Architektur und Programmierung von Grafik- und Koprozessoren

Die Grafik Pipeline

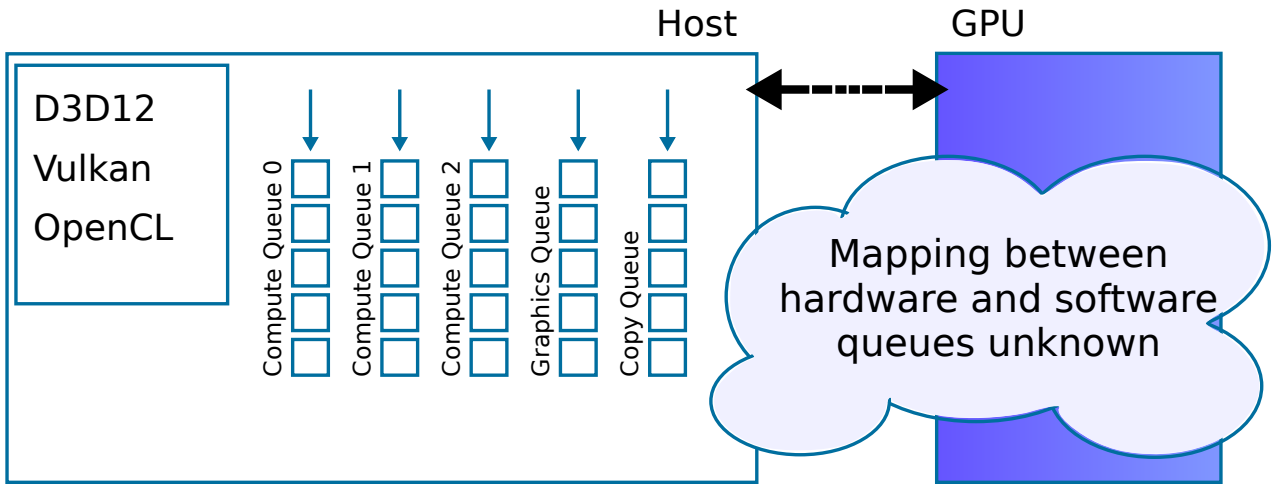
Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Command Processor

Queues / engines etc. die vom API exponiert werden, mappen nicht notwendigerweise auf tatsächliche Hardware Queues.



Command Processors - AMD GCN

Beispiel AMD Graphics Core Next (z. B. Radeon R9 390X): Acht *Asynchronous Compute Engines* können über Queue 0-7 angesteuert werden. 3D Command Processor: Queue 0. Zusätzlich noch dedizierte DMA Buffer für Datentransfer.

