Vorstellung der Hardware

Seminar: Paralleles Rechnen auf der Grafikkarte
Andreas Beyer – Uni-Heidelberg - SS 2009

Inhalt

- Was zeichnet eine Grafikkarte aus
- Die Grafik macht die Karte
- Komponenten & Aufbau der Hardware
- Evolution der Pipeline zu Unified Shadern
- Performancevergleich & Skalierbarkeit
- Beispiele reiner „Number-Cruncher“
- Ausblick & Fazit

Wir werden uns anschauen was die interessanten Eckdaten einer GraKa sind die sie der CPU gegenüber attraktiv machen

An einem simplen Beispiel wird die originale Funktionsweise einer GraKa erklärt um zu zeigen wieso die Hardware in ihrer heutigen Form genau so besteht

Die Pipeline und ihre „Evolution“ werden betrachtet – letztlich mit der Erkenntnis, dass CPU die Prinzipien von GPUs nachahmen und umgekehrt GPUs des GPGPU-Computings wegen mittlerweile klassische CPU-Elemente realisiert.

Die in der Pipeline befindlichen frei programmierbaren Stufen werden vorgestellt und Mechanismen die der Ausführungs-Optimierung dienen erklärt.
(multithreading / Unified Shader / StreamProzessoren)

Motivation

- GPU auf spezielle Probleme beschränkt
  - keine große Flexibilität
  - kein großer Verwaltungsaufwand

  ➔ „Viel Platz“ für Rechenoperationen
- Unter bestimmten Voraussetzungen enorme Geschwindigkeitssteigerung gegenüber CPUs
- GPU arbeitet massiv parallel
- Bessere Speicheranbindung als CPU

Im Vergleich zur CPU und der umgebenden Architektur bietet die GPU samt Peripherie einige Eigenheiten die sie üfr bestimte Problemstellung attraktiv macht. Diese sollen im Weiteren erörtert werden.

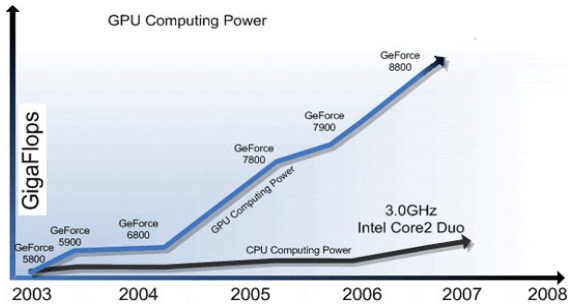
Vergleich der Geschwindigkeiten

Komponente	**Bandbreite**
GPU Speicherinterface	130 GB/sec
PCI Express Bus (×16)	8 GB/sec
Speicherinterface (FSB)	12 GB/sec
Speicherinterface (QIP/HT)	20GB /sec

Vergleich der Latenzen

Komponente	Zugriffszeit
GPU Speicher GDDR5	0,4 ns
GPU Speicher GDDR4	0,8 ns
CPU Speicher DDR(1/2/3)	8-12ns

Leistungszuwachs: GPUs



- …sind schnell
  - Core 2 XE QX 6700:
    - 23 GFLOPS
    - 20 GB/sec max (QPI)
  - GeForce GT 200:
    - 1000 GFLOPs
    - 130 GB/sec max
- ...werden schneller schneller
  - CPUs: jährlich x1,5
    → in 10 Jahren x60
  - GPUs: jährlich > x2,0
    → in 10 Jahren > x1000

Wie arbeitet eine Grafikkarte?

- 3D-Daten liegen als Menge von Punken vor
- Diese werden in verschiedenen Schritten transformiert / komponiert
- Die Abfolge dieser hintereinander geschalteten Operationen nennt man Pipeline
- hohen Parallelisierbarkeit der Aufgaben daher mehrere Pipelines auf einem Chip
- Zum Parametrisieren solcher Schritte werden Grafik-APIs verwendet
- Diese abstrahieren die darunter liegende Hardware

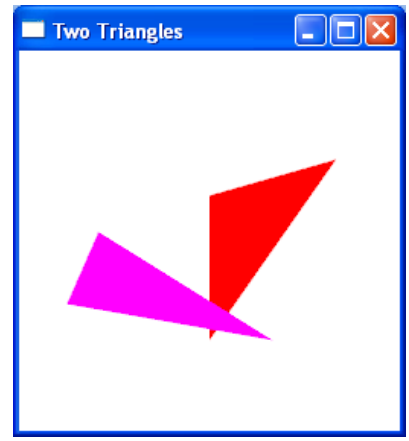
Es wird die prinzipielle Arbeitsweise einer GPU beschrieben, wie sie zur originalen Verwendung, also der Verarbeitung und Darstellung von 3D/2D-Szenen üblich ist. Als Beispiel dient openGL. Ziel ist es mit diesem Wissen zu verstehen wie man selbige Arbeitsschritte zum lösen mathematischer Problemstellungen (in Matrixform) nutzen kann.

Exemplarisch für OpenGL

- OpenGL wurde als Zustandsautomat entworfen
- entstand ursprünglich aus dem von Silicon Graphics 1984 entwickelten IRIS GL
- der Standard wird 1992 von 3DLabs, Apple, AMD/ATI, Dell, IBM, Intel, NVIDIA, SGI und Sun entwickelt und kann seither frei verwendet werden
- herstellerunabhängig und für viele Sprachen verfügbar
- durch „verpacken“ einer mathematischen Problemstellung in eine grafische Umgebung kann diese auf der GPU gelöst werden.

OpenGL Codebeispiel in C (1)

- In diesem Beispiel werden die OpenGL Erweiterungen **G***raphics* **L***ibary* **U***tility* **T***oolkit* in einem C-Programm genutzt
- ein Ausgabefenster wird definiert
- darin zwei Dreiecke dargestellt



Codebeispiel wie aus einfach zu lesendem openGL-Code ein primitives Bild generiert wird. Die Komposition verschiedener Matrizen kann später zur Lösung mathematische Probleme „zweckentfremdet“ werden.

OpenGL Codebeispiel in C (2)

Setup

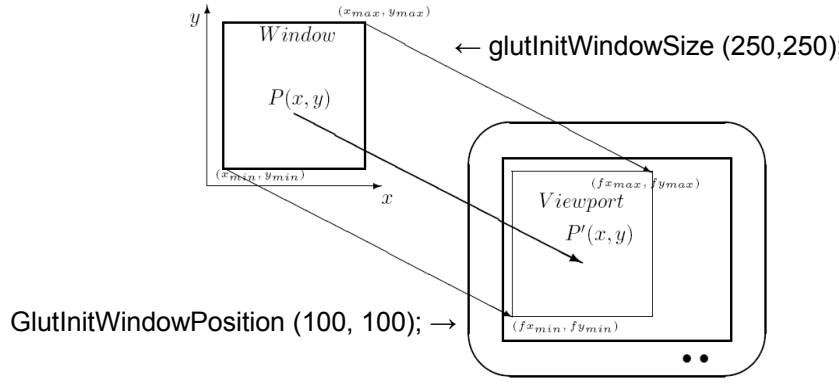
```
#include <GL/glut.h>
#include <stdio.h>

int main(int argc,char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE |
     GLUT_RGB);
  glutInitWindowSize (250,250);
  glutInitWindowPosition (100, 100);
  glutCreateWindow („Two Triangles");
  glClearColor (1.0, 1.0, 1.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(-120.0, 120.0, 15.0, 120.0);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Execution

