Lecture 13

Image filters

OpenCL

GPU computing with GLSL

OpenGL Compute shaders

Lecture questions

1) What kind of devices will OpenCL run on?

2) What does an OpenCL work group correspond to in CUDA?

3) What geometry is typically used for shader-based GPU computing?

4) Are scatter or gather operations preferrable? Why?

Lab 5

- Image filtering with shared memory
- Low-pass filters
- Median filter

Intended as continuation of previous image filtering lab.

Lab 6

- OpenCL
- Reduction
- Sorting using bitonic merge sort

A few more features

Dynamic shared memory

Example of dynamic parallelism

Tensor and RT cores

Dynamic shared memory

You can adapt the allocation after compilation!

<<gridsize, blocksize, *dynamic_shared*>>

One single buffer, split usage manually.

Something like this

Set pointers to the dynamic buffer!

```
dynamic_shared_size = N*sizeof(double) + M * sizeof(char);
mykernel<<<gridsize,blocksize,dynamic_shared_size>>>()

__global__ void mykernel()
{
  extern __shared__ char array[];

  double *somedata = (double *)array; // N doubles, statt of arrray
  char *somemoredata = &array[N*sizeof(double)]; // Rest of array

// Process data
}
```

Demo on dynamic parallellism

Requires compure capability 3.5 = not a problem.

Simple demo, just saving thread numbers
but uses two kernels.

Note: Requires modifier -rdc=true

How about Tensor and RT cores?

To do: Really tight demo.

Tensor core demos are always doing too big things!

One more CUDA-like library

HIP = *Heterogeneous-computing Interface for Portability*

From AMD! "Thin layer" on AMDs ROCm, very similar to CUDA, also giving single-file capability.

Big player!

Image filters (lab 3 and 5)

Linear filters: Convolution

Box filter

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

/25

Gaussian approximation

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

/256

And others

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Sobel (gradient)

0	-1	0
-1	4	-1
0	-1	0

Laplace

Separable filters

$$\begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix}/5 \oplus \begin{bmatrix}1&1&1&1&1\end{bmatrix}/5 = \begin{bmatrix}1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\end{bmatrix}/25$$

$$\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix}/16 \oplus \begin{bmatrix}1&4&6&4&1\end{bmatrix}/16 = \begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&36&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}/256$$

Repeated box filters converge to Gaussian!

$$\begin{matrix}1\\4\\6\\4\\1\end{matrix}/16 = \begin{matrix}1\\2\\1\end{matrix}/4 \oplus \begin{matrix}1\\2\\1\end{matrix}/4 = \begin{matrix}1\\1\end{matrix}/2 \oplus \begin{matrix}1\\1\end{matrix}/2 \oplus \begin{matrix}1\\1\end{matrix}/2 \oplus \begin{matrix}1\\1\end{matrix}/2$$

Central limit theorem

Compare to dice!

Non-linear filters

Median filter

Outputs median of neighborhood.

Requires some method to find the median.

Possible application: Noise suppression.
Preserves edges!

Separable only as approximation.

Median filters

Example

1	2	2
1	9	3
1	7	10

1 1 1 2 2 3 7 9 10

Average: 4
Median: 2

Not separable

1	2	2	2
1	9	3	3
1	7	10	7
1	7	3	

Rows first or columns first both end up 3 for this case

Works as approximation

How to filter edges

The filter kernel reaches outside the image!

My answer: clamp! Use min(max()) or the clamp() function.

Solved for you in the Lab 5 code.

Why? Avoid branching!

```
    if (x < imagesizex && y < imagesizey)
    {
// Filter kernel (simple box filter)
    sumx=0;sumy=0;sumz=0;
    for(dy=-kernelsizey;dy<=kernelsizey;dy++)
        for(dx=-kernelsizex;dx<=kernelsizex;dx++)
        {
            // Use max and min to avoid branching!
            int yy = min(max(y+dy, 0), imagesizey-1);
            int xx = min(max(x+dx, 0), imagesizex-1);

            sumx += image[((yy)*imagesizex+(xx))*3+0];
            sumy += image[((yy)*imagesizex+(xx))*3+1];
            sumz += image[((yy)*imagesizex+(xx))*3+2];
    }
out[(y*imagesizex+x)*3+0] = sumx/divby;
out[(y*imagesizex+x)*3+1] = sumy/divby;
out[(y*imagesizex+x)*3+2] = sumz/divby;
}
```

Remember texture memory?

Clamp and repeat

This is perfect and automatic!

or this

You are used to this

ERROR	ERROR	ERROR	ERROR
ERROR	1	2	ERROR
ERROR	3	4	ERROR
ERROR	ERROR	ERROR	ERROR

Now you can get this

4	3	4	3
2	1	2	1
4	3	4	3
2	1	2	1

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

In the lab

1. Shared memory
Use shared memory to reduce global memory access.
Major part of the lab!

2. Separable filters
Easy if step 1 is done right.

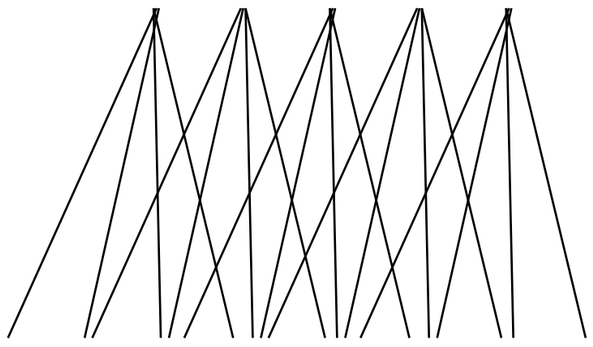
3. Weighted kernels
One size is enough.

4. Median filter
Variable size, modest demands.

Trivial filter

Just loop over kernel

Inefficient! Multiple reads from global memory!

