



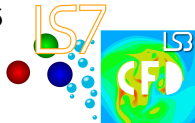
Introduction to data-stream based computations on graphics hardware

Dipl.-Inform. Dominik Götdeke

(dominik.goeddeke@math.uni-dortmund.de)

Mathematics III: Applied Mathematics and Numerics
Computer Science VII: Computer Graphics
University of Dortmund

ASIM 2005 – 18th Symposium on Simulation Technique
Workshop Parallel Computing and Graphics Processors
Erlangen, Germany, September 12th-15th 2005





Overview

- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 GPGPU Techniques and 'Hello World'
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





Overview

- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 GPGPU Techniques and 'Hello World'
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





Overview

- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 GPGPU Techniques and 'Hello World'
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





Overview

- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 GPGPU Techniques and 'Hello World'
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





Overview

- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 GPGPU Techniques and 'Hello World'
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





Some typical benchmark results

Benchmark: FEM building blocks: **SAXPY_C**, **SAXPY_V** (variable coefficients), **MV_V** (9 band Q_1 matrix with variable coefficients) and **DOT** for increasing problem size.

Left: CPU timings on Opteron 244 (3 GFLOP/s LINPACK): highly optimized, cache-friendly FEM package FEAST.

GPU implementation: "quick and dirty" (timings on GeForce 6800, right)



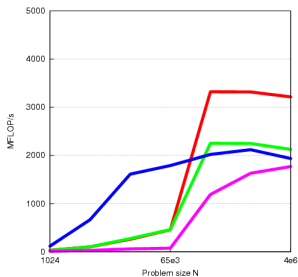
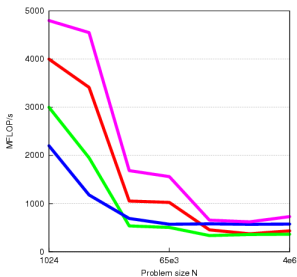


Some typical benchmark results

Benchmark: FEM building blocks: **SAXPY_C**, **SAXPY_V** (variable coefficients), **MV_V** (9 band Q_1 matrix with variable coefficients) and **DOT** for increasing problem size.

Left: CPU timings on Opteron 244 (3 GFLOP/s LINPACK): highly optimized, cache-friendly FEM package FEAST.

GPU implementation: "quick and dirty" (timings on GeForce 6800, right)





Why GPUs, then?

GPUs are fast: > 60 GFLOP/s peak, > 35 GB/s memory bandwidth.

GPUs deliver this performance especially for **larger** problems, no performance penalty like on the CPU due to poor cache coherence.

Most kernels and building blocks in scientific computing map quite well to GPUs. Published results include numerical linear algebra, several solvers for linear systems, multigrid etc., all reporting speed-ups of factor 5 to 20.

For rapid prototyping and interactive applications, simulation data is already in place for visualization.

Accuracy can be an issue, but can be overcome by using the GPU only as **numerical co-processor** (see next talk).





Overview

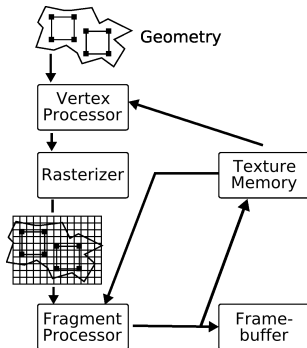
- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 GPGPU Techniques and 'Hello World'
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





The Graphics Pipeline

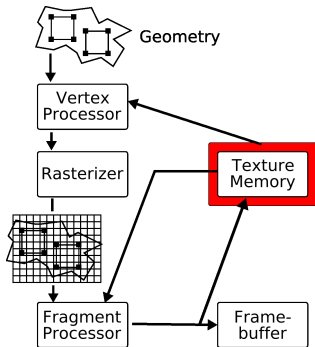
- Each stage in the graphics pipeline can be independently configured through graphics APIs like OpenGL or DirectX.
- Some parts have fixed functionality and require just a few parameters, others are fully programmable.
- Next couple of slides: More detailed look at the different stages: What they offer, how they can be used and abused.





