Architektur und Programmierung von Grafik- und Koprozessoren

General Purpose Programmierung auf Grafikprozessoren

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

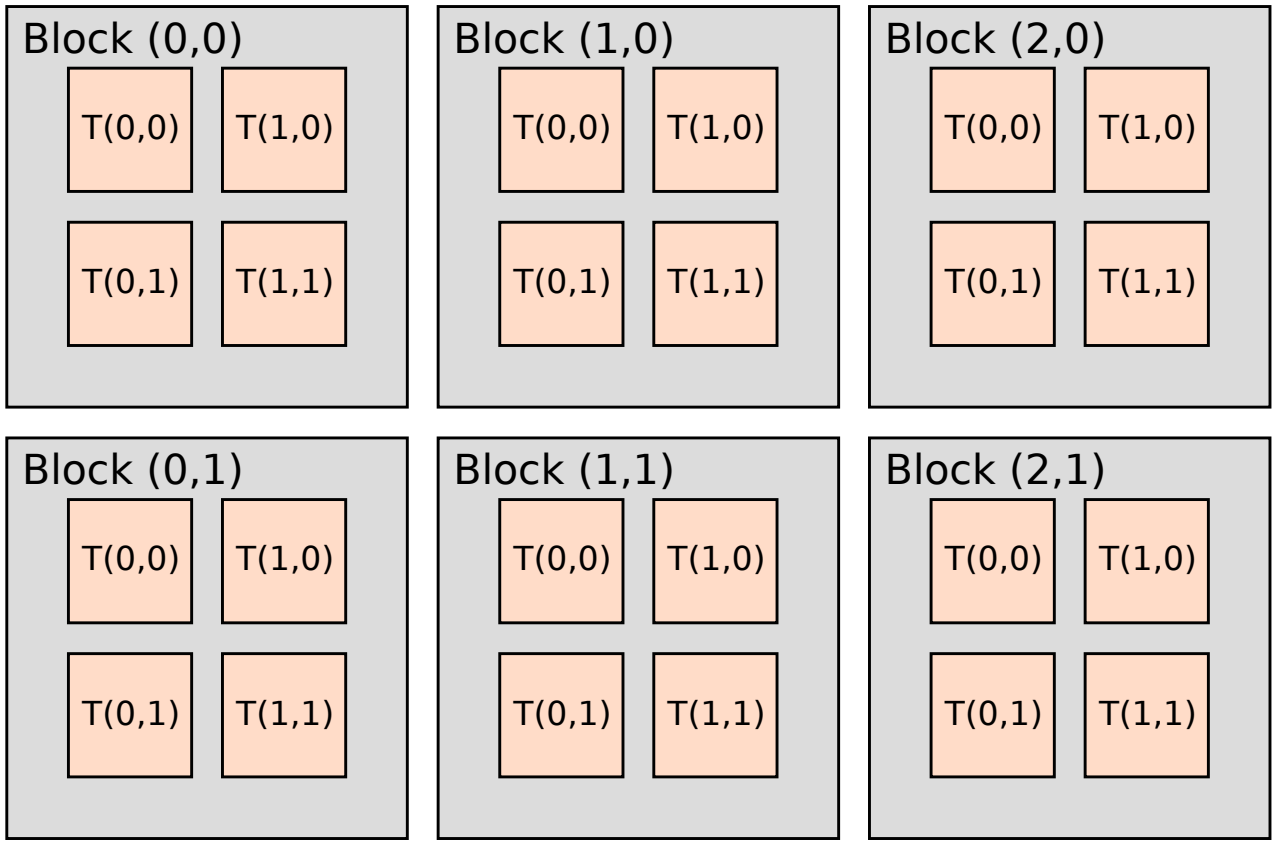
GPU Many-Core Skalierung

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.