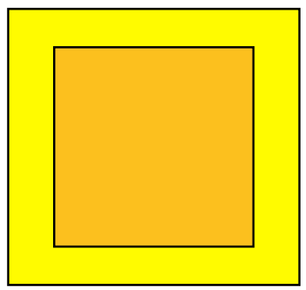


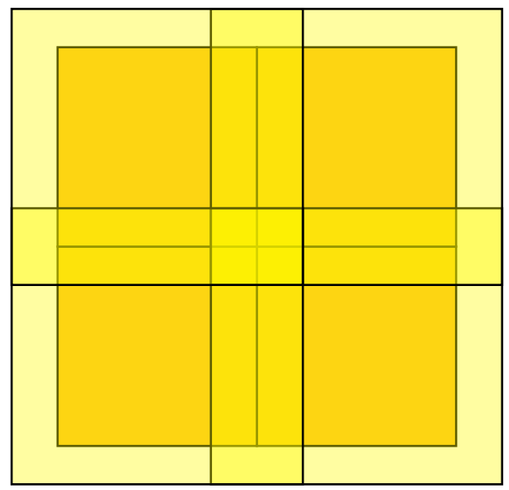
Better filter

Use shared memory!

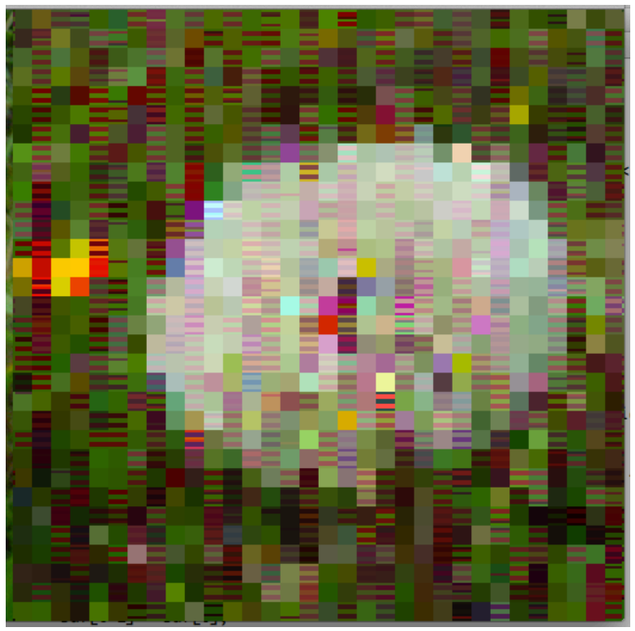
How can you minimize global reads,
or at least significantly reduce them?

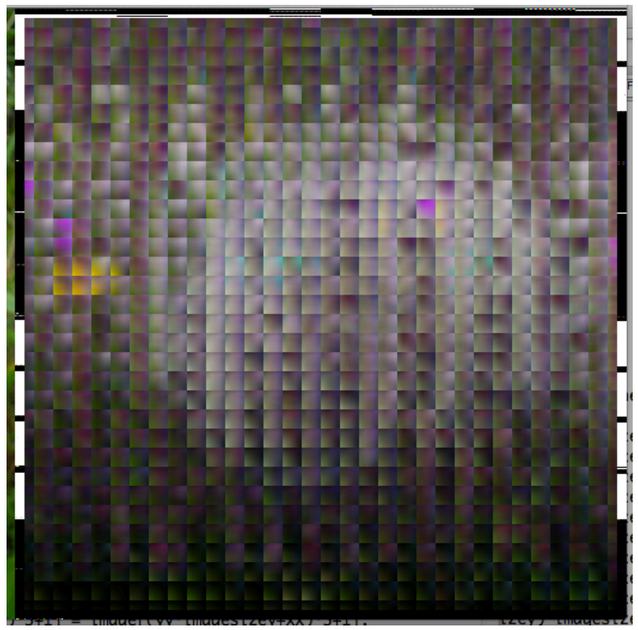
Note the edges of the patch
computed by each block!





Bonus: Unintentional fun!





Coding filters in CUDA is like a box of chocolate…

GPU Computing with fragment shaders
”Classic GPGPU”

Use graphics shaders for general-purpose computing.

Adapt your data and computing to fit the graphics pipeline.

Hot until CUDA arrived, now overshadowed by CUDA and OpenCL.

Why is classic GPGPU interesting?

- Highly suited to all problems dealing with images, computer vision, image coding etc

- Parallelization ”comes natural”, you can’t avoid it and good speedups are likely. Fewer pitfalls.

- Highly optimized (for graphics performance).

- Compatibility is vastly superior!