Textures

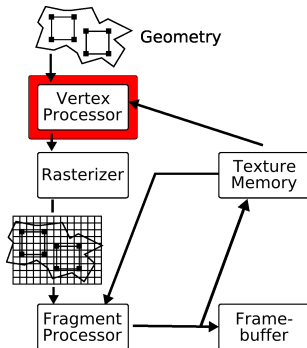
- Textures are the equivalent of arrays.
- Size limitation: 4096 texels in each dimension.
- Native data layout: Rectangular (2D) textures.
- Data formats: One channel (LUMINANCE) to four channels (RGBA).
- Supported floating point formats: 16bit (s10e5), 32bit (s23e8, NVIDIA), 24bit (s16e7, ATI), **almost IEEE 754**.
- ⇒ *Carefully choose data mapping from arrays to textures for optimal performance!*
Highly dependent on vendor and model, no "rule of thumb" available.





Programmable Vertex Processor

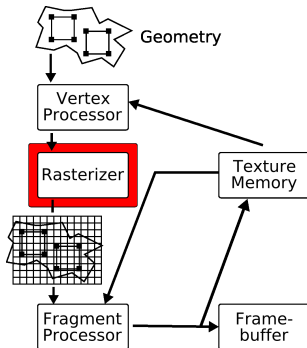
- Input: Stream of geometry.
- Transforms each vertex in homogenous coordinates ($xyzw$) independent of the other vertices, works on 4-tupels simultaneously.
- Capable of scatter (change vertex position) and gather (only by texture fetch).
- Capable of setting additional vertex attributes.
- Up to 6 VP working in parallel.
- Output: Stream of transformed vertices and triangles.
- Fully programmable in assembly and high level shading languages like Cg, GLSL, HLSL.





Fixed-function Rasterizer

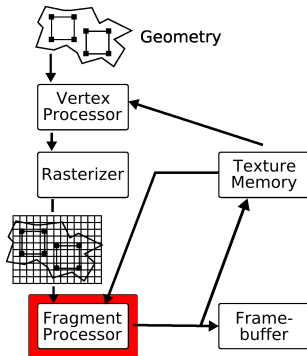
- Input: Stream of transformed vertices and triangles.
- Generates fragment for each pixel covered by transformed geometry.
- Interpolates vertex attributes linearly.
- Output: Stream of fragments.
- Fixed-function part of the pipeline.





Programmable Fragment Processor

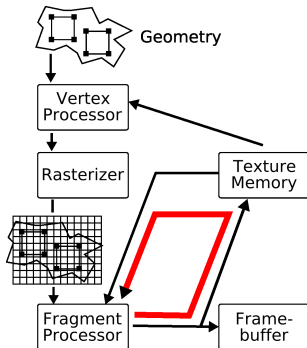
- Input: Stream of fragments with interpolated attributes.
- Applies fragment program to each fragment independently.
- Capable of gather (from up to 16 different input textures), incapable of scatter.
- Works on 4-tupels (RGBA) in parallel without performance penalty.
- Up to 24 FP working in parallel make it the computational workhorse in most applications.
- Output: Pixels to be displayed.
- Fully programmable in assembly and high level shading languages like Cg, GLSL, HLSL.





Feedback: Render-To-Texture

- Textures can be used as render targets!
- Textures are either read-only or write-only.
- Feedback loop: Render intermediate results into a texture, use it as input in subsequent pass.
- Visualization: Render single quad into framebuffer textured with last intermediate result.
- Further processing on CPU: Readback texture data.





Overview

- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 **GPGPU Techniques and 'Hello World'**
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





Hello World

Example: BLAS saxpy() operation for vectors $\mathbf{y}, \mathbf{x} \in \mathbb{R}^N$ and scalar α .

Typical CPU implementation (when not linking to some actual BLAS library): single loop over all elements.

```
for (int i=0; i<N; i++)  
    y[i] = y[i] + alpha*x[i];
```





Concept 1: Arrays = Textures

- Pick optimal data format, here for simplicity a rectangular single-channeled texture for each vector.
- Since textures are either read-only or write-only, three textures are needed: y_{old}, y_{new} will alternately serve as input and output texture (commonly called **ping pong technique**), x is input texture only.
- Create suitable mapping from CPU array to GPU textures, in this case, simple row-wise mapping from array of length N to texture of size $\sqrt{N} \times \sqrt{N}$.
- Pick floating point internal format for these textures.
- Allocate three textures in GPU memory.
- Download texture data into GPU memory.