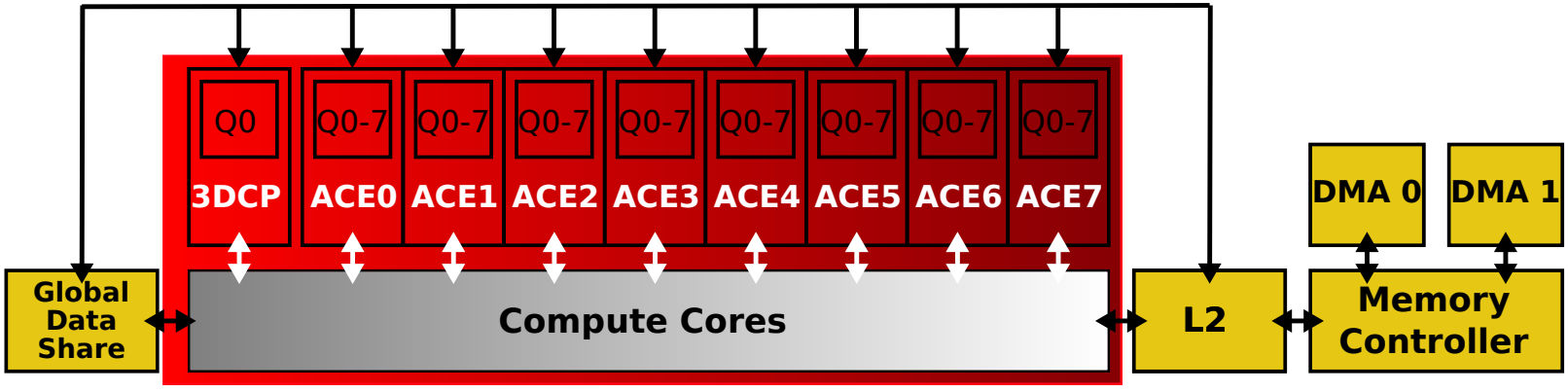


Abbildung: vgl. AMD Whitepaper: Asynchronous Shaders - Unlocking the full potential of the GPU.

API Order

- APIs haben eine strikte Regel: Zeichenkommandos müssen *dem Anschein nach* in der Reihenfolge ausgeführt werden, in der sie spezifiziert wurden ("API Order"). Damit einhergehend: die Reihenfolge, in der Dreiecke spezifiziert werden, determiniert, in welcher Reihenfolge sie gezeichnet werden.
- Große Hürde beim Design hochparalleler Architekturen.
- Design Entscheidungen bei Grafikkartenarchitekturen wesentlich durch API Order beeinflusst.

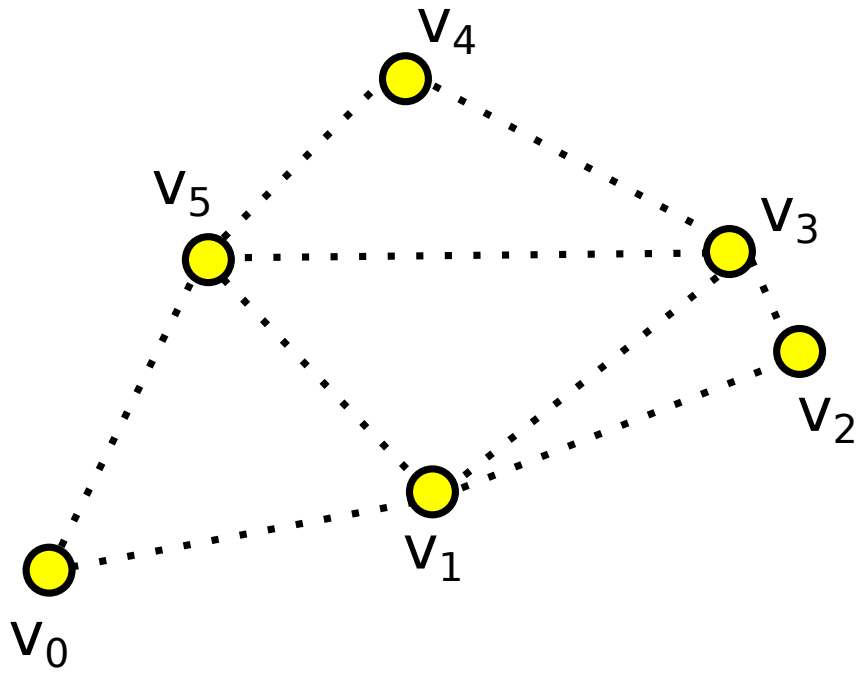
Vertex Phase

Vertex Phase

- Vertices werden transformiert. Kann einfache Matrix Transform sein, oder komplizierte Operation, die im Vertex Programm beschrieben wird.
- Vertex Attribute wie Normalen, Texturkoordinaten, Vertexfarben etc. werden ebenfalls transformiert.
- 1:1 Mapping: es verlassen so viele Vertices die Vertex Phase, wie hereinkommen (wir ignorieren Geometrie Shader, Tesselation Shader etc.).