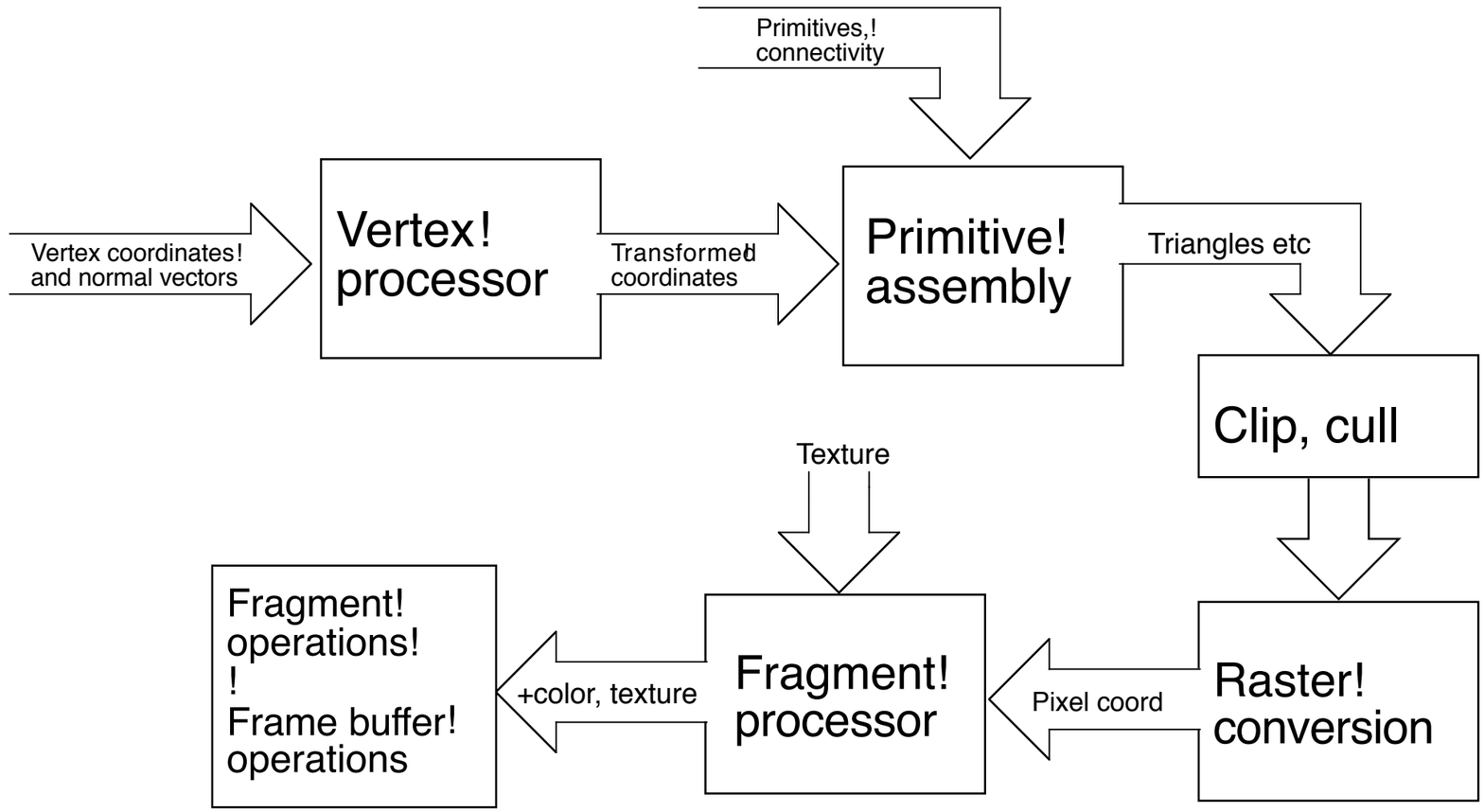
- Very much easier to install!

So what is not so good?

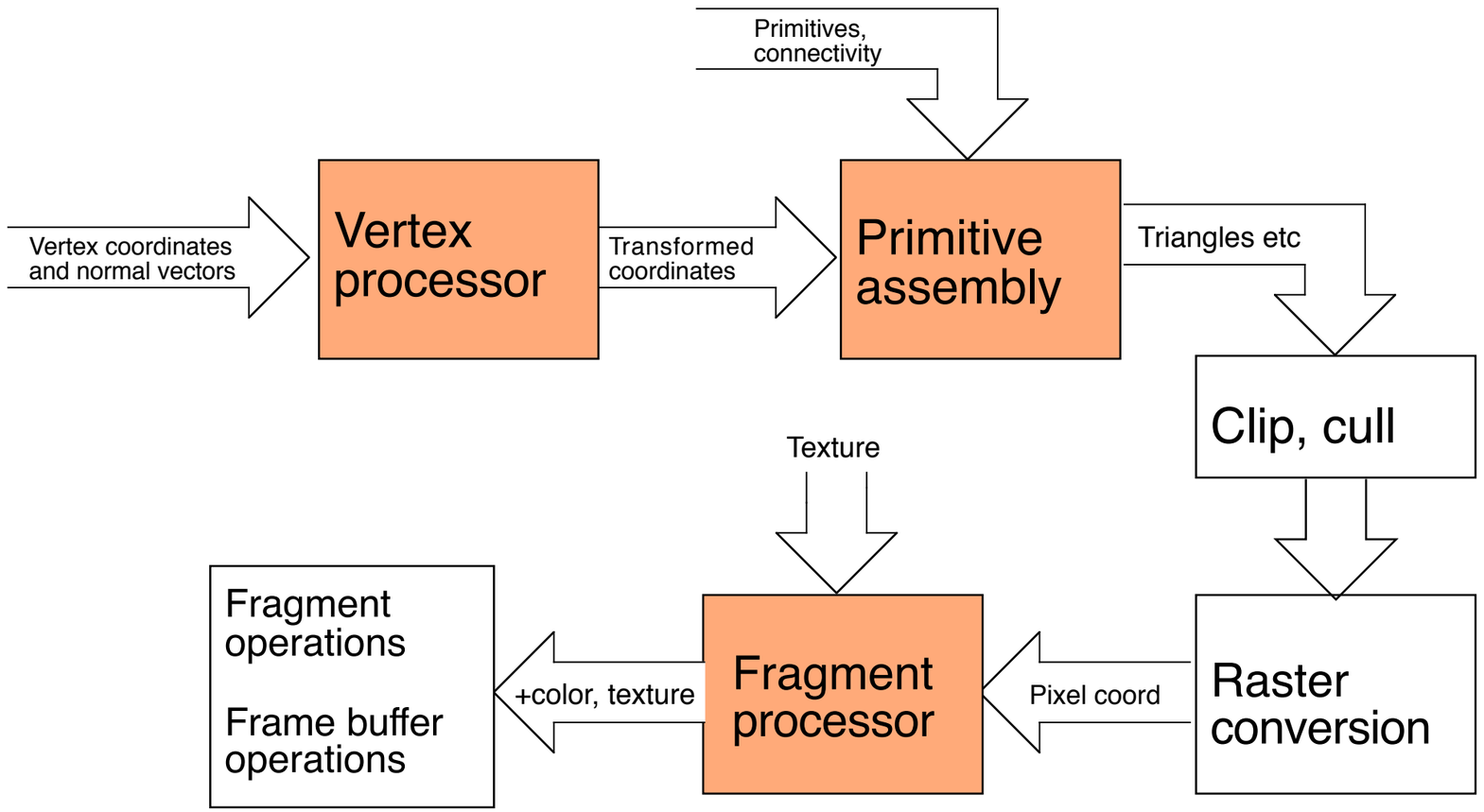
- Must map data to image data
- Computing controlled by pixels in output image
- No shared memory access

However: OpenGL 4 adds much flexibility, moves closer to CUDA and (especially) OpenCL. Writable textures, atomics, synchronization...

