LOBACHEVSKY STATE UNIVERSITY
of NIZHNI NOVGOROD
National Research University

Computing Mathematics and Cybernetics faculty

Software department

CS255. Computer Graphics Introduction Course

Графический 3D конвейер и синтез изображений

Graphics Pipeline. Shaders

проф. Турлапов В.Е.,
vadim.turlapov@cs.vmk.unn.ru

3D-сцена и графический конвейер

Геометрическая стадия.

Шейдеры п.п.3-5

1. **Wireframe** (Каркасное) моделирование поверхности объектов с учетом видимого объема (Camera, Frustum, View Volume). Формирование списка отображаемых объектов.

2. **Tesselation**. Тесселяция или триангуляция (triangulation): разбиение поверхности на плоские полигональные элементы. Вместо криволинейной поверхности – **полигональная модель**, представленная вершинами (vertex)

3. **Transformation** (трансформация) : перемещение, изменение формы посредством матричных преобразований вершин в пределах видимого объема

4. **Lighting**. Расчет освещенности вершин для закрашивания (shading) в последующем треугольников полигональных поверхностей с интерполяцией по: Гуро (Gouraud shading), Фонгу (Phong shading)

5.**Camera-ViewPort**. Проецирование 3D-объекта с сохранением информации о расстоянии (о глубине) каждой из вершин до плоскости проекции

6. **Triangle setup**. Подготовка (компоновка) треугольников объекта: генерация текстурных координат; сортировка вершин; отбор и отбрасывание нелицевых граней (culling)

3D-сцена и графический конвейер

Стадия рендеринга

(rendering -преобразование, представление)

1. **HSR (Hidden Surface Removal)** – Удаление скрытых, для текущей точки наблюдения, поверхностей. Алгоритмы: z-сортировка; z-буферизация

2. **Texture mapping (пиксельные шейдеры).** Текстурирование – первый этап растровой графики. Тексели-элементы текстуры формата 2^m x 2^n. Соответствие пикселов и текселов устанавливается по результатам проецирования.
Приемы: **MIP-mapping** (текстуры с различным разрешением); **perspective corrected texture mapping** (коррекция перспективы); **Filtering** (BLF, LF, TLF, Anisotropic); **bump mapping** (наложение Мелкомасштаб-го рельефа); мультитекстурирование (конвейерное) – 4n блоков текстурирования;

3. **α-blending and fogging** (моделирование полупрозрачности, коррекция цвета: α- смешивание и затуманивание)

4. **Anti-aliasing** (Коррекция зазубренности границ: edge AA (краевой) и full screen AA (полный ~ FSAA)
Приемы: super sampling (супер- и мультисэмплинг); tile based архитектура; техника аккумулятора.

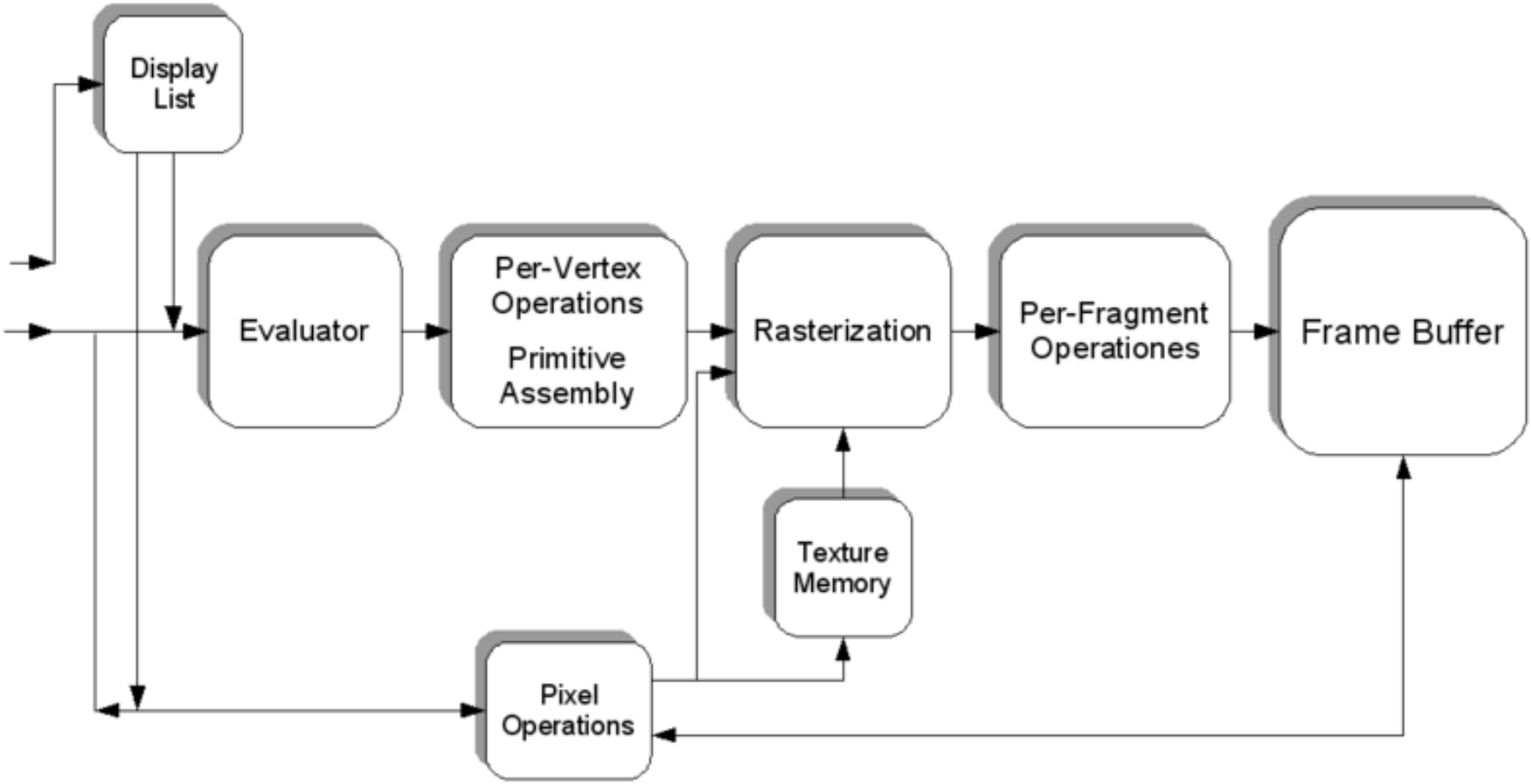
5. **Dithering**. Интерполяция недостающих цветов (для индексированногоцвета)

6. **Frame buffer**. Формирование кадрового буфера для формирования выходного аналогового сигнала. Приемы: double buffering двойная буферизация ~ формирование 2-го начинается до того как закончится передача в ЦАП (RAM DAC) первого

7. **Post-processing.** Пост-обработка: двумерные эффекты над целым кадром.

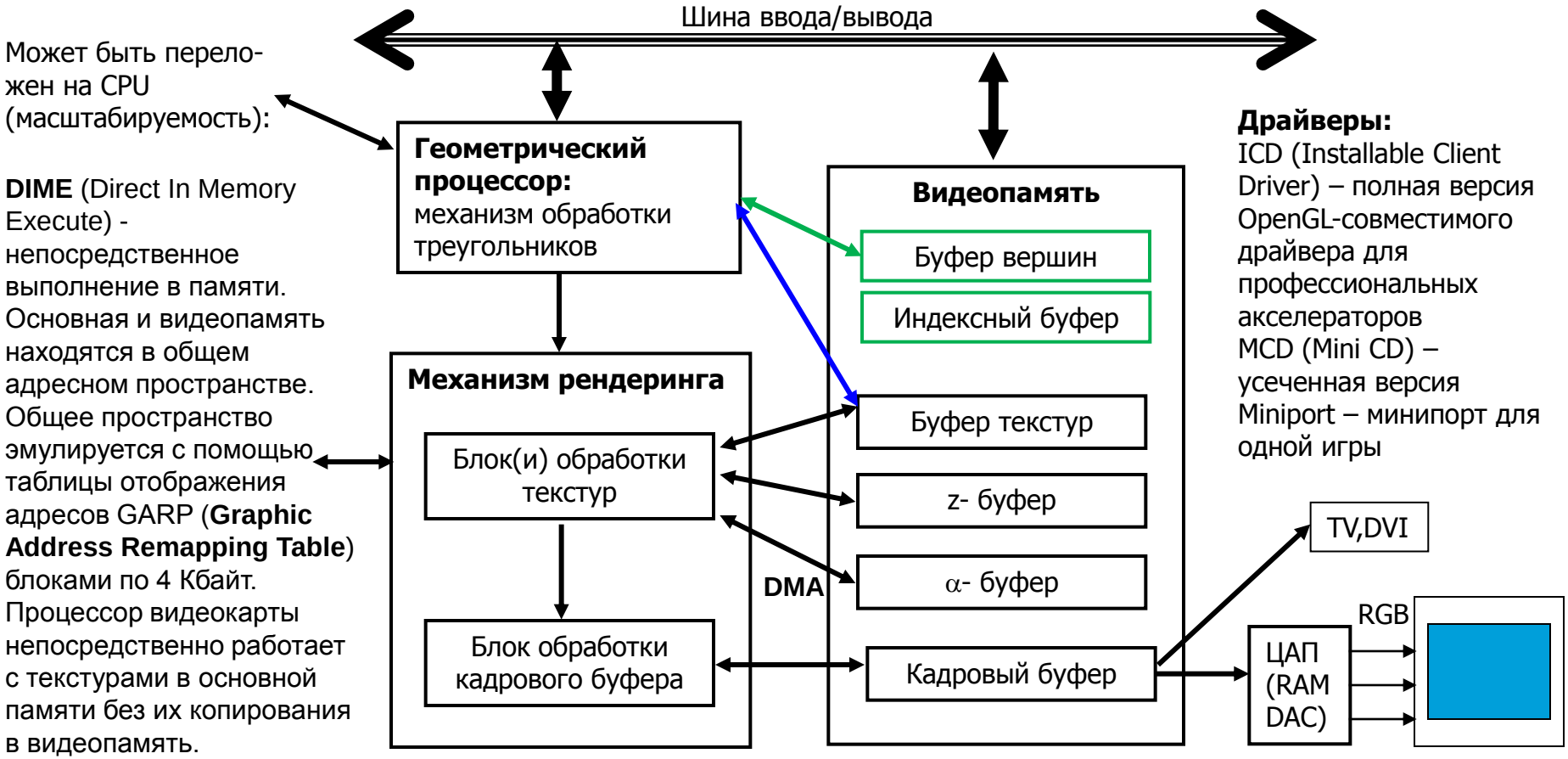
3D-сцена и графический конвейер

Концепция OpenGL 4.0 pipeline



Обобщенная структура 3D-акселератора (видеокарты, GPU)

