Case Study: NVIDIA GeForce 3 Series

gf3-1

Overview

Early programmable GPU.

Available 2001, discontinued.

Specifications (GeForce3 Ti 500)

Memory: 64 MiB

Bandwidth: 8 GB/s.

Programmable vertex processor (shader).

References

gf3-2

Description of GeForce 3 Vertex Processor Microarchitecture

Good technical description in top-tier graphics conference.

Erik Lindholm, Mark J. Kilgard, Henry Moreton, "A User-Programmable Vertex Engine," SIGGRAPH 2001, p.149-

Product Overview

Manufacturers product description page, `http://www.nvidia.com/page/geforce3.html`

Slides describing GeForce3 with good coverage of instruction set.

Michael McCool, Mauro Steigleder, "Graphics Accelerators: State of the Art: NVIDIAs GeForce3", `http://www.cgl.uwaterloo.ca/Projects/rendering/Talks/StateArt2.ppt`

gf3-3

Specification of Vertex Processor API

Ostensibly, an API for programming, not the true set of machine instructions...
... however Lindholm 2001 strongly implies it is close to true instruction set.

NV_Vertex_Program specification,
`http://www.ece.lsu.edu/gp/refs/nv-vertex-program.txt`

gf3-4

GeForce3 Major Units

Command and Data Fetch

Vertex Processor

- Single Unit
- Programmable
- This unit described in detail here.

Primitive Assembly Setup

Texture Shader

- Four Units
- An important unit, but not covered in detail until good reference found.

Z-Test, Blend, Frame Buffer Update

gf3-5

Operating Modes

Render Mode:
GPU processing vertices as vertex attributes arrive from CPU.

In render mode when processing string of `glVertex` OpenGL commands.

Setup Mode:
GPU changing state (configuration) in response to non-vertex data from CPU.

Setup might be needed for change of:

Transformation matrices.

Vertex program.

Lighting parameters.

Preliminaries: Quad Data Type

gf3-6

Quad Data Type

Just one data type, the **quad**.

Quad:
Set of four 32-bit FP numbers in IEEE 754 format, so total size is 128 bits.

Format follows IEEE 754 standard but arithmetic does not:

Many arithmetic operations not done to full precision.

No arithmetic exceptions.

Just one rounding mode (not four).

$0 \times x = 0 \quad \forall x$, (including non-numbers)

No integer type (with one special-purpose exception).

gf3-7

Data Type Rationale

Thirty-two bits sufficient for graphics.

Many graphics operations one 4-element vectors, including homogeneous coordinates and RGBA data.

True IEEE 754 arithmetic adds to cost but not to value (at least before GPGPU applications).

Preliminaries: Swizzling

gf3-8

Swizzling (Vector Element Rearrangement and Duplication)

Swizzle:

To rearrange or duplicate elements of a vector. For example, $(1, 2, 3, 4)$ can be swizzled to $(4, 2, 2, 3)$.

Swizzle Notation

Let `R1` be the name of something that stores a quad.

The symbols `x`, `y`, `z`, and `w` denote the four elements (`x` is first element, etc.).

Name followed by four letters (*e.g.*, `R1.zyxx`), rearrange as shown. *E.g.*, for `R1.zyxx`: $(1, 2, 3, 4) \longrightarrow (3, 2, 1, 1)$. (Note duplication of x.)

Vertex Assembly Notation: One letter (*e.g.*, `R0.y`): duplicate, equivalent to `R0.yyyy`. *E.g.*, $(1, 2, 3, 4) \longrightarrow (2, 2, 2, 2)$.

GL Shader Language Notation: Name followed by $x \in [1, 4]$ letters: vector of length x swizzled as shown. *E.g.*, let `R1` $= (1, 2, 3, 4)$; then `R1.y` $= (2)$ (note difference with vertex assembly notation).

Preliminaries: Vertex Attribute

gf3-9

GeForce 3 Vertex Attribute:

One of 16 quads describing some aspect of a vertex.

Attributes are numbered and each has a specific meaning.

Attribute 0 is the vertex coordinate, attribute 2 is normal, etc.

Attribute numbers are exposed to the APIs (OpenGL, Direct3D).

Attributes number used as register number in several places.

gf3-10

Unit: Command and Data Fetch

In rendering mode, reads attributes from CPU.

Data from CPU in variety of formats (8-bit integer, 32-bit float, etc.) ...
... and may not be full 4-element vectors.

Unit coverts data to quads and writes to Vertex Attribute Buffer.

Missing array elements are initialized to 0 or 1.

Vertex Attribute Buffer (VAB):
Set of 16 quad registers, each register corresponds to a vertex attribute.

Hardware implementation of command / data fetch unit not described.

gf3-10

Vertex Processor Overview

Purpose: Apply transform & lighting computations.

Operation: Read data from VAB, write to OB.

