Architektur und Programmierung von Grafik- und Koprozessoren

Die Grafik Pipeline

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

API Erweiterungen

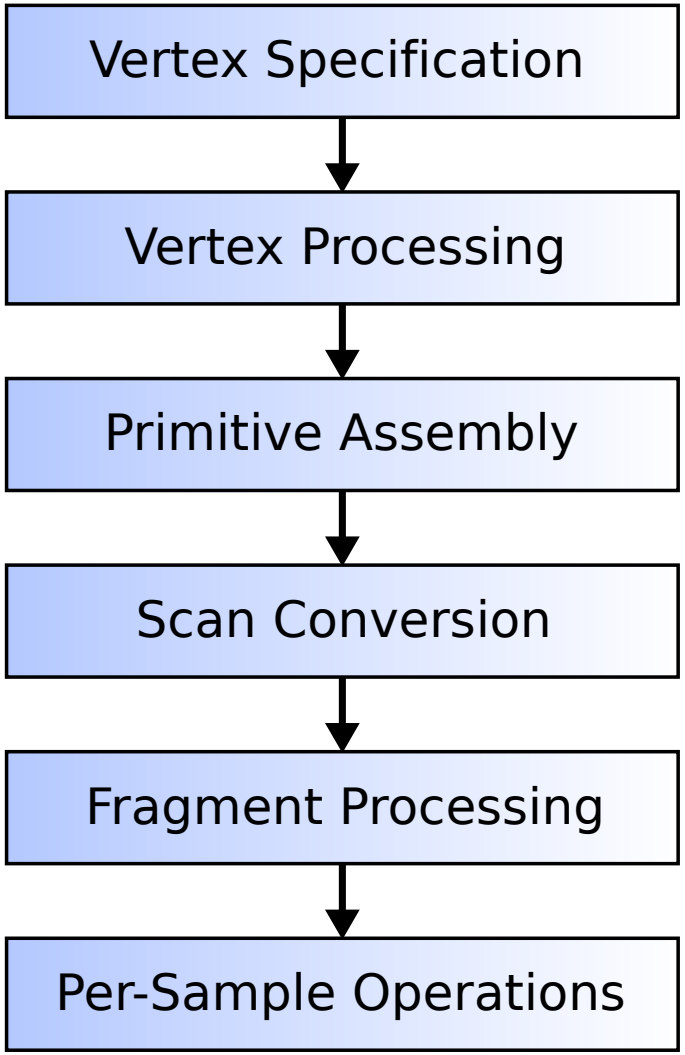
- Unterschiedliche Dynamik:
- Direct3D: Microsoft entscheidet im wesentlichen, schnelle Deprecation Mechanismen, schnelle Assimilation neuer Features.
- OpenGL (Khronos Group API): API unterstützt Extensions. Lade Extension per Function Pointer aus Treiber (wenn nicht vorhanden, ist FP nach laden `NULL`).
- OpenGL Extension Review Mechanismus:
  - erst Vendor: `NV_shader_thread_group`, `AMD_multi_draw_indirect` etc.
  - dann *Extension* (z. B. wegen Konsens mehrerer Vendors: `EXT_sparse_texture2`.
  - im Verlauf entscheidet ggf. *Architecture Review Board* (ARB), ob Extension sinnvoll: `ARB_bindless_texture`.
  - Zum Schluss wird Extension ggf. Teil des *Core Profils*.
- Deprecation bei OpenGL ähnlich langwierig.

Historische Entwicklung von Grafikprozessoren

Grafikbeschleuniger

- **1993** - SGI Grafik Workstations.
- **1996–1998** - Grafikprozessoren:
  - 3dfx Voodoo & Voodoo2.
  - Nvidia Riva TNT & TNT2.
- **1999** - Nvidia GeForce 256 ("GPU").
- **2006** - Nvidia GeForce 8800 GTX
  - "Unified Shader Architecture".
  - **2007** - CUDA, G80 Chip erste CUDA-fähige GPU.
- **2007** - Nvidia Tesla (vermarktet als "GPU ohne Grafikausgang, basiert auf G80 Chipsatz, zielte auf High Performance Computing Markt ab).
- **2009** - Nvidia Fermi Architektur: double-precision Berechnungen, single-precision fused multiply-add.