Поколения акселераторов, роль OpenGL

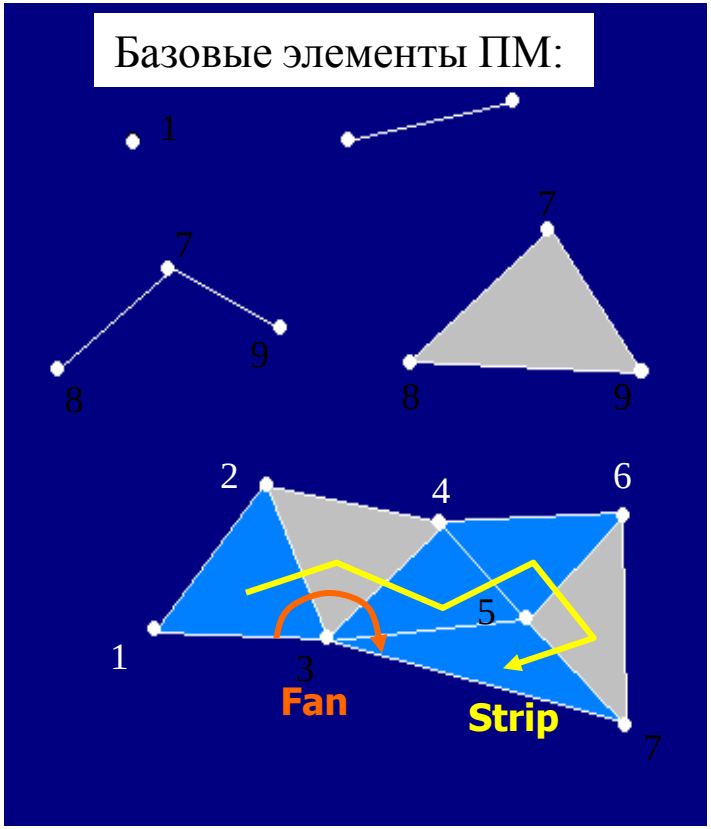


Геометрический процессор

Структура данных о полигональной модели

Данные о вершинах. Потоки. Нормали. Координаты текстуры → Буфер вершин

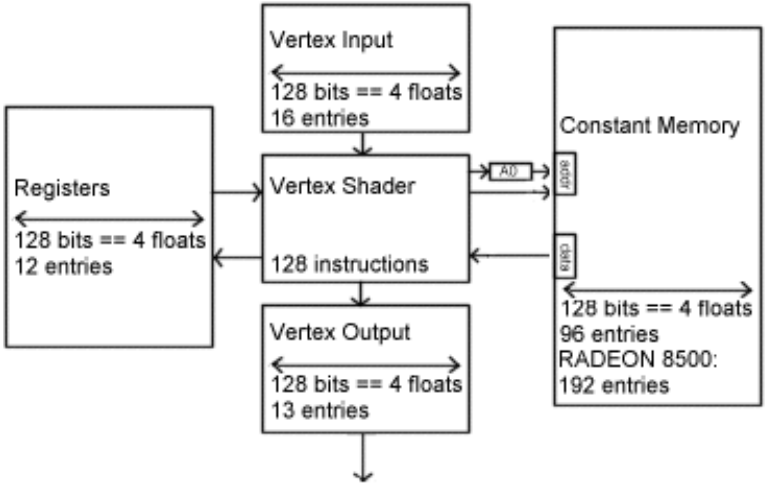
```
struct VS_INPUT
{
  float2 Pos      : POSITION0;
  float2 UV       : TEXCOORD0;
  float  ZPos     : POSITION1;
  float3 Norm     : NORMAL;
  float3 Col      : COLOR0;

  static const DWORD FVF;
};

const DWORD MYVERTEX::FVF =
D3DFVF_XY | D3DFVF_TEX1 | D3DFVF_Z
| D3DFVF_NORMAL | D3DFVF_DIFFUSE;
```



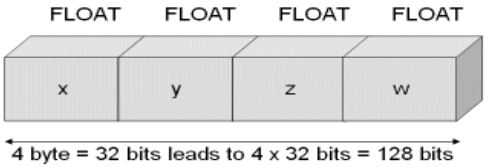
Архитектура шейдеров

➢ Вершинные шейдеры (vertex shaders)

Vertex Shader 128bit Architecture



Регистр:



Специализация регистров шейдера

Registers:	Number of Registers	Properties
Input (v0 - v15)	16	RO
Output (o*)	GeForce 3/4TI: 9; RADEON 8500: 11	WO
Constants (c0 - c95)	vs.1.1 Specification: 96; RADEON 8500: 192	RO
Temporary (r0 - r11)	12	RW
Address (a0.x)	1 (vs.1.1 and higher)	WO (W: only with mov)

Выходные регистры вершинного шейдера:

Name	Value	Description
oDn	2 quad-floats	Цвет для пиксельного шейдера diffuse (oD0) и specular (oD1).
oPos	1 quad-float	Output position в однородном усеченном пространстве
oTn	до 8 quad-floats	Выходные координаты текстуры
oPts.x	1 scalar float	Размер точки (скаляр)
oFog.x	1 scalar float	Коэффициент затуманивания

Вершинные шейдеры. Ассемблер

1.Объявление вершинного шейдера в C++, установка константных регистров

```
float c[4] = {0.0f,0.5f,1.0f,2.0f};
DWORD dwDecl_0[] = {
D3DVSD_STREAM(0), //VSD-Vertex Shader Declaration
D3DVSD_REG(0, D3DVSDT_FLOAT3 ),  //inpRegister v0
D3DVSD_REG(5, D3DVSDT_D3DCOLOR), //inpRegister v5
// set a few constants
D3DVSD_CONST(0,1),*(DWORD*)&c[0],*(DWORD*)&c[1]
,*(DWORD*)&c[2],*(DWORD*)&c[3],
D3DVSD_END()};
pd3dDevice->SetVertexShaderConstant(28,
&matProjTransp,4); // 4 строки
pd3dDevice->CreateVertexDeclaration(dwDecl_0,
&pVertexDeclaration );
```

2.Компиляция и создание, установка и удаление вершинного шейдера

```
D3DXAssembleShaderFromFile("PShader.psh",0,0,0,
    &pCode,0); // for Assembler Shader
D3DXCompileShaderFromFile(strPath,0,0,"Perlin",
"vs_1_1",dwShaderFlags,&pCode,0,&pConstantTable));
pd3dDevice->CreateVertexShader((DWORD*)pCode->
GetBufferPointer(),&pVertexShader);
pCode->Release();
m_pd3dDevice->SetVertexShader(pVertexShader);
m_pd3dDevice->DeleteVertexShader(pVertexShader);
```

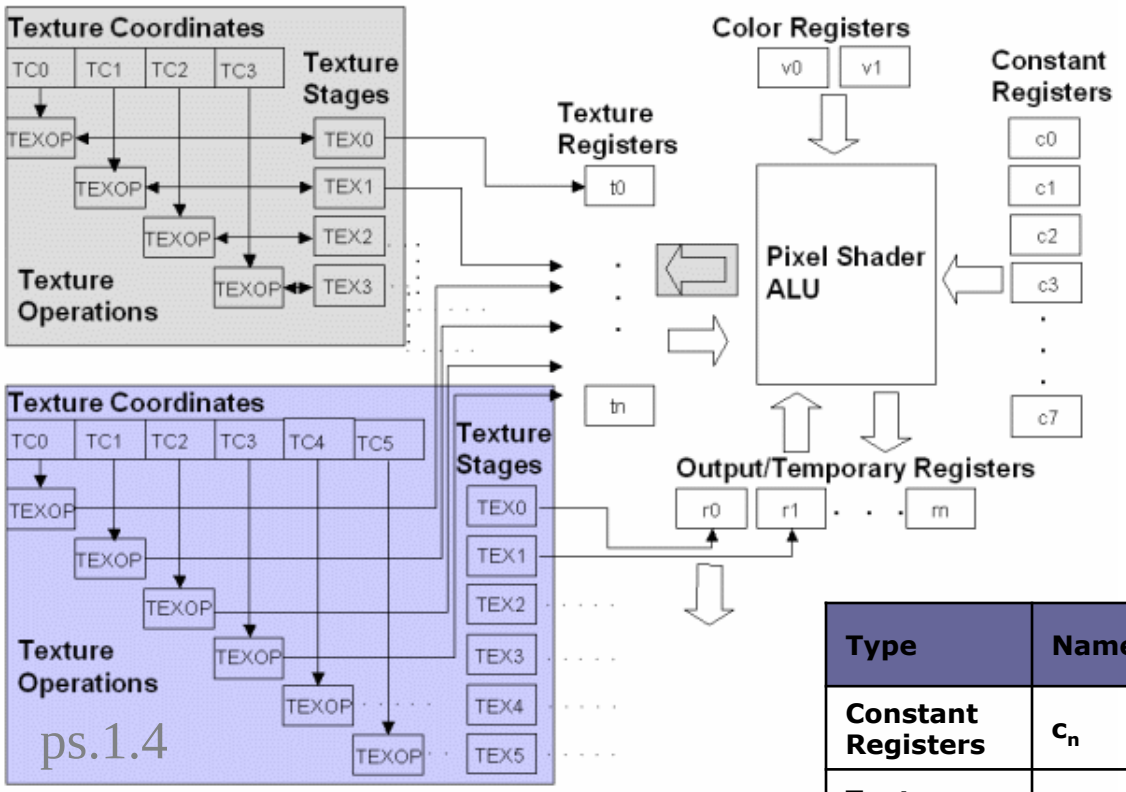
3.Примеры вершинного шейдера **на ассемблере**

```
mov a0.x,r1.x
m4x3 r4,v0,c[a0.x + 9];
m3x3 r5,v3,c[a0.x + 9];
          ; Scale by fog parameters :
; c5.x = fog start
; c5.y = fog end
; c5.z = 1/range
; c5.w = fog max
dp4 r2, v0, c2 ; r2 = distance to camera
sge r3, c0, c0 ; r3 = 1, sge=(>=?)
; camera space depth (z) - fog start
add r2, r2, -c5.x
; 1.0 - (z - fog start) * 1/range
mad r3.x, -r2.x, c5.z, r3.x
mad r4,r3,c9,r4 ; r4=r3*c9+r4

mov R1, -R2.xyyz  ; swizzling example
mov  R1.xw, R2    ; masking example

; r0 = r1 x r2 (3-vector cross-product)
mul  r0, r1.yzxw, r2.zxyw
mad  r0, -r2.yzxw, r1.zxyw, r0
;
mov   oD0,r4
```

