

Grafikprozessor (GPU)

Ein Grafikprozessor (Graphics Processing Unit, GPU) ist eine spezialisierte elektronische Schaltung, die entwickelt wurde, um die Berechnung und Darstellung von Bildern und Grafiken auf einem Computerbildschirm zu beschleunigen. GPUs sind besonders gut geeignet für parallele Verarbeitung großer Datenmengen, was sie ideal für Anwendungen wie Videospiele, 3D-Modellierung und maschinelles Lernen macht.

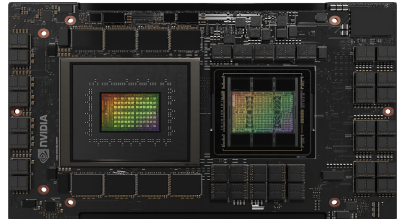
Leistungsfähigkeit von GPUs

Die Leistungsfähigkeit von GPUs wird oft in Teraflops (TFLOPS) gemessen, was die Anzahl der Billionen Gleitkommaoperationen pro Sekunde angibt, die sie ausführen können. Moderne GPUs können je nach Modell und Anwendungsbereich unterschiedliche Leistungsstufen erreichen. Typische Rechengenauigkeiten sind 32-Bit (FP32) und 16-Bit (FP16) Gleitkommazahlen. Bei 64-Bit (FP64) Gleitkommazahlen ist die Leistung in der Regel deutlich geringer.

| GPU- Klasse | Leistung |
|----------------------------|------------------|
| Einsteiger- / Mittelklasse | 2 - 10 TFLOPS |
| High-End-Gaming | 10 - 100 TFLOPS |
| Workstation- / AI-GPUs | 20 - 1000 TFLOPS |

Quelle: GPU Monkey Benchmark NVIDIA RTX 6000 ADA FP32

Quelle: NVIDIA H100 Datasheet



Aufbau und Funktionsweise einer GPU

- **Architektur:** Viele einfache Rechenkerne in *Streaming Multiprocessors (SM)*, optimiert für parallele Berechnung.
- **Speicherhierarchie:**
 - *Global Memory (VRAM)*: Großer, langsamer Speicher für die gesamte GPU.
 - *Shared Memory*: Schneller Zwischenspeicher pro SM.
 - *Register*: Klein, extrem schnell, lokal pro Kern.
- **SIMD-Prinzip:** Gleiche Operation auf mehrere Daten gleichzeitig (Single Instruction, Multiple Data).
- **Thread-Modell:** Threads in *Thread-Blocks* organisiert, diese in einem *Grid*.
- **Spezialisierte Hardware:** Rasterisierung, Texturierung und Shading für Grafikoperationen.

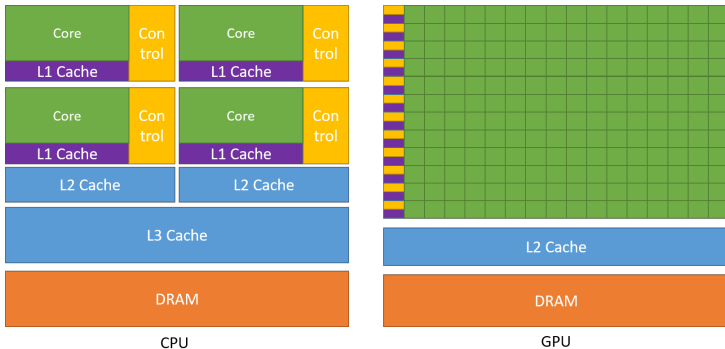


Figure: Quelle: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

Was macht die GPU schnell?

Die GPU ist schnell, wenn sie **sehr viele ähnliche Rechnungen gleichzeitig** machen kann.

- Bild: viele Pixel werden parallel berechnet
- Geometrie: viele Vertices werden parallel transformiert
- KI/Mathe: viele Zahlen in großen Matrizen werden parallel verarbeitet

Was ist wichtig beim Programmieren?

- **Große Aufgaben auf einmal:** lieber 1 große Rechnung als 1000 kleine
- **Daten nicht ständig hin- und herschieben:** CPU ↔ GPU kostet Zeit
- **Wenig Verzweigungen:** wenn viele Threads unterschiedliche Wege gehen, wird es langsamer

GPU-Programmierung: Wie nutzt man sie?