Grafik Pipeline



Grafikprozessoren um 1996

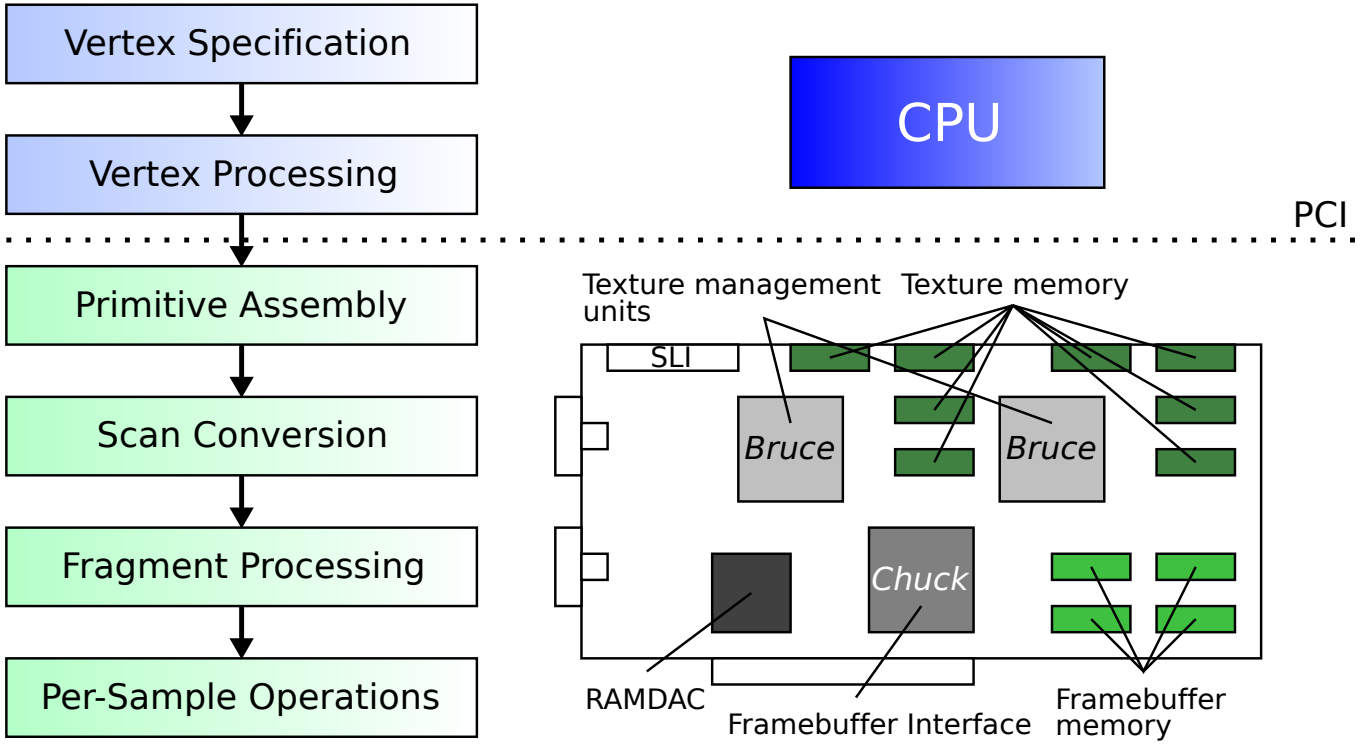


Abbildung: vgl. VOODOO^2 Graphics High Performance Graphics Engine For 3D Game Acceleration (Rev. 1.16, Dec 1, 1999), 3Dfx Interactive

“GPU” um 1999

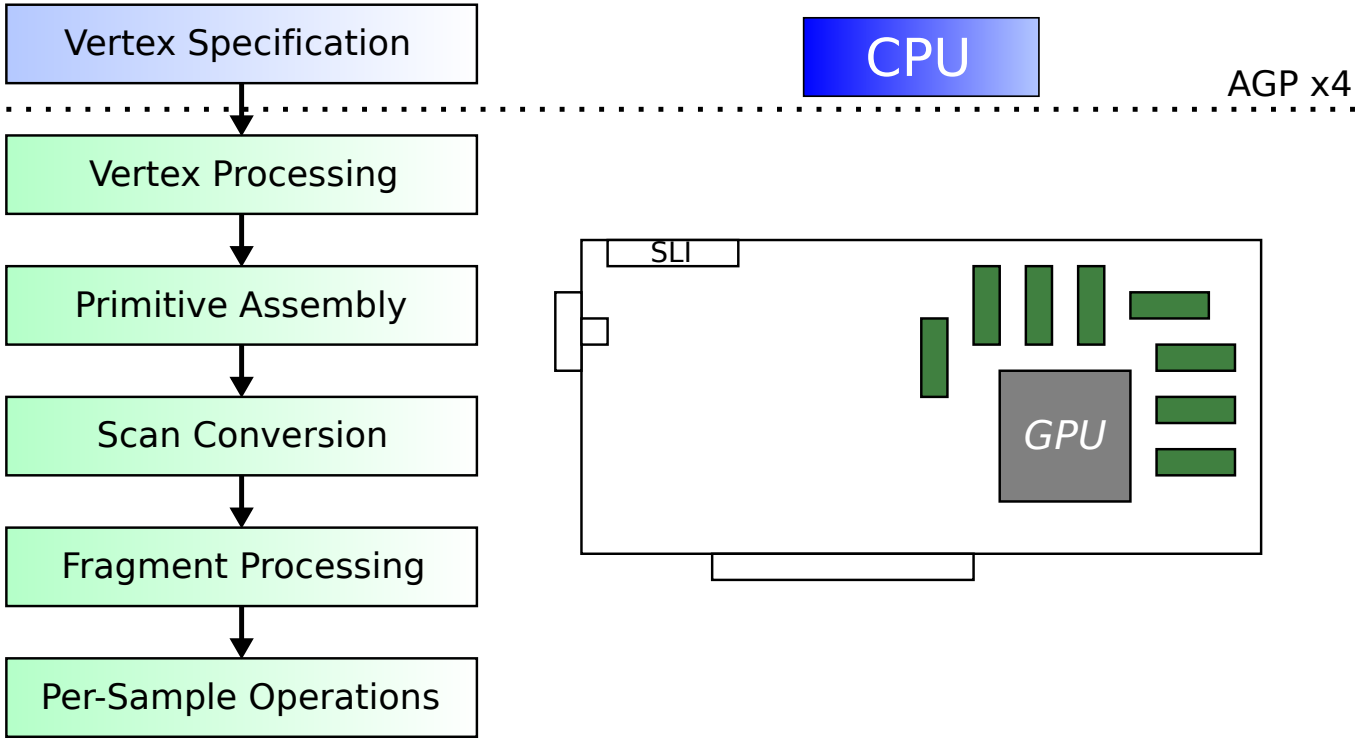


Abbildung: z. B. Nvidia GeForce 256, vermarktet als *Graphics Processing Unit* (GPU). Erster Koprozessor, der Vertex Verarbeitung und Pixel Operationen abbildete. OpenGL 1.2, DirectX 7.