Архитектура шейдеров

➢ Пиксельные шейдеры (pixel shaders)

Pixel Shader 128bit Architecture



Этапы работы с пиксельными шейдерами

- Проверка поддержки Pixel Shader
- Установка Flags для текстуры (with D3DTSS_* flags)
- Установка Texture (SetTexture())
- Определение констант (with SetPixelShaderConstant()/def)
- Pixel Shader инструкции
  - о Адресации текстуры
  - о Арифметические
- Ассемблирование Pixel Shader
- Создание Pixel Shader
- Установка Pixel Shader
- Освобождение ресурсов

Type	Name	ps.1.1	Read/ Write
Constant Registers	c_n	8	RO
Texture Registers	t_n	4	RW → RO
Temporary Registers	r_n	2→n	RW
Color Registers	v_n	2	RO

Пиксельные шейдеры. Ассемблер

1.Объявление пиксельного шейдера в C++, установка константных регистров

```
if(pCaps->PixelShaderVersion
          < D3DPS_VERSION(1,1)) return E_FAIL;
//Flags
m_pd3dDevice->SetTextureStageState( 0/*Stage*/,
D3DTSS_COLORARG1, D3DTA_TEXTURE ); //…
m_pd3dDevice->SetTextureStageState( 0,
D3DTSS_COLOROP, D3DTOP_MODULATE );
//Set
m_pd3dDevice->SetTexture(0, m_pWallTexture);
//Define Constants
HRESULT SetPixelShaderConstant(DWORD Register,
  CONST void* pConstantData,DWORD ConstantCount);
def c0, 0.30, 0.59, 0.11, 1.0 ; [-1.0..1.0]
```

2.Компиляция и создание, установка и удаление пиксельного шейдера

```
DXUtil_FindMediaFile(Shad,_T("environment.psh"));
if(FAILED(D3DXAssembleShaderFromFile(Shad,0,NULL,
&pCode,NULL)))    return E_FAIL;

if( FAILED(m_pd3dDevice->CreatePixelShader(
(DWORD*)pCode->GetBufferPointer(),&m_dwPixShader)
)) return E_FAIL;
m_pd3dDevice->SetPixelShader( m_dwPixShader );
m_pd3dDevice->DeletePixelShader( m_dwPixShader );
```

3.Примеры пиксельного шейдера на ассемблере

```
; t1 holds the color map
; bump matrix set with the
; D3DTSS_BUMPENVMAT* flags
ps.1.1 ; start of asm. shader
tex t0    ; bump map with du,dv,lum data
texbeml t1, t0   ; compute u, v
          ; sample t1 using u, v
          ; apply luminance correction
mov r0, t1 ; output result

texcoord t0 ; convert texture
            ; coordinates to color
mov r0, t0  ; move color into output

tex t0      ; color map
texdp3 t1, t0 ; t1 = (t1) dot (t0)
mov r0, t1 ; output result
lrp r0, v0, t0, r1 ; v0*t0+(1-v0)*r1

; (t1) holds row #1 of the 3x2 matrix
; (t2) holds row #2 of the 3x2 matrix
; t0 holds normal map
tex t0 ; normal map
texm3x2pad t1, t0  ;calc. z from row #1
; calculates w from row #2
; stores a result in t2 depending on
; if (w == 0) t2 = 1.0;
; else t2 = z/w;
texm3x2depth t2, t0
```

Microsoft HLSL (High Level Shading Language)

файлы эффектов, техники (technique) и проходы (pass)

1.Mex. Пример файла эффектов с шейдерами на ассемблере

```
#define C_EYE_POSITION      27
#define C_DISPLACEMENTS     30
// light direction
float3 L = normalize(float3(-0.2f,-0.8f,.4f));
// light intensity
float4 I_a = float4(0.3f, 0.3f, 0.3f, 1.0f);
float4 I_d = float4(0.6f, 0.6f, 0.6f, 1.0f);
float4 I_s = float4(0.8f, 0.8f, 0.8f, 1.0f);
// material specular
float4 k_s = float4(1.0f, 1.0f, 1.0f, 1.0f);
// transformations
float4x4 World      : WORLD;
float4x4 View       : VIEW;
float4x4 Projection : PROJECTION;
// eye position
float3 Eye;
// textures
texture FinTex;
texture ShellTex;
static const float4 vOne = float4(1, 1, 1, 1);
VertexShader ShellVS = asm { vs.1.1
          dcl_position v0
          dcl_normal   v3
          dcl_color0   v4
          dcl_texcoord v6
```

2.Вода. Пример файла эффектов на языке MS HLSL

```
#include "light_scattering_constants.h"
// transformations
float4x4 mWorldViewProj: WORLDVIEWPROJECTION;
float4 vCameraPos: worldcamerapos;
float4 sun_color : suncolor =
{0.578f,0.578f,0.578f,0.0f};
float4 xAxis = {1.0f, 0.0f, 0.0f, 0.0f};
float4 vHalf = {0.5f, 0.5f, 0.5f, 0.0f};
float3 waterColor0 = {0.15f, 0.4f, 0.5f};
float3 waterColor1 = {0.1f, 0.15f, 0.3f};
texture tex0 : TEXTURE; // blend mask
struct VS_INPUT
{   float2 Pos       : POSITION;
    float  ZPos0     : POSITION1;
    float2 Norm0     : NORMAL0;
    float  ZPos1     : POSITION2;
    float2 Norm1     : NORMAL1;       };
struct VS_OUTPUT
{   float4 Pos       : POSITION;
    float3 Col       : COLOR0;
    float3 T0        : TEXCOORD0;
    float3 T1        : TEXCOORD1;     };

VS_OUTPUT VS(VS_INPUT v)
{
  VS_OUTPUT Out = (VS_OUTPUT)0;
```


Архитектура GPU и параллельные вычисления

или

За что стоит любить массивно-параллельные (massive parallel) архитектуры

Два закона (Амдала и Густафсона) и одно общее следствие

Закон Амдала (Amdahl's law, 1967):

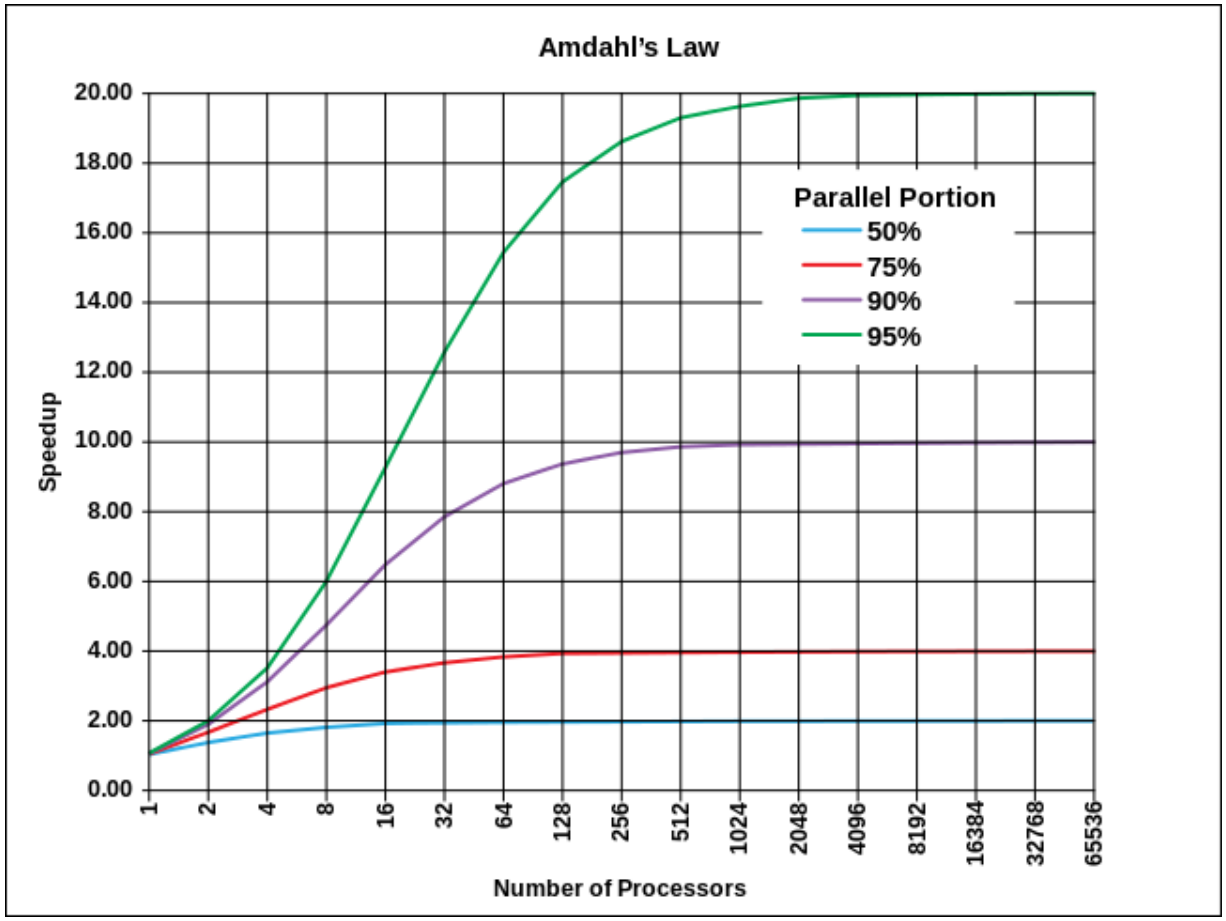
$$S(n) = \frac{1}{f + (1 - f)/n}$$

где n – число процессоров, S – ускорение, f– часть кода, не поддающаяся распараллеливанию. → часто **не стоит наращивать *n*, но стоит снижать *f***

