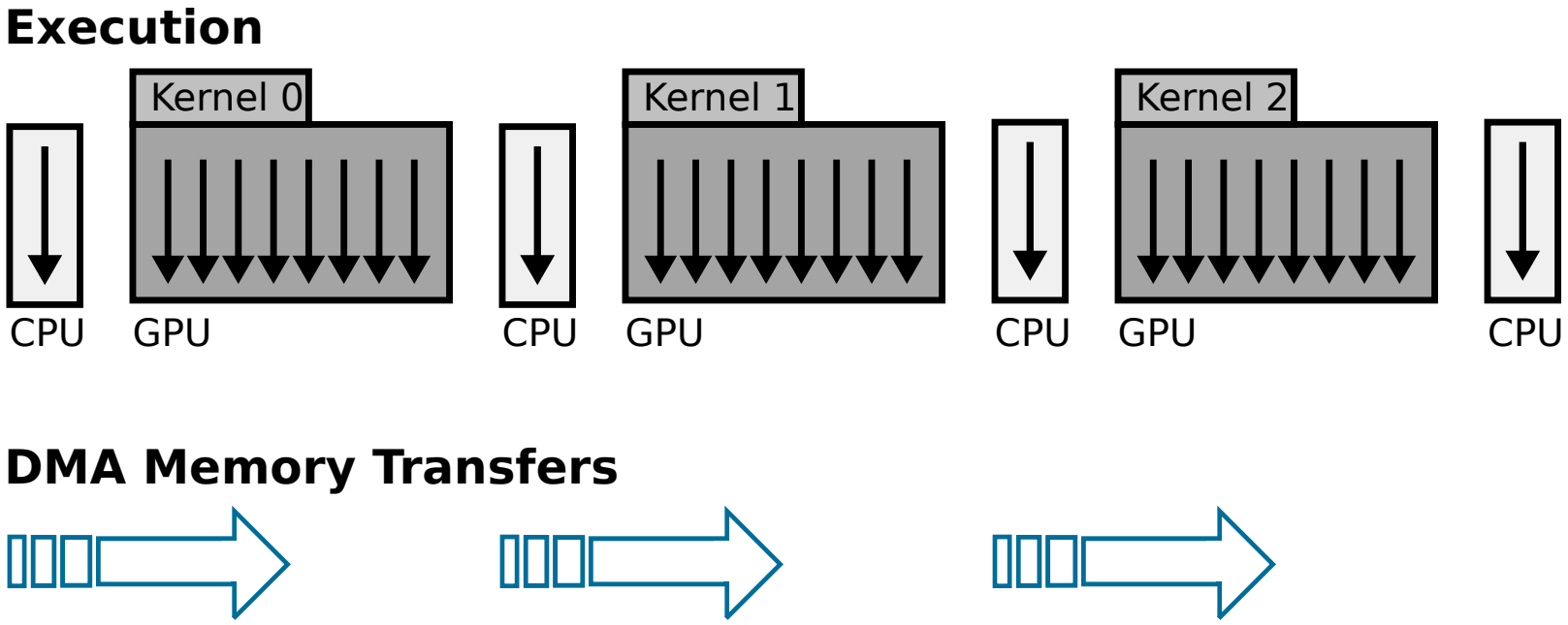



Abbildung: vgl. CUDA Toolkit Programming Guide.

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 $G_x = \lceil \frac{X}{B_x} \rceil$, $G_y = \lceil \frac{Y}{B_y} \rceil$, $G_z = \lceil \frac{Z}{B_z} \rceil$.