Input Assembly

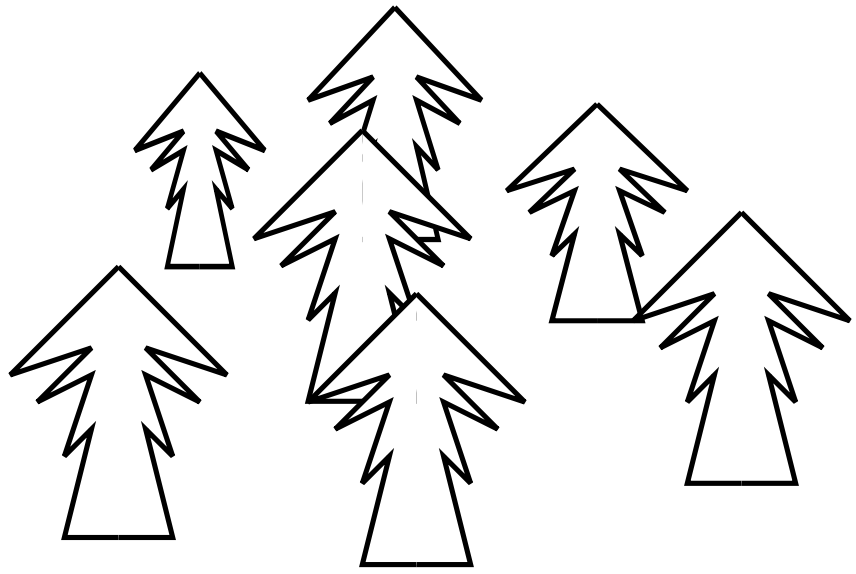
Indizierte Geometrie – Vertices werden üblicherweise von mehreren Dreiecken geteilt.




```
vertex_buffer_data(v0, v1, v2, v3, v4, v5, v6);
index_buffer_data({0,1,5}, {1,2,3}, {1,3,5}, {3,4,5});
```

Input Assembly

Instancing – instanziiere komplexe Geometrie, um Bandbreite zu sparen.



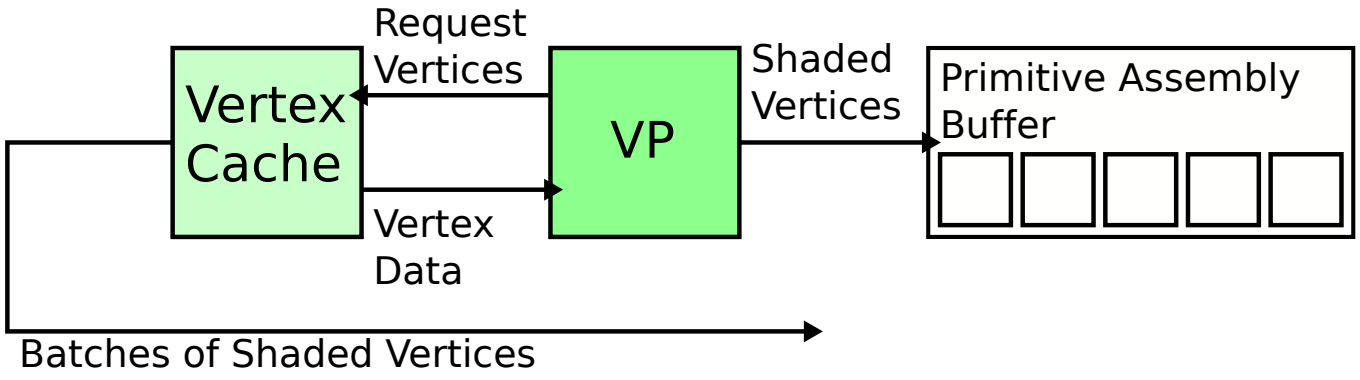
Input Assembly

- Input Assembly Stage: lese aus Index und Instance Buffers und reiche Vertices entsprechend an Vertexprozessoren weiter.
- Nicht so einfach: Vertex Strom nicht nur aus Speicher: da i. d. R. Vertices von 4-5 Dreiecken geteilt werden, werden transformierte Vertices in Cache gespeichert.
- Die sich anschließende Vertex Phase ist hochparallel: berücksichtigt keine Index Buffer, geteilte Vertices werden dupliziert. Index Buffer dienen im wesentlichen dazu, PCIe Bandbreite zu sparen.