```
void display(void) {
  glClear
     (GL_COLOR_BUFFER_BIT);
  glColor3f (1.0,0.0,0.0);
  glBegin(GL_POLYGON);
    glVertex2i (0, 20);
    glVertex2i (100, 110);
    glVertex2i (0, 80);
  glEnd();
  glColor3f (0.0,0.0,1.0);
  glBegin(GL_POLYGON);
    glVertex2i (-90, 50);
    glVertex2i (-90, 100);
    glVertex2i (40, 40);
  glEnd();
glFlush ();
}
```

Auswahl des Sichtbereichs



- Liefert als Ergebnis alle nötigen Informationen über unser Koordinatensystem

Darstellungsfenster und Inhaltsbereich definieren

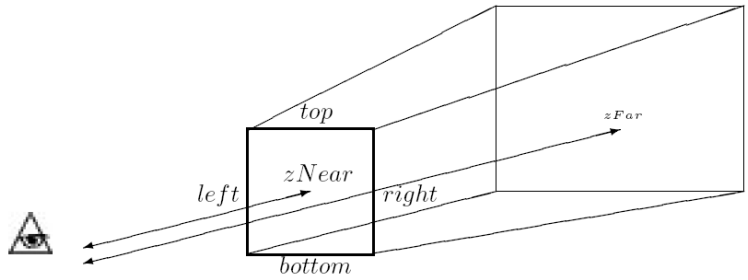
Der Farbraum

- Hintergrundfarbe wird definiert

  glClearColor (1.0, 1.0, 1.0, 0.0);

  - RGBA → 100% jedes Farbkanals
  - 0% Transparenz
- Wir zeichnen auf weißen Grund
  - Dieser Zustand kann mit glClear() wieder erreicht werden

Zur 3D Sicht: Frustum-Matrix



- Darstellungsraum wird in 3D definiert:
  - Ausschnittsgröße (top, bottom, left, right)
  - Blickwinkel (mittig von zNear zu Ausschnitt)
  - Blickdistanz (zNear bis zFar)

Hier wird der Raum definiert in dem dann Grafikprimitive (Punkte, Linien, Flächen, Körper) plaziert und manipuliert werden können.

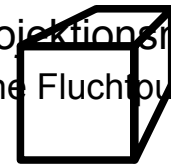
Frustum als Matrix

$$\begin{pmatrix} \frac{2zNear}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2zNear}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{zFar+zNear}{zFar-zNear} & -\frac{2zFarzNear}{zFar-zNear} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- Informationen als Matrix repräsentiert
  - Im folgenden in Kette der Berechnungen verwendet

Verschiedene Sichten

- OpenGL kennt die Sichen
  - perspektivische Projektionsmatrix
    - perspektivische Verzerrung von Objekten
  - Orthogonale Projektionsmatrix
    - Darstellung ohne Fluchtpunkt



  - In unserem Beispiel wird eine orthogonale 2D-Projektion initiiert: gluOrtho2D(left,right,top,bottom);

Zeichnen: glutDisplayFunc(display)

- Darstellung laut Funktion → void display(void) { … }
- Geometrische Objekte
  - Festlegung der Farbe → glColor3f (1.0,0.0,0.0);
  - Form der folgenden Punkte definieren

```
glBegin(GL_POLYGON);
   ...
glEnd();
```

  - Liste der Raumkoordinaten bildet Primitive

