Architektur und Programmierung von Grafik- und Koprozessoren

Wiederholung ausgesuchter Themen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Performanzbegriff

Gesucht: Performanz von *Computerprogramm*, das auf einem Prozessor ausgeführt wird.

Wall clock time / CPU execution time: absolute Zeit, die das Programm benötigt, um all seine eingabeabhängigen Befehle abzuarbeiten:

$$\textit{CPU time} = \textit{Anzahl Taktzyklen}(I, D) \times \textit{Zeit pro Taktzyklus}, \quad (1)$$

wobei

$$\textit{Anzahl Taktzyklen}(I, D) = \#I \times \textit{CPI}. \quad (2)$$

Performanzbegriff

Wichtig zu merken:
Unser Begriff von der Performanz hängt von *drei* Größen ab:

- Anzahl an Instruktionen im Programmlauf.
- Durchschnittliche Taktzyklen pro Instruktion.
- Zeit pro Taktzyklus.

Oft beurteilt man Performanz (z. B. eines Prozessors) nur aufgrund *einer* dieser Größen (z. B. Taktfrequenz). Dies ist eine isolierte Betrachtungsweise, die zu Fehlannahmen führen kann.

Leistungsaufnahme

- Leistungsaufnahme: Eingangsspannung V_{dd} sinkt mit schrumpfenden Transistoren (sogar quadratisch) ("Dennard Scaling").
- *Leakage Spannung* jedoch gleich, egal wie groß der Transistor.
  - Daher verbauen Prozessorhersteller Chip Elemente, die in inaktivem Zustand keinen Strom oder wenig verbrauchen, z. B. größere Caches, mehr Prozessorkerne.

Parallelismus daher auf allen möglichen Ebenen. Aber (**wichtig**): Parallelismus gibt es nicht erst seit SIMD & Multi-Core. Parallelismus erstreckt sich von ILP über Hardware Pipelining und SIMD bis zu Cluster Computing.

Pipelining

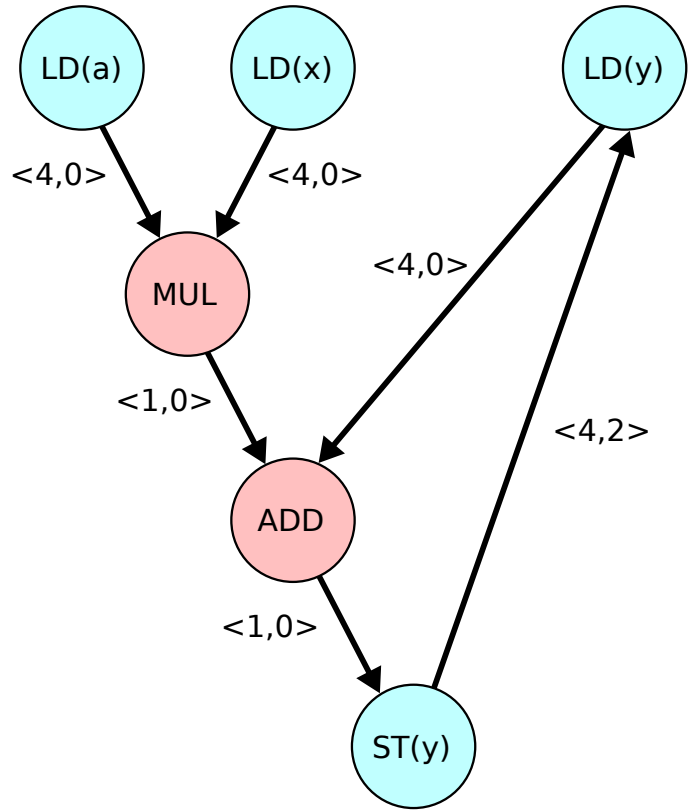
Software Pipelining (Compiler) mittels Modulo Scheduling.

- Schedule Schleifendurchläufe überlappend.
- Finde minimales Initiation Interval, das gültig
  - bzgl. der zur Verfügung stehenden Ressourcen (z. B. ALUs, MMUs, etc.)
  - bzgl. Datenabhängigkeiten zwischen Instruktionen.

Ressourcen $\Rightarrow$ **Reservierungstabellen**.
Datenabhängigkeiten $\Rightarrow$ **Datenabhängigkeitsgraphen**.

Modulo Scheduling

1. MII := max(ResMII, RecMII)
2. II := MII
3. Unrolle Schleife
4. Versuche, Schleife mit II zu planen. Falls gültig: fertig.
5. Sonst: II := II + MII, gehe zu 4.