Post Transform Cache

- Fixed-function: Cache mit fixer Anzahl an Vertices.
- Heute: Vertex kann mit variabler Anzahl Vertex Attributen ausgestattet sein $\Rightarrow$ je nach Anzahl Konfiguration passen mehr oder weniger Vertices in den Cache.
- Außerdem: Unified Shader sind für Durchsatz anstatt Latenz optimiert $\Rightarrow$ transformiere Batches von Vertices auf einmal (fixed-function: ein Vertex pro Vertex Shading Einheit). Vertex Cache entsprechend ausgelegt.

Vertex Phase



Post Transform Cache: nachdem Vertex Shader durchlaufen, werden geshadete und transformierte Vertices in den Cache geschrieben. Primitive Assembly Stage (nachfolgend) wird entweder aus Cache und durch noch zu transformierende Vertices bedient, die noch nicht im Cache stehen.

Vertex Caching komplex, da die Pipeline keine Konnektivität von Vertices speichert. Um zu bestimmen, ob zwei Vertices gleich, werden Vertex Attribute verglichen.

Vertex Phase

Vertex Phase (Beispiel Nvidia Fermi):[3]

- "Prozessor Cluster" (Gruppe von Cores) prozessieren Vertex Batches. 32 Threads ("Warp") pro Core.
- Threads verarbeiten Vertex Programm Instruktionen in lock-step. Wartende Threads (Branching) werden ausmaskiert (implizites Programmiermodell).
- Scheduler: Cores führen eine Warp auf einmal aus. Wartet eine Warp z. B. auf Speicheroperation, kann auf Core eine andere Warp geplant werden, die Arithmetik ausführt.
- Schnelles Umschalten zwischen Warps: jede Warp hat eigene Register im Register File $\Rightarrow$ Register knappe Ressource. Je mehr Register durch Shader alloziert, umso weniger Warps können gescheduled werden.

[3] vgl: Life of a triangle - NVIDIA's logical pipeline

Vertex Phase

Warp / Thread Group Scheduler

- z. B. Nvidia Kepler Architektur (GTX 680): 4 “Prozessor Cluster”, 192 Shader Cores pro Prozessor Cluster, 4 Warp Scheduler pro Prozessor Cluster.
- Instruktionen laufen bei Scheduler *in Gruppen* auf, Scheduler plant Warps möglichst sinnvoll.
- Warp führt eine Reihe (z. B. 2 oder 4) Instruktionen auf dem Shader Core aus, auf dem sie geplant wird.
- So kann Latenz versteckt werden: plane erst Warp, die aus Speicher liest; während diese wartet, plane Instruktionen von anderen Warps mit niedrigerer Latenz.
- Wegen Unified Shader Architekturen: Scheduling Logik für Vertex Phase und Fragment Phase gleich.

Vertex Phase

- Ausgabe fixed-function: Vertices in Normalized Device Coordinates.
  - Vertex Programme: Entwickler kann im Grunde selber entscheiden. Nach Vertex Phase jedoch Viewport Transformation (fixed-function)!
- Auf neuen Architekturen schließen sich noch weitere programmierbase Pipeline Stages an: Geometrie Stage, Hull & Domain Shader Programm. Diese generieren evtl. weitere Geometrie.
- Diese Pipeline Stages vernachlässigen wir bei unserer Betrachtung.

Primitive Assembly

Primitive Assembly

Bisher:

- Reine Verarbeitung auf Vertex Ebene, Konnektivität irrelevant.
- **Primitive Assembly**: führe Vertices mit Indizes aus *Index Buffer* zusammen.
  - GPU Primitive: Punkte, Linien, Dreiecke, Polygone.
  - Punkte: einfach.
  - Linien vernachlässigen wir, evtl. dedizierter Code Pfad.
  - Polygone auf Dreiecke abbildbar.
- **Culling / Clipping Phase**: jedes Dreieck wird am sichtbaren Frustum geclipped.
  - Einfache Dreiecke: vollständig innerhalb oder außerhalb Frustum.
  - Dreiecke, die Frustum *schneiden*, müssen geclipped werden.