Zwei typische Wege

- **Grafik (Shader):** Programme für Vertex/Pixel in OpenGL/Vulkan/Direct3D/Metal/WebGPU
- **Rechnen (Compute):** GPU als Parallelrechner (Compute Shader oder CUDA/HIP/OpenCL)

PyTorch als "GPU-Programmierung ohne viel API-Aufwand"

In PyTorch sind viele Tensor-Operationen intern GPU-Kernels. Man schreibt Code in Python, aber die GPU rechnet die großen Operationen.

Lernziel

Die **GPU** ist besonders stark bei großen Matrixrechnungen.
Aber: Man muss die Daten **auf der GPU lassen**, sonst frisst Kopieren den Vorteil.

Idee

Wir berechnen einmal eine Matrixmultiplikation auf CPU und einmal auf GPU.

PyTorch Mini-Beispiel: CPU vs GPU (kurz)

```
import torch, time

def run(n, device):
    a = torch.randn(n, n, device=device)
    b = torch.randn(n, n, device=device)

    a @ b                                # warm-up
    if device == "cuda": torch.cuda.synchronize()

    t = time.time()
    a @ b
    if device == "cuda": torch.cuda.synchronize()
    return time.time() - t

print("CPU:", run(2048, "cpu"))
print("GPU:", run(8192, "cuda"))
```

Wichtig

GPU rechnet "nebenbei". Für faire Messung braucht man `torch.cuda.synchronize()`.

Lehrbeispiel: typischer Fehler (zu viel Kopieren)

Problem

Wenn man in einer Schleife immer wieder Daten auf die GPU kopiert, kann das langsamer sein als CPU-Rechnen.

Merksatz

Einmal kopieren, dann viel rechnen.

PyTorch: "schlecht" vs. "besser" (passt auf Folie)

```
# schlecht: in jeder Runde erneut auf die GPU kopieren
for x in xs:                                # xs liegt auf CPU
    y = (x.to("cuda") @ x.to("cuda"))

# besser: einmal kopieren, dann rechnen
xs_gpu = [x.to("cuda") for x in xs]
for x in xs_gpu:
    y = (x @ x)
```

Was man daraus lernt

Nicht der Rechenkern ist das Problem, sondern oft das "Hin- und Hertragen" der Daten.

Optional: Warum "Tensor Cores" so große Zahlen liefern

Einfach erklärt

Es gibt in vielen GPUs **Spezial-Einheiten für Matrixrechnung**. Die sind extrem schnell, aber sie arbeiten oft mit **reduzierter Genauigkeit** (z. B. FP16/BF16/TF32 statt "echtem" FP32).

Wichtig für TFLOPS-Angaben

- **FP32 TFLOPS** = klassische Rechenwerke
- **Tensor TFLOPS** = Spezial-Einheiten für Matrizen (oft viel größer)

Lehrbeispiel (PyTorch): Blur als Faltung

Idee

Ein Blur kann als **Faltung (Convolution)** verstanden werden: Jeder Pixel wird durch einen gewichteten Mittelwert seiner Nachbarn ersetzt.

Warum ist das GPU-freundlich?

- Jeder Output-Pixel wird nach dem **gleichen Rezept** berechnet
- Sehr viele Pixel \Rightarrow viel Parallelität
- Passt gut zur Bildverarbeitung und zu vielen Rendering-Posteffekten

PyTorch: Blur mit conv2d (kurz, GPU-tauglich)

```
import torch
import torch.nn.functional as F

dev = "cuda" # oder "cpu"
img = torch.rand(1, 1, 1024, 1024, device=dev) # N,C,H,W

k = torch.ones(1, 1, 9, 9, device=dev) / (9*9) # Box-Blur Kernel
out = F.conv2d(img, k, padding=4) # gleiche Größe

# out ist das geblurrtte Bild (weiter auf GPU nutzbar)
```

Didaktischer Punkt

Das gleiche Rechenmuster wird für jeden Pixel parallel ausgeführt. Wenn `img` und `k` auf der GPU liegen, läuft auch die Faltung auf der GPU.

CPU-RAM → GPU-VRAM: wie lange dauert das grob?

Grobe Größenordnung

Die Kopie läuft typischerweise über **PCI Express (PCIe)**:

- **Große, zusammenhängende Datenblöcke**: typischerweise Millisekunden pro 100 MB
- **Sehr große Transfers (GB-Bereich)**: typischerweise Zehner-Millisekunden pro GB
- **Viele kleine Transfers**: oft **deutlich langsamer**, weil pro Transfer ein fester Overhead anfällt

Warum kleine Kopien besonders schlecht sind

PCIe ist paketbasiert und jede Kopie hat Overhead. Deshalb:
lieber wenige große Kopien statt vieler kleiner.

