GPGPU

When GPUs turned to non-graphics problems

GPGPU
General purpose

A crazy idea that became a smash hit: Try to use the computing power of the GPU for other purposes than graphics.

gpgpu.org

Started when shader programs became powerful enough.

CUDA, OpenCL etc arrived after shaders proved that the path was viable.

GPGPU

Application examples:

- **Image processing**
- **Image analysis**
- **Equation systems**
- **Wavelet transform**
- **Fourier transform**
- **Cosine transform**
- **Level sets**
- **Video codning**

Really just about anything that is computationally heavy and of parallel nature!

GPGPU

Problem:

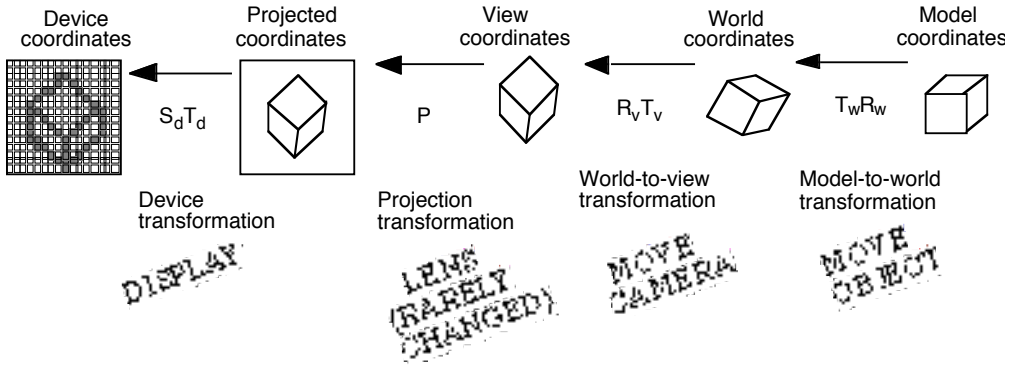
- **Algorithms must be parallellized - more than with CUDA. No intermediate results from neighbors can be used.**

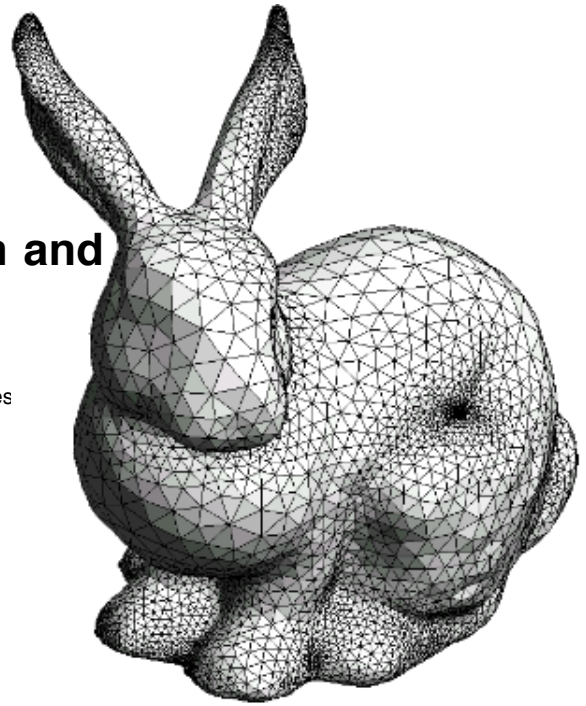
- **No access to shared memory. (But access to constant memory and easy access to texture memory.)**

Does it pay to use shaders for GPU Computing?

Typical OpenGL situation

- Complex geometry
- Many transformations
- Perspective projection
- Lighting and material calculations for the surfaces
- Many texture accesses for interpolation and supersampling





Typical GPGPU processing (also used in filtering in graphics):

- Render to a single rectangle covering the entire image buffer.
- Use FBOs for effective feedback
- Floating-point buffers
- Ping-ponging, many pass with different shaders

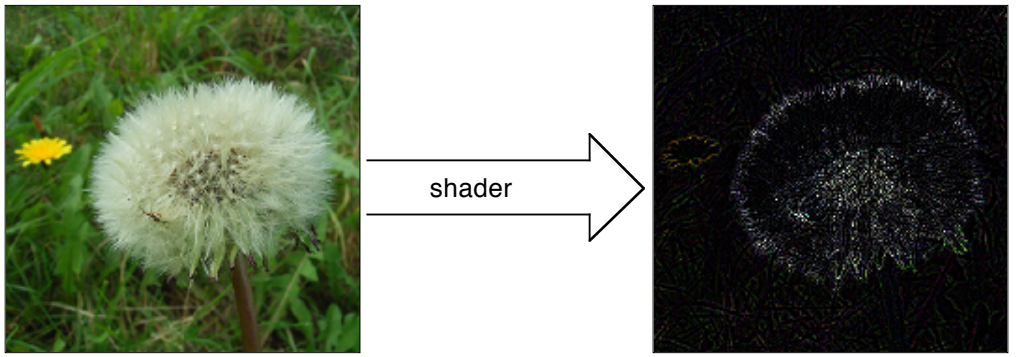
The GPGPU/shaders model

- Array of input data = texture
- Array of output data = resulting frame buffer
- Computation kernel = shader
- Computation = rendering
- Feedback = switch between FBO's or copy frame buffer to texture