	0	1	2	3	4	5	6	7
MMU		x	x				x	x
ALU	x			x	x	x		

SIMD und Multithreading

Werden explizit(er) programmiert. SIMD Instruktionsfluss in "lock-step", Multithreading bedarf Synchronisation.

Verschiedene Low-Level und High-Level Sprachkonstrukte, die bei der Programmierung verwendet werden können, z. B. `pragmas`, Libraries mit Parallelen Primitiven, Auto-Vektorisierung, direkte Programmierung von Threads mit Hilfe von Betriebssystem-APIs, etc.

Synchronisation kann Auswirkungen auf Cache Effizienz haben, z. B. Atomic Variablen mit unalignierten Adressen auf NUMA Systemen.

Parallele Algorithmen

Können abstrakt für PRAM oder für Work/Time Modell ("Arbeit vs. Zeit Paradigma") formuliert werden.

Schrittkomplexität: Anzahl der benötigten *Zeitschritte* des PRAM Algorithmus.
Arbeitskomplexität: Anzahl der insgesamt ausgeführten *Operationen*.

Parallele Algorithmen sind *kostenoptimal*, wenn die Zeit, die sie zur Abarbeitung benötigen, mal der Anzahl paralleler Ressourcen, der seriellen Komplexität des Algorithmus entspricht.

PRAM Speichermodell

- **EREW** - **E**xclusive **R**ead, **E**xclusive **W**rite: alle Speicherzugriffe erfolgen exklusiv.
- **CREW** - **C**oncurrent **R**ead, **E**xclusive **W**rite: lesender Zugriff erfolgt gleichzeitig, schreibender Zugriff exklusiv.
- **CRCW** - **C**oncurrent **R**ead, **C**oncurrent **W**rite: alle Prozessoren können gleichzeitig lesend und schreibend auf den Speicher zugreifen.

Wir präferieren Algorithmen, die von exklusiven Speicherzellen lesen und an exklusive Speicherzellen schreiben. (Achtung: PRAM Modell sieht lock-step Verarbeitung vor, manchmal müssen auch EREW Speicherzugriffe synchronisiert werden, vgl. “Paralleles Reduzieren.)

Rasterisierungsalgorithmus

Der Rasterisierungsalgorithmus ist ein **CRCW** Algorithmus: Primitive werden in unbestimmter Reihenfolge in Tiefenpuffer gerastert (Prozessoren sind nicht Rasterpositionen zugeordnet). (Grafik APIs: Reihenfolge durch API-Order vorgegeben.)

Arbeitskomplexität: $W(n) = O(V \times VP)$.
Zeitkomplexität: $S(n) = O(VP)$.

Scan-Konvertierung: **Pineda Algorithmus**.

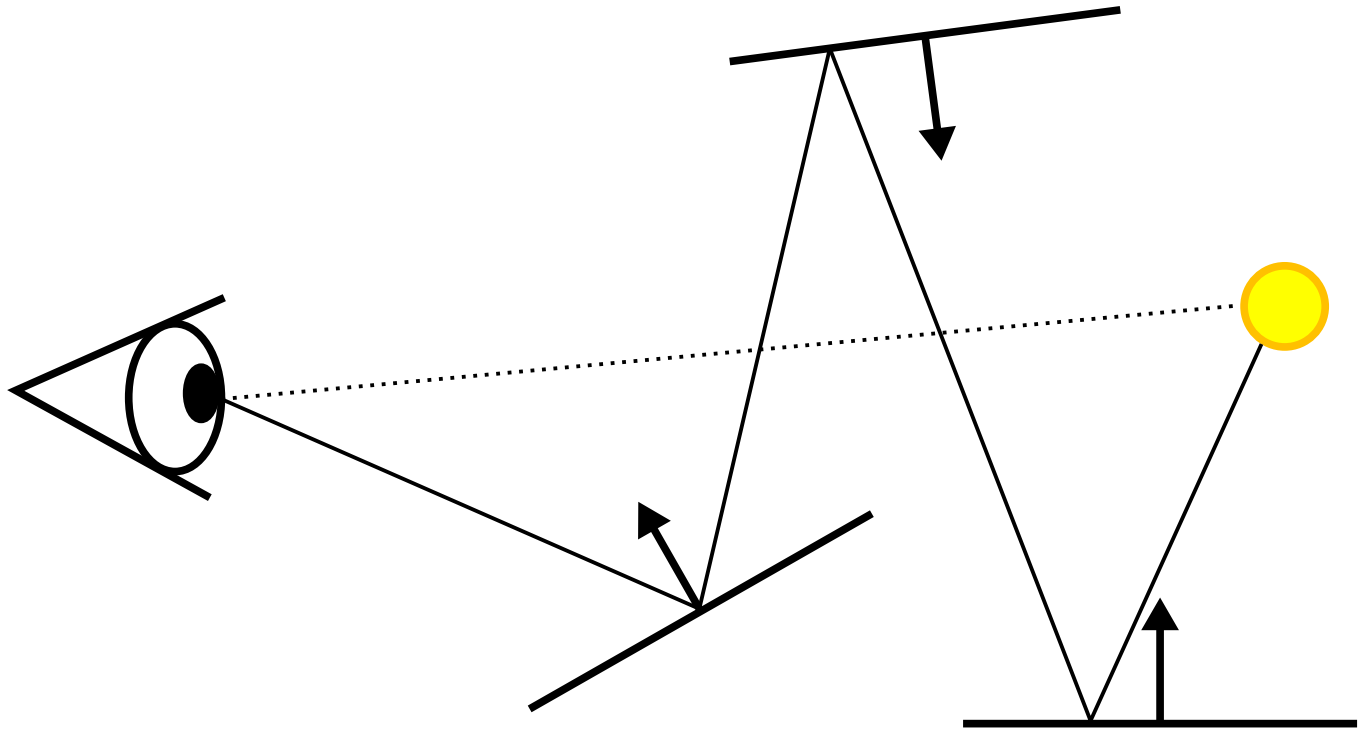
Wir haben mehrere Formulierungen des Algorithmus kennengelernt, die sich auf die Grafik Pipeline beziehen.