Primitive Assembly

Idee 1

"Richtiger" Clipping Algorithmus, z. B. *Cohen-Sutherland*. Führe auf Dreiecken aus, generiert weitere Dreiecke.

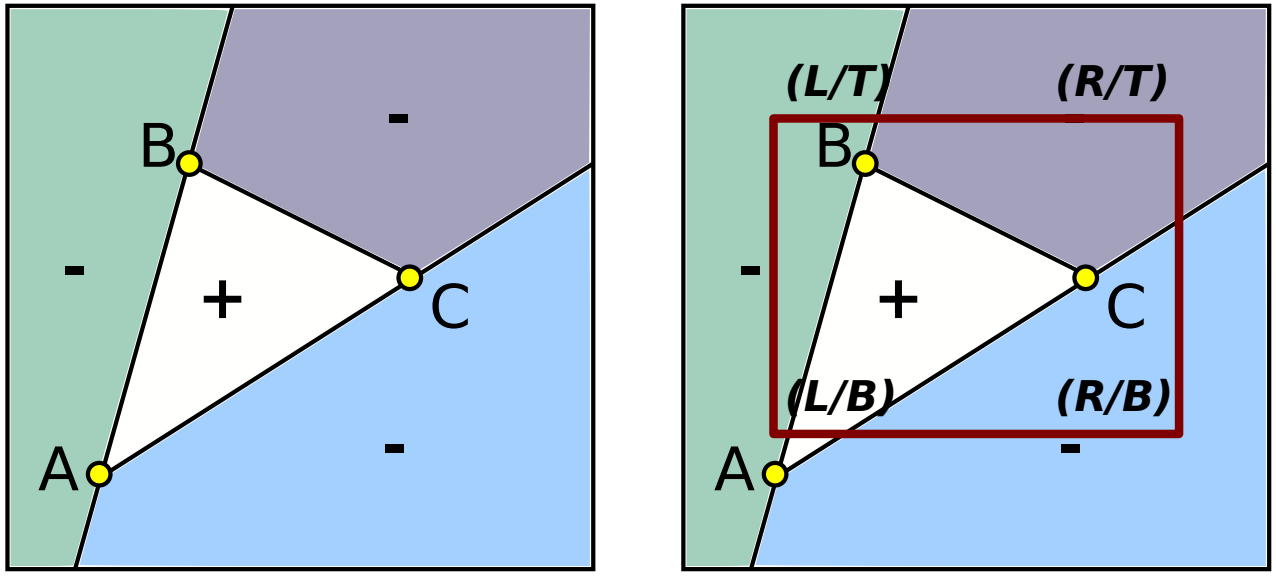
Idee 2

Wir erinnern uns an den Scan Konvertierungs Algorithmus aus Vorlesungsteil 2.

Anstatt weitere Dreiecke in die Pipeline einzufügen, identifizieren wir *konservativ* die Dreiecke, die echt außerhalb des Frustums liegen.

Anstatt zu clippen, führen wir Clipping Ebenen einfach als weitere *Kantengleichungen* für den Scan Konvertierungs Algorithmus ein
$\Rightarrow$ Clipping implizit durch Raster Engine, einfachere Hardware.

Implizites Clipping durch Raster Engines



- Anstatt zu clippen, übergebe weitere Kantengleichungen an Raster Engines.
  - Dreieckskanten: $A - C$, $B - A$, $C - B$.
  - Frustum Kanten: $(L/T) - (L/B)$, $(R/T) - (L/T)$, $(R/B) - (R/T)$.
- Fensterkoordinaten: valide z-Werte zwischen 0 und 1 werden interpoliert. Clipping mit z-Near / z-Far trivial.