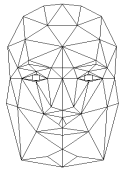
Computation = rendering

Typical situation:

- Texture and frame buffer same size
- Render the polygon over the entire frame buffer



Texture Frame buffer



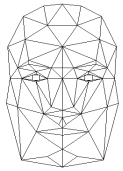
Kernel = shader

Shaders are read and compiled to one or more program objects. A GPGPU application can use several shaders in conjunction!

Activate desired shader as needed using glUseProgramObjectARB();

The fragment shader performs the computation:

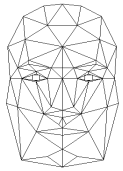
```
uniform sampler2D texUnit;
void main(void)
{
   vec4 texVal  = texture2D(texUnit, gl_TexCoord[0].xy);"

   gl_FragColor = sqrt(texVal);
}
```



Render a single polygon

- Texture and frame buffer same size
- Render polygon over entire frame buffer

```
glBegin(GL_QUADS);
   glTexCoord2f(0, 0);
   glVertex2i(0, 0);
   glTexCoord2f(0, 1);
   glVertex2i(0, m);
   glTexCoord2f(1, 1);
   glVertex2i(n ,m);
   glTexCoord2f(1, 0);
   glVertex2i(n, 0);
glEnd();
```



Feedback

We must be able to pass output from one operation as input of the next!

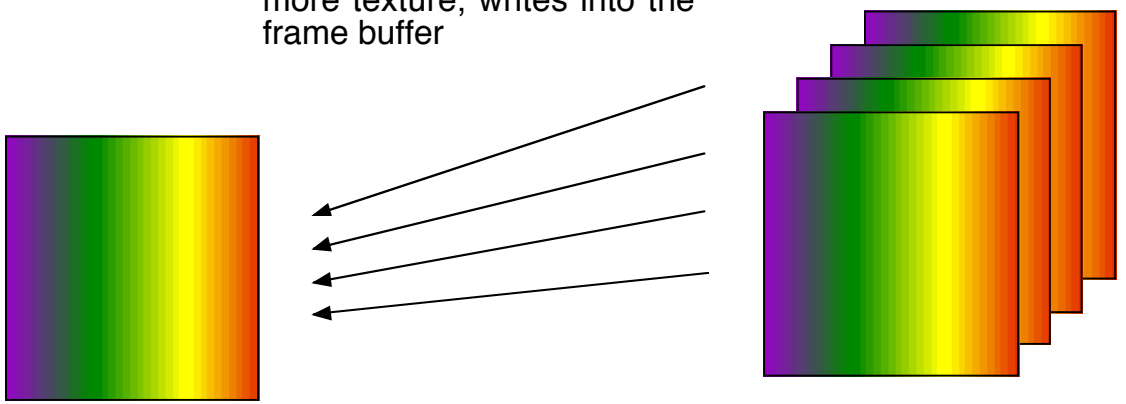
Stable but not the fastest: glCopyTexSubImage2D
Copies frame buffer to texture!

```
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, n, m);
```

Faster solutions are newer members of the standard.
Best: Framebuffer Objects.

“Ping-pong”-ing

The kernel reads from one or more texture, writes into the frame buffer



Using “framebuffer objects” the output image can be a texture

Input data is a number of textures. Limited by the number of texturing units available.

Ping-ponging in practice

Set source:

```
glBindTexture(GL_TEXTURE_2D, tx1);
```

Set destination:

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, tx2, 0);
```

Set shader:

```
glUseProgramObjectARB(shaderProgramObject);
```

Render! Repeat!

Filtering, convolution

Common problem, highly suited for shaders.

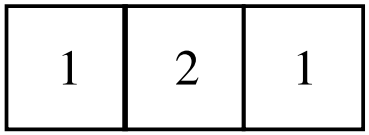
All kinds of linear filters:

- **Low-pass filtering (smoothing)**
- **Gradient, embossing**

Must be done by gather operations, not scatter!

3x1 filter