Rasterisierungsalgorithmus

Wichtig zu merken:

- Rasterisierungsalgorithmus setzt sich aus mehreren *Einzelalgorithmen* zusammen:
  - Transformationen, Clipping, Scan Konvertierung, Beleuchtung, Texture Mapping, Alpha Blending, **etc.**
- "Herz" des Rasterisierungsalgorithmus: Scan Konvertierung (Pineda).
- Rasterisierungsalgorithmus **Grundlage der Grafik Pipeline**.
- Rasterisierungsalgorithmus kann auch wieder "Building Block" sein:
  - Beispiel Shadow Mapping: rendere erst von der Lichtquelle aus, danach **2. Rendering Pass**, benutzt Sichtbarkeitsinformation als Schwarz-Weiß Textur.
  - Beispiel Deferred Shading..
- Komplexität ist u. a. proportional zur Anzahl Lichtquellen!

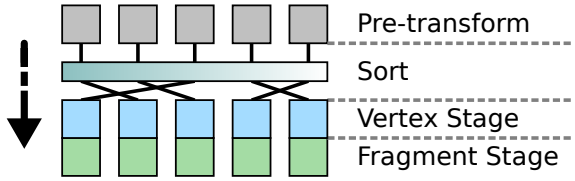
Strahlverfolgung



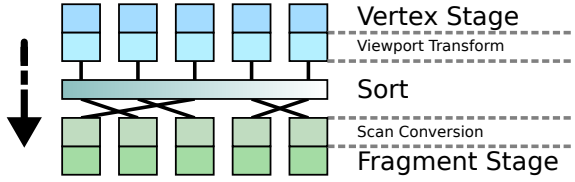
- Andere Familie von Algorithmen.
- Sichtbarkeitstest aufwendiger als bei Rasterisierung, dafür flexibler.
- Häufig (aber nicht notwendigerweise) Offline Rendering.

Paralleles Rendern

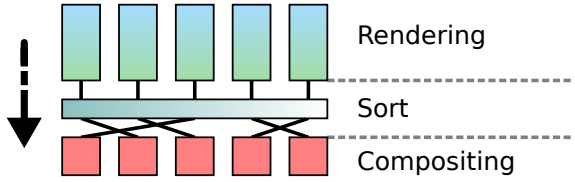
- Molnars Taxonomie: Paralleles Rendering als Sortierproblem.
- Sort-First: verteile Geometrie auf Prozessoren für Bildschirmbereiche.