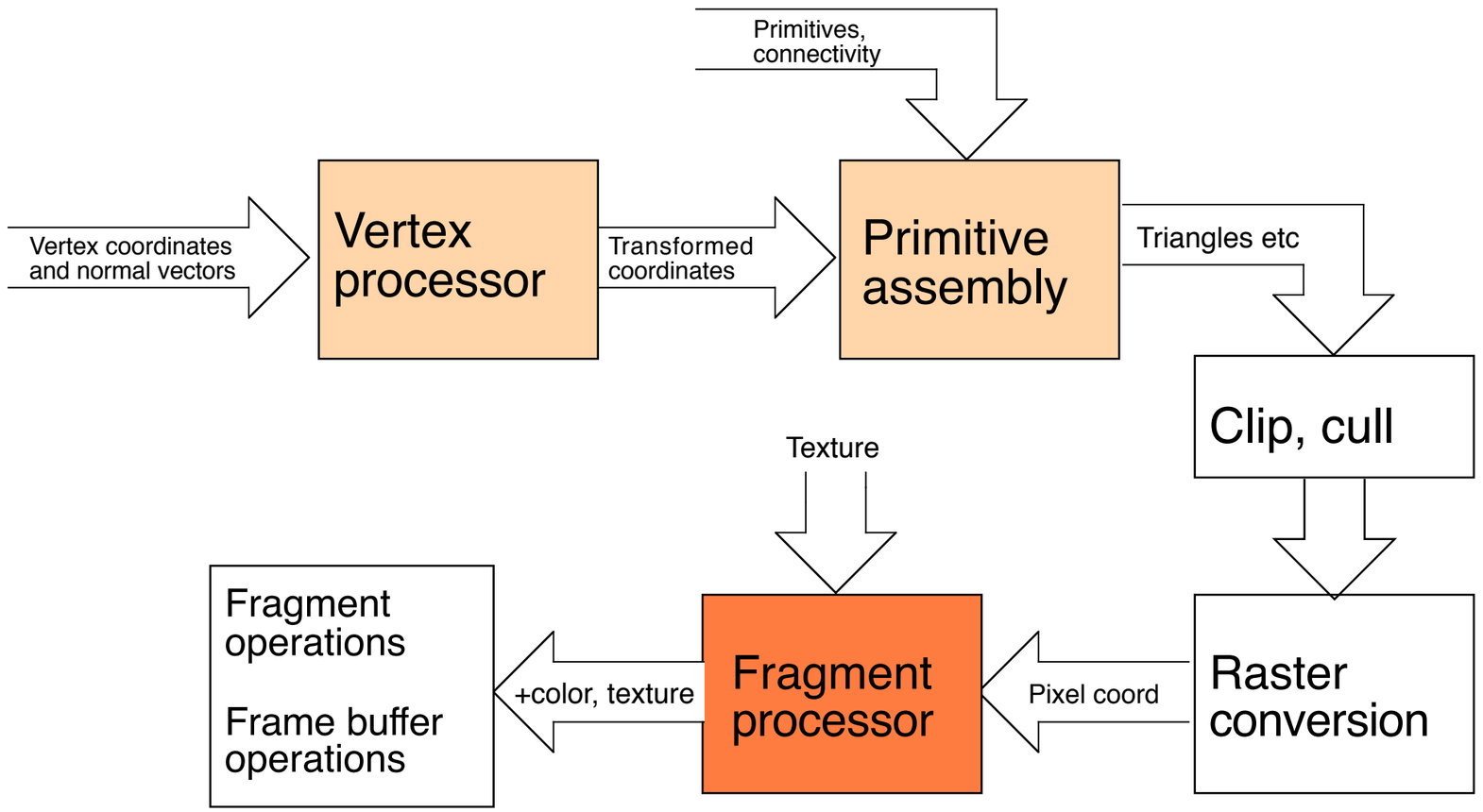
The OpenGL pipeline



Out of these, three are programmable!

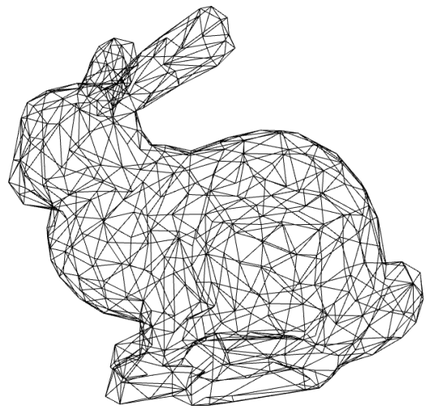


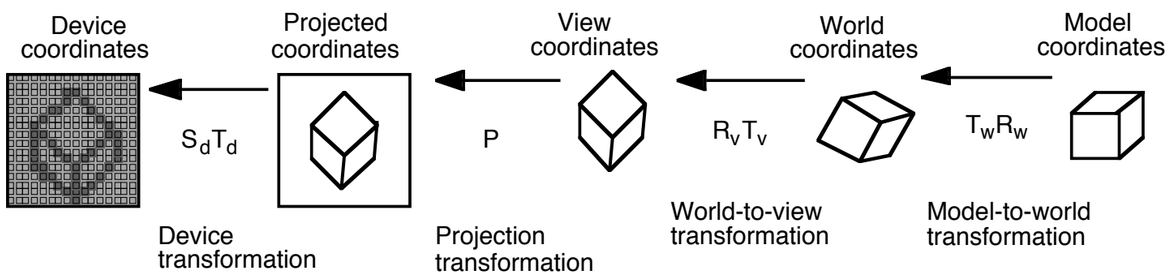
But only one creates easily accessible output data!