```
uniform sampler2D texUnit;
uniform float texSize;
void main(void)
{
    float offset = 1.0 / 256.0;
    vec2 texCoord = gl_TexCoord[0].xy;
    vec4 c  = texture2D(texUnit, texCoord);
    texCoord.x = texCoord.x + offset;
    vec4 l  = texture2D(texUnit, texCoord);
    texCoord.x = texCoord.x - 2.0*offset;
    vec4 r  = texture2D(texUnit, texCoord);
    texCoord.x = texCoord.x - offset;
    gl_FragColor = (c + c + l + r) * 0.25;
}
```



Separable filters

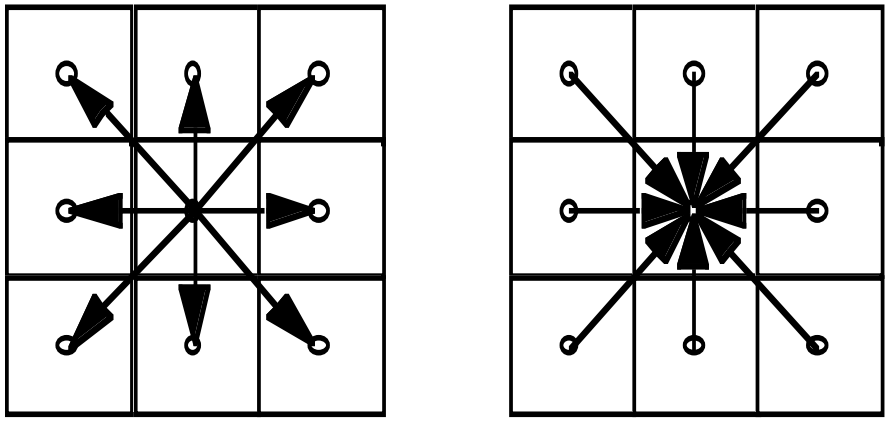
1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

$=$

$$\begin{bmatrix}1 & 2 & 1\end{bmatrix} \otimes \begin{bmatrix}1 & 2 & 1\end{bmatrix} \otimes \begin{bmatrix}1\\2\\1\end{bmatrix} \otimes \begin{bmatrix}1\\2\\1\end{bmatrix}$$

Implemented as ping-ponging passes.
Optimization possibilities!

Scatter vs gather



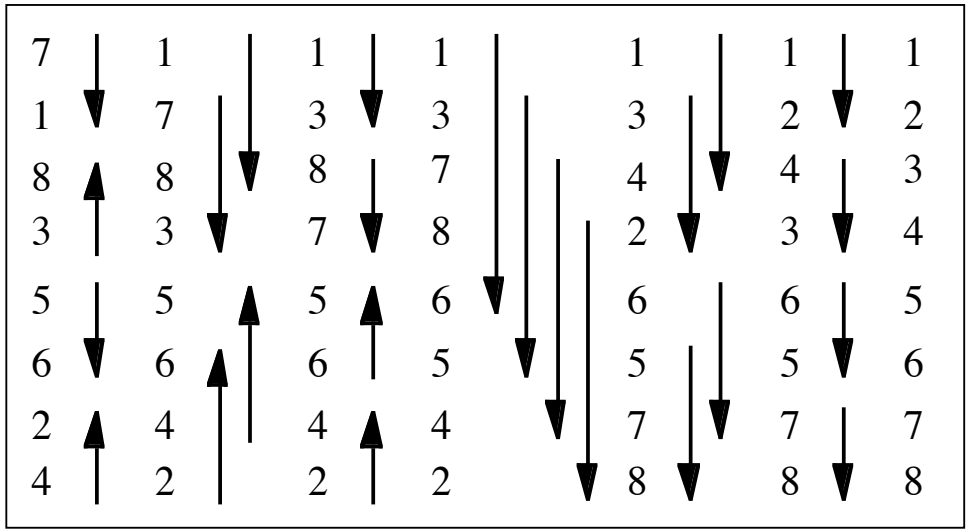
Scatter Gather

Shaders give output for *one* pixel -> gather only!

Sorting

QuickSort hard to implement in shaders

Bitonic Merge Sort fits shaders well

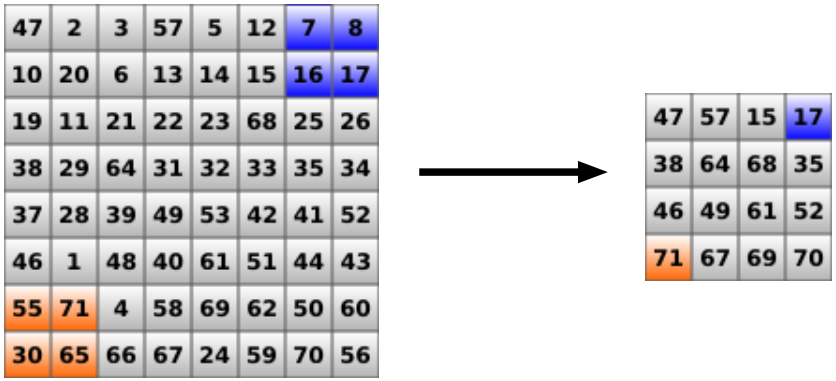


Reduction

Reduction algorithms are implemented by a ping-ponging pyramid

Maximum, minimum, global average...

Output smaller than input



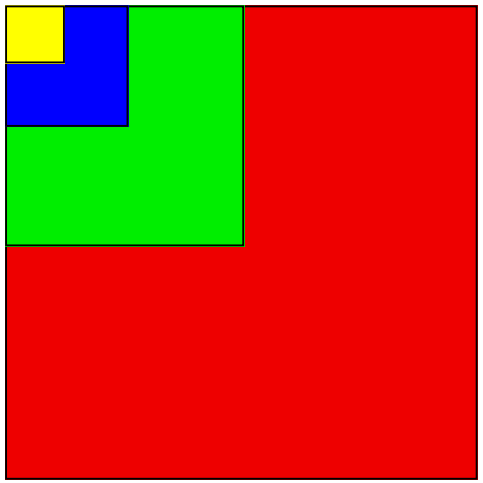
