



Information Coding / Computer Graphics, ISY, LiTH

# **GPGPU**

**When GPUs turned to non-graphics  
problems**

Information Coding / Computer Graphics, ISY, LiTH

# **GPGPU**

## **General purpose**

**Interesting trend: Try to use the computing power of the GPU for other purposes than graphics.**

**[gpgpu.org](http://gpgpu.org)**

**Argument in favor: GPU performance grows much faster than CPU's. CPUs are stuck in the "power wall" and "memory wall".**

**(Note that GPU performance advancements have also slowed down a bit lately.)**

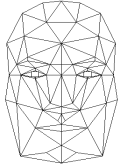


# GPGPU

## Application examples:

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

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



# GPGPU

## **Problem:**

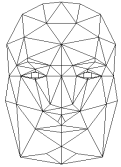
- **Algorithms must be parallelized. No intermediate results from neighbors can be used.**
- **The CPU interface is a bottleneck. In early days (AGP), it was a serious bottleneck. The processing time can be less than the time needed to read the result.**

**Does it pay to use GPGPU?**



## **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**



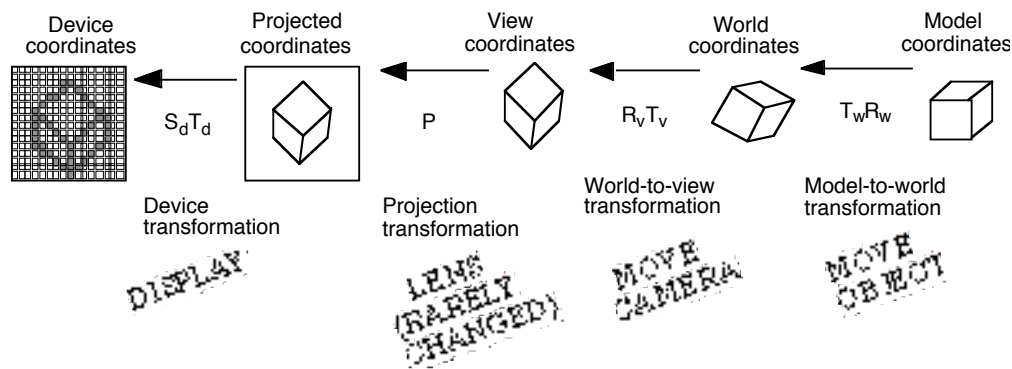
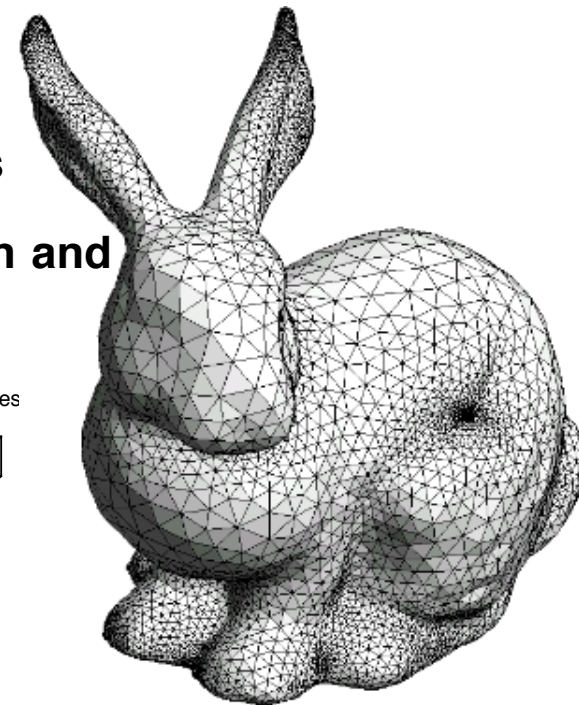
## The GPGPU model

- 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



# Typical OpenGL situation

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





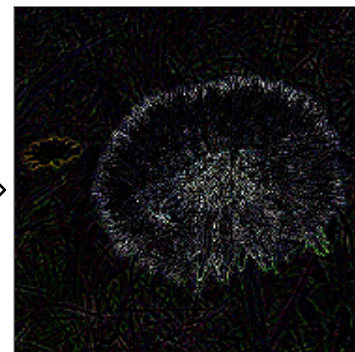
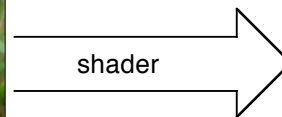
# Computation = rendering

Typical situation:

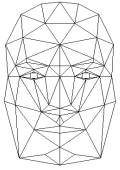
- Texture and frame buffer same size
- Render the polygon over the entire frame buffer



Texture



Frame buffer



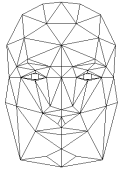
## Kernel = shader

**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:**

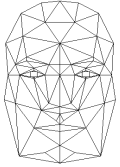
```
uniform sampler2D texUnit;  
void main(void)  
{  
    vec4 texVal = texture2D(texUnit, gl_TexCoord[0].xy);"  
    gl_FragColor = sqrt(texVal);  
}
```