GPUs um 2003

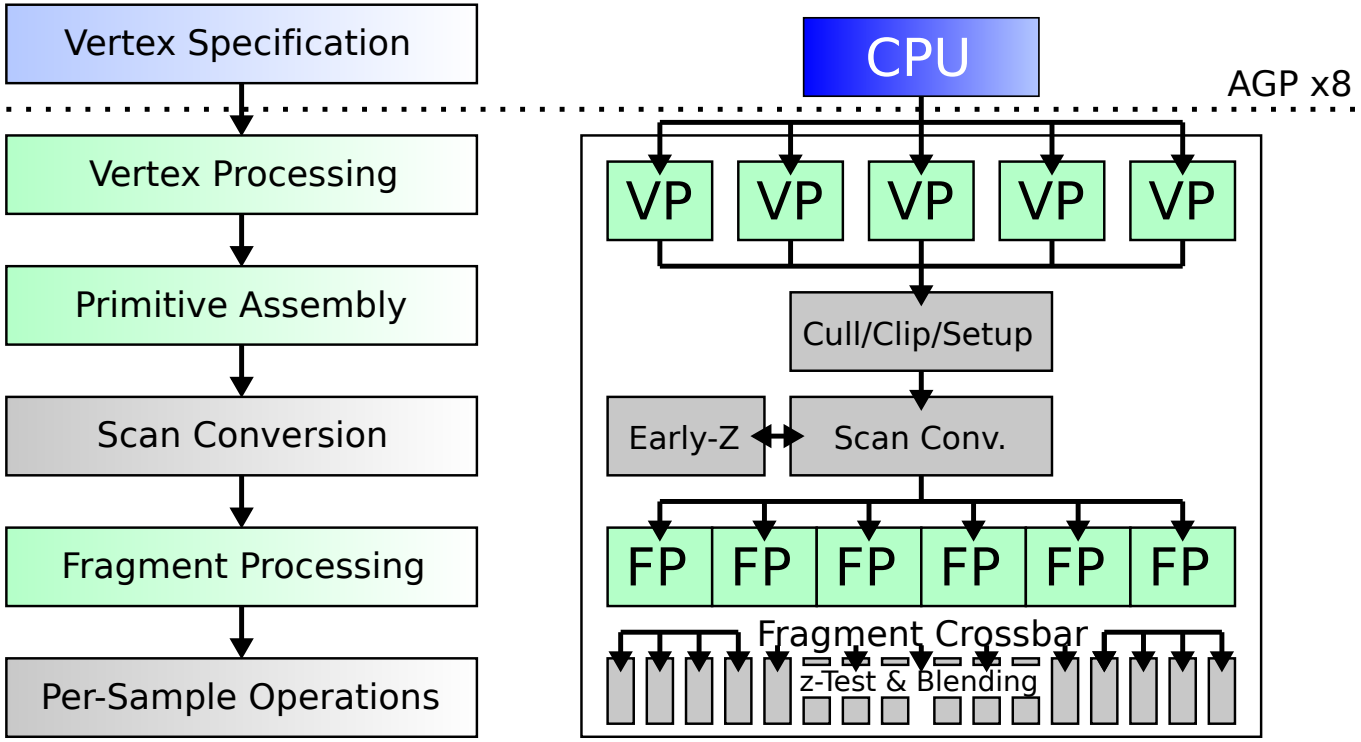


Abbildung: vgl. GPU Gems 2: The GeForce 6 Series GPU Architecture. Erste Generation von GPUs mit frei programmierbaren Vertex und Fragment Stages, dedizierten Vertex- und Fragmentprozessoren mit Branching Units, breiter 256-bit Memory Bus etc.

GPUs um 2003

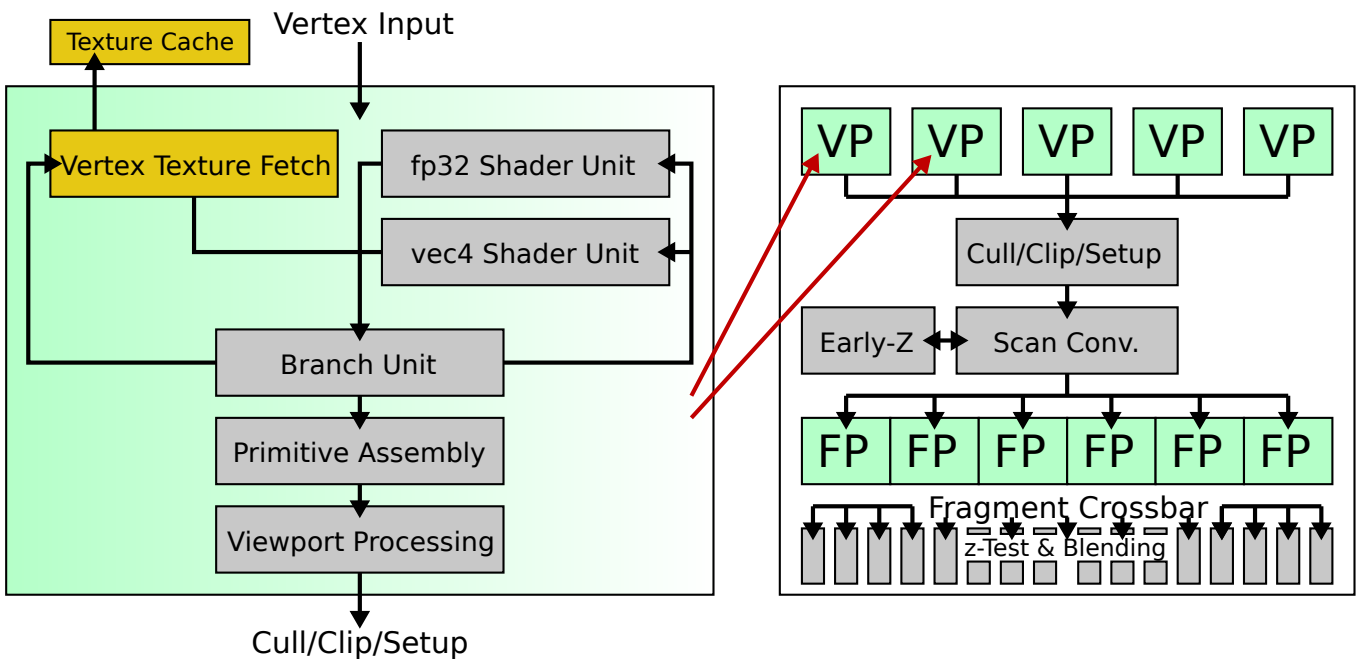


Abbildung: vgl. GPU Gems 2: The GeForce 6 Series GPU Architecture.