```
glVertex2i (0, 20);          ← zweidimensional mit Integer-Werten
glVertex2i (100, 110);
glVertex2i (0, 80);
```

  - In diesem Fall also ein rotes Dreieck

Ähnlichkeitstransformationen

Drehen:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\beta & -sin\beta \\ sin\beta & cos\beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Dehnen (Breite b, Höhe h):

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} b & 0 \\ 0 & h \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Kursivieren:

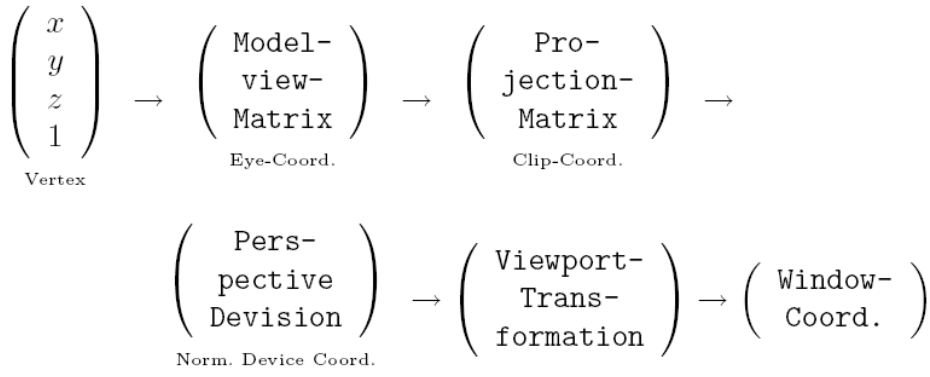
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

- Ebenfalls als Matrixoperationen definiert
  - Hier in 2D → 3D analog
  - Bezugspunkt ist Ursprung des Koordinatensystems

Zum Manipulieren der Grafikprimitive

Die Berechnungsschritte

- Abfolge der Matrixoperationen



Sukzessive Anwendung aller definierten Matrizen auf jeden Punkt der Grafikprimitive liefert das letzten Endes dargestellte Bild

For any point on object $p=(x\ y\ z\ 1)^T$, applying N, the new point $q=(x'\ y'\ z'\ w')^T$, where

$$x'=x,\ y'=y,\ z'=\alpha z+\beta,\ w'=-z$$

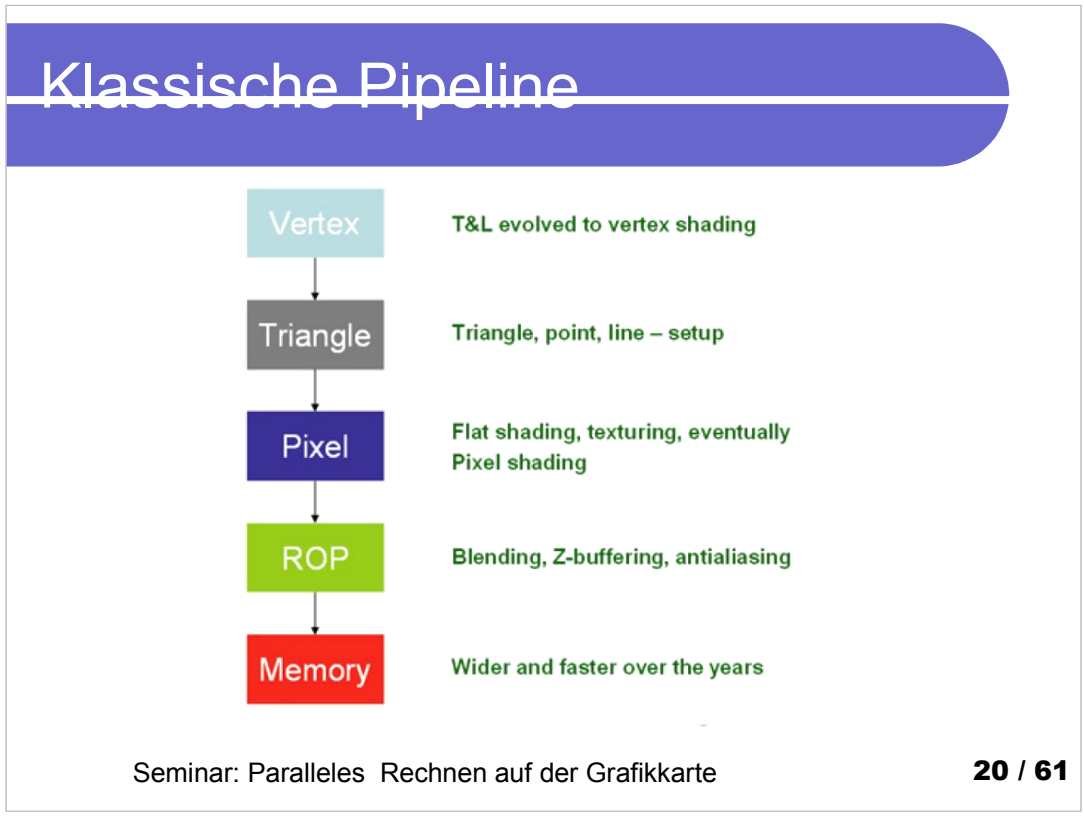
After perspective division, the point $(x, y, z, 1)$ goes to

$$x'' = -x/z,\ y'' = -y/z,\ z'' = -(\alpha + \beta/z)$$

Komponenten & Aufbau

- Klassische Grafikpipeline
- Hierarchisches Layout mit Vertex-/Fragment Prozessoren
- Konzept der Streaming-Prozessoren
- Layout ohne Hierarchie

Detailierte Betrachtung alle beteiligter Komponenten



Starre Abfolge

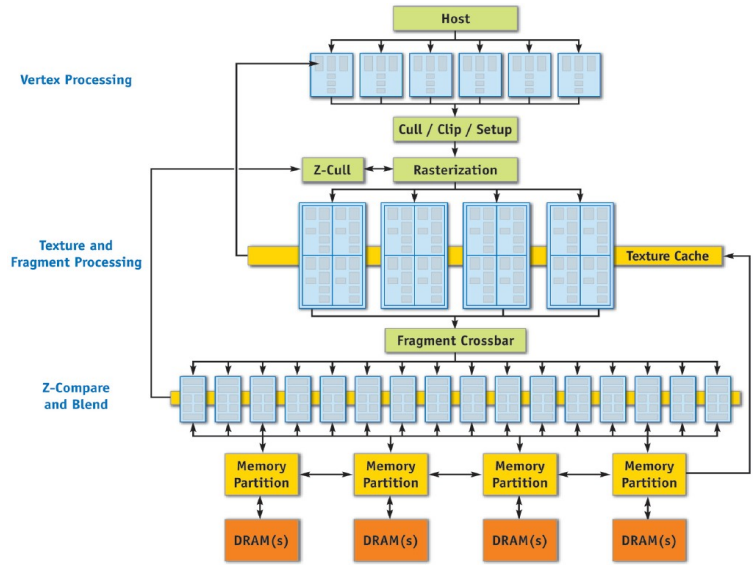
Nutzen zur nichtgrafischen Berechnung lediglich durch „einbetten der Aufgaben in Grafik“

Klassische Pipeline

- Nachteile:
  - traditionell mehr Pixelshader-Anweisungen als Vertexshader-Programme verwendet
  - Modifikation bereits berechneter Daten durch einen erneuten Pipelinedurchlauf ist kompliziert
  - ALU oft im Leerlauf

die neue Architektur ist in der Lage, Daten, die den Shadercore bereits durchlaufen haben, erneut im Shadercore zu modifizieren. Der Output des Streamprozessors wird dabei in ein Register, einen extrem schnellen Zwischenspeicher, geschrieben und anschließend entweder erneut durch den Unified Shader Core geschleust (eine „Loop-Funktion“) oder an die ROPs weitergegeben.

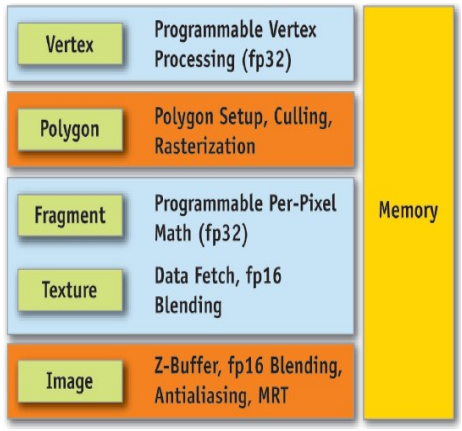
Programmierbars altes Layout (GeForce6-Serie)



GeForce 6-Reihe

Neue Möglichkeiten gezielt eigenen Code auszuführen, da V-P und T&F-P frei programmierbar sind (nächsten 2 Folien)

Vereinfachte Sicht der Architektur

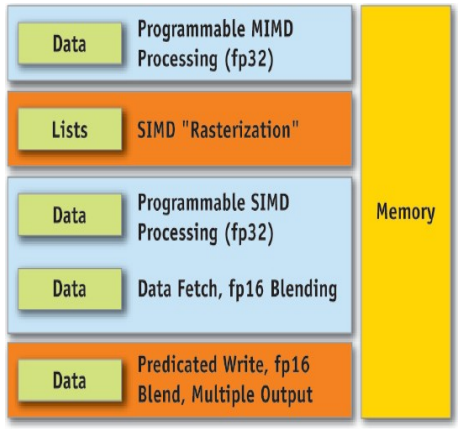


Grafikanwendungen:

- programmierbarer Vertex Prozessor
- programmierbarer Fragment Prozessor
- Texturfilter
- Tiefentest / Blending

Eine Einheit, da parallelisierbarer Code von der Karte selbst an Prozessoren verteilt wird

Alternative Sicht einer Recheneinheit



wissenschaftliches Rechnen

- zwei serielle Prozessoren:
- Vertex-Prozessor und Fragment-Prozessor
- Beide rechnen mit einfacher FP-Genauigkeit
- greifen über die Textur-Einheit mit 35GB/sec auf den Speicher zu

Flynnsche Klassifikation / Flynn'sche Taxonomie

SISD (Single Instruction Stream, Single Data Stream)
SIMD (Single Instruction Stream, Multiple Data Streams)

SIMD-Computer, auch bekannt als Array-Prozessoren oder Vektorprozessor, dienen der schnellen Ausführung gleichartiger Rechenoperationen auf

mehrere gleichzeitig eintreffende oder zur Verfügung stehende Eingangsdatenströme und werden vorwiegend in der digitalen Bildverarbeitung eingesetzt.

MISD (Multiple Instruction Streams, Single Data Stream)
MIMD (Multiple Instruction Streams, Multiple Data Streams)