## 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!**

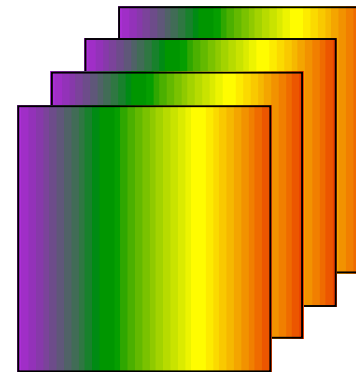
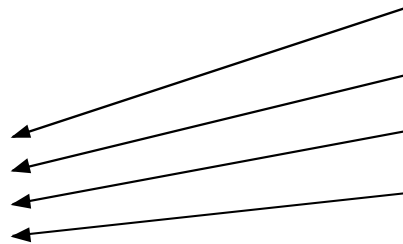
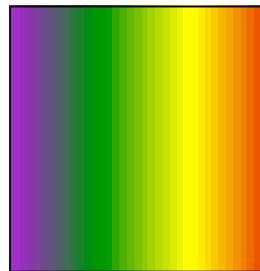
```
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, n, m);
```

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



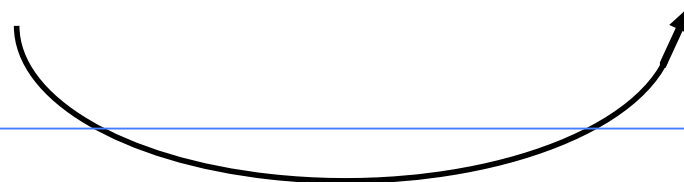
# “Ping-pong”-ing

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!**



# Ping-ponging in working code

```
void runfilter(GLhandleARB shader, GLuint texin,
              GLuint texin2, GLuint fboout)
{
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,
                        fboout);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texin2);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texin);

    glViewport(0, 0, width, height);
    if (fboout == 0)
        glViewport(0, 0, lastw, lasth);

    // Clean matrices!
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glUseProgramObjectARB(shader);

    // Polygon over framebuffer
    glBegin(GL_POLYGON);
    glTexCoord2f(0, 0);
    glVertex2f(-1, -1);
    glTexCoord2f(1, 0);
    glVertex2f(1, -1);
    glTexCoord2f(1, 1);
    glVertex2f(1, 1);
    glTexCoord2f(0, 1);
    glVertex2f(-1, 1);
    glEnd();

    // Restore
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}
```



## ...gets pretty nice in the end

```
// Run treshold shader from tex 1 to tex/fbo 2
runfilter(tresholdShader, tex1, 0, fb2);

// Filter several times
for (i = 0; i < loops; i++)
{
    runfilter(lpVShader, tex2, 0, fb3);
    runfilter(lpHShader, tex3, 0, fb2);
}

// Output result
runfilter(0, tex2, 0, 0);
```



# Filtering, convolution

**Common problem, highly suited for shaders.**

**All kinds of linear filters:**

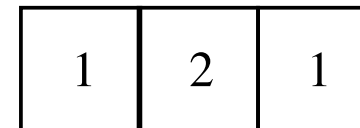
- **Low-pass filtering (smoothing)**
  - **Gradient, embossing**

**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

=

1	2	1
---	---	---

⊗

1	2	1
---	---	---

⊗

1
2
1

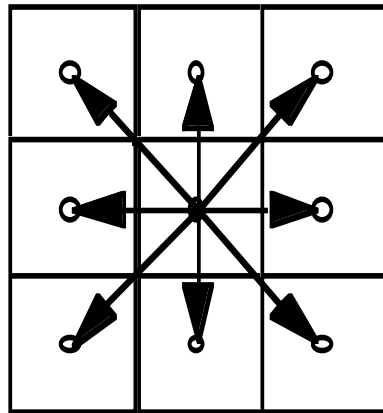
⊗

1
2
1

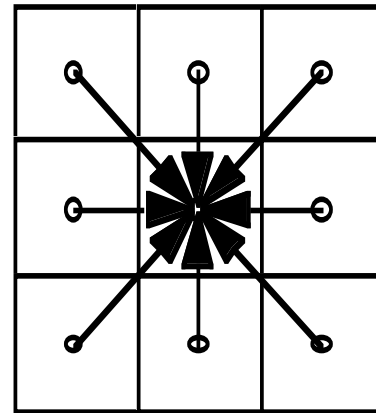
**Implemented as ping-ponging passes.  
Optimization possibilities!**