GPUs um 2003

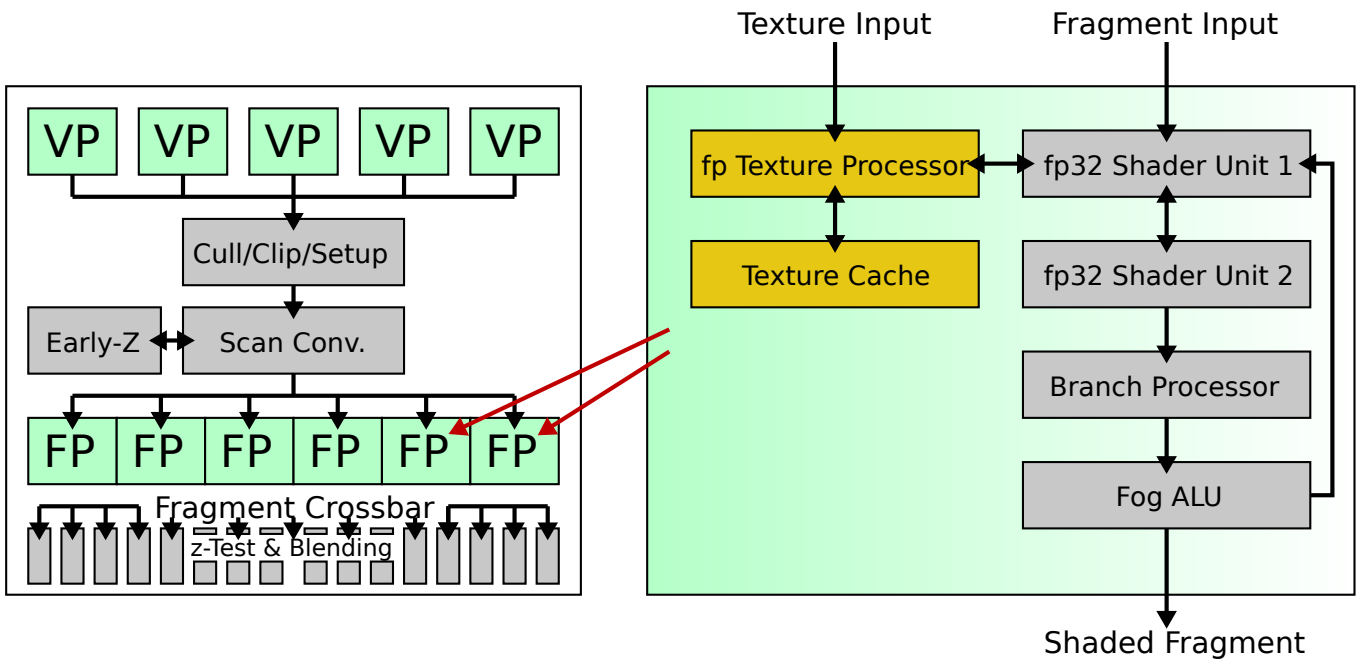
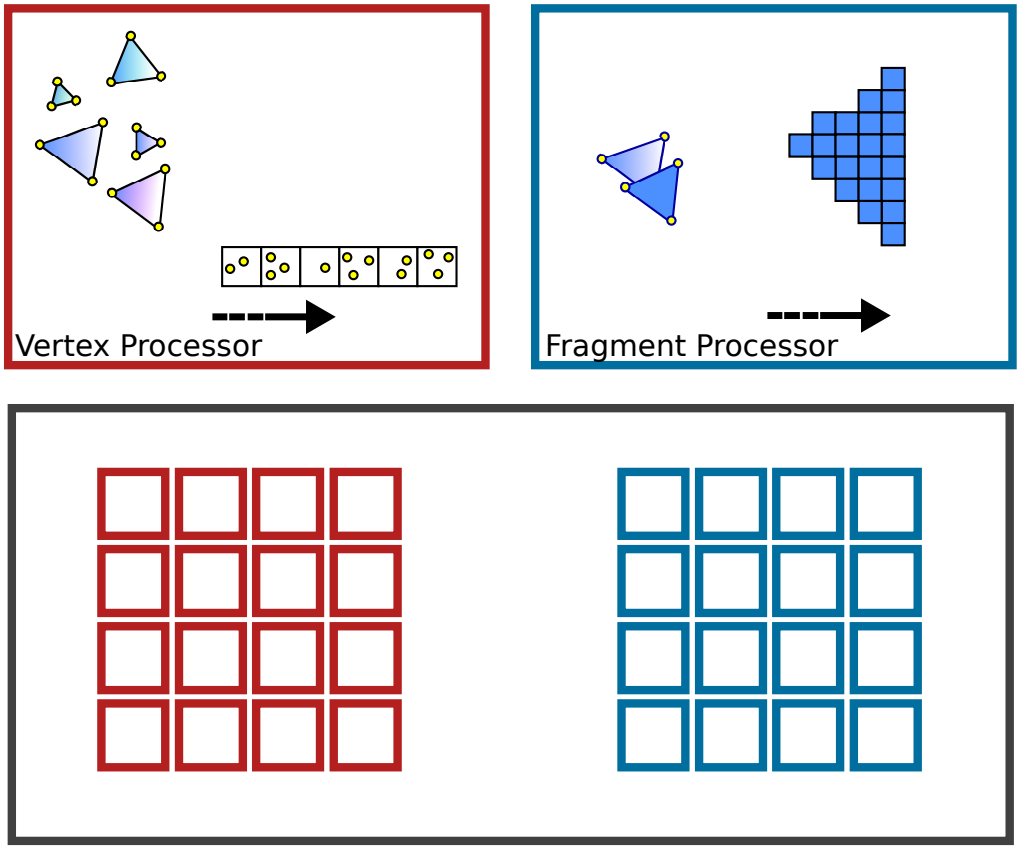


Abbildung: vgl. GPU Gems 2: The GeForce 6 Series GPU Architecture. FP verarbeitet vier Pixel auf ein Mal (für Texture Derivatives). SIMD Fragment Verarbeitung hunderter Fragmente, versteckt Latenz für Texture Fetch.

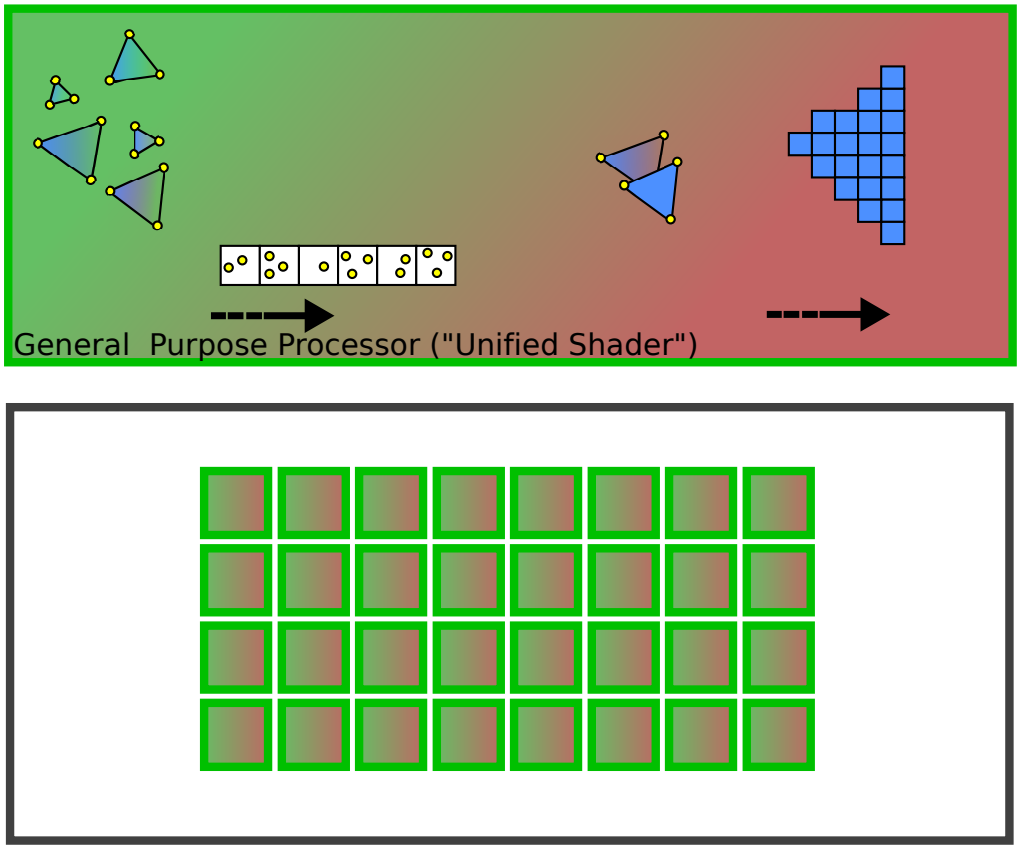
Traditionelle GPU Pipeline

Traditionelle Architekturen hatten separate Vertex und Fragment Prozessorkerne. Dies führte zu Lastimbalanzen, falls die Anzahl an Dreiecken und Fragmenten nicht gleich verteilt war.