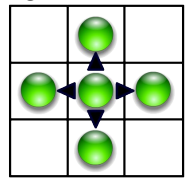
MIMD-Computer führen gleichzeitig verschiedene Operationen auf verschieden gearteten Eingangsdatenströmen durch,

wobei die Verteilung der Aufgaben an die zur Verfügung stehenden Ressourcen meistens durch einen oder mehrere Prozessoren

des Prozessorverbandes selbst zur Laufzeit durchgeführt wird. Jeder Prozessor hat Zugriff auf die Daten anderer Prozessoren.

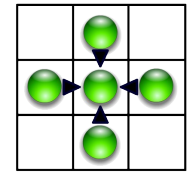
Limitierungen der zwei Prozessoren

Vertex Prozessor
- programmierbar (SIMD / MIMD)
- Kann eine MultiplyAdd-Operation auf Vektor mit drei Elementen ausführen
- Kann eine Funktion auf einen Skalar anwenden
- Wahlfreier schreibender Zugriff auf Textur-Puffer



Scatter

Fragment Prozessor
- programmierbar (SIMD)
- Kann ENTWEDER eine MultiplyAdd-Operation auf Vektor mit drei Elementen ausführen oder eine Funktion auf einen Skalar anwenden
- Wahlfreier lesender Zugriff auf Textur-Puffer



Gather

Beide teilen sich den Texture-Cache

Mittlerweile gibt es diese Unterscheidung nicht mehr, die folgenden Folien erläutern das Konzept des unified Shaders

Wie zieht man daraus Nutzen?

- genaue Strukturen und Mechanismen der Grafikchips sind geheim
- früher mussten mathematische Probleme in Matrixform in die Pipeline eingebracht werden
- seit DirectX 8.0 und den verbundenen Layoutforderungen Zugriff über Hochsprachen und Frameworks

GPGPU Konzept

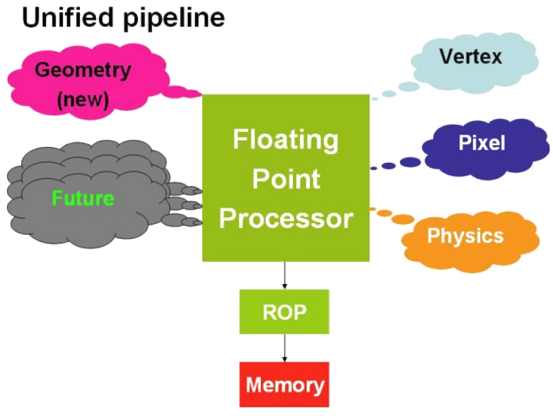
- Quadratische Textur = Eingabematrix
- Fragment Program = Computational Kernel
- One-to-one Pixel zu Texel Mapping:
  - Data-Dimensioned Viewport
  - Orthographic Projection.
- Viewport-großes Quadrat = Berechnungsstart
- Copy-to-Texture = Feedback / Ausgabe

1) Matrizen werden in der GPU-Programmierung in Form von (quadratischen) Texturen abgespeichert und verarbeitet
2) Das Gesamtproblem sollte zerlegbar / parallelisierbar sein., nur dann werden die Teilprobleme als „Programm“ des StreamingProzessors genutzt
3) Die Ergebnisse liegen als Texel vor, um eine 1:1-Beziehung zu Pixeln herzustellen wird das „Ergebnis“ über einen passend konfigurierten Viewport betrachtet
4) Die Neuberechnungen der GraKa werden durch eine Bildänderung provoziert, nach Laden des Programms wird nun also ein Quadrat aufgespannt, welches den gesammten Viewport und somit alle Pixel und zugehörigen Texturen abdeckt
5) Ein so gewonnenes Bild kann in der Grafikkarte wieder als Textur abgelegt werden um nun mit den Ergebnissen der vorherigen Rechnung weiterzuarbeiten

Aktuell: GPU wird homogener

- Prozessoren nicht mehr nach Aufgaben getrennt
- Thread-Prozessor verteilt Shader-Code an Shader-Einheiten
- Ergebnis kann als Eingabe in Core zurückgeschrieben werden

DX10: Unified Shader

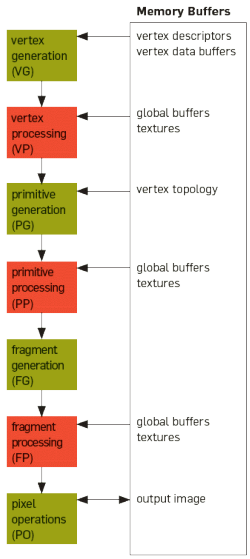


- Konzept des Streaming-Prozessor löst hierarchische Pipeline ab
- „Innenleben“ wohl gehütetes Geheimnis
- Rechenleistung nutzbar durch von Entwicklern bereitgestellte Tools
- Kann alles

Für Grafik wichtig: früher kam es durch festgelegtes Verhältnis der verbauten Vertex-Shader und Fragment-Shader zu suboptiimaler Auslastung, der generische unified-Shader löst dieses Problem.

Für GPGPU wichtig: statt je nach Aufgabe eine der Komponententypen wählen zu müssen und an den starren Ablaof der Pipeline gebunden zu sein wird nun ein mehrfachdurchlauf der einzelnen Komponenten möglich.

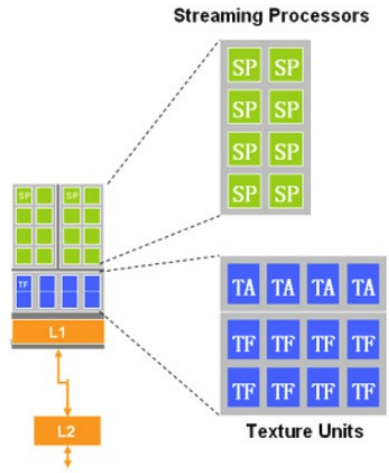
Neu: programmierbare Pipeline mit Streamprozessoren



- Punkte definieren 3D-Gitter
- Meist 2D-Sicht des 3D-Input
- Ordnet Primitive in Stream
- Splittet oder vereint Primitive
- Rasterisierung
- Texturen und Reflektionen