(Images by Dominik Göddeke)

Reduction

1) Texture pyramid, typically 2x2

2) Constant texture size, use smaller and smaller parts of the texture!

Same performance! The geometry coverage is what counts!



Special considerations

- vec4 or scalar?
- Texture size limitations
- Interpolation

vec4 or scalar?

GPUs are/were designed to process 4-component vectors! (NVidia less so today.)

Packing data in groups of four values (RGBA) can be needed for maximizing performance - especially on AIT boards.

This will complicate algorithms. The neighbor of data[100].a is data[101].r!

Texture size limitations

Maximum 4096 elements! That means 16384 floating-point values!

Larger arrays must be packed in 2D or 3D!

Again, edges get complicated. The neighbor of data[0,255] is data[1,0] (for a 256 item wide texture)!

Interpolation

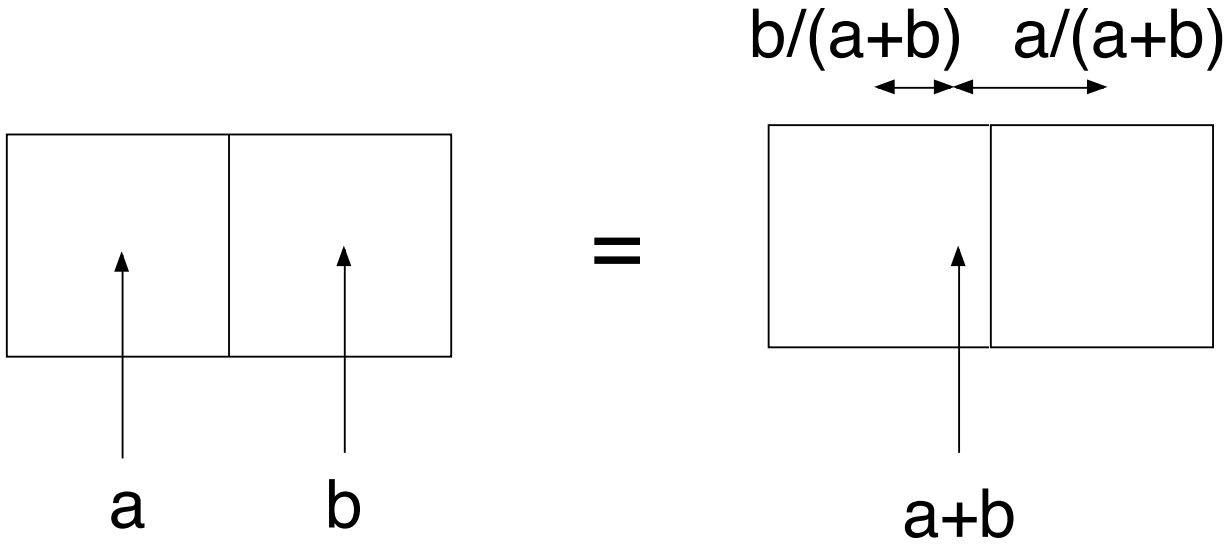
Computation tricks when optimizing

Texture access provides hardware accelerated linear interpolation!

Access texture data on non-integer coordinates and the texture hardware will do linear interpolation automatically!

Can be used for many calculations, e.g. filters.

Interpolation



Texture accesses and calculations hardware accelerated!