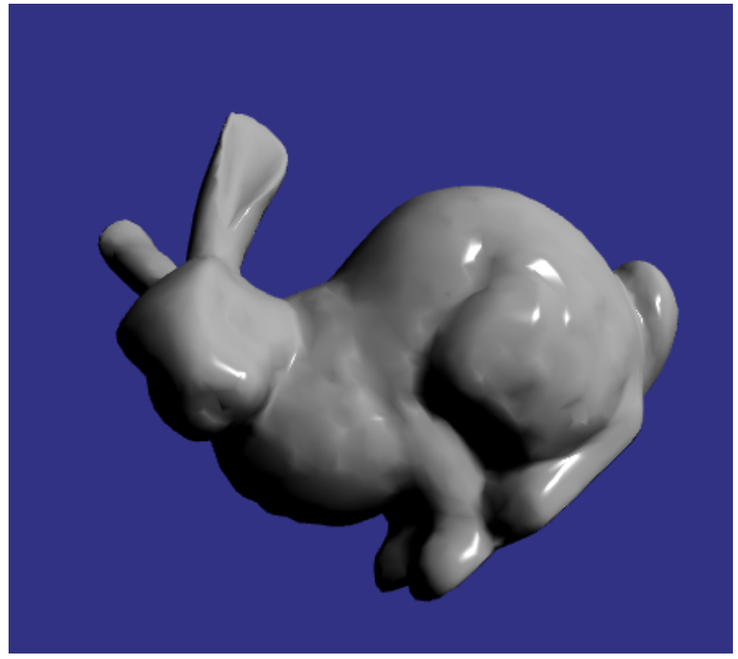


Typical OpenGL situation

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

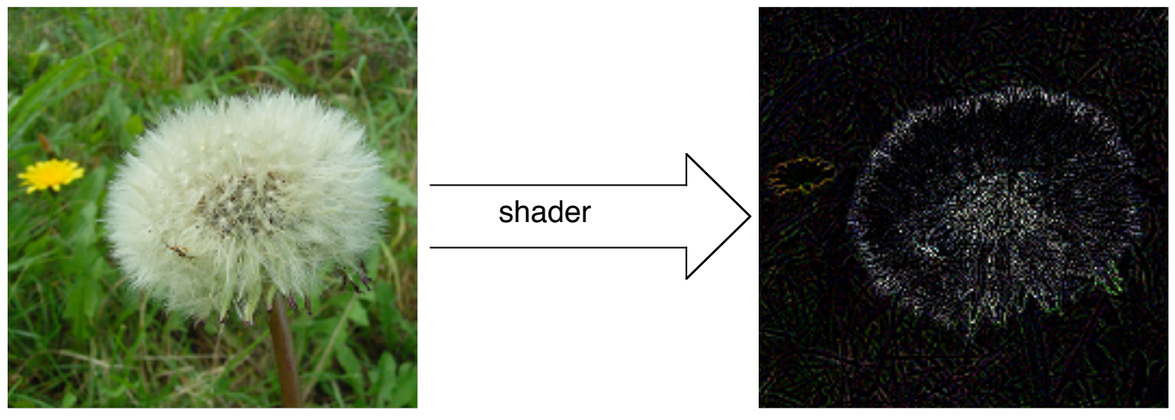






Typical GPU Computing with fragment shaders (also used in filtering in graphics):

- 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



Render image 1:1 Output

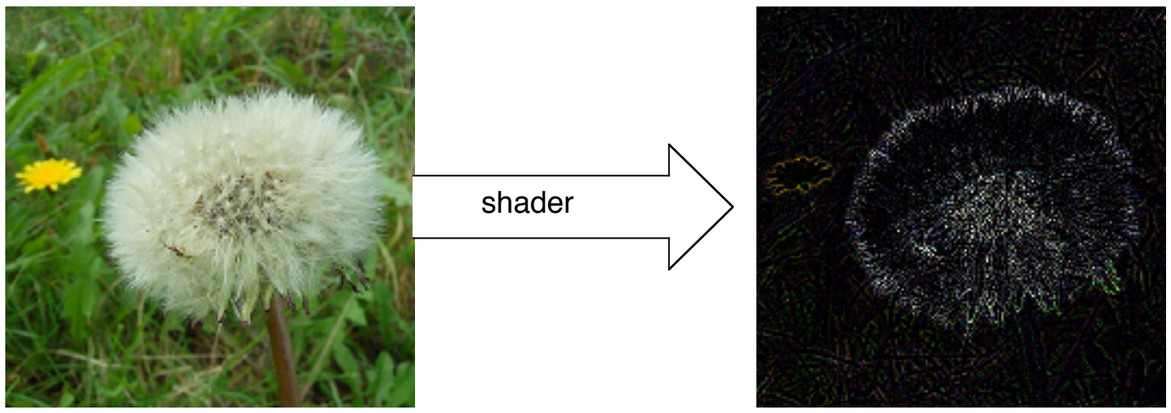
Computing model

- 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 glUseProgram();

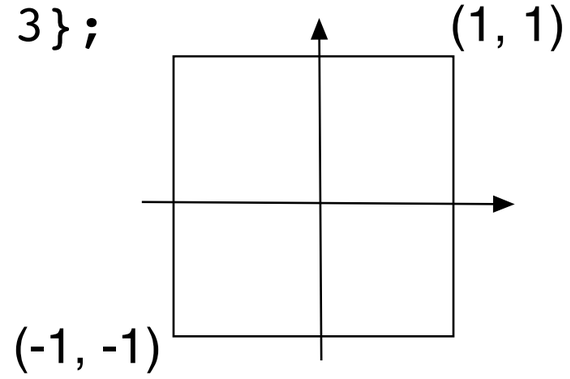
The fragment shader performs the computation:

```
uniform sampler2D texUnit;
in vec2 texCoord;
out vec4 fragColor;

void main(void)
{
   vec4 texVal  = texture(texUnit, texCoord);
   fragColor = sqrt(texVal);
}
```