Moderne GPU Pipeline

Moderne GPUs verfügen über sog. Unified Shader Kerne, die genereller als die alten Vertex und Fragment Kerne sind und beide Aufgaben übernehmen können.



Moderne GPU Pipeline

- Ausgehend davon (State Machine + Unified Shader Architektur) wollen wir uns später anschauen, wie moderne Architekturen den Weg vom Dreieck zum Bildschirmpixel hardwareseitig implementieren.
- Dabei betrachten wir den Weg vom CPU Programm (z. B. C/C++ API) über das Betriebssystem, den Grafikbus und die GPU bis zur Anzeige.

Unified Shader GPUs um 2007

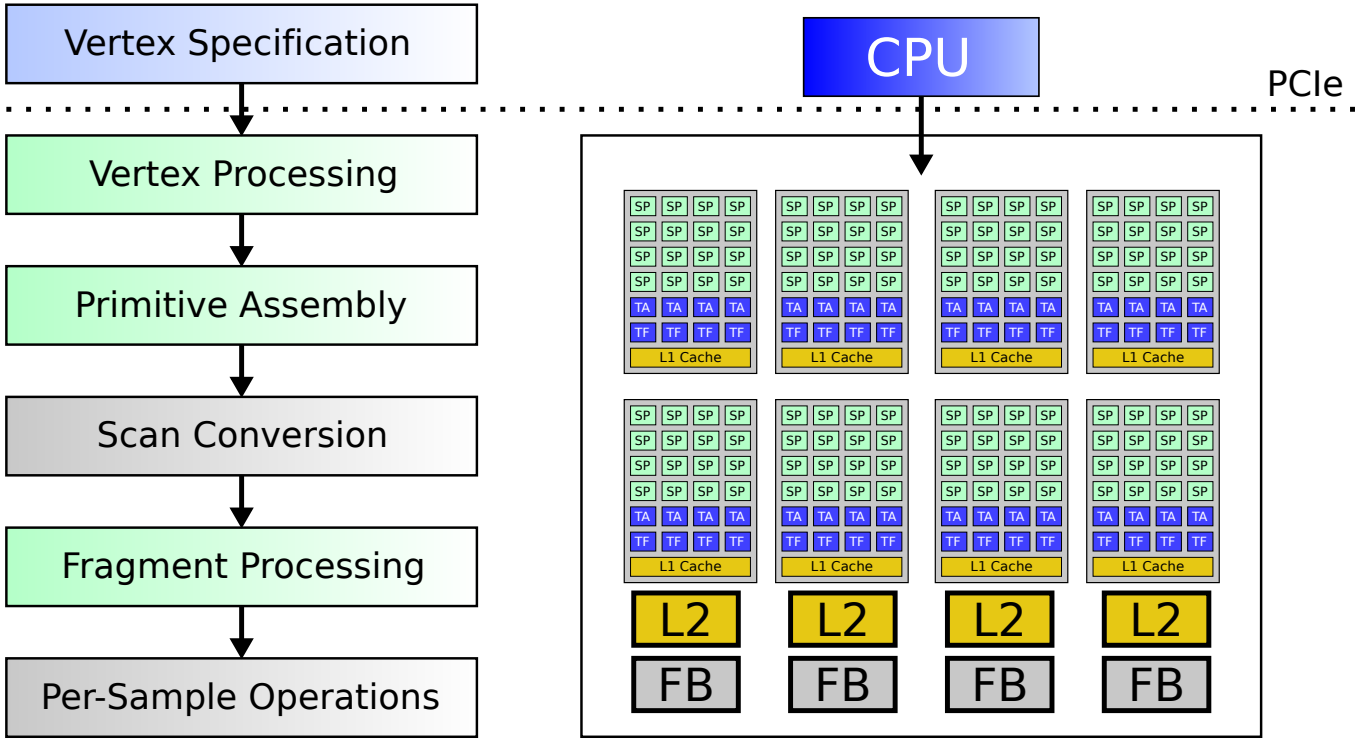
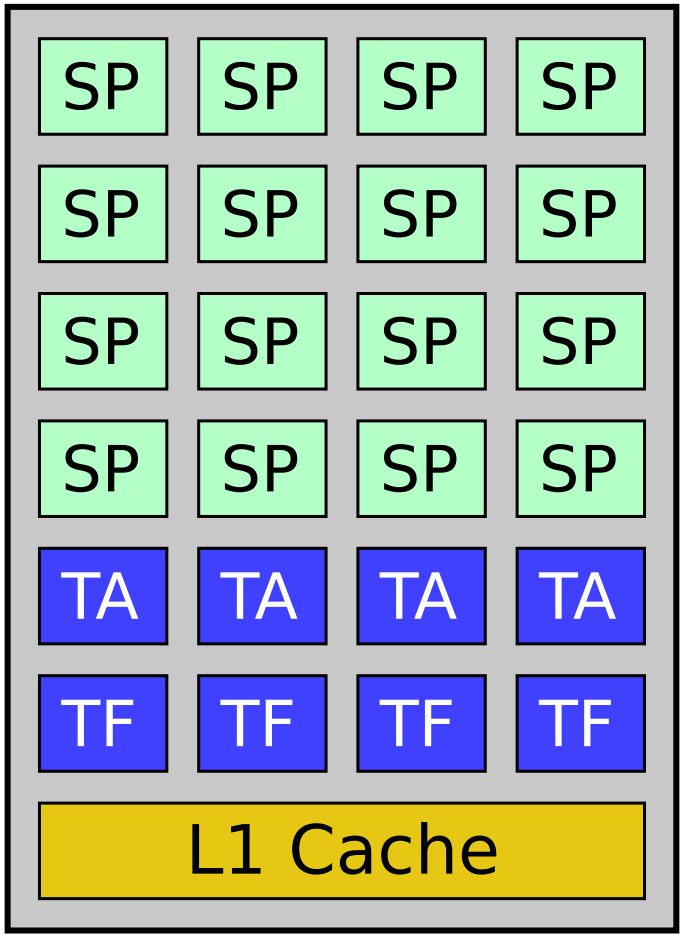