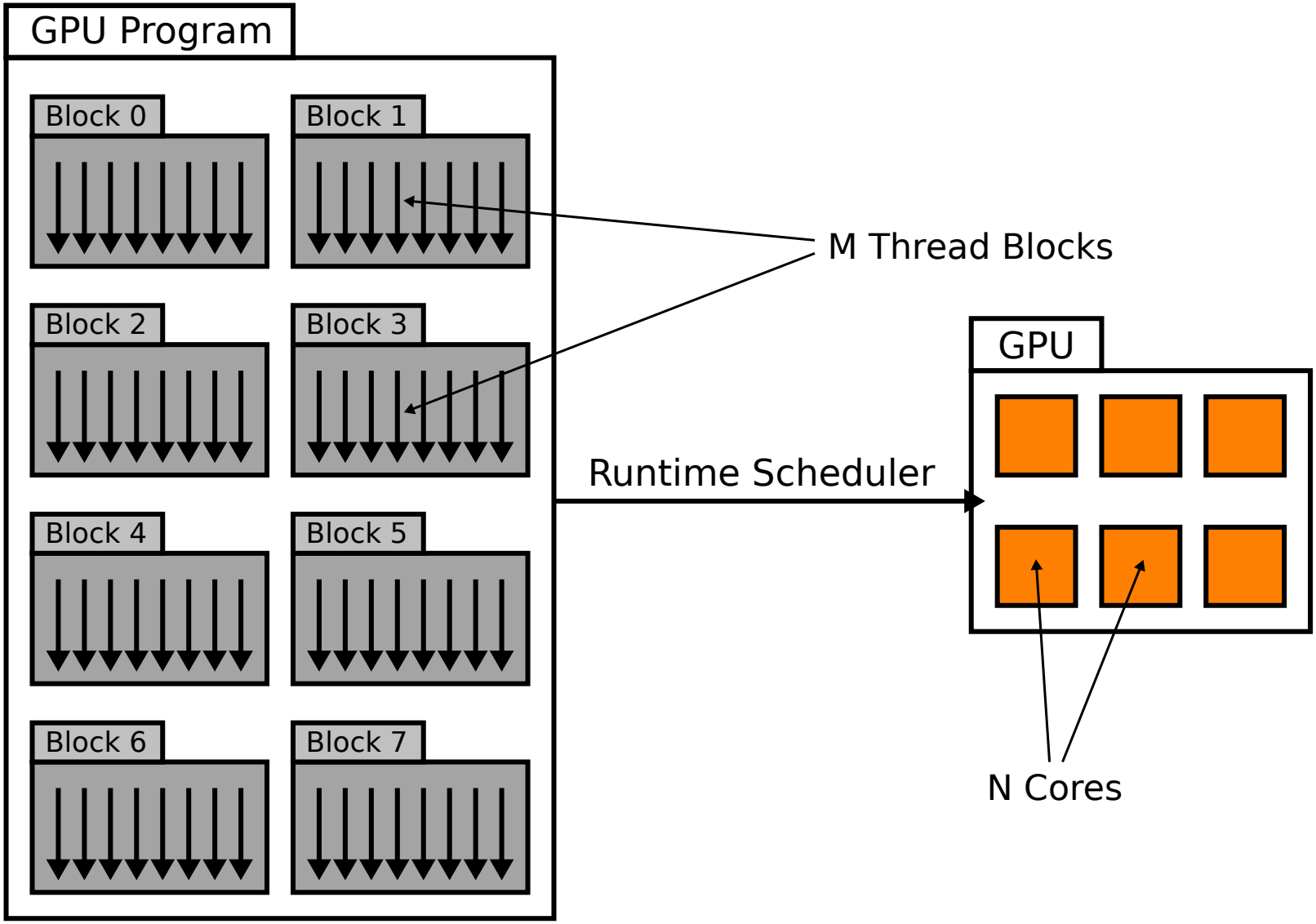
GPU Many-Core Skalierung

- 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

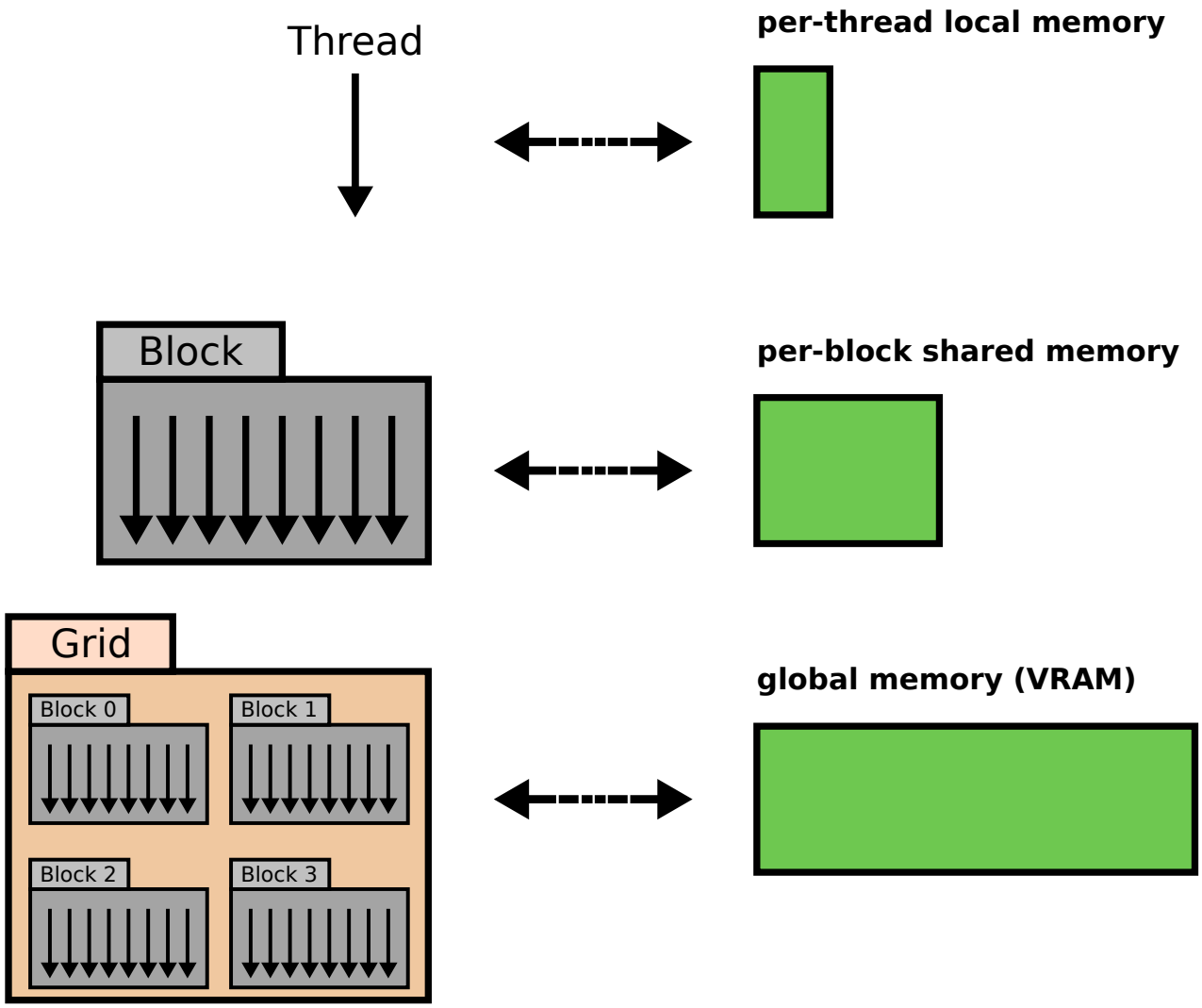


Abbildung: vgl. CUDA Toolkit Programming Guide.

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.

Registerallokation ohnehin schon *schwieriges* Optimierungsproblem. Durch mehrere Warps nun noch komplizierter.