Render a single polygon

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

```
GLfloat quadVertices[] = { -1.0f, -1.0f, 0.0f,
                           -1.0f, 1.0f, 0.0f,
                            1.0f, 1.0f, 0.0f,
                           1.0f, -1.0f, 0.0f};
  GLuint quadIndices[] = {0, 1, 2, 0, 2, 3};
```



Program structure:

- Set up OpenGL
- Upload data to texture
- Load shaders from file and compile
- Draw quad on screen (of off screen) using OpenGL
- Data is computed by the fragment shader, per pixel
- Output can be downloaded as image data

Examples…

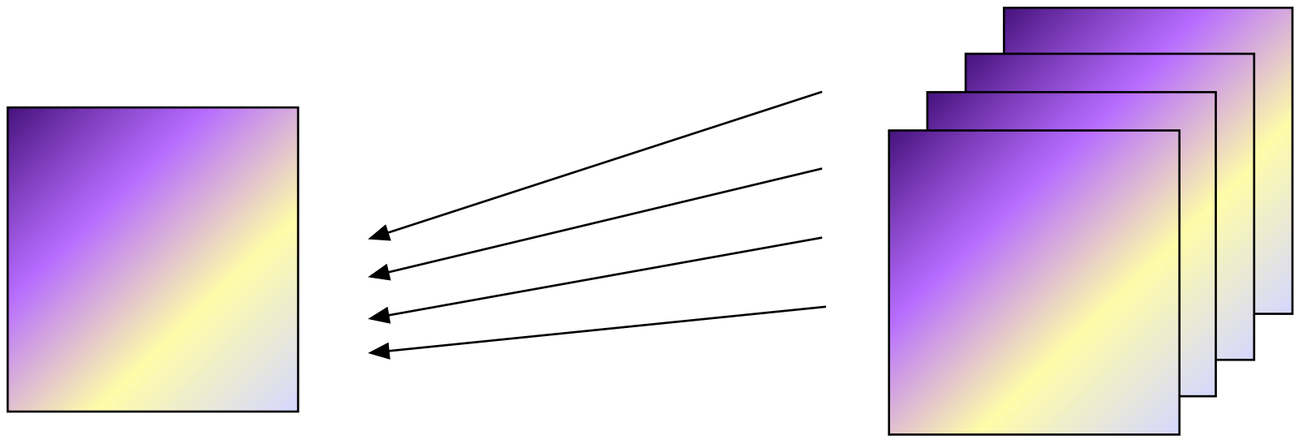
Feedback

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

Solution: Render to texture, ”framebuffer objects”, create a texture used as input for a later stage

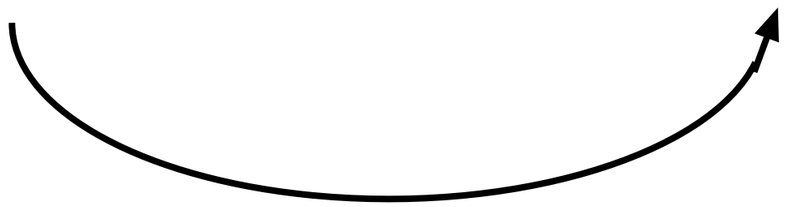
“Ping-pong”-ing

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.



Filtering, convolution

Common problem, highly suited for shaders.

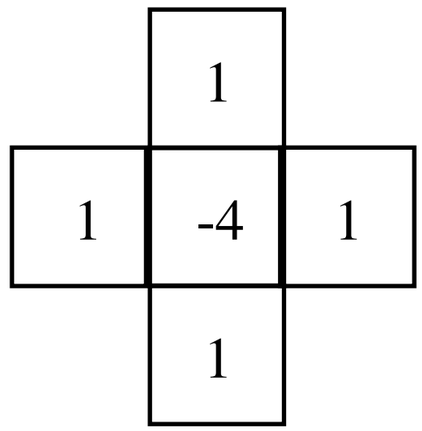
All kinds of linear filters:

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

Must be done by gather operations, not scatter!

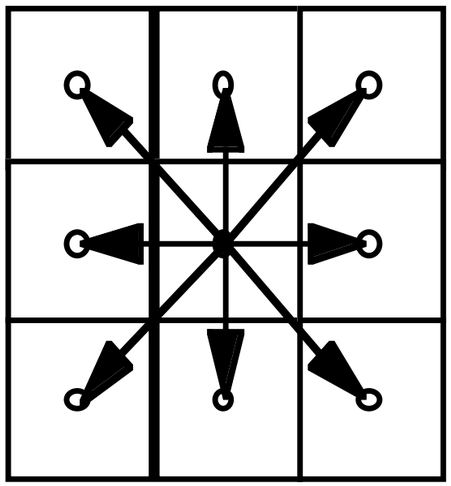
Example: high pass filter

```
#version 150

out vec4 outColor;

in vec2 texCoord;
uniform sampler2D tex;

void main(void)
{
   float h, v;
   const float offset = 1.0/512.0;

   vec4 c  = texture(tex, texCoord);
   vec4 r  = texture(tex, texCoord + vec2( offset,  0.0));
   vec4 l  = texture(tex, texCoord + vec2(-offset,  0.0));
   vec4 u  = texture(tex, texCoord + vec2(    0.0, offset));
   vec4 d  = texture(tex, texCoord + vec2(    0.0,-offset));
   outColor = (-4.0*c + r + l + u + d);
}
```

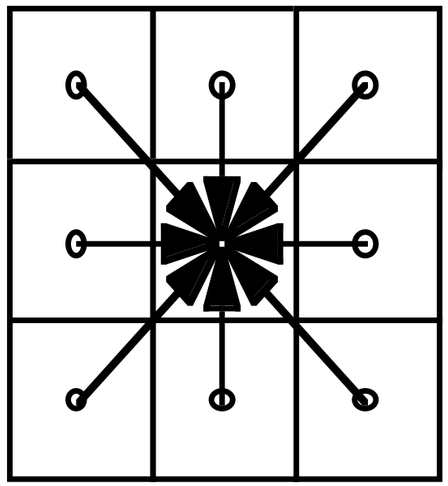


More graphics heritage: Index data by steps of 1/size, not 1!