Implemented as very simple microprogrammed processor.

VP Registers

Input Buffer (implements, Vertex Attribute Registers):
A set of 16 quad registers holding vertex attributes, these registers are read-only by vertex processor. Each vertex processor has several input buffers.

Number of input buffers not available.

The number might have been chosen to match operation latency.

Constant Registers (implements, Program Parameter Registers):
A set of 96 quad registers that are read only by vertex processor.

Constant registers do not change from vertex to vertex.

They hold data such as transformation matrices and lighting parameters.

VP Registers

gf3-13

Temporary Registers:

A set of 12 quad registers that can be read or written by vertex processor.

Address Register:

Effectively a single 32-bit integer register, but defined as a four-element vector of 32-bit integers. Can only be written by one instruction, `ARL`. Value can only be used for indexed addressing of constant (parameter) registers.

Output Buffer (implements Vertex Result Registers):

A set of 16 quad registers that are write only. Each VP has multiple output buffers.

gf3-14

Vertex Attribute (Input Buffer) Register Names and Purpose (Table X.2)

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

gf3-14

gf3-15

Vertex Result (Output Buffer) Register Names and Purpose (Table X.1)

Vertex Result Register Name	Description	Component Interpretation
HPOS	Homogeneous clip space position	(x,y,z,w)
COL0	Primary color (front-facing)	(r,g,b,a)
COL1	Secondary color (front-facing)	(r,g,b,a)
BFC0	Back-facing primary color	(r,g,b,a)
BFC1	Back-facing secondary color	(r,g,b,a)
FOGC	Fog coordinate	(f,*,*,*)
PSIZ	Point size	(p,*,*,*)
TEX0	Texture coordinate set 0	(s,t,r,q)
TEX1	Texture coordinate set 1	(s,t,r,q)
TEX2	Texture coordinate set 2	(s,t,r,q)
TEX3	Texture coordinate set 3	(s,t,r,q)
TEX4	Texture coordinate set 4	(s,t,r,q)
TEX5	Texture coordinate set 5	(s,t,r,q)
TEX6	Texture coordinate set 6	(s,t,r,q)
TEX7	Texture coordinate set 7	(s,t,r,q)

gf3-15

Vertex Attribute Buffer and Input Buffer

gf3-16

Vertex attribute buffer (VAB) to input buffer (IB) transfer.

Data automatically copied from VAB to IB.

Transfer is triggered by a write to VAB attribute 0 (vertex position).

The 16 VAB registers are copied to the 16 registers of one of the IBs.

IB chosen in round-robin fashion.

Dirty bits used to avoid copying data that's unchanged.

Note automatic triggering of copy by write of attribute 0.

VP Instruction Set Architecture

gf3-17

Instruction Sets

True Instruction Set

Instructions recognized by vertex processor hardware.

These are not documented . . .
. . . but are likely some kind of microinstructions.

Exposed Instruction Set

Instructions recognized by API calls.

Documented in OpenGL NV_Vertex_Program specification.

Lindholm 2001 implies close match to true instruction set.

NVIDIA-provided software translates exposed instruction set to true one.

Description here is of exposed instruction set.

VP Instruction Set Architecture

gf3-18

Register Name Assembly Syntax

Based on output of NVIDIA compiler.

Input Buffer (Vertex Attribute) Register Names:

vertex_program notation: `v[0]-v[15]` or `v[OPOS]-v[TEX7]`.

NVIDIA compiler: `vertex.position`, `vertex.normal`, etc.

Constant Register Names: `c[0]-c[95]`.

Temporary Register Names: `R0-R11`.

Output Buffer Register Names:

vertex_program notation: `o[0]-o[15]`.

NVIDIA compiler: `result.position`, `result.color`, etc.

Example:

```
MAD result.position, vertex.position.w, c[14], R0;
```

Instruction Sources

Instructions can have up to three register source operands:

```
MAD R1, R2, R3, R4;
```

Any source operand can read IB, temporary, or constant registers:

```
ADD R1, R2, R3 (Read temporary.)
ADD R1, R2, c[3] (Read constant.)
ADD R1, R2, vertex.position (Read input buffer.)
```

Any source operand can be arbitrarily swizzled:

```
ADD R1, R2.x, R3.wzyx (Reverse order of last operand's components.)
```

Any source operand can be negated:

```
ADD R0.y, R0, -R0.z;
```

Constant register can be indexed using **address** (not memory) register, `A0`:

```
ADD R1, -R2, c[A0];
```

There are no immediates (instead, place constant in constant register).

VP Instruction Set Architecture

gf3-20

Instruction Destinations

Any instruction can write temporary and output buffer registers.

Un-exposed instructions may be able to write constant memory.