Закон Густафсона-Барсиса (*Gustafson – Barsis's law,* **1988**: пользователи стремятся не сократить время работы текущей версии задачи, а перейти к новой версии, обеспечивающей новый уровень параллельности решения):

$$S(n) = n - f \cdot (n - 1)$$

→ **всегда стоит наращивать *n* и снижать *f***



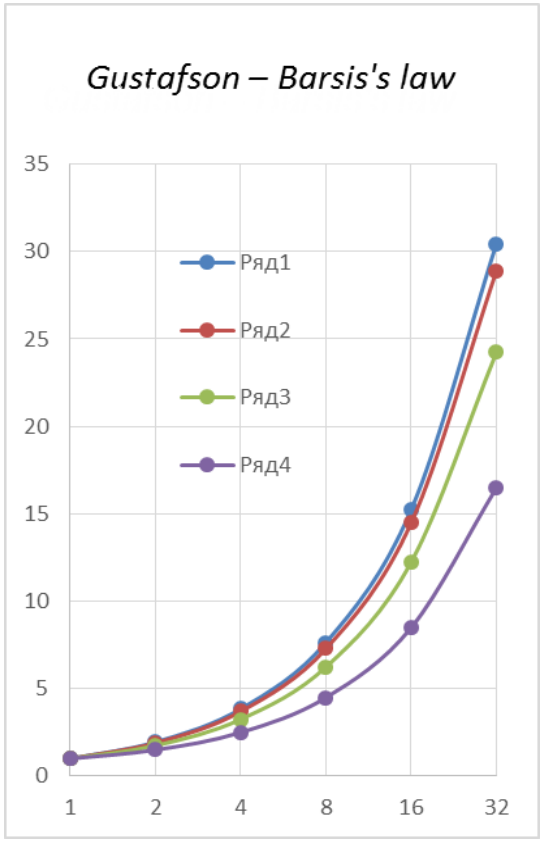
"AmdahlsLaw" by Daniels220 at English Wikipedia - Own work based on: File:AmdahlsLaw.png. Licensed under CC BY-SA 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg#mediaviewer/File:AmdahlsLaw.svg

Два закона (Амдала и Густафсона) и одно общее следствие

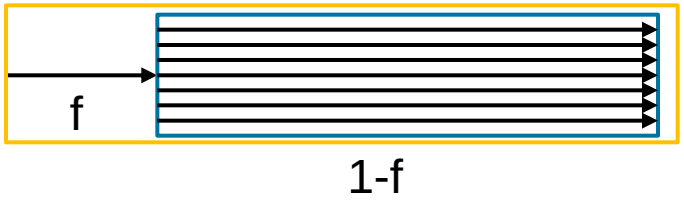
Закон Густафсона-Барсиса
(*Gustafson – Barsis's law,* 1988: перейти к новой версии, обеспечивающей новый уровень оптимизации и параллельности решения):

$$S(n) = n - f \cdot (n-1)$$

n	95%	90%	75%	50%
1	1	1	1	1
2	1.95	1.9	1.75	1.5
4	3.85	3.7	3.25	2.5
8	7.65	7.3	6.25	4.5
16	15.25	14.5	12.25	8.5
32	30.45	28.9	24.25	16.5



Почему так? Что произошло?
Ответ:
Последовательная часть f исполняется в каждом потоке

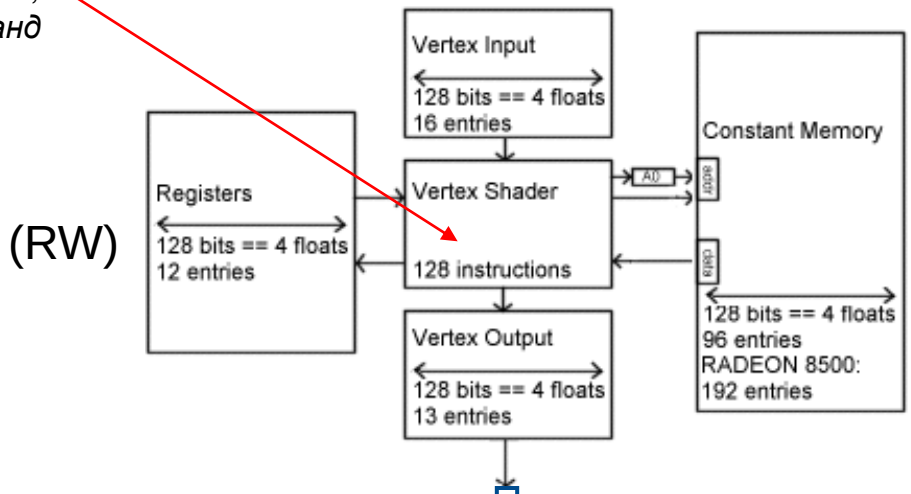


Следствие:
Всегда наряду с оптимизацией следует сводить алгоритм к массивно-параллельному (т.е. с нулевым f в смысле закона Амдала)

Архитектура GPU и параллельные вычисления

Данные уникальные *для каждой вершины* **attribute**-переменные (RO)

Окно исполнения:
короткие адреса, несколько команд в 1 регистре



(RW)

Константы и **uniform**-переменные общие для всех потоков (RO)

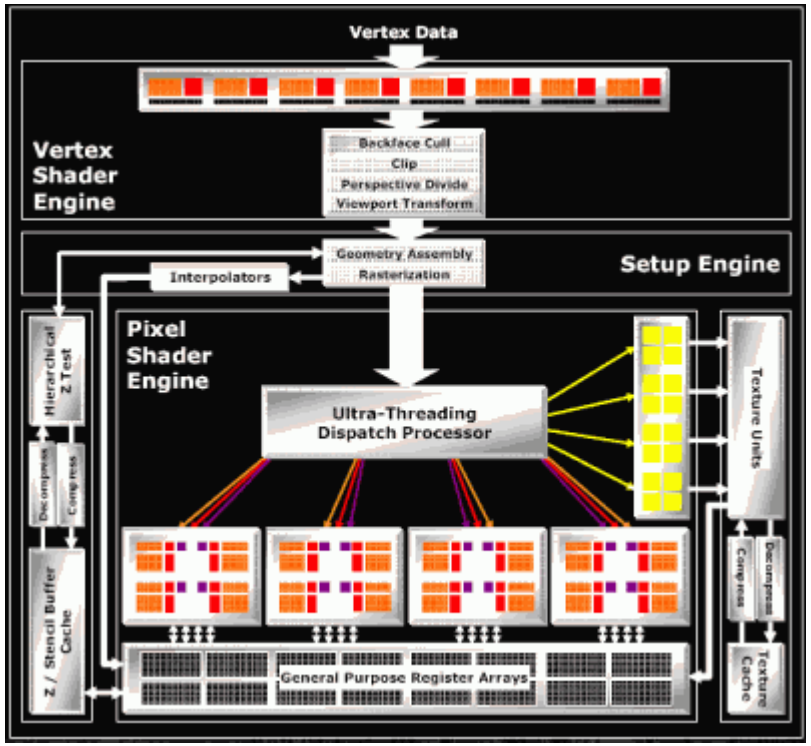
Аппаратная линейная интерполяция

varying-переменные

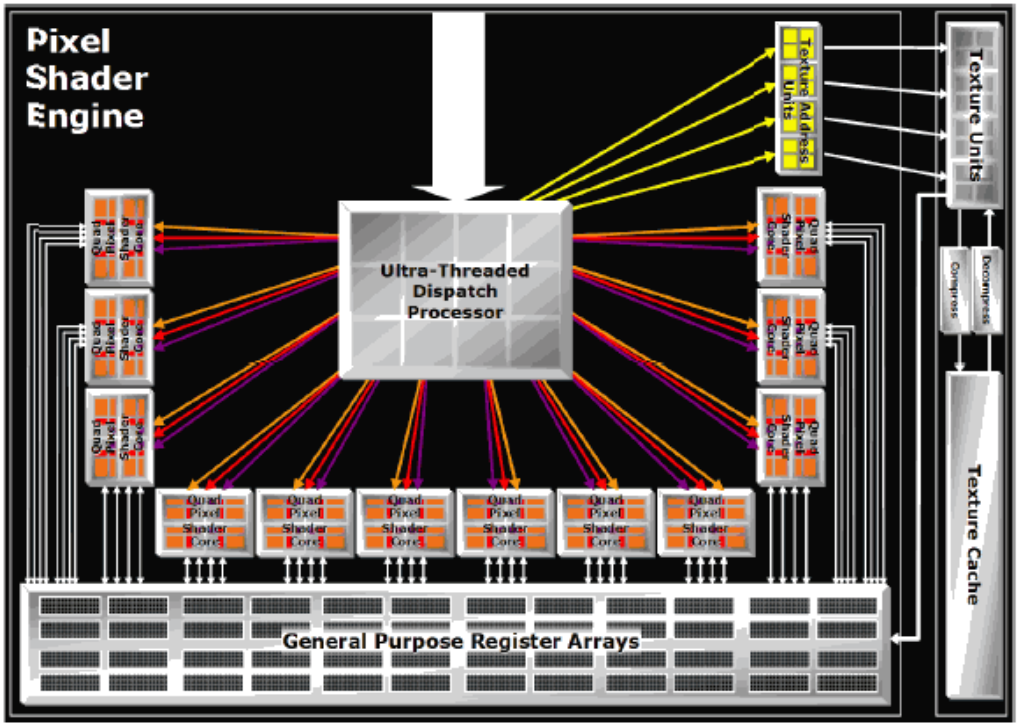
Грубая оценка веса одного потока – 32 регистра → Много потоков → Эффективный кэш

Этапы развития.
Видеокарты 3 поколения (2008)

- **ATI Radeon X1800, X1900**



16 пиксельно-шейдерных конвейеров



48 пиксельно-шейдерных конвейеров

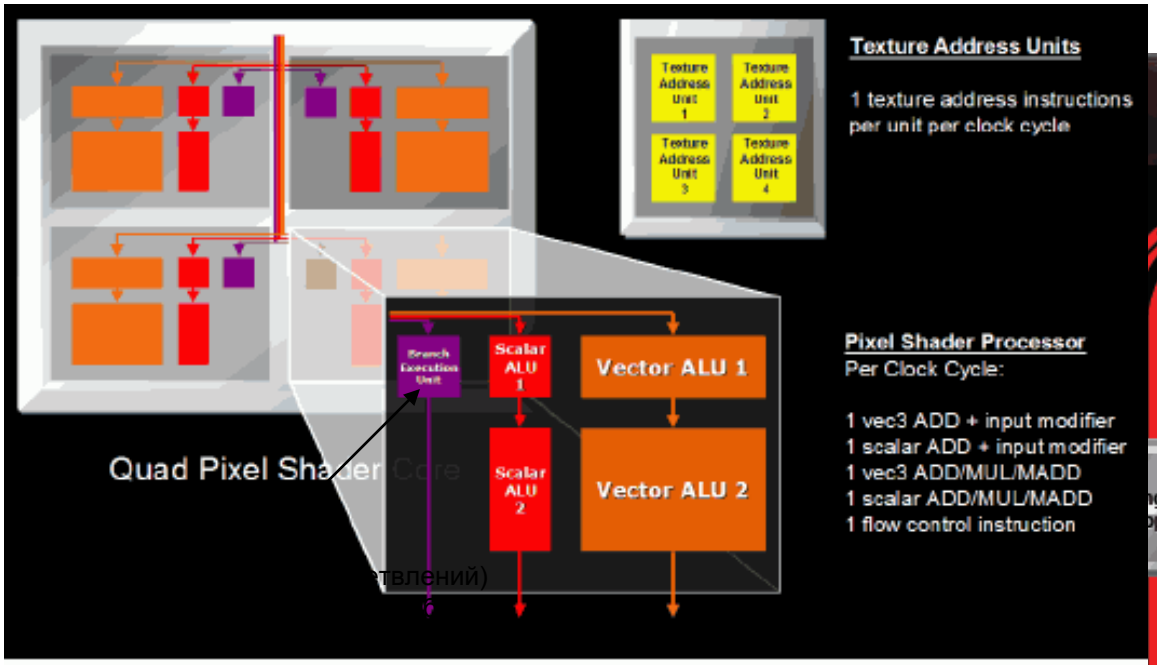
Этапы развития.
Видеокарты 3 поколения (2007)