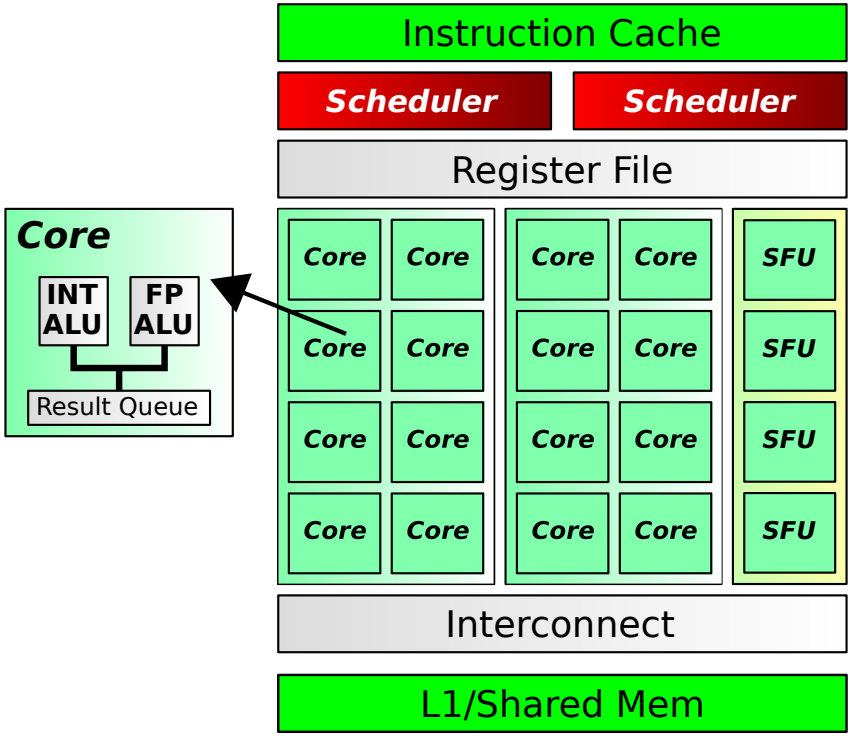
Abbildung: vgl. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview, November 2006 TB-02787-001_v0.9. Streaming Architektur, Shader Cores übernehmen diverse Aufgaben, u. a. auch Geometry Shader (neu).

Unified Shader GPUs um 2007



- Unified Shader Core:
  - *Streaming Prozessoren* (SP).
  - *Texture Address Units* (TA).
  - *Texture Filtering Units* (TF).
- Völliges Überlappen von Texture Fetch und Math Instructions ⇒ Extremes Verstecken von Speicherzugriffslatenz.

2009: Double Precision Berechnungen



Nvidia's Fermi Architektur: "Special Function Units" für Trigonometrie etc.; Hardware double-precision Support; Support für 32-bit IEEE-754 compliant fused multiply-add (`a * b + c` in einer Instruktion).[1]

[1] vgl. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi

Skalierbarkeit von GPU Architekturen

CPUs vs. GPUs

CPU

- Hohe Taktfrequenz
- Moderat viele Cores
- Moderat tiefe Pipelines
- Große Caches, Branch Prediction Units, Out-of-Order Execution

GPU

- Niedrige Taktfrequenz
- Viele Cores
- Sehr tiefe Pipelines
- Für hohen Durchsatz optimiert, Speicher- und sonstige Interfaces mit hoher Bandbreite und hoher Latenz. “Latency Hiding” über Durchsatz

CPUs vs. GPUs

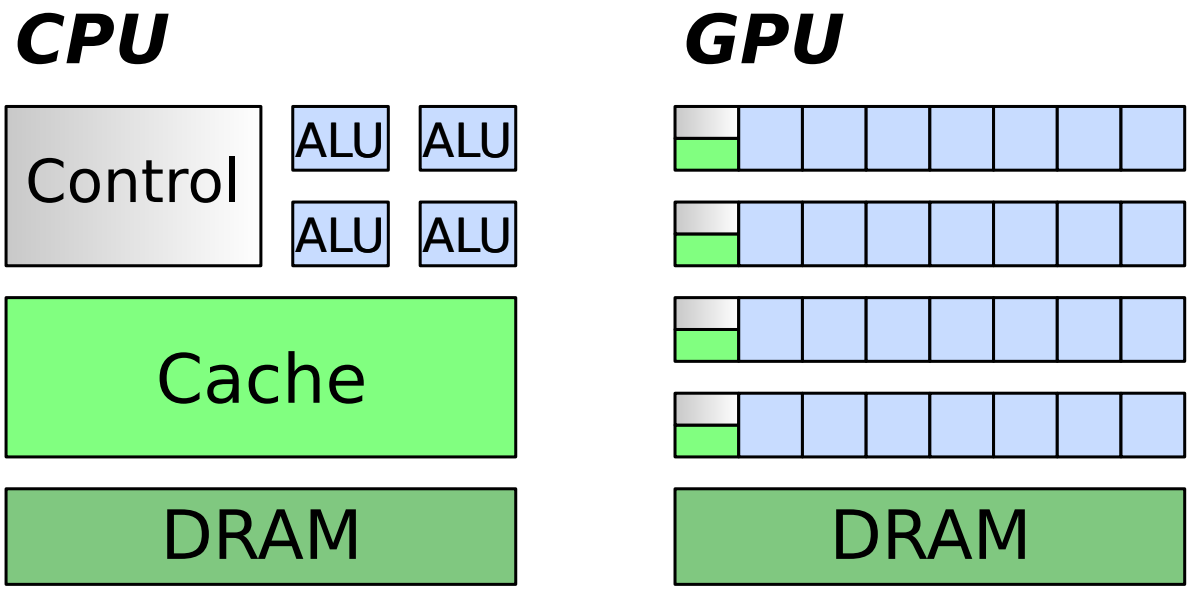


Abbildung: vgl. Modern GPU Architecture, Roger Crawfis, Game Design and Project.

Anforderungen an Skalierbarkeit

- GPUs sind hochparallele Architekturen. Dies äußert sich jedoch nicht bloß darin, dass GPUs extrem viele Kerne auf einem Die unterbringen.
- Anforderungen an Skalierbarkeit entlang der gesamten Pipeline. Skaliert eine Pipeline Stage nicht mit zunehmender Eingabedatenmenge, beeinflusst das die gesamte Pipeline.
- GPUs haben programmierbare Funktionseinheiten (Shader Prozessoren / Shader Cores) und nicht-programmierbare Funktionseinheiten (Textureinheiten, Raster Engines, Render Output Units (ROPs)).
- Außerdem "Host Interface": Treiber, Command Prozessoren, etc.
- GPU Architektur muss Skalierbarkeit bzgl. all dieser Einheiten gewährleisten.