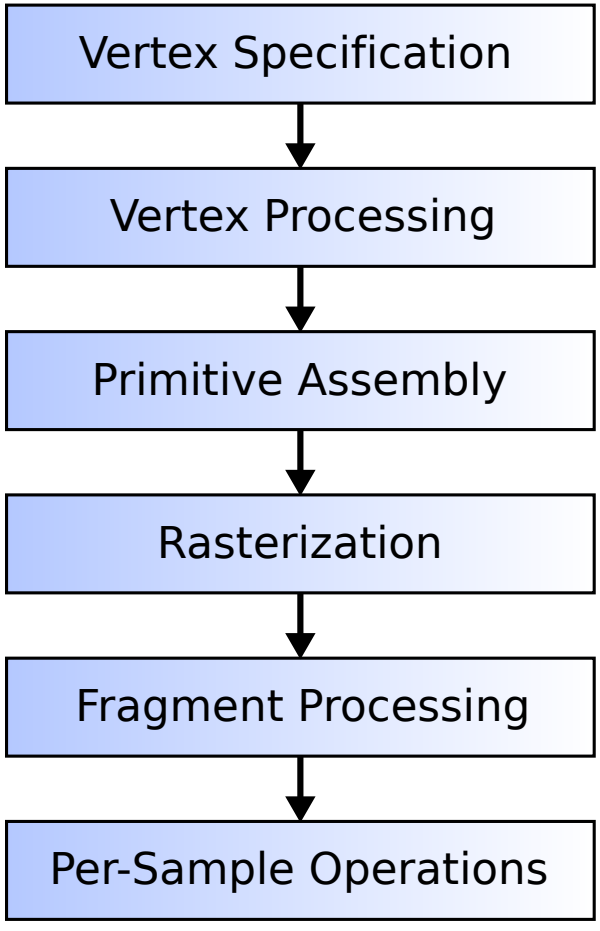
- Sort-Middle: verteile in Bildschirmkoordinaten transformierte Geometrie auf Prozessoren für Bildschirmbereiche.



- Sort-Last: verteile Geometrie auf dedizierte Prozessoren, unabhängig vom Bildschirmbereich, den sie überlappt.



Grafik Pipeline



Grafik Pipeline: technische Sicht auf den Rasterisierungsalgorithmus. (Rasterization := Scan Konvertierung.)

Grafik Pipeline

Wichtig zu merken:
Grafik Pipeline *heute* mittels programmierbarer und nicht programmierbarer Stages implementiert.

Progammierbare Stages werden von "Unified Shader Cores" bearbeitet $\Rightarrow$ Lastverteilung.

GPU Architektur heutzutage: sehr viele Threads, deren Zustand dauernd persistent in riesigen Register Files gehalten wird $\Rightarrow$ Umschalten zwischen Threads sehr schnell. Schnelles Umschalten $\Rightarrow$ Potenzial um Latenz zu verstecken.

Grafik Pipeline

Wichtig zu merken:
Durchsatzraten auf verschiedensten Levels:

- 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 texturieren und zusammensetzen (Compositing).
- Texturspeicher: Größe der maximal nutzbaren Texturen.
- Display Bandbreite: Bandbreite, mit der Pixel aus Grafikspeicher an Displays verteilt werden können.

Heutige Grafik Pipelines haben *paralleles Host Interface*, mehrere Raster Engines (zur Scan Konvertierung), mehrere Textureinheiten und mehrere ROPs.

APIs

Grafik Pipeline Grundlage jedes Grafik APIs (OpenGL, DirectX, Vulkan,..).

Vulkan API: sehr explizit, Objektmodell abstrahiert gesamtes Software und Hardware Umfeld der GPU.

Dedizierte APIs für GPGPU: Compute Shader; CUDA; OpenCL etc.

Wichtig:
Grafikkarten haben viel kompliziertere Speicherhierarchie als z. B. CPUs. Textureinheiten mit dedizierten Caches; Konstantenspeicher, der von tausenden Threads geteilt wird; Shared Memory (on-chip, ähnlich wie L1), der im Programm explizit verwendet werden kann. GPGPU APIs exponieren dieses Speichermodell.

APIs

CUDA

- Stream Computing Modell.
- Parallele Ausführung in *Kernels*.
- Gitterbasiertes Threading Modell.
- Datentransfers (Host/Device, Device/Host, Device/Device) explizit.
- Performante GPU Programme maximieren die *GPU Auslastung*.
- Vermeide Branching, vermeide Vorberechnung von Zwischenergebnissen.
- Alternativen: OpenCL, Metal, Compute Shader, Microsoft AMP, AMD ROCm,...

Probeklausur

Do. 19.7.2018, 12:00h, im üblichen Vorlesung + Übung Slot.

Fr. 20.7.2018: Besprechung ausgesuchter Aufgaben.

Zur Probeklausur (aber natürlich nicht zur Prüfung) dürfen Sie beliebige Unterlagen und Hilfsmittel mitbringen.

Viel Erfolg beim Lernen und für die Klausur!