ATI Radeon X1900

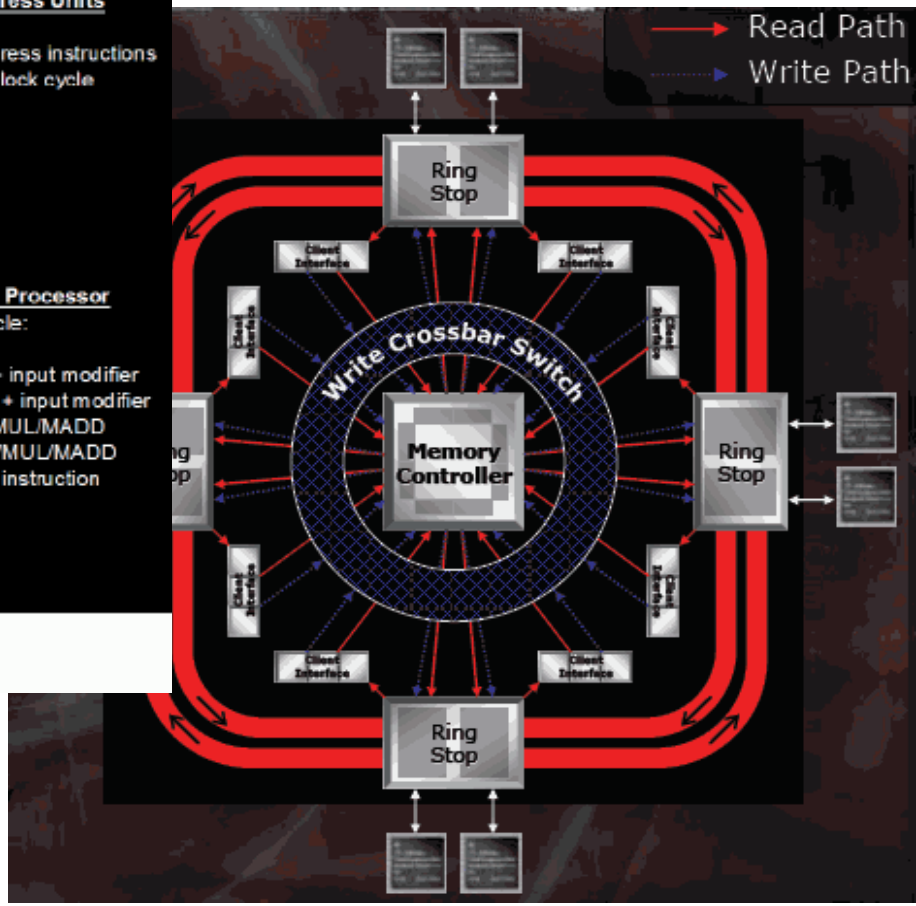
Ключевые спецификации ATI Radeon X1900:
Полноценная поддержка шины PCI-Express X16 (250-500MB/sec x 16=4-8GB/sec)
Полная поддержка Microsoft DirectX 9.0 Shader Model 3.0
Восемь вертексных геометрических процессоров
48 конвейеров обработки пиксельных шейдеров (пиксельно-шейдерных процессоров). Для сравнения: 16 - у X1800 (R520), 8 - у X1600 (RV530), 4 - у X1300 (RV515)
16 текстурных блоков (TMU, Texture Mapping Unit)
256 Мб или 512 Мб (до 1 Гб) 8-канальной графической памяти GDDR3 (с перспективой поддержки GDDR4).
Внутренняя 512-битная кольцевая шина памяти, 256-битный интерфейс (используется 8 чипов 512-битной памяти GDDR3); новый дизайн ассоциативного текстурного, цветового и Z/stencil кэшей, иерархический Z-буфер, Z-компрессия со сжатием без потерь (до 48:1), быстрая очистка Z-буфера
Декодирование всех форматов DTV/HDTV
Режимы AA - 2x/4x/6x, со сжатием без потерь при соотношении до 6:1 во всех разрешениях
Режимы анизотропной фильтрации (Anisotropic Filtering) - 2x/4x/8x/16x
Ultra-Threaded Shader Engine - многопоточная обработка данных, одновременно до 512 пиксельных тредов, полноскоростная обработка 128-данных с плавающей запятой
Улучшенный механизм предсказания ветвления и обработки тредов
До 1536 инструкций за проход
Поддержка обработки текстур с высоким разрешением (до 4k x 4k , ARGB → 64Мб))
Текстурное сжатие с 64-битной FP точностью
Высококачественная 4:1 компрессия для нормального и освещённого текстурирования (normal and luminance mapping)
Поддержка OpenGL 2.0. High Dynamic Range (HDR) рендеринг Доступность текстур из Vertex Shader

Этапы развития. Видеокарты 3 поколения (2007)

ATI Radeon X1900, Pixel Shader Processor



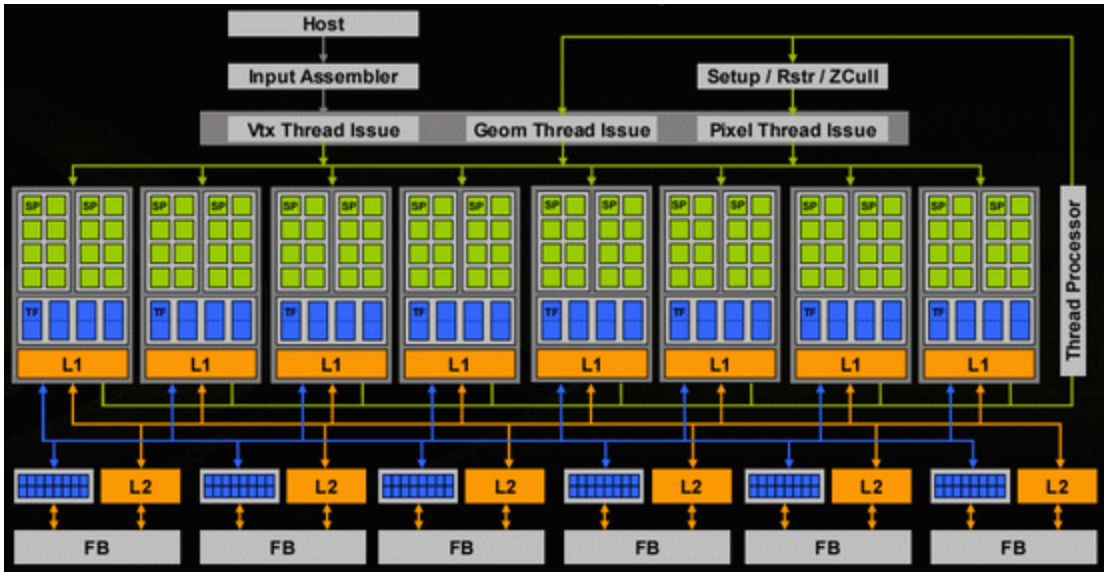
Radeon X1900 Pixel Shader Processor Detail



Кольцевая архитектура контроллера памяти с 2 шинами по 256 = 8 x 32 бит

Суть технологии Fetch4 Текстурные блоки спроектированы для одновременного моделирования и фильтрации четырёх текселов адреса (для 16 текстур). За счет технологии быстрого управления потоком данных Ultra-Threading и функции выбора текстур Fetch4 чипы Radeon X1900 могут обеспечить вывод мягких оттенков быстрее, нежели традиционная технология теневого текстурирования.

GPU с шейдерами 4 поколения.
Архитектура NVIDIA GeForce 8800 GTX

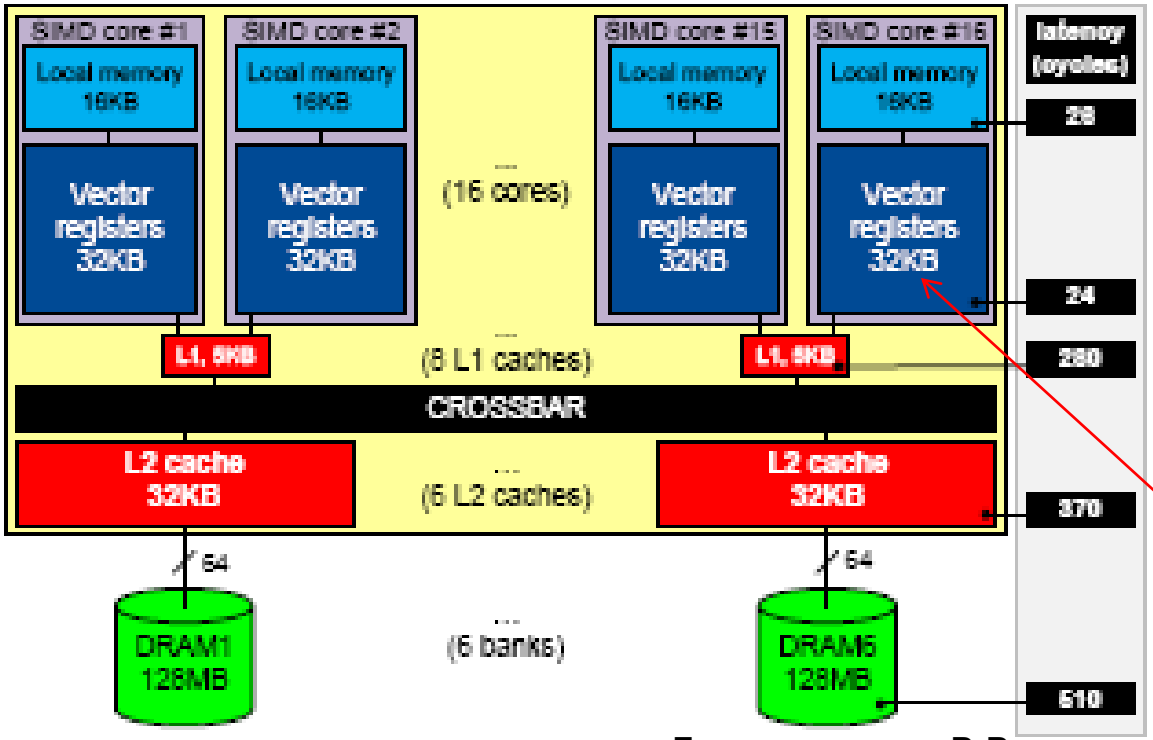


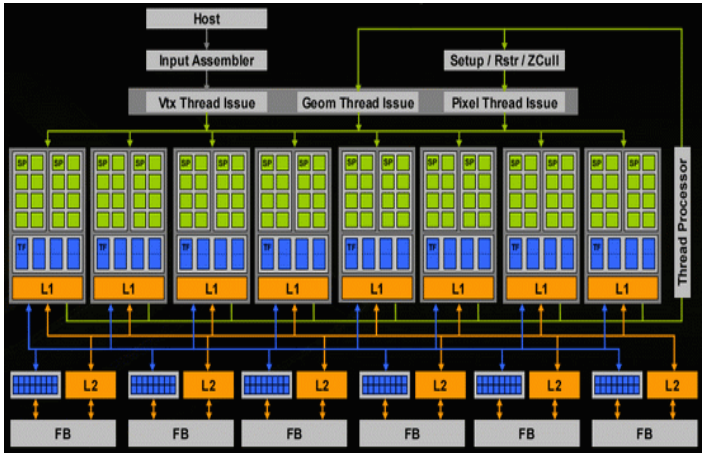
NVIDIA GeForce 8800 GTX **имеет: 128 скалярных потоковых процессоров (scalar processors), которые сгруппированы в 8 блоков (мультипроцессоров) по 16 процессоров, каждый блок должен быть оснащен 4 текстурными модулями и общей кэш-памятью L1.**