Concept 2: Loop bodies = kernels

- Recall CPU implementation: The addition is carried out sequentially by iterating over the elements in the arrays:

```
for (int i=0; i<N; i++)
    y[i] = y[i] + alpha*x[i];
```

- Instructions inside the loop are the **kernel**.
- Write similar instructions in a fragment program which is executed in parallel for *all* fragments. Retrieve values from two input textures and return sum of these values.
- Example in Cg, a high level shading language:

```
float saxpy (in float2 coords: WPOS,
            uniform samplerRECT texY,
            uniform samplerRECT texX,
            const float alpha) : COLOR {
    float y = texRECT(texY, coords);
    float x = texRECT(texX, coords);
    return y+alpha*x;
}
```





Concept 2: Loop bodies = kernels

- Recall CPU implementation: The addition is carried out sequentially by iterating over the elements in the arrays:

```
for (int i=0; i<N; i++)
    y[i] = y[i] + alpha*x[i];
```

- Instructions inside the loop are the **kernel**.
- Write similar instructions in a fragment program which is executed in parallel for *all* fragments. Retrieve values from two input textures and return sum of these values.
- Example in Cg, a high level shading language:

```
float saxpy (in float2 coords: WPOS,
            uniform samplerRECT texY,
            uniform samplerRECT texX,
            const float alpha) : COLOR {
    float y = texRECT(texY, coords);
    float x = texRECT(texX, coords);
    return y+alpha*x;
}
```





Concept 3: Computing by drawing

- Set texture y_{new} as render target.
- Set viewport transform to orthogonal projection, enabling a 1:1 pixel:texel mapping.
- Draw viewport-aligned fullscreen quadrilateral with appropriate texture coordinates.
- The rasterizer generates a fragment for each texel / array element.
- The FP is executed in parallel for each texel / array element, output is written to fixed memory location (array index).
- Result texture can be read back to the CPU for further processing, or can be used as input for the next computation pass (here: swap role of y textures).





Advanced topics (I): Domain decomposition

Avoid branches and too general programs, instead use multiple passes and specialized shaders.

Example: Handling of boundary conditions:

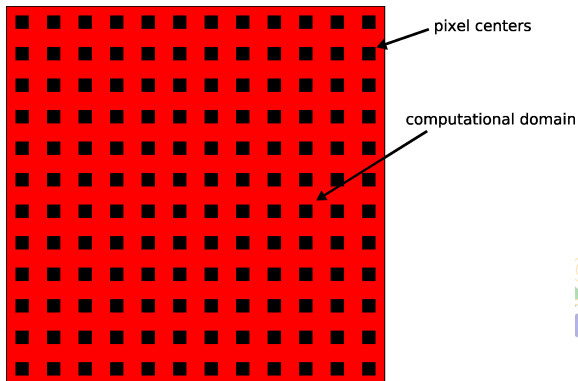




Advanced topics (I): Domain decomposition

Avoid branches and too general programs, instead use multiple passes and specialized shaders.

Example: Handling of boundary conditions:

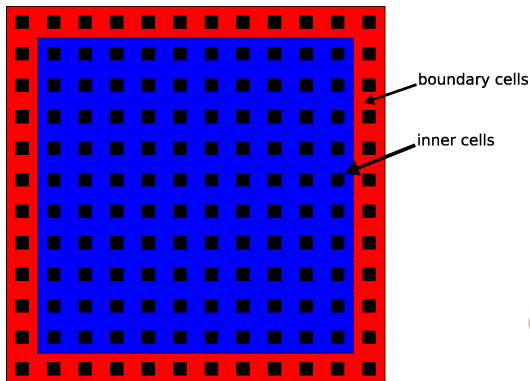




Advanced topics (I): Domain decomposition

Avoid branches and too general programs, instead use multiple passes and specialized shaders.

Example: Handling of boundary conditions:





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.

4	7	2	3	5	7	5	12	7	8
10	20	6	13	14	15	16	17		
19	11	21	22	23	68	25	26		
38	29	64	31	32	33	35	34		
37	28	39	49	53	42	41	52		
46	1	48	40	61	51	44	43		
55	71	4	58	69	62	50	60		
30	65	66	67	24	59	70	56		





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.

47	2	3	57	5	12	7	8
10	20	6	13	14	15	16	17
19	11	21	22	23	68	25	26
38	29	64	31	32	33	35	34
37	28	39	49	53	42	41	52
46	1	48	40	61	51	44	43
55	71	4	58	69	62	50	60
30	65	66	67	24	59	70	56





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.