Scatter vs gather



Scatter



Gather

Shaders give output for one pixel -> gather only!

How about CUDA/OpenCL?

Scatter vs gather: You usually prefer gather. Less synchronization! (Remember, synchronization comes for a cost!)

Separable filters: Optimization just as valid for all techniques!
(But particularly common in shaders, for images.)

Reduction, sorting

Same methods as I have mentioned before.

Bitonic sort suitable.

Reduction by tree structure.

In the past: Fixed output per thread. This is getting less fixed.

- Write to texture possible.
- Synchronization supported.

Conclusions:

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

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

- **How to do GPGPU with shaders**

FBOs, Ping-ponging, algorithms, special considerations.

But stay tuned for Compute Shaders to change things...

Introduction to OpenCL

Open Compute Language


T!
M?
PEL!
Zlatan efter nya succén:
VI ÄR
REDO
FÖR CL

- Motivation
- Overview
- Examples
- Performance comparison

Origins of OpenCL

Initiated by Apple

Managed by Khronos group

Many supporting parties

Many providers

Image removed

Why?

- The market could not let CUDA rule the world
- Support for other platforms
- Open standard
- Similarity with OpenGL

For programming ”all” parallel architectures

Supported architectures (not complete!)

GPU

Intel compatible CPUs

ARM

FPGA

CELL

Intel Xeon Phi

Who decides? Any company making its own OpenCL implementation!

”Open”?

Means open specification

Like OpenGL

Many providers making their own implementation

There is not one OpenCL library.

No free lunch

Model does not fit all architectures

One size fits all - platform dependent optimizations hard to do

OpenCL for GPU Computing

Mostly similar to CUDA both in architecture and performance!

Messy setup - but you get used to it

Kernels similar to CUDA

Easier for NVidia to be first with new features

OpenCL vs CUDA terminology

OpenCL	CUDA
compute unit	multiprocessor (SM)
work item	thread
work group	block
local memory	shared memory
private memory	registers

And CUDA local memory =?
OpenCL local memory (= CUDA shared memory)

Oh, that "local memory"...

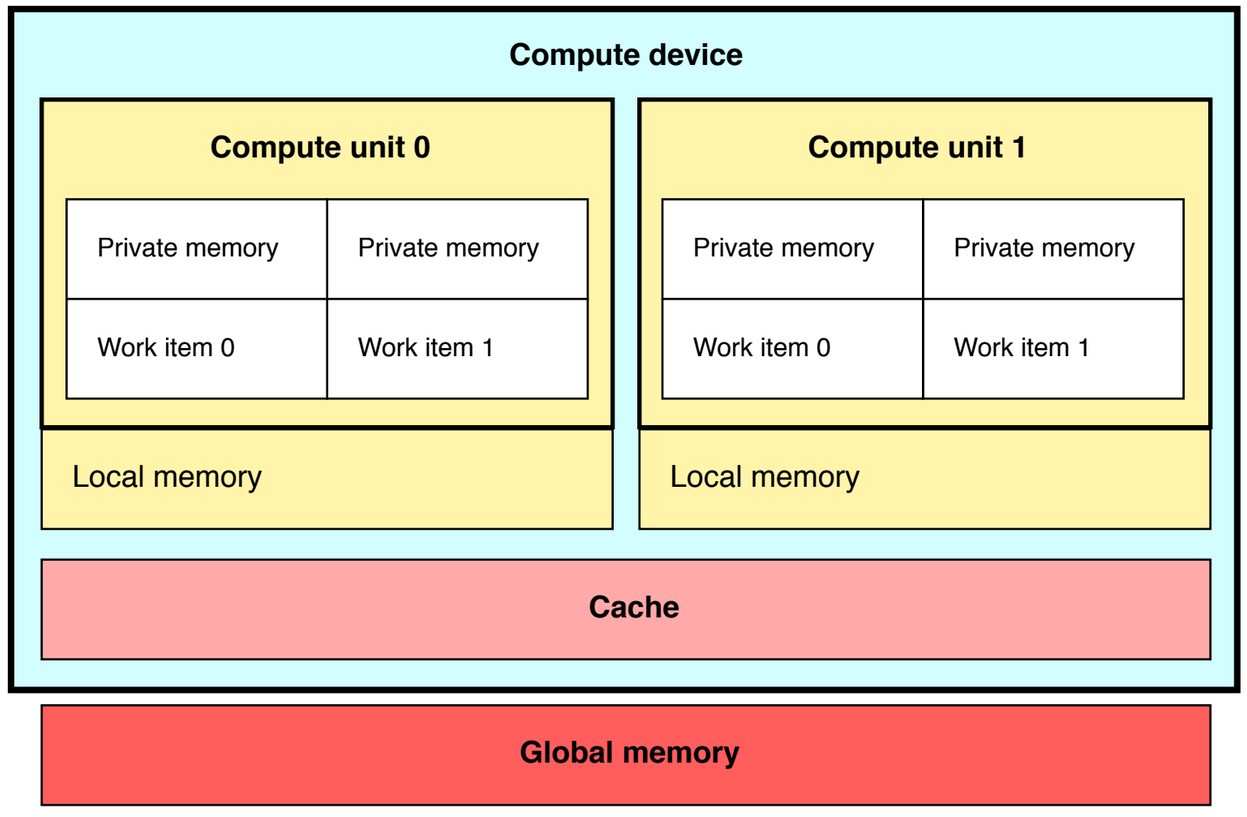
CUDA local memory = global memory accessible only by one thread (like registers but slower)

CUDA shared memory = OpenCL local memory = memory local inside the SM, shared within block/work group

Anyone else who thinks this makes sense?