Vtx (Geom|Pixel) Tread Issue – исток для вершинных (геометрических / пиксельных шейдеров)

Каждый блок делится на два шейдерных процессора (каждый из которых состоит из 8 потоковых процессоров), имеющих SIMD архитектуру. И все восемь блоков имеют доступ к любому из шести L2 кэшей и к любому из шести массивов регистров общего назначения (РОН). Таким образом, данные обработанные на одном шейдерном процессоре могут быть использованы другим шэйдерным процессором.

GPU. SIMD Memory Access
On-Chip Memory Hierarchy. Latency

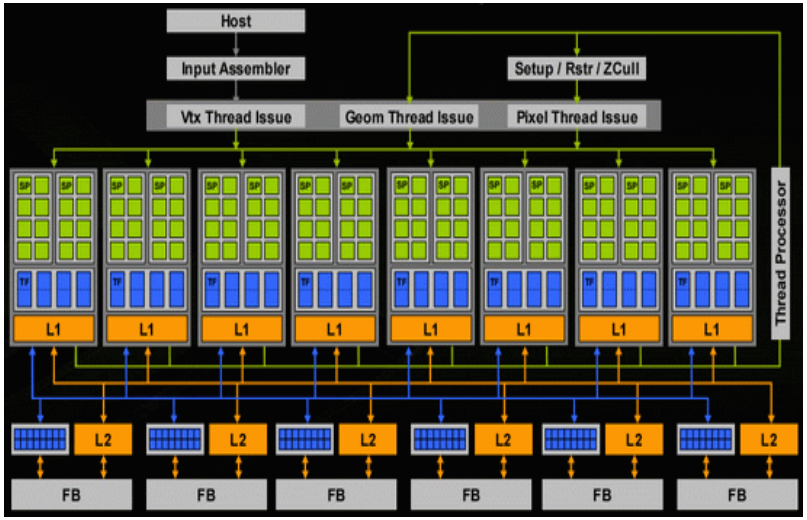




потоки

Латентности по В.Волкову

GPU с шейдерами 4 поколения.
Архитектура NVIDIA GeForce 8800 GTX

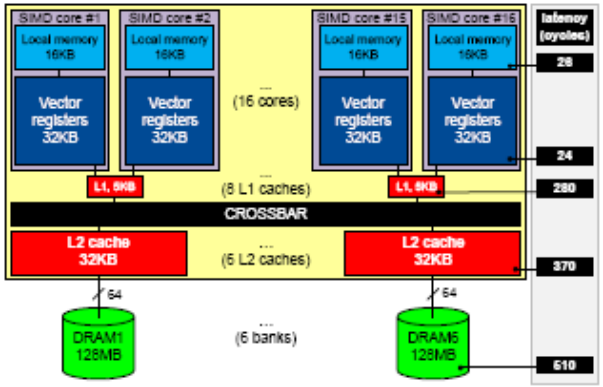


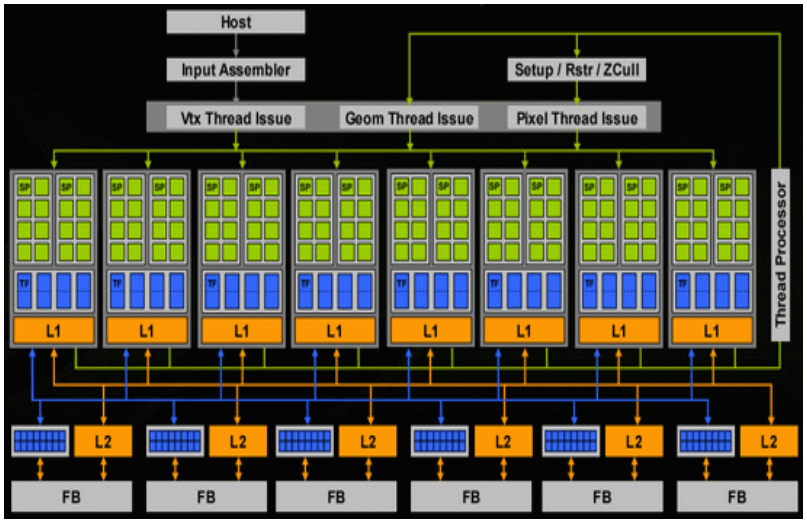
Имеется одно устройство выборки/распаковки на каждый мультипроцессор. Оно использует 4 цикла для исполнения одной инструкции для всего warp-а [NVIDIA 2007, Ch. 5.1.1.1]. Т.о., **мультипроцессор есть SIMD устройство** со скалярными процессорами эффективно используемыми как SIMD линейки. Мы обращаемся к ним как к SIMD ядрам для их размещения в контексте других современных многоядерных систем, таких как Core2 и Cell: одно ядро GPU имеет 19–23 Gflop/s для multiply-and-add операций, что подобно пиковой производительности CPU ядер (~21.3 Gflop/s/core для 2.66GHz Intel Core2) или SPE устройств Cell процессора (25.6 Gflop/s per SPE).

Другим важным свойством GPU явл. многопотоковость, что позволяет скрыть латентности памяти и конвейера. Для облегчения низкой стоимости переключения в контексте, все одновременно выполняемые потоки держат свои регистры состояния в одном файле регистров. Число регистров захватываемое потоком зависит от программы.

Имеется небольшая локальная разделяемая память на каждый потоковый блок. Пользователь может создать больше потоков, чем могут одновременно вместить регистры и локальная память. Тогда некоторые потоки не инициируются пока другие не закончат (что не сулит выгод в производительности).

GPU. SIMD Memory Access On-Chip Memory Hierarchy





•GPU производит высокоскоростную безкэшевую загрузку SIMD-памяти и сохранение. Non-cached non-SIMD memory operations также поддерживаются, но они менее эффективны по пропускной способности.

•GPU также производит выборки из кэшированной памяти. Это не требует организации SIMD паттернов для лучшей производительности. Однако, кэшированный доступ требует высокой пространственной локализованности внутри каждой векторной сборки из-за малого размера кэша. Все эти типы доступа к памяти представлены в ISA как индексированные слияния (сборки) и индексированные разборки. Это подразумевает использование лишних индексов в случае SIMD доступа.

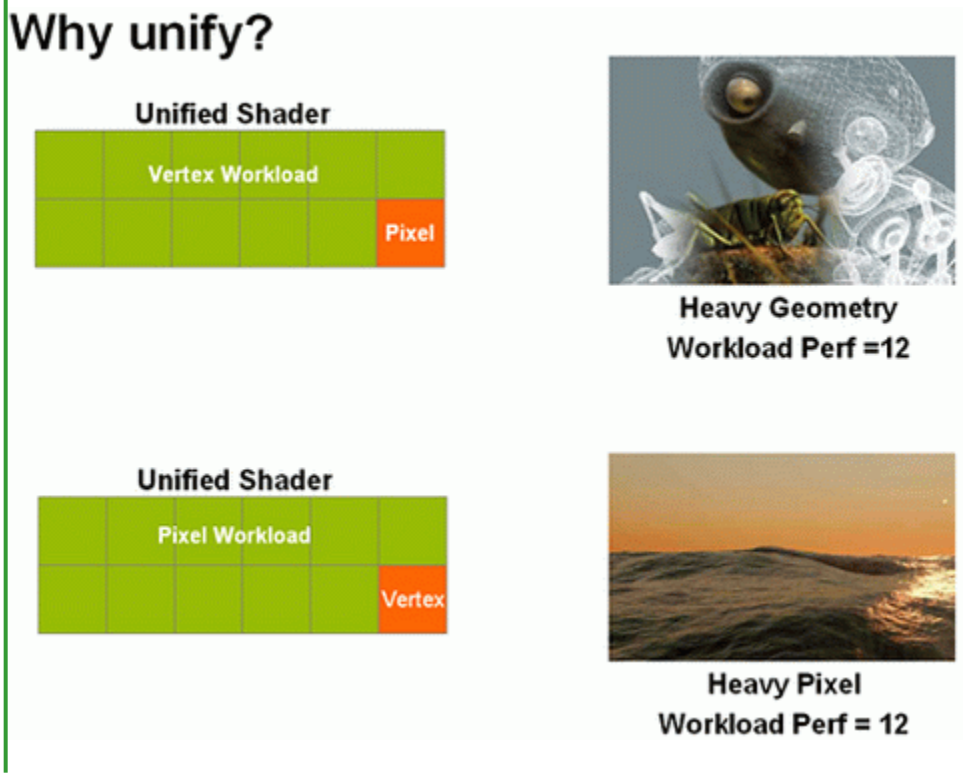
Каждое SIMD ядро имеет 32KB register file разделенный между SIMD линейками. Для GeForce 8800 GTX это составляет до 512KB на один чип, что больше, чем любой другой частный уровень иерархии памяти на чипе. Это мотивирует различные другие решения, используемые в суперскалярных процессорах, которые имеют относительно небольшое число регистров (e.g. 128−256 bytes in SSE units) и массивные кэши.

Второй более высокий уровень on-chip memory hierarchy – это локальная память (16KB per SIMD ядро) и 256 KB всего. Это может быть эффективно использовано как скалярные регистры и также разрешает индексированный досуп. Другие важные типы on-chip memories - L2 и L1 read-only кэши, объемом до 192KB и 40KB на 8800GTX для нашего исследования.