## Scatter vs gather



**Scatter**



**Gather**

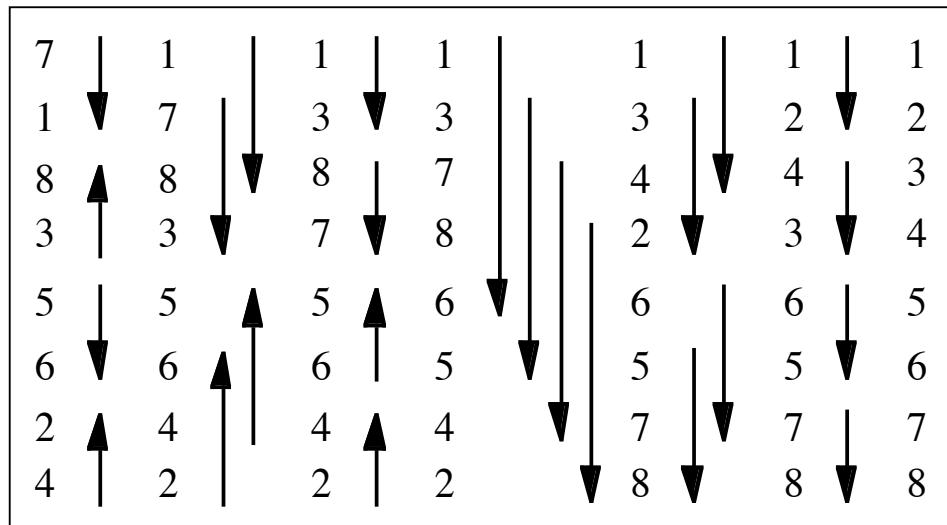
**Shaders give output for *one* pixel -> gather only!**



# 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

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

→

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

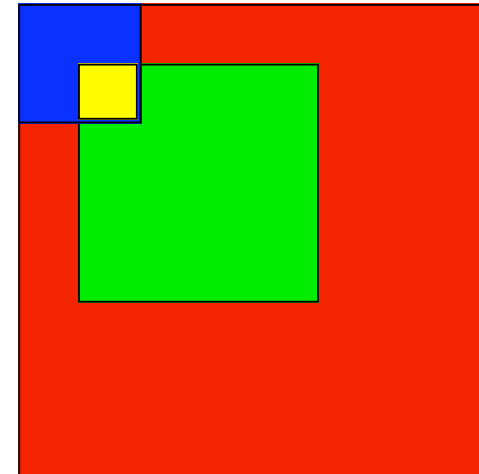
(Images by Dominik Göddeke)



## 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

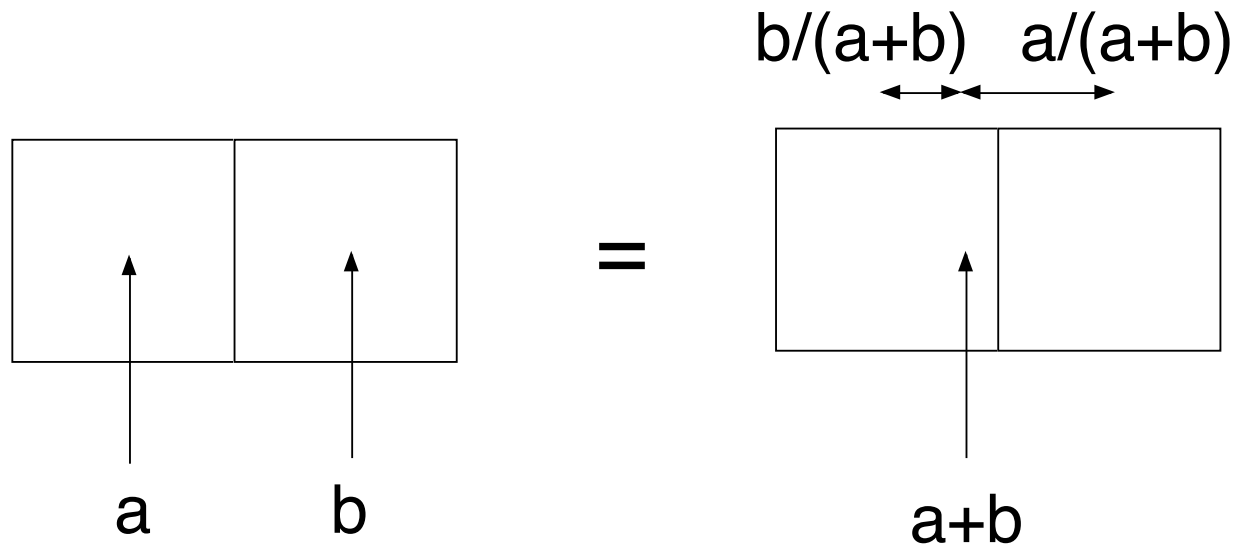
Texture access provides hardware accelerated linear interpolation!

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

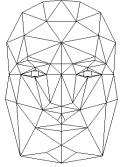
Can be used for many calculations, e.g. filters.



# Interpolation



**Texture accesses and calculations hardware accelerated!**



## Conclusions:

- **Shader-based GPGPU is not dead, it is just not hyped**

Superior compatibility and ease of installation makes it highly interesting for the foreseeable future. Especially suitable for all image-related problems.

- **GLSL**

The OpenGL shading language was introduced.

- **How to do GPGPU with shaders**

FBOs, Ping-ponging, algorithms, special considerations.