- Ausgabe sichtbarer Elemente

Rot → frei programmierbar

Die Streaming-Prozessoren



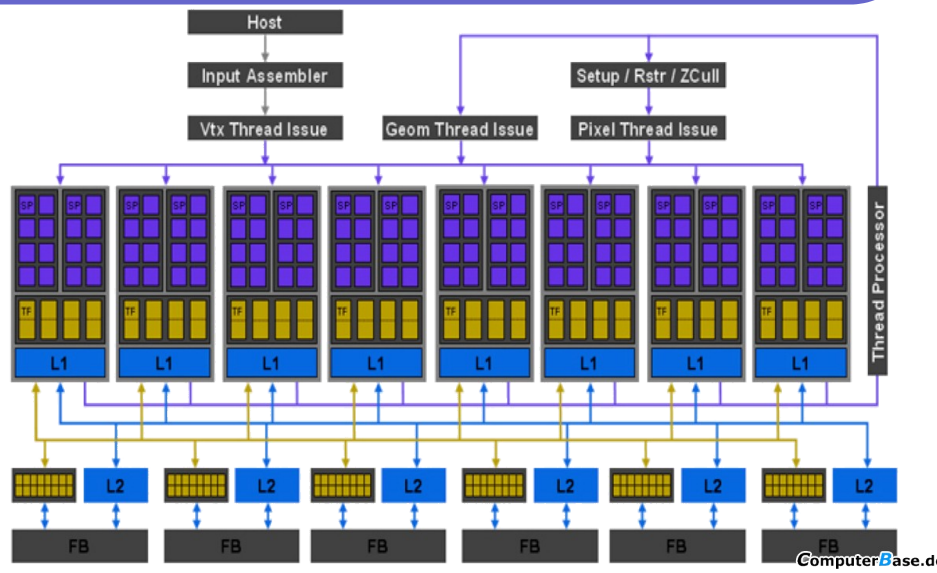
- SP = Streaming Processors
- TF = Texture Filtering Unit
- TA = Texture Address Unit
- L1/L2 = Caches

Nvidia G80: Aufbau im Detail

Auf dem G80 können 64 Pixel gefiltert, aber nur 32 Pixel adressiert werden.

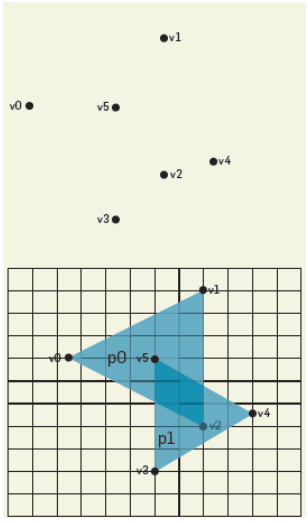
Bei einfacher bilinearer Filterung können insgesamt 32 Texel pro Takt fertiggestellt werden, wobei die restlichen 32 Textureinheiten quasi brach liegen. Wenn man nun allerdings einen 2-fachen bilinearen anisotropen Filter einsetzt, benötigen die 32 Textureinheiten bereits zwei Takte, um die Pixel zu filtern. Mit Hilfe der 32 restlichen Einheiten kann dieser Schritt nun schon in einem Taktdurchlauf erledigt werden, weswegen 2xAF mit bilinearer Filterung theoretisch keinerlei Leistung kostet.

aktuelles Layout ohne Hierarchie (GeForce8-Serie)



GeForce 8800

Pipelineoperationen (1)



- „ VertexGeneration“ sechs Punkte des output stream definieren zwei Dreiecke
- Entsprechend „VertexProcessing“ und „PrimitiveGeneration“ werden sie positioniert und gruppiert (p0 & p1)

Pipelineoperationen (2)

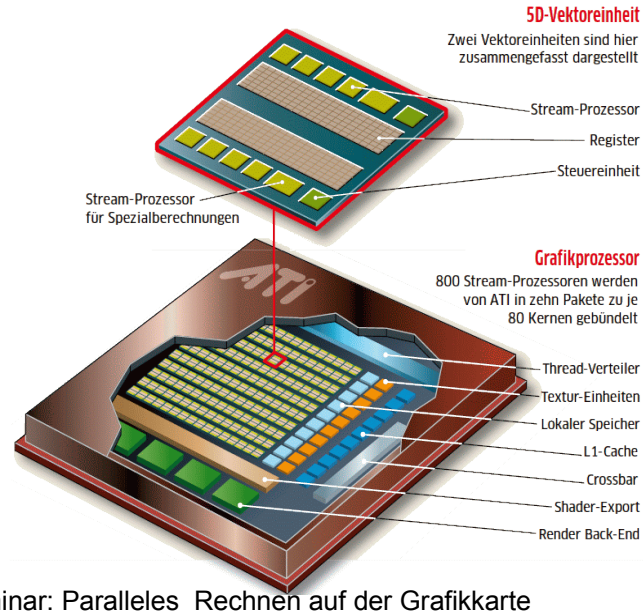
- „FragmentGenerator“ stellt diese als Fragmente frei
- „FragmentProcessor“ bestimmt Oberfläche jedes Fragment
- „PixelOperations“ trägt sichtbare Fragmente in output ein

Facts & Figures

EVGA GEFORCE GTX 280	
	
TECHNISCHE DATEN	
Chip	nVidia GT-200
Transistoren	ca. 1,4 Mrd.
Chiptakt	602 MHz
Shader-Takt	1.296 MHz
Speicher	1.024 MByte GDDR3
Speichertakt	1.107 MHz
Rechenleistung	ca. 933 GFLOPs

MSI R 4870 T2D 512	
	
TECHNISCHE DATEN	
Chip	ATI RV770
Transistoren	ca. 965 Mio
Chiptakt	750 MHz
Shader-Takt	750 MHz
Speicher	512 MByte GDDR5
Speichertakt	1.800 MHz
Rechenleistung	ca. 1.200 GFLOPs

ATI RV7700 - Aufbau



800 Stream-Prozessoren

Eine MADD Anweisung pro Takt

Je fünf zusammengefaßt (5D) zu ienem Shader-Kern (ähnlich einem nvidia-SP)

→ 160 Shader-Kerne

10 Kerne bilden einen SIMD-Core mit 16kB Speicher

→ 16 SIMD Cores

Jeder Core hat ein TextureCluster aus vier Textur-Einheiten, eine Adresseinheit und einen L1-Texture-Cache

Verbunden sind die Cores über einen Bus mit 16kB-Cache

Mit dem Speicher Kommuniziert wird über vier L2-Caches

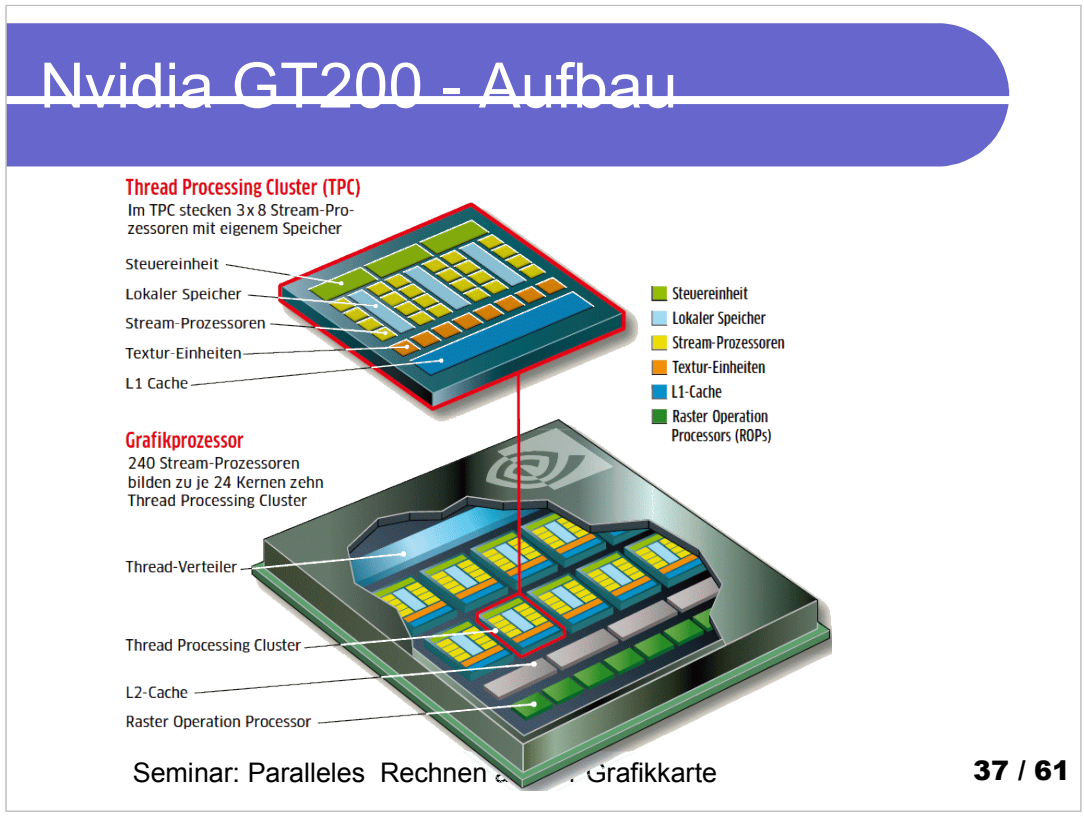
Verteilt werden die Aufgaben vom UltraThreadDispatchProcessor (UTDP)

Dazu 16 ROPs

256 bit breit zum Speicher (512MB)

1200 GFLOP/s

Kann 64bit-FP (DP)



240 Stream-Prozessoren

1D: eine skalare Berechnung (R, G, B oder A) oder eine MADD pro Takt

Plus eine MUL-Anweisung

8 Alus bilden StreamingMultiprocessor (SM) und haben 16kB-Cache

→ 30 SMs

Je 3 bilden mit einem 8 Textureineiten und einem L1-Cache einen TexturProcessingCluster

Intern SIMT (SingelInstructionMultipleThreads), untereinander MIMD

Dazu 32 ROPs

512 bit breit zum Speicher (1024MB)

933 GFLOP/s

Kann 64bit FP (DP)

Wie umgeht man Amdahls Gesetz?!

- Gar nicht!
- Amdahls Gesetz:
  - Ein jedes Programm hat parallelisierbaren Anteil P und sequentiellen Teil 1-P
  - Ist P ideal parallelisierbar beträgt der (lineare) Speedup bei Ausführung auf N Knoten P/N

$$S = \frac{1}{(1-P)+\frac{P}{N}} \leqslant \frac{1}{1-P}$$

  - → ist P = 95% schafft man maximal Faktor 20

Nicht parallelisierbar: Initialisierung / Speicherallokation

Super-linearen-Speedup diskutieren wir hier nicht (eher unwahrscheinlich)

Meist verzerrter Vergleich durch Caching-Effekte während Parallelisierung

Eigentlich felht in der Formel noch o(N) für wachsenden Kommunikations- und Synchronisations-Overhead