Характеристики GPU nVidia G80

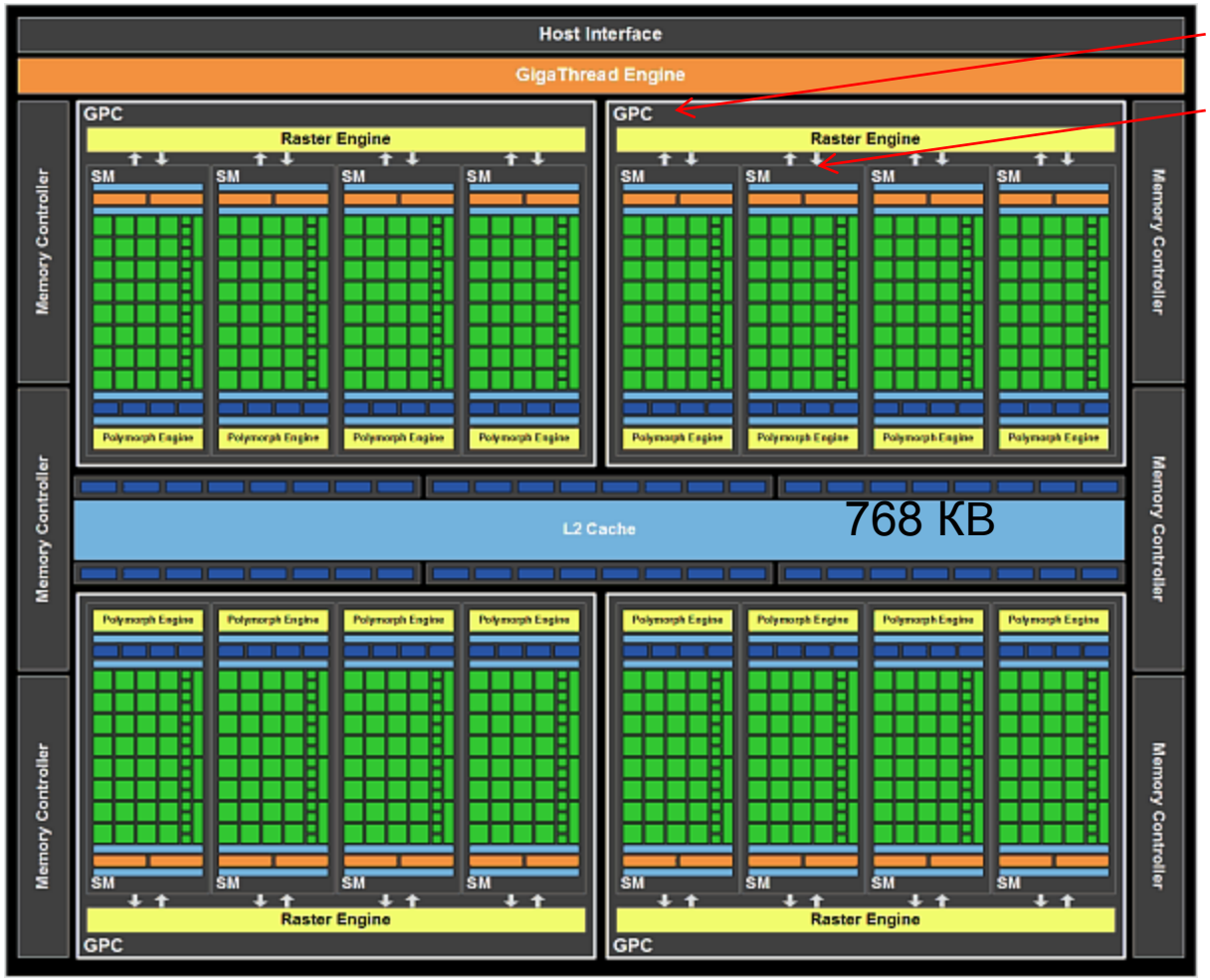
GPU name	GeForce 8800GTX	Quadro FX5600	GeForce 8800GTS	GeForce 8600GTS
# of SIMD cores	16	16	12	4
core clock, GHz	1.35	1.35	1.188	1.458
peak Gflop/s	346	346	228	93.3
peak Gflop/s/core	21.6	21.6	19.0	23.3
memory bus, MHz	900	800	800	1000
memory bus, pins	384	384	320	128
bandwidth, GB/s	86	77	64	32
memory size, MB	768	1535	640	256
flops:word	16	18	14	12

Эти процессоры имеют широкие возможности к гибкому программированию, описанные в руководстве CUDA (CUDA programming guide).
Однако, использование этой гибкости может стоить 10-100x потерь в производительности. GPU представлен как многопотоковая SIMD (SIMT) архитектура.



Этапы развития. 5 поколение

NVIDIA Fermi GF-100 architecture



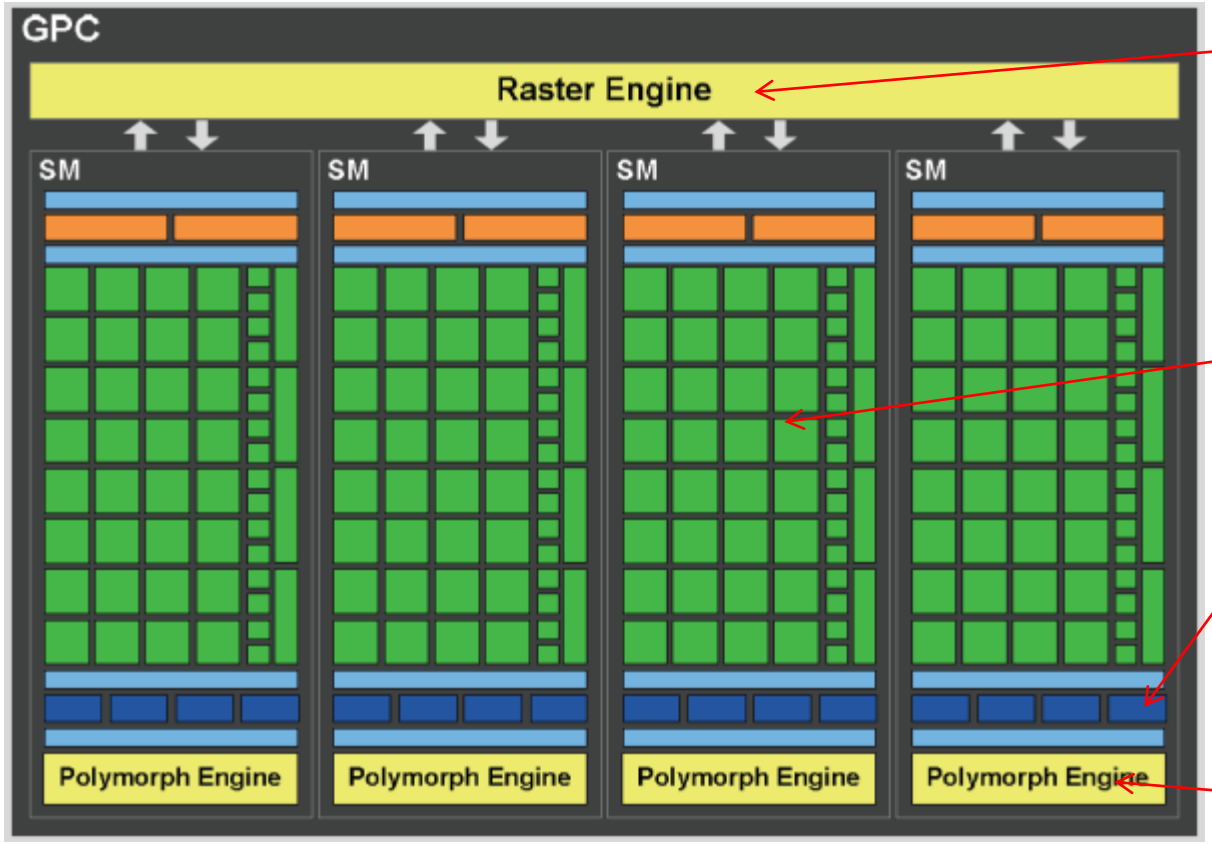
Graphics Processing Clusters (GPC) - 4

Streaming Multiprocessors – 4 / GPC

GigaThread Engine is the center of the chip, it generates and distributes blocks of threads between different multiprocessors.

Multiprocessor distributes warps (warp, a group of 32 threads) between stream processors (CUDA cores, **Compute Unified Device Architecture**), and other execution units.

Graphics Processing Clusters (GPC) architecture



Own Raster Engine on GPC
(triangle setup, rasterization, z-cull)

Streaming Processors – 32 / SM

Texture Processing Blocks – 4/ SM

PolyMorph Engine - 1/ SM
(Vertex attribute fetch, tessellation)

Graphics Processing Clusters architecture

Warp Scheduler + Instruction Dispatch Unit
2 Warp Scheduler → 2 Warps simultaneously

IEEE 754-2008
Standard Calc.

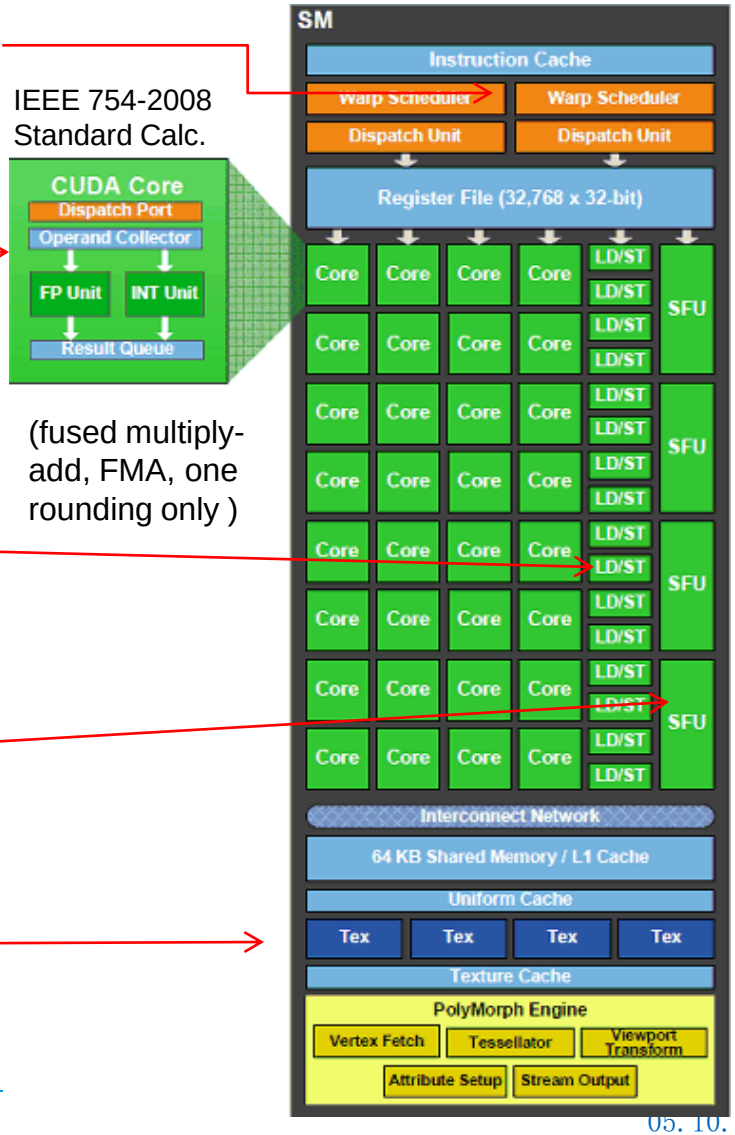
Streaming CUDA core (processors) – 32 / SM