Anforderungen an Skalierbarkeit

Insb. muss Skalierbarkeit bzgl. der folgenden Schlüsselmetriken [2] gewährleistet sein:

- Eingabedatenrate: Rate, mit der Kommandos und Geometrie an die GPU geschickt werden können.
- Dreiecksrate: Rate, mit der Geometrie transformiert, geclipped und als Dreiecke an Raster Engines geschickt wird.
- Pixelrate: Rate, mit der Raster Engines Dreiecke zu Fragmenten rastern, diese texturiert und zusammensetzen (Compositing).
- Texturspeicher: Größe der maximal nutzbaren Texturen.
- Display Bandbreite: Bandbreite, mit der Pixel aus Grafikspeicher an Displays verteilt werden können.

[2] vgl. Pomegranate: A Fully Scalable Graphics Architecture, Eldridge et al. Siggraph 2000

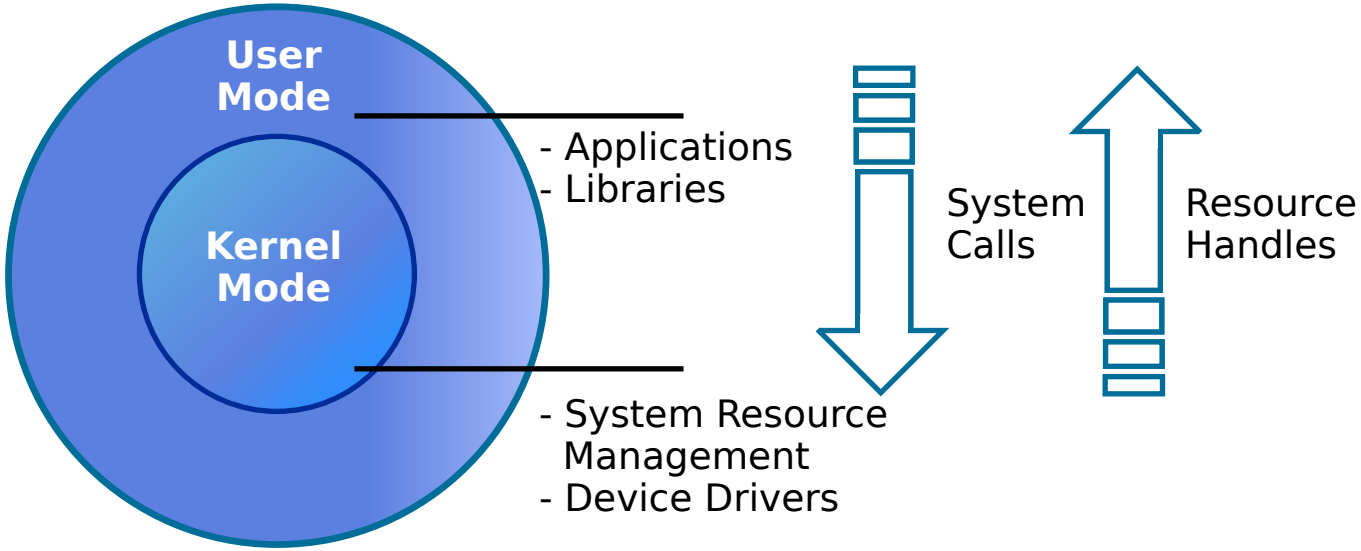
Host CPU und Grafiktreiber

Host Interface

- Applikation submittiert Vertices, Zeichenbefehle, State etc.
- Geht in Form von *Kommandos* durch das "Host-Interface":
  - Runtime und Treiber auf Seite der CPU.
  - Command Processor auf Seite der GPU.

Betriebssysteme - Protected Mode

Operationsmodi von CPUs und Betriebssystemen



Betriebssysteme - Protected Mode

- CPUs regulieren den Zugriff auf Systemressourcen (Speicher, Netzwerk, Geräte, etc.) mittels einer Ringstruktur (bei x86 Systemen: Ring 0 und Ring 3).
- Software, die im inneren Ring (Ring 0) ausgeführt wird, ist bzgl. Ressourcenzugriff hochpriviligiert ("Kernel Mode"), Software im äußeren Ring ist niedrigprivilegiert.
- API für Ressourcenzugriff: *System calls*. Applikation fragt beim Kernel mittels system call Ressource an und bekommt, so die Ressource verfügbar ist, ein *Ressourcen Handle*. Über das Ressourcen Handle wird die Ressource verwendet und freigegeben.
- Alle Operationen, die Kommunikation mit dem Kernel erfordern, sind vergleichsweise zeitintensiv.

Betriebssysteme - Protected Mode

Beispiel Dateisystemzugriff

C-Library Aufruf (vgl. `man 3 fopen`, `man 3 fclose`).

```
/* fopen.c */
#include <stdio.h>
int main() {
    FILE* fp = fopen("/home/zellmans/fopen.c", "r+w");
    fclose(fp);
}
```

System Call (vgl. `man 2 open`, `man 2 close`), Output mit `strace`.

```
...
open("/home/zellmans/fopen.c", O_RDWR) = 3
close(3)                               = 0
exit_group(0)                          = ?
++ exited with 0 +++
```

Betriebssysteme - Protected Mode

- Monolithisches Kernel Modell: Betriebssystem läuft im Protected Mode. Funktionalität wird duch *Kernel Module* ergänzt, die zur Laufzeit geladen oder ausgestoßen werden können.
  - Viele moderne Betriebssyteme wie Linux oder BSD basieren auf dem monolithischen Kernel Modell.
- *Gerätetreiber* werden als Kernel Module realisiert, die auf Ressourcen wie externe Hardware zugreifen können. Treiber i. Allg. sind Programme, die sich auf User Mode und Kernel Mode verteilen dürfen (in diesem Fall müssen die jeweiligen Treiber Bestandteile als separate Betriebssystemprozesse ausgeführt werden).

Grafiktreiber - Kernel und User Mode

Betriebssysteme definieren Schnittstelle, die Grafikkartenhersteller (*independent hardware vendor* (IHV)) implementieren müssen, z. B. *Windows Display Driver Model* (WDDM).