Dann ergeben sich auch die aus der Praxis bekannten Glocken (statt Asymptoten)

Wie umgeht man Amdahls Gesetz?!

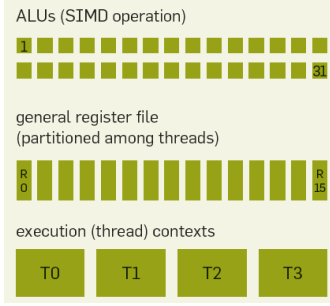
- … aber man kann tricksen ;-)
  - *Hardware-Multithreading*
  - Ein Core initialisiert x Threads und hält immer x-1 Kontexte im Speicher während pro Takt an einem Thread gearbeitet wird
  - Spekulativ wird (ohne Auftrag) initialisiert
  - Somit wird der sequentielle Teil des Threads „vorauseilend“ ausgeführt
- Im Folgenden ein Beispiel:

Nicht parallelisierbar: Initialisierung / Speicherallokation

Super-linearen-Speedup diskutieren wir hier nicht (eher unwahrscheinlich)

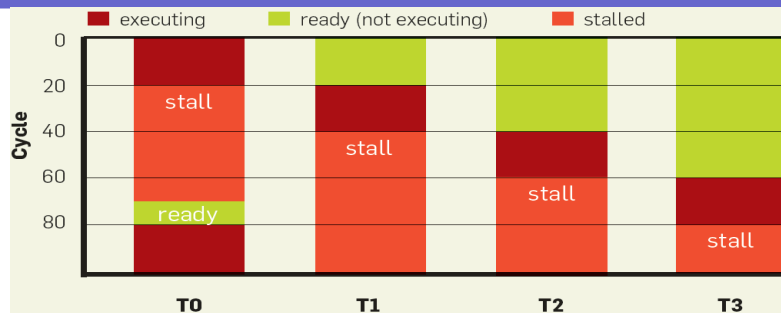
P=95% auf 240 Cores → Faktor 18,5
P=95% auf 800 Cores → Faktor 19,5

SIMD & dynamisches HW-Thread Scheduling (1)



- Nur ein Thread wird bearbeitet, aber vier Kontexte gespeichert
- Eine Instruktion bearbeitet 32 Komponenten des Vektors
- Die 16 Register fassen je einen Vektor aus 32 Floats und werden unter den Threads aufgeteilt
- Wartezeiten entstehen nur durch Kommunikation mit dem Textur-Puffer
- Dies kann 50 Zyklen dauern

SIMD & dynamisches HW-Thread Scheduling (2)



- Core (alle 20 ALUs) immer voll ausgelastet
- Konzept fußt darauf, dass immer ausreichend Fragmente zu berechnen sind
  - Hier: 1 Chip * 4 Threads * 32=128 Fragmente nötig
  - Realistischer: 16Chips * 16 Threads * 32 = **8192**

NumberCruncher – ohne VGA-Out

- Tesla-Reihe (nVidia)
- Torrenza (AMD)
- Firestream 9170 (AMD/ATI)
- Larrabee (Intel)
- Tsubame-Rechner (Sun)

Tesla-Reihe

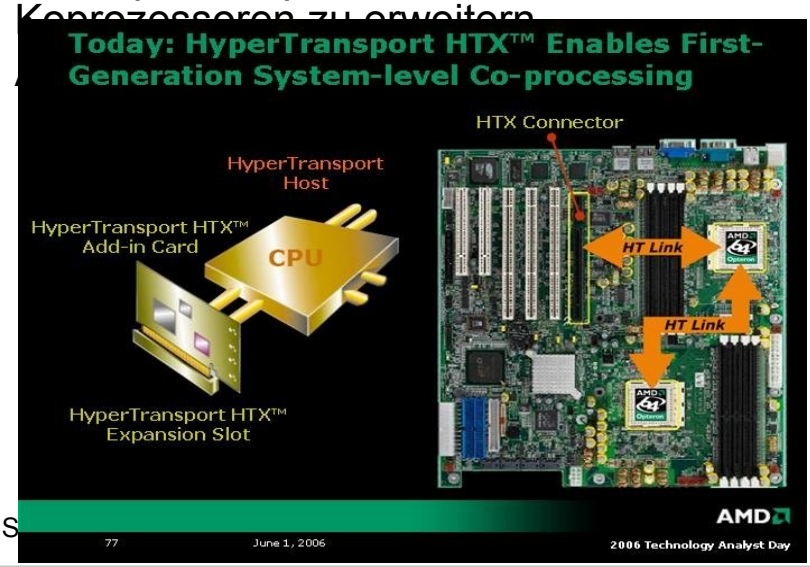
- 128 Prozessoren, 700 Mio. Transistoren liefern 518 Gigaflops
- Prinzipiell ein GeForce-8800-Serie mit 1,5 GB Ram
- "nVidia Tesla S870 GPU Computing Server " besteht aus 8 Karten

Der Tesla-Prozessor kostet ungefähr 1500 $, der Supercomputer 7500 $ und die Server-Variante sogar 12000 $

Zahlen sind aber wohl ein Jahr alt...

Torrenza

- Konzept um Opteron-Server mit Koprozessoren zu erweitern
- A



S

Wichtig ist die Art des Anshclußes, also die schnelle Anbindung über Hypertransport

Genau genommen läßt das Konzept offen ob hierbei eine weitere CPU, eine GPU oder ein FPGA angeschlossen werden

AMD Firestream 9170

- an RV670 Chips angelehnter Koprozessor
- 320 Stream-Prozessoren welche Fließkommaberechnungen in doppelter Genauigkeit ausführen
- 2GB GDDR3 Ram
- 800Mhz
- theoretische 500GFLOPs bei 150W Leistungsaufnahme
- zeitgleich erscheinendes auf Brook+ basierendes SDK

2000$ mit Spoiler

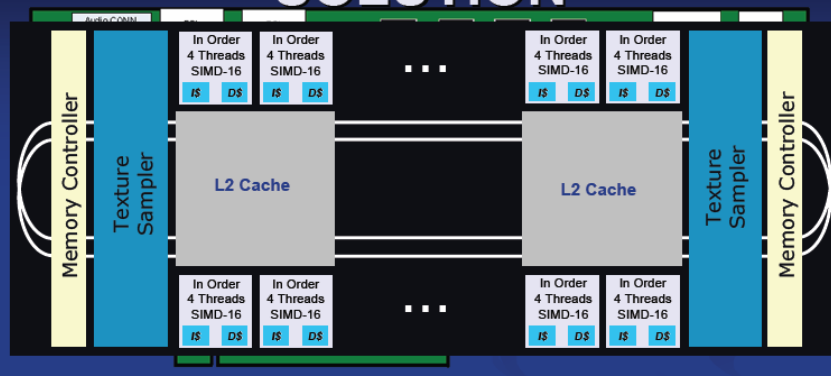
AMD/ATI sprechen manchmal von Brooke+ (was die Ausgangssprache bezeichnet) oder nennen ihr Framework CAL → Computer Abstraction Layer

Oder schlicht und ergreifent CTM → close to the metal

Larrabee

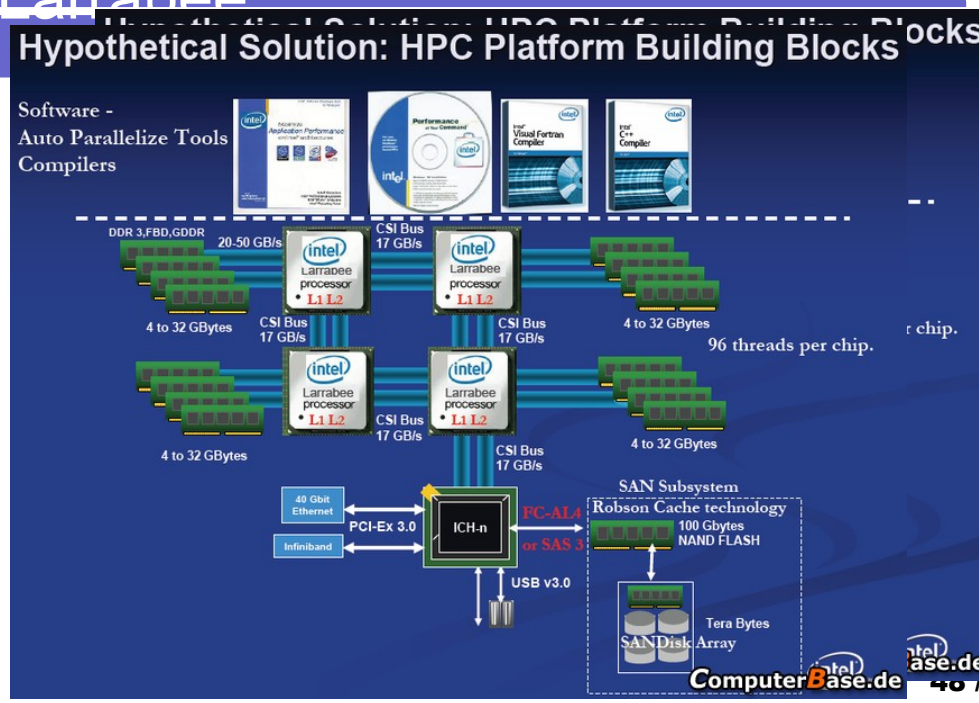
- 16 einfache Prozessoren, von denen jeder pro Takt 4 Threads ausführen kann
- Performance im Teraflop-Bereich
- bei einem Takt von 2 GHz nur ca. 150 Watt Leistungsaufnahme
- Pro Kern 32 KB L1- und 256 KB L2-Cache
- zwei Speichercontroller mit einer Gesamtbandbreite von 128 GB/s
- Pro Takt zwischen 128 und 256 Floating-Point-Operationen doppelter Genauigkeit

LARRABEE – TERA-SCALE SOLUTION



- Discrete high end GPU on general purpose platform
- TeraFlops of fully programmable performance
- **GPU -** >16 cores @ ~2.0GHz, >150W
- JPEG textures, physics acceleration, anti-aliasing, enhanced AI, Ray Tracing etc.

Larrabee



Tsubame-Rechner

- Cluster am Tokioter Technologieinstitut
- Mit 102 TeraFLOP/s auf Platz 16 der Top500
- Betrieben mit u.a. 360 PCI-X-Karten von Clearspeed
- je 192 Streaming-Prozessoren doppelter Genauigkeit