Quelle: NVIDIA Blog (Transfers bündeln, pinned memory)

Quelle: NVIDIA Forum (realistische PCIe4 x16 Durchsatzwerte, Overhead bei kleinen Transfers)

Warum das ein Flaschenhals ist

Das Nadelöhr: PCIe vs. GPU-internes Speichertempo

- PCIe 4.0 x16 liegt grob bei **einigen Dutzend GB/s**, PCIe 5.0 x16 bei **bis zu 64 GB/s** (theoretisch).
- Der Speicher **auf der GPU** (VRAM/HBM) ist dagegen in einer ganz anderen Liga: Datacenter-GPUs erreichen **bis über 2 TB/s** Speicherbandbreite.

Praktische Konsequenz (Merksatz)

Wenn Daten jeden Frame / jeden Iterationsschritt über PCIe nachgeladen werden müssen, dann bestimmt oft der Transfer die Gesamtgeschwindigkeit. Deshalb: Daten einmal auf die GPU laden und dort möglichst viel damit machen.

Quelle: NVIDIA A100 Datasheet (GPU Memory Bandwidth > 2 TB/s)

Quelle: ComputerBase (PCIe 5.0 x16 ~ 64 GB/s, dual-simplex erklärt)

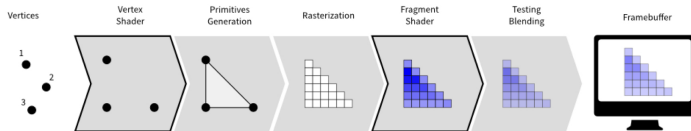
Schaubild: warum „pinned“ und „Batching“ helfen

Anschauliches Diagramm (NVIDIA)

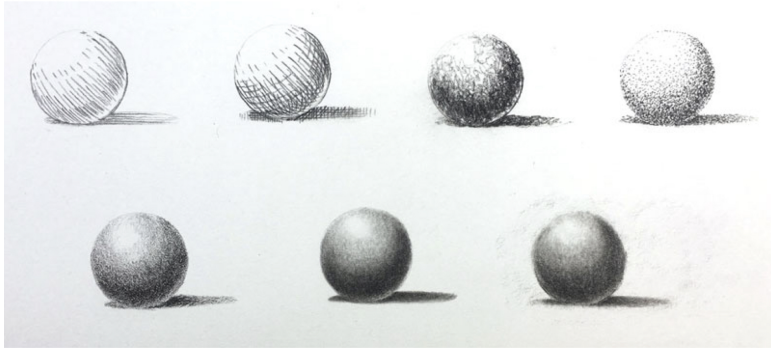
Das Schaubild zeigt, dass **page-locked/pinned memory** und **größere Transfers** die effektive Bandbreite erhöhen.

NVIDIA Blog: [How to Optimize Data Transfers in CUDA C/C++](#)

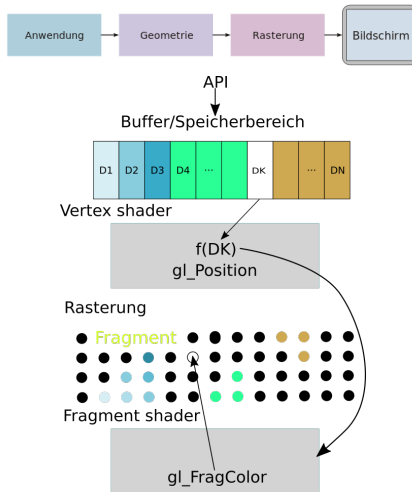
Vertex und Fragmentshader



Shader=Schattierer



Shaderprogramm



```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
    attribute vec2 a_position;
    uniform float t;
    varying float T;
    void main() {
        // gl_Position = vec4(a_position, 0.0, t);
        T = t;
        gl_Position = vec4(a_position[0], a_position[1], 0.0, 1.0);
    }
</script>

<script id="2d-fragment-shader" type="x-shader/x-fragment">
    precision mediump float;
    varying float T;
    void main() {
        gl_FragColor = vec4(0.0 ,1.0,0.0,1.0);
    }
</script>
```