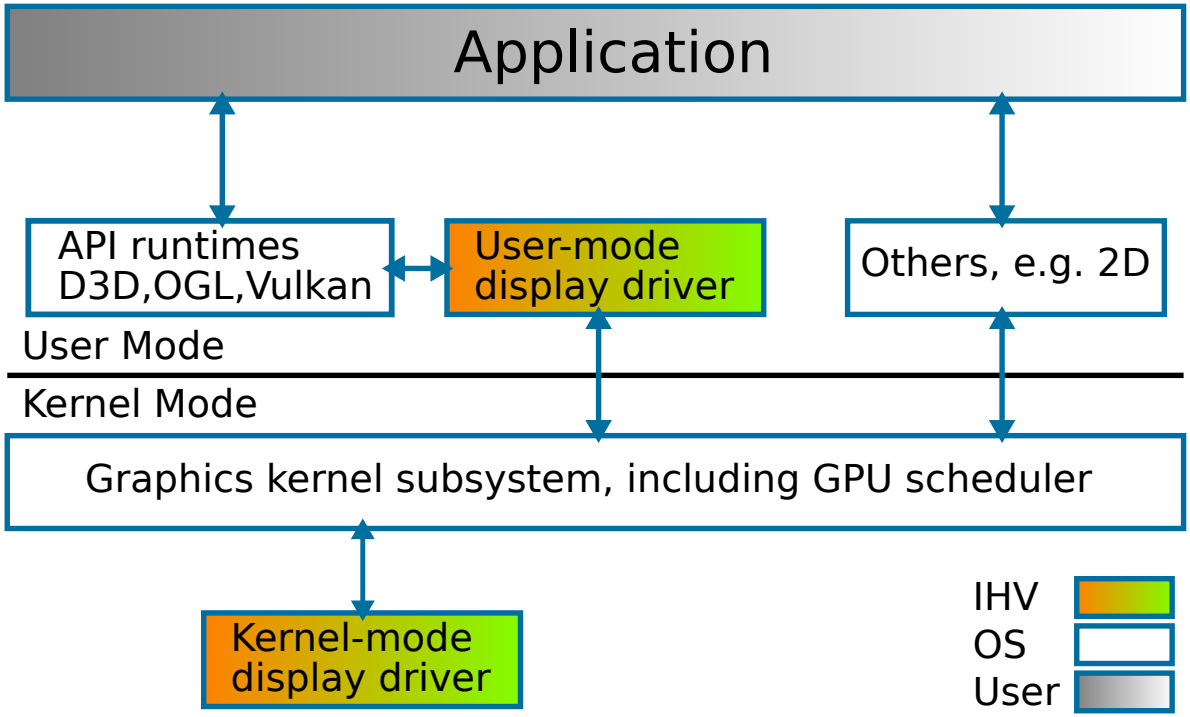


Abbildung: Darstellung (vereinfacht) gemäß Microsoft WDDM Design Guide: WDDM Architecture 14.03.2018.

Grafiktreiber - Kernel und User Mode

- Microsoft WDDM exemplarisch, andere OSs ähnlich.
- IHVs stellen Kernel- und User Mode Treiberkomponenten bereit, GPU scheduler kommt vom OS.
- User Mode Treiber übernimmt Aufgaben wie Patchen und Übersetzen von Shader Programmen in Intermediär Code (PTX, AMD IL) etc.
  - bei modernen APIs wie Vulkan oder D3D12 wird per Default keine Runtime Shader Compilation mehr durchgeführt.
- Prinzip: Verlagerung von möglichst vielen Aufgaben in User Mode verhindert, dass einzelne Programme das gesamte Grafik Subsystem zum Absturz oder Einfrieren bringen.

Grafiktreiber - Kernel und User Mode

- Virtualisierung: User Mode Prozesse können nicht auf den Grafikspeicher anderer Prozesse zugreifen. Virtualisierung in User Mode, nicht im Treiber.
- Prozess-Scheduling und Command Preemption durch User Mode Runtime.

GPU Applikation - Operationsfluss

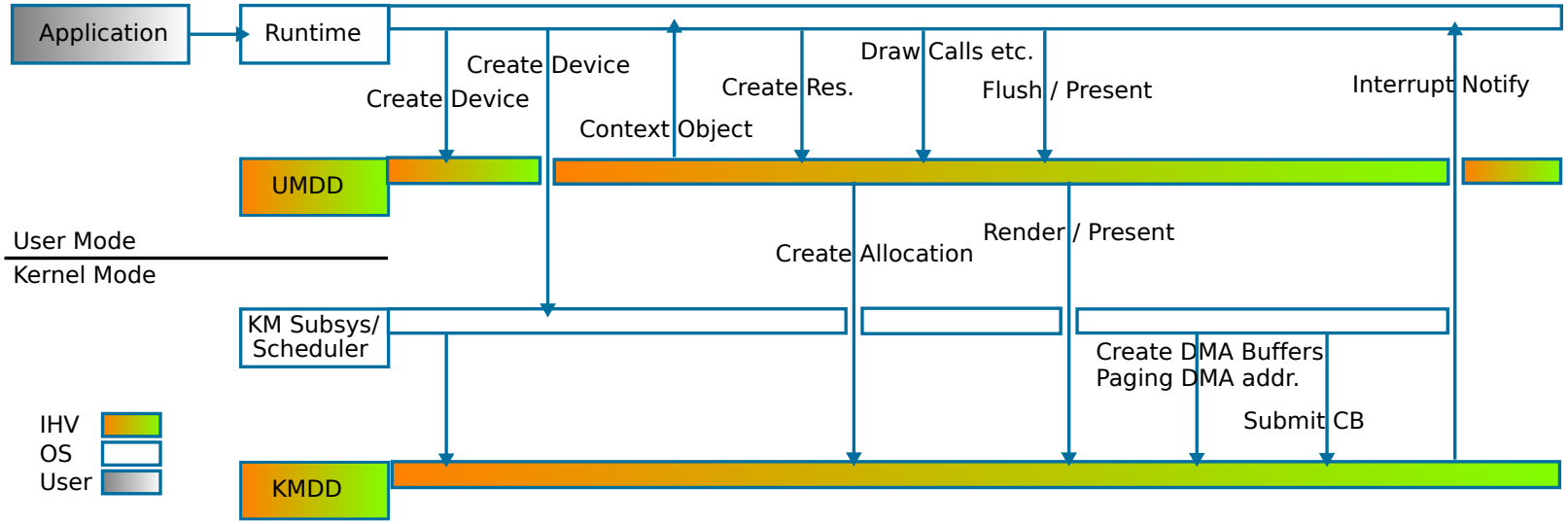


Abbildung: Darstellung (vereinfacht) gemäß Microsoft WDDM Design Guide: WDDM Operation Flow 14.03.2018.