Clearspeed Advanced X620 schlappe 8000$

• The Tokyo Tech Supercomputer and UBiquitously Accessible Mass storage Environment (TSUBAME) redefines supercomputing

• 648 Sun Fire™ X4600 servers deliver 85 TeraFLOPS of peak raw compute capacity

• 42 Sun Fire X4500 Data Servers provide access to 1 petabyte of networked storage

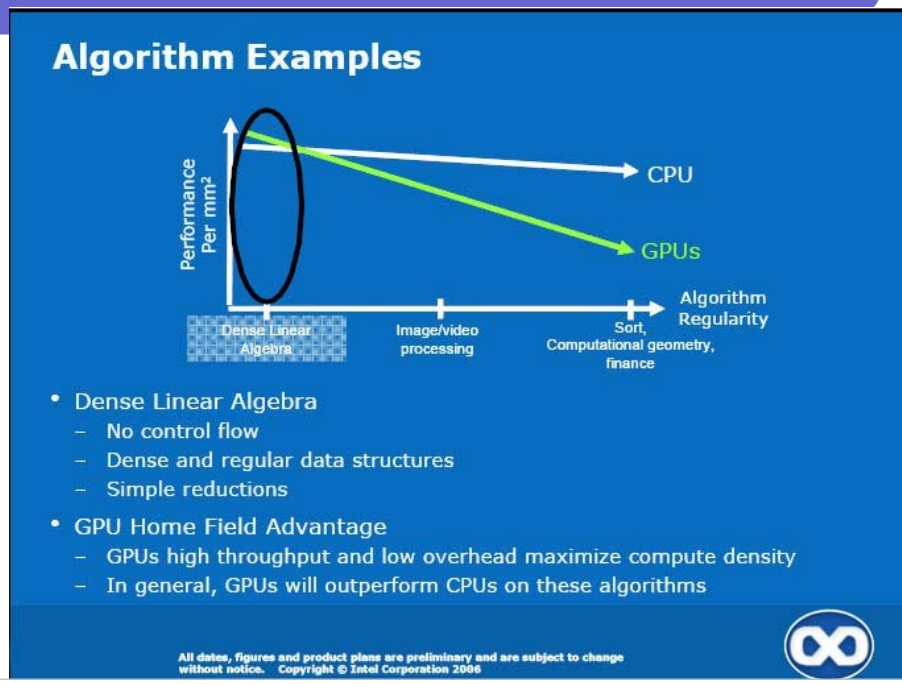
• ClearSpeed Advance accelerator boards configured in 360 compute nodes help the grid exceed 47 TeraFLOPS sustained Linpack performance

• Eight Voltaire Grid Director ISR9288 high-speed InfiniBand switches keep traffic in the grid moving

• Sun N1™ Grid Engine software distributes jobs across systems in the grid

• An innovative and integrated software stack enables common off-the-shelf applications, including PC applications, to run on the grid

Ausblick

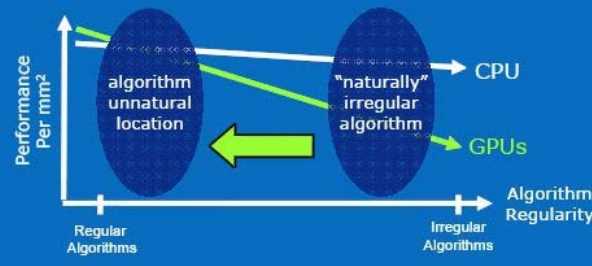


Recht naive Darstellung des prinzipiellen Problems: es gibt keine triviale Lösung sequenziellen Code zu parallelisieren – wie bei der multiprozessor-Programmierung muß auch bei GPGPU im Vorfeld die Parallelität „erdacht“ werden.

→ Aufgabe des Entwicklers

Ausblick

Dragging Applications "To the Left"



- Applications have a natural location on the continuum
- Possible to reformulate applications to use more regular algorithms
  - Usually requires significant programmer effort
  - Often implies algorithms not as well suited to application
  - HW efficiency improvements more than offset SW inefficiency
- Optimal solution is function of overall efficiency (HW, SW, programmer)

Fazit

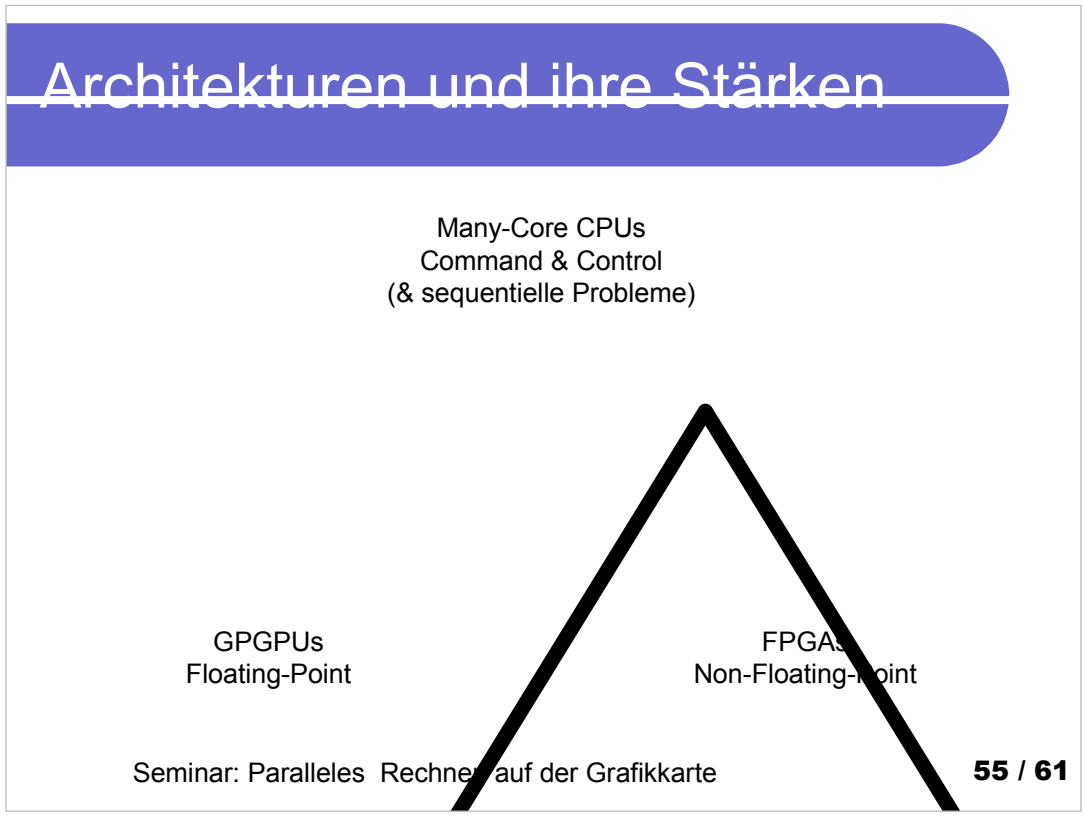
- Hardware ausgelegt für massiv parallele Berechnungen und schnelle I/O
- Frameworks erleichtern „Zweckentfremdung“ der programmierbaren Einheiten
- Doppelte Genauigkeit mittlerweile unterstützt
- Optimale Performance (nur) bei optimalen Problemen

Wo Licht ist, ist auch Schatten

- ALLE gut skalierenden Algorithmen werden verlagert...
- Adobe Creative Suite ۴ & TMPEG Express ۴ berichten:
  - „Wir arbeiten bereits auf der GPU“
- Global Secure Systems sagt:
  - „Mit CUDA wurde die Berechnung von WPA- und WPA۲-Schlüsseln um ۱۰.۰۰۰ Prozent beschleunigt“
- Svarychevski Michail Aleksandrovich stellt kostenlos seinen GPU-MD۵-Cracker zum Download bereit:
  - ۱۰۸ Mio Schlüssel /s auf einem Core ۲ Duo ۳,۰ GHz mit SSE۲-Befehlen
  - ۳۵۰ Mio Schlüssel /s auf einer GeForce ۹۶۰۰ GT

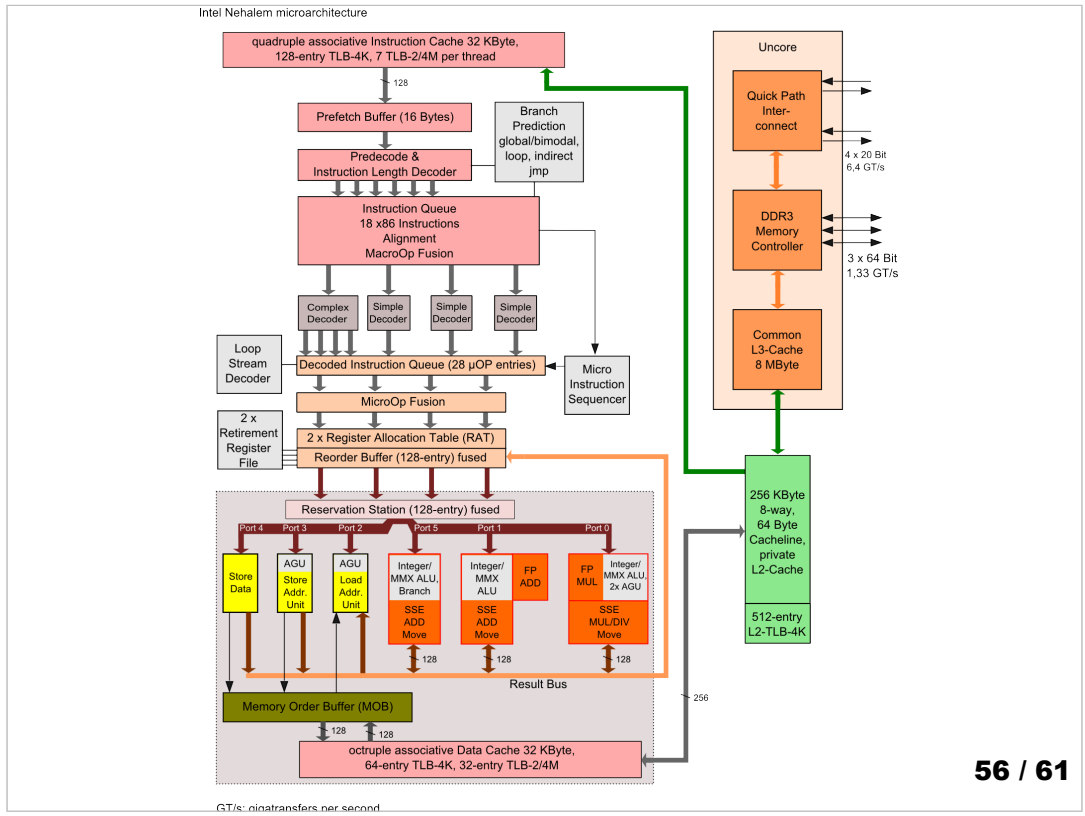
Kurz Angerissen

- CPUs bedienen sich der bei GPUs bewährten Mechanismen
  - Intels „Nehalem“-Architektur
- Der dritte große Sektor leistungsfähiger und massiv paralleler Hardware
  - FPGAs



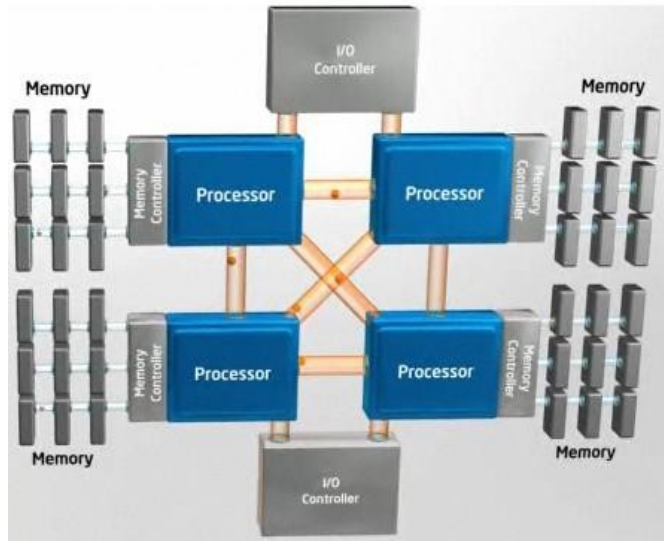
Jedes Problem hat eine dafü prädestinierte Hardware, neben CPU und GPU gibt es weiter, zB FPGAs.

Auch diese haben ein eigenes Programmiermodell samt stärken und Schwächen, auch diese nehmen dem Entwickler nicht die „Denkarbeit“ ab, wichtig ist, zu verstehen dass jede Architektur eine ganz Problemklasse abdeckt.



Beispiel um zu zeigen dass in neue CPU-Architekturen Ideen der GPU-Architekturen einfließen. (und umgekehrt)

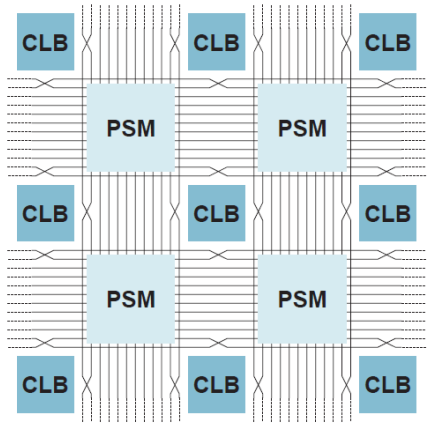
Nehalem Quad



Nehalem

- FSB weicht QuickPath Interconnect
  - Punkt-zu-Punkt Netzwerk ähnlich AMDs HyperTransport
- Ram-Anbindung über internen Controller
  - Aktueller Flaschenhals
- Simultaneous Multithreading
  - In getrennten Pipelines mehrere unabhängige Threads gleichzeitig zur Ausführung durch einen Core vorgehalten
    - Aha ;-)

Struktur eines FPGA



- **C**onfigurable **L**ogic **B**lock
  - Ram-Baustein der sechsstellige Wertetabelle eines boolschen Ausdrucks beinhaltet als LookupTable