4	7	2	3	5	7	5	12	7	8
10	20	6	13	14	15	16	17		
19	11	21	22	23	68	25	26		
38	29	64	31	32	33	35	34		
37	28	39	49	53	42	41	52		
46	1	48	40	61	51	44	43		
55	71	4	58	69	62	50	60		
30	65	66	67	24	59	70	56		

47	57	14	17
38	64	68	35
46	49	61	52
65	67	69	70





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.

4	7	2	3	5	7	5	12	7	8
10	20	6	13	14	15	16	17		
19	11	21	22	23	68	25	26		
38	29	64	31	32	33	35	34		
37	28	39	49	53	42	41	52		
46	1	48	40	61	51	44	43		
55	71	4	58	69	62	50	60		
30	65	66	67	24	59	70	56		

47	57	14	17
38	64	68	35
46	49	61	52
65	67	69	70





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.

4	7	2	3	5	7	5	12	7	8
10	20	6	13	14	15	16	17		
19	11	21	22	23	68	25	26		
38	29	64	31	32	33	35	34		
37	28	39	49	53	42	41	52		
46	1	48	40	61	51	44	43		
5	5	7	1	4	5	8	6	9	6
3	0	6	5	6	6	7	2	4	5

4	7	5	7	1	4	1	7
3	8	6	4	6	8	3	5
4	6	4	9	6	1	5	2
6	5	6	7	6	9	7	0

6	4	6	8
6	7	7	0





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.

4	7	2	3	5	7	5	12	7	8
10	20	6	13	14	15	16	17		
19	11	21	22	23	68	25	26		
38	29	64	31	32	33	35	34		
37	28	39	49	53	42	41	52		
46	1	48	40	61	51	44	43		
5	5	7	1	4	5	8	6	9	6
30	6	5	6	6	7	2	4	5	6

4	7	5	7	1	4	1	7
3	8	6	4	6	8	3	5
4	6	4	9	6	1	5	2
6	5	6	7	6	9	7	0

6	4	6	8
6	7	7	0





Advanced topics (II): Reductions

Use **parallel reductions** for *vector to scalar operations* like maximum/minimum/average, norms, dot products: Do logarithmic number of passes, rendering only a quarter of the texture in each pass. This is more efficient than reading all N values from a single fragment.

4	7	2	3	5	7	5	12	7	8
10	20	6	13	14	15	16	17		
19	11	21	22	23	68	25	26		
38	29	64	31	32	33	35	34		
37	28	39	49	53	42	41	52		
46	1	48	40	61	51	44	43		
5	5	7	1	4	5	8	6	9	6
30	6	5	6	6	7	2	4	5	9

4	7	5	7	1	4	1	7
3	8	6	4	6	8	3	5
4	6	4	9	6	1	5	2
6	5	6	7	6	9	7	0

6	4	6	8
6	7	7	0

70





Performance

Some rules of thumb for better performance:

- Try to keep texture lookups spatially coherent for improved texture cache performance.
- Try to hide texture lookup latency by doing lots of independent math after each texture fetch.
- Try to achieve high **arithmetic intensity** for good performance. Rule of thumb: 8 ops per fetch.
- Drawback: So far, all FEM building blocks presented here exhibit arithmetic intensity ≈ 1 .
- Future research: Reformulate solvers to increase their intensity, thereby increasing *total efficiency*.





Overview

- 1 Motivation: Why GPUs?
- 2 GPU resources
 - The Graphics Pipeline
 - Textures
 - Programmable Vertex Processor
 - Fixed-function Rasterizer
 - Programmable Fragment Processor
 - Feedback
- 3 GPGPU Techniques and 'Hello World'
 - Concept 1: Arrays = textures
 - Concept 2: Loop bodies = kernels
 - Concept 3: Computing by drawing
 - Advanced techniques
 - Performance
- 4 Further reading





Further reading

Articles, books, conferences:

- Theory and article overview: *A Survey of General Purpose Computations on GPUs* by Owens et al., Eurographics 2005 State of the art report.
- Practice: *GPU GEMS II: General Purpose Computations on GPUs: A Primer*
- Community web page: <http://www.gpgpu.org>: SIGGRAPH2004,2005 and VIS2004 courses, article overview, active forum, example codes, Beginner's FAQ

Streaming languages and compilers that abstract from graphical context:

- BrookGPU (Stanford)
- libSH: Metaprogramming GPUs (Waterloo)