(fused multiply-add, FMA, one rounding only)

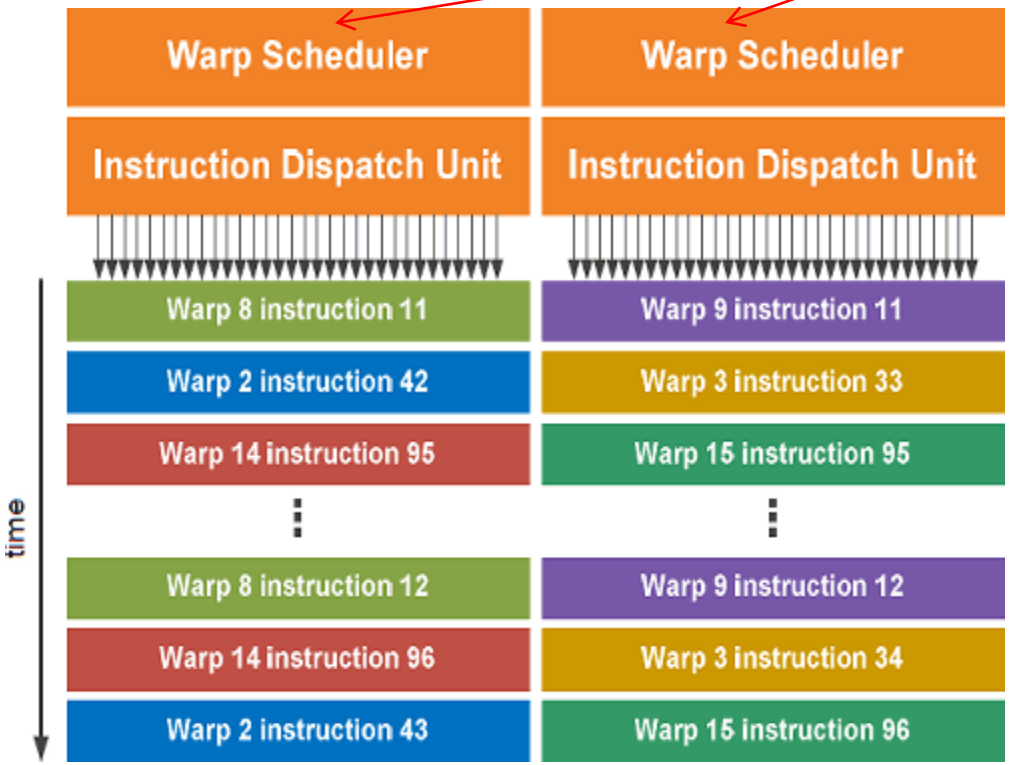
Load/store unit, LD/ST or LSU - 16/ SM
(16 threads per clock)

Special Function Units, SFU – 4/SM
(sin, cos, square root, interpolation)

Texture Processing Blocks – 4/ SM



Double Warp Scheduler on SM



2 Warp Scheduler + Instruction Dispatch Unit
→ 2 Warps simultaneously

VLIW instructions dispatching

One instruction of the warp on 16 core simultaneously (SM/ 2)

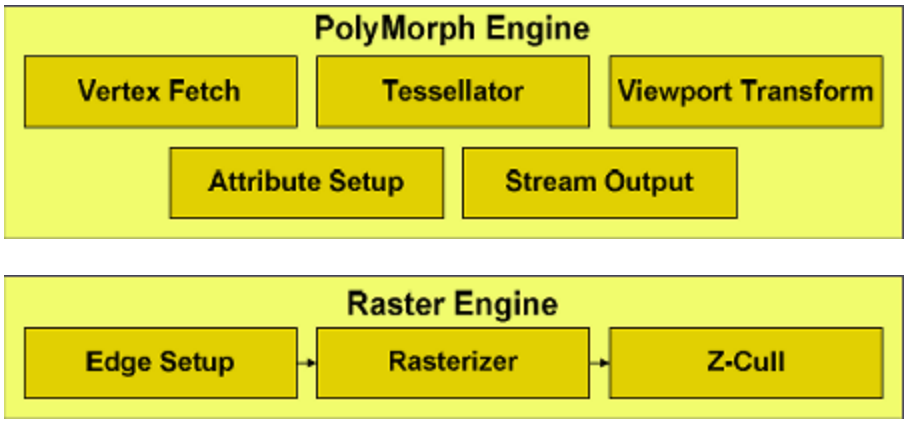
Fermi Memory Hierarchy

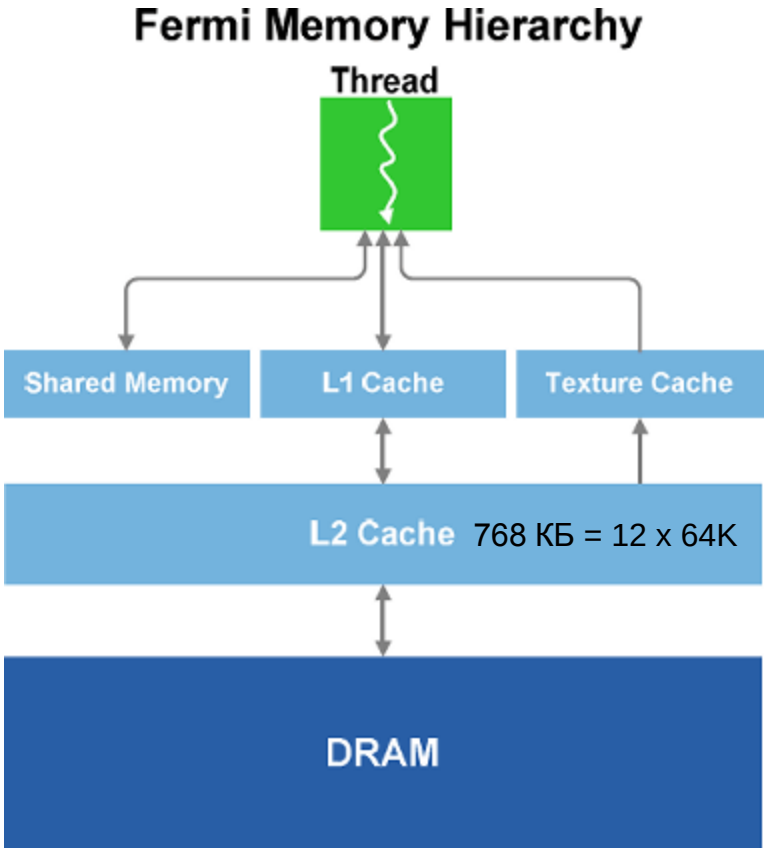
Parallel Geometry Processing & Raster Engine

Clock rate (MHz)
Core:772; Shader:1544; Memory: 4008

SM of GF100 has 64KB on-chip memory
in 2 possible configurations:
•48 shared memory/ 16 L1 Cash - Graphics
•16 shared memory/ 48 L1 Cash - GPGPU

Bandwidth (GB/s): 192,4
6 Memory Controllers





Additional Information:
GeForce; GeForce 500 Series
NVIDIA Fermi GF-100 architecture

Этапы развития. 6 поколение

NVIDIA Kepler GK-110 architecture

NVIDIA **Kepler** GK110 **Architecture** Whitepaper.docx (local)

Streaming Multiprocessors – 4 / GPC

GigaThread Engine is the center of the chip, it generates and distributes blocks of threads between different multiprocessors.

Multiprocessor distributes warps (warp, a group of 32 threads) between stream processors (CUDA cores), and other execution units.

Этапы развития. 6 поколение

NVIDIA Kepler GK-110 architecture

NVIDIA **Kepler** GK110 **Architecture** Whitepaper.docx (local)

SMX Processing Core Architecture:
192 single-precision CUDA cores,
64 double-precision units,
32 special function units (SFU)
32 load/store units (LD/ST).

•Each of the Kepler GK110 SMX units feature 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic logic units.

•Kepler retains the full IEEE 754-2008 compliant single- and double-precision arithmetic introduced in Fermi, including the fused multiply-add (FMA) operation.

•One of the design goals for the Kepler GK110 SMX was to significantly increase the GPU's delivered double precision performance, since double precision arithmetic is at the heart of many HPC applications.

•Kepler GK110's SMX also retains the special function units (SFUs) for fast approximate transcendental operations as in previous-generation GPUs, providing 8x the number of SFUs of the Fermi GF110 SM.

Этапы развития. 6 поколение

NVIDIA Kepler GK-110 architecture

NVIDIA **Kepler** GK110 **Architecture** Whitepaper.docx (local)

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

Источники:

1. Торн А. Графика в формате DirectX 9. Полное руководство по использованию 3D-пространства: Пер.с англ. - М.: НТ-пресс, 2007. - 288с.
2. Евченко Александр. OpenGL и DirectX. Программирование графики. – СПб: Питер, 2006. -350с. (+ CD-ROM)
3. Хилл Ф. OpenGL. Программирование компьютерной графики. – С.Пб: Питер, 2002. 1088с.
4. Поляков А.Ю., Брусенцев В.А. Программирование графики: GDI+ и DirectX. – СПб.: БХВ-Петербург, 2005. -368с. (Visual C++,.NET, прилагается CD).
5. Гайдуков С.А. OpenGL. Профессиональное программирование трехмерной графики на C++. СПб.: БХВ-Петербург, 2004. -736с.(GLUT, NVidia SDK, ATI SDK, прилагается CD)
6. Верма Р.Д. Введение в OpenGL. – М.: Горячая линия – Телеком, 2004. - 303с.
7. Allison Klein (Senior Lead Program Manager MS Direct3D). Introduction to the Direct3D 11 Graphics Pipeline (ppt)