Write can target any subset of components:

```
DP3 R0.x, R0, R1; (Leave R0's y, z, and w unchanged.)
```


gf3-20

VP Instruction Set Architecture

gf3-21

Complete Instruction Set

From 2.14.1.9:

Opcode	Inputs (scalar or vector)	Output (vector or replicated scalar)	Operation
ARL	s	address register	address register load
MOV	v	v	move
MUL	v,v	v	multiply
ADD	v,v	v	add
MAD	v,v,v	v	multiply and add
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DST	v,v	v	distance vector
MIN	v,v	v	minimum
MAX	v,v	v	maximum
SLT	v,v	v	set on less than
SGE	v,v	v	set on greater equal than
EXP	s	v	exponential base 2
LOG	s	v	logarithm base 2
LIT	v	v	light coefficients

Instruction Descriptions

gf3-22

Selected instructions described below.

For descriptions of all instructions see vertex_program Section 2.14.1.10.

gf3-23

Instruction: RCP destination, source0

Reciprocal

```
t.x = source0.c;
if (negate0) {t.x = -t.x;}
if (t.x == 1.0f) {u.x = 1.0f;} else {u.x = 1.0f / t.x;}
if (xmask) destination.x = u.x;
if (ymask) destination.y = u.x;
if (zmask) destination.z = u.x;
if (wmask) destination.w = u.x;
```

Precision: $\texttt{u.x} - \texttt{IEEE}(\texttt{1.0}/\texttt{t.x}) < 2^{-22}$.

EE 7700-1 Lecture Transparency. Formatted 10:08, 12 March 2010 from set-study-gf3.
gf3-23

gf3-24

Instruction: EXP destination, source0

Exponential Base 2

```
t.x = source0.c;
if (negate0) {t.x = -t.x;}
q.x = 2^floor(t.x);
q.y = t.x - floor(t.x);
q.z = q.x * APPX(q.y);   // Approximation of 2^q.y
if (xmask) destination.x = q.x;
if (ymask) destination.y = q.y;
if (zmask) destination.z = q.z;
if (wmask) destination.w = 1.0;
```

x component holds approximate result, y and z hold values needed to compute exact result.

VP ISA: Sample Code

gf3-25

Vertex transformation only (no lighting).

Source Code (OpenGL Shader Language):

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

vertex_program Assembler Code (Output of NVIDIA compiler)

```
PARAM c[5] = { program.local[0],
                state.matrix.mvp.transpose };
TEMP R0;
MUL R0, vertex.position.y, c[2];
MAD R0, vertex.position.x, c[1], R0;
MAD R0, vertex.position.z, c[3], R0;
MAD result.position, vertex.position.w, c[4], R0;
END
# 4 instructions, 1 R-regs
```

gf3-26

Transformation and Lighting

Source Code (OpenGL Shader Language):

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
vec3 norm_e = gl_NormalMatrix * gl_Normal;
vec4 light_pos = gl_LightSource[1].position;
float phase_light = dot(norm_e, normalize(light_pos - vertex_e).xyz);
float phase_user = dot(norm_e, -vertex_e.xyz);
float phase = sign(phase_light) == sign(phase_user) ? abs(phase_light) : 0.0;
const vec3 ambient = gl_LightSource[1].ambient.rgb;
const vec3 diffuse = gl_LightSource[1].diffuse.rgb;
vec4 new_color;
new_color.rgb = gl_Color.rgb * ( phase * diffuse + ambient );
new_color.a = gl_Color.a;
gl_FrontColor = new_color;
gl_BackColor = gl_Color;
```

gf3-27

vertex_program Assembler Code (Output of NVIDIA compiler)

```
PARAM c[15] = { { 0 },
                state.matrix.modelview.transpose,
                state.matrix.modelview.inverse.row[0..2],
                state.light[1].ambient,
                state.light[1].diffuse,
                state.light[1].position,
                state.matrix.mvp.transpose };
TEMP R0; TEMP R1; TEMP R2;
MUL R0, vertex.position.y, c[2];
MAD R0, vertex.position.x, c[1], R0;
MAD R0, vertex.position.z, c[3], R0;
MAD R2, vertex.position.w, c[4], R0;
ADD R1, -R2, c[10];
DP4 R0.w, R1, R1;
RSQ R0.w, R0.w;
MUL R0.xyz, vertex.normal.y, c[6];
MAD R0.xyz, vertex.normal.x, c[5], R0;
MAD R0.xyz, vertex.normal.z, c[7], R0;
MUL R1.xyz, R0.w, R1;
DP3 R0.w, R0, -R2;
DP3 R0.x, R0, R1;
```

gf3-28