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

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

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

CUDA Speichermodell

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

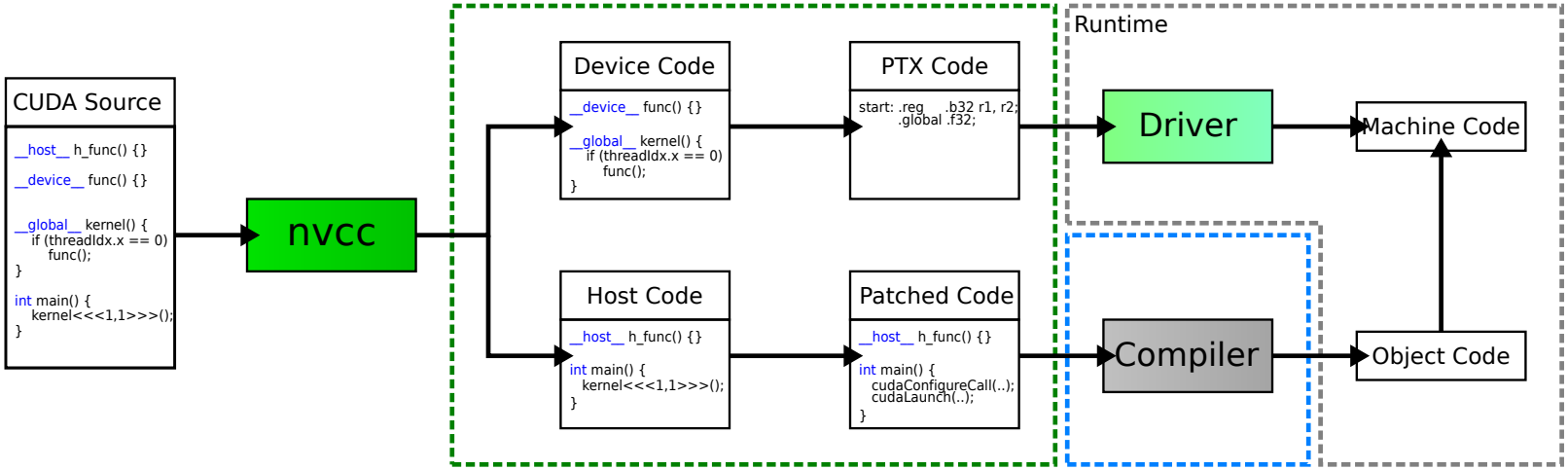
Sehr kleiner (je nach Architektur etwa 64 kb *insgesamt*), gecachter Speicher für Konstanten.

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.

Kompilieren mit nvcc und CUDA Runtime API

- Device Compiler Teil von nvcc übersetzt nach PTX (und alternativ nach cubin).
- Treiber übersetzt *zur Laufzeit* PTX Code in Maschinencode (just-in-time (JIT) compilation).
  - Maschinencode wird im User-Verzeichnis gecached, JIT nur bei erstem Programmstart nach Rekompilieren.
  - JIT Compiler kann dediziert für Target-Platform / GPU Architektur optimieren.
- Übersetzen nach cubin $\Rightarrow$ PTX Code editierbar, kann nach nvcc Lauf handoptimiert werden.
- PTX unterstützt 64-bit. Entweder gesamte Toolchain (Host & Device) 32-bit oder 64-bit.

CUDA Programmiermodell

Single Instruction Multiple Thread (SIMT). Ähnlich wie SIMD, jedoch nicht *explizit*.

Keine expliziten SIMD Instruktionen. Thread Funktionen (Kernels) exponieren Instruktionsfluss eines einzelnen Threads.

Nvidia PTX ISA: keine SIMD opcodes (anders als AMD GPUs).

Implizit: alle Threads in Warp führen die gleichen Instruktionen aus. Betritt ein Thread aus der Warp einen Branch (`if..else`), warten alle anderen Threads inaktiv.

CUDA Programmiermodell

Auf einem SM laufen i. d. R. mehrere Warps gleichzeitig. Alle Threads/Warps auf SM haben geteilten (on-chip) Speicher (*shared memory*) (siehe CUDA Speichermodell) $\Rightarrow$ Barrier Synchronisation einzelner Warps.

Kernels werden von CPU aus angestoßen. CPU initiiert auch Speichertransfers. Kernels und andere Operationen asynchron.