- **P**rogrammable **S**witch **M**atrix
  - Verbindet CLBs um komplexe Ausdrücke zu realisieren
- dazu RAM und spezialisierte Blöcke (IO / PPC / AD&DA)

Eckdaten (FPGA / GPU / CPU)

Architektur	**FPGA**	**Microprocessor**	**GPU**
Art	Konfigurierbare Logic	On Chip Register	Stream Prozessoren
Transferrate	• TB/sec	• 100's GB/sec	• ?
Zellengröße	• 10's KB	• 1-10's KB	• 128 B
	Internal RAM	L1-L3 Cache	L1-L2 Cache
Transferrate	• 100's GB/sec	• 100's GB/sec	• 100's GB/sec
Zellengröße	• 100's KB	• 1-10's KB	• 16 KB shared
	Local Memories	DDR	GDDR
Transferrate	• ~10GB/sec	• 12GB/sec	• ~35GB/sec
Zellengröße	• 10-1000's MB	• 1-4 GB	• 0,5-1 GB
	System Memory	System Memory	System Memory
Transferrate	• 100-1000's MB/sec	• 100-1000's MB/sec	• 100-1000's MB/sec
Zellengröße	• Terabytes	• Terabytes	• Terabytes
Takt	• 400Mhz	• 3Ghz	• 1Ghz
Leistung	• 25W	• 240W	• 310W

„Konventioneller“ Entwurf

statisch

- Im ISE
  - Designs in VHDL beschreiben
  - in Bitfile übersetzen (Synthese und Place&Route)
- Mit PlanAhead
  - Floorplanning (UCF-File)
  - Timing modellieren
- Mit PinAhead
  - Pinout planen
- Im EDK
  - PowerPC ansteuern & einrichten
- Via iMPACT
  - alles auf FPGA laden und testen

dynamisch (zzgl)

- Im ISE
  - Komponenten in VHDL beschreiben
  - Synthese der einzelnen Komponenten
- Mit PlanAhead
  - Chip in dynamische und statische Area(s) einteilen.
  - Der/den dynamischen Area(s) die eben sythetisierten Komponenten zuordnen
  - Jeweils Place&Route durchlaufen
- Im EDK
  - Scheduling für PowerPC entwickeln

Programmierung auf FPGAs

- Wegen fehlendem gemeinsamem Problem noch keine generische API
- Sprachen, Strukturen und Komplexität des Toolflows schrecken ab
- Unschlagbar in interner Datentransferrate und Parallelisierungsgrad primitiver Operationen

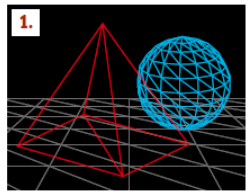
Quellen

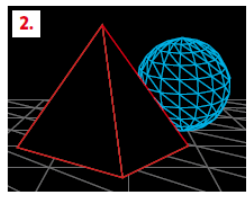
- SIGGRAPH
- www.GPGPU.org
- Tweakguides - Graphics
- WebLecture: David Kirk / Wen-mei Hwu
- Tesla
- Torrenza
- Larrabee
- Tsubame
- Beyond3D
- ComputerBase - GeForce 8800GTX - Technik im Detail
- Chip 10/2008 S 60 „So funktionieren Top Grafikkarten“
- ACM Vol51No10 S 50 „A closer look to GPUs“
- Computergrafik1 Skript von Frau Dr. Krömker

Danke

- Fragen?

Schritte der Grafikpipeline (1)

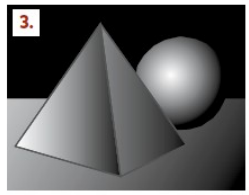




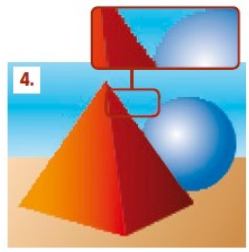
- Vertex-Shader erzeugt aus Koordinaten ein 3D-Gitter
- Verdeckte Elemente werden entfernt

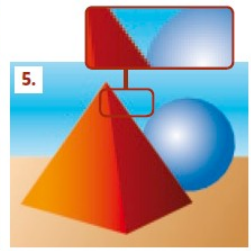
Alternative Darstellung der Arbeitsschritte moderner Graikpipelines

Schritte der Grafikpipeline (2)



- Lichtquellen werden gesetzt
- Pixel-Shader und Textureinheiten färben Oberflächen





- Anti-Aliasing glättet Kanten

Datenblatt aktueller Karten

	Radeon HD 4890	GeForce GTX 285
Chip	RV790	GT200b
Transistoren	ca. 959 Mio.	ca. 1,4 Mrd.
Fertigung	55 nm	55 nm
Chiptakt	850 MHz	648 MHz
Shadertakt	850 MHz	1.476 MHz
Shader-Einheiten (MADD)	160 (5D)	240 (1D)
FLOPs (MADD/ADD)	1.360 GFOP/s	1.063 GFLOPs
ROPs	16	32
Pixelfüllrate	13600 MPix/s	20736 MPix/s
TMUs	40	80
TAUs	40	80
Texelfüllrate	34000 Mtex/s	51840 MTex/s
Shader-Model	SM 4.1	SM 4
Hybrid-CF/-SLI	X	X
Speichermenge	1.024 MB GDDR5	1.024 MB GDDR3
Speichertakt	1.950 MHz	1.242 MHz
Speicherinterface	256 Bit	512 Bit
Speicherbandbreite	124800 MB/s	158976 MB/s

MultiplyAdd Werte sind theoretische Spitzenwerte.

In der Praxis teilweise zu 80% erreichbar

Bei ähnlichen Werten sind diese Karten doch konzeptionell sehr unterschiedlich aufgebaut.

ATI setzt auf sehr hohe Rechenleistung für mathematische Aufgaben, nVidia auf leistungsstarke Textureinheiten