```
SLT R0.y, R0.w, c[0].x;
SLT R0.z, c[0].x, R0.w;
ADD R0.w, R0.z, -R0.y;
SLT R0.z, R0.x, c[0].x;
SLT R0.y, c[0].x, R0.x;
ADD R0.y, R0, -R0.z;
ADD R0.y, R0, -R0.w;
ABS R0.y, R0;
SGE R0.y, c[0].x, R0;
ABS R0.y, R0;
ABS R0.x, R0;
SGE R0.y, c[0].x, R0;
MAD R1.x, -R0, R0.y, R0;
MUL R0, vertex.position.y, c[12];
MUL R1.xyz, R1.x, c[9];
MAD R0, vertex.position.x, c[11], R0;
ADD R1.xyz, R1, c[8];
MAD R0, vertex.position.z, c[13], R0;
MUL result.color.xyz, vertex.color, R1;
MAD result.position, vertex.position.w, c[14], R0;
MOV result.color.back, vertex.color;
MOV result.color.w, vertex.color;
END
```


gf3-28

gf3-29

```
# 35 instructions, 3 R-regs
```


gf3-29

VP Instruction Set Architecture

gf3-30

Instruction Set Design Choices

Based on analysis of fixed-functionality vertex processing code:

Used about 50% of time: MOV, MUL, ADD, MAD

Used about 40% of time: DP3, DP4.

RCP: Instead of divide because it's faster.

RSQ: Within 1.5 bits of IEEE precision.

VP Microarchitecture

gf3-31

Register sets listed above.

Instruction memory has room for 128 instructions.

Executes at rate of one instruction per cycle.

200 MHz clock.

Two functional units.

gf3-32

Functional Units:

Two exposed functional units (SIMD, Special).

SIMD Vector Unit

Three source operands.

MOV, MUL, ADD, MAD, DP3, DP4, DST, MIN, MAX, SLT, SGE

Special Functional Unit

Single source operand.

RCP, RSQ, LOG, EXP, LIT

Possible additional units for fixed-function use.

All instructions have **same latency.**

gf3-33

Program Sequencing

In setup mode:

- Program loaded to program memory.
- Constants loaded into constant registers.

In render mode:

- Program run for particular IB/OB pair.
- Program starts each time an IB fills.
- Program completion signals primitive assembly unit to proceed.
- Execution multithreaded.

gf3-34

Program Execution

Assumed Stages (Timing and number of stages unknown, μ-insn fetch omitted):

RR: Register Read.

SN: Swizzle and Negate.

Ei: Execute stage i. This likely takes multiple cycles and fully pipelined.

WB: Writeback.

Multithreaded execution is used in GeForce 3.

gf3-35

Single Thread (Not Multithreaded) Execution

A design option **not used** for GeForce 3.

Finish data from one IB before starting another.

Consider a pair of dependent instructions:

```
ADD r1, c[2], v[3]      RR SN E1 E2 WB
MUL o[4], r1, c[5]         RR ----> SN E1 E2 WB
```

`MUL` stalls two cycles waiting for result of `ADD`.

In GF3 number of stalls would be higher since there are more Ei.

+Just need one μPC and one set of temporary registers.

-Multi-cycle stalls.

-To avoid stalls need bypass paths or scheduling opportunities.

gf3-36

Multithreaded Execution

Used in GeForce 3 (and most if not all modern GPUs).

Work on data from several input buffers simultaneously.

Each **thread** accesses data from one input buffer.

Let ti denote thread i.

Thread i has its own set of temporary registers and μPC.

Thread i reads IB i registers, writes output buffer i registers.

gf3-37

Same pair of dependent instructions as last example.

Five threads active.

```
#   Cycle                  0  1  2  3  4  5  6  7  8  9
t0: ADD r1, c[2], v[3]     RR SN E1 E2 WB        <- v[3] in IB 0  r1 in set 0
t1: ADD r1, c[2], v[3]        RR SN E1 E2 WB     <- v[3] in IB 1  r1 in set 1
t2: ADD r1, c[2], v[3]           RR SN E1 E2 WB
t3: ADD r1, c[2], v[3]              RR SN E1 E2 WB
t4: ADD r1, c[2], v[3]                 RR SN E1 E2 WB
t0: MUL o[4], r1, c[5]                    RR SN E1 E2 WB   <- Also for t1-t4
#   Cycle                  0  1  2  3  4  5  6  7  8  9
```

+No stalls.

+No bypass paths needed.

-Need multiple sets of temporary registers.

Number of IB chosen to cover execution latency.

EE 7700-1 Lecture Transparency. Formatted 10:08, 12 March 2010 from set-study-gf3.
gf3-37

Vertex Processor Design Factors

gf3-38

Exploits vertex program code characteristics:

No memory access: no memory port.

Small program size: tiny program memory.

Limited purpose: specialized instructions.

Vertex independence: easy multithreaded execution.

Repeated execution: data-triggered sequencing.

gf3-39

Vertex Processors in More Recent GPUs

Limited control-transfer instructions (branching).

Access to memory.

Features carefully controlled to preserve multithreading and simplify memory access.

gf3-39