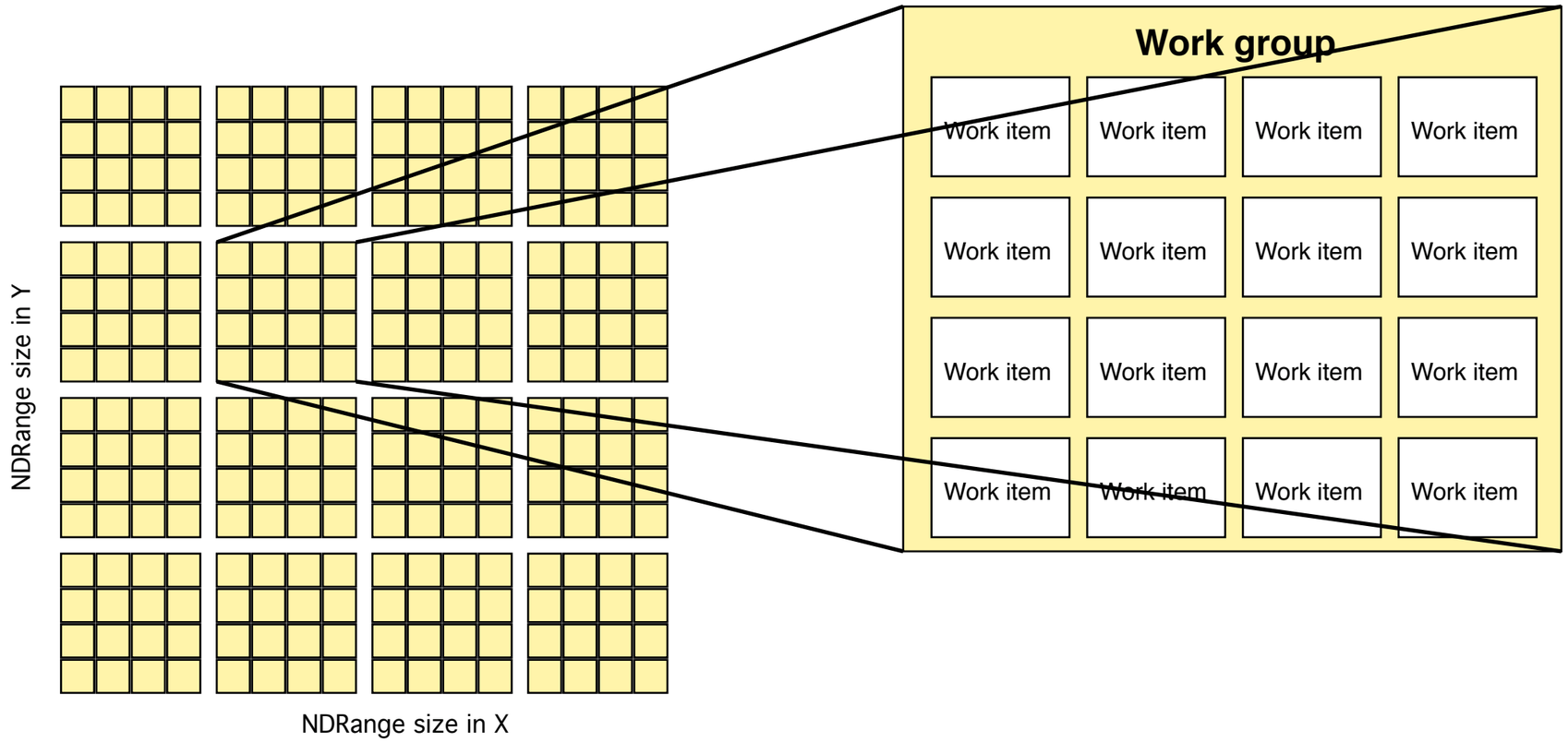
Images removed

OpenCL memory model



Been there, done that...

OpenCL execution model



Anyone who see ”blocks” and ”threads”?

Synchronization

Kernels can synchronize within a work group:

`barrier(CLK_LOCAL_MEM_FENCE)`

Synchronizes memory access. You choose which kind of memory access to synchronize (global, local).

The host (CPU) can synchronize on global level:

Available for:
tasks (e.g. clEnqueueNDRangeKernel)
Memory(e.g.clEnqueueReadBuffer)
events (e.g. clWaitforEvents)

Heterogenous

Some differences from CUDA: Designed for heterogenous systems!

Several devices may be active at once

You can specify which device to launch a task to

Query devices and device characteristics

Some overhead compared to CUDA, and the reward is flexibility!

Hello World!

More complex setup

Manual, somewhat tedious passing of data

No language extensions!

Similar kernel

Example using local (shared) memory:

* Rank sorting in sorting OpenCL

```
__kernel void sort(__global unsigned int
*data, __global unsigned int *outdata,
const unsigned int length)
{
    unsigned int pos = 0;
    unsigned int i, b;
    unsigned int val;
    unsigned int this;

    unsigned int __local buf[128];

    // loop until all data is covered

    this = data[get_global_id(0)];

    for (b = 0; b < length; b += 128)
    {
        // Get data
        buf[get_local_id(0)] = data[get_local_id(0) + b];

        // Synch
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        //find out how many values are smaller
        for (i = 0; i < 128; i++)
            if (this > buf[i]) // data[b + i])
                pos++;

        // Synch
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
    }

    outdata[pos] = this;
}
```

How about that setup?

1) Get a list of platforms

2) Choose a platform

3) Get a list of devices

4) Choose a device

5) Create a context

6) Load and compile kernel code

Then we can start working

7) Allocate memory

8) Copy data to device

9) Run kernel

10) Wait for kernel to complete

11) Read data from device

12) Free resources

1-5: Where to run

Simplified here - might fail!

```
cl_platform_id platform;
unsigned int no_plat;
err =  clGetPlatformIDs(1,&platform,&no_plat);

// Where to run
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
if (err != CL_SUCCESS) return -1;
```

Context

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
if (!context) return -1;
commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands) return -1;
```

6: Kernel

```
    // What to run
program = clCreateProgramWithSource(context,
1, (const char **) & KernelSource, NULL, &err);
if (!program) return -1;

err = clBuildProgram(program, 0, NULL, NULL,
NULL, NULL);
if (err != CL_SUCCESS) return -1;
kernel = clCreateKernel(program, "hello", &err);
if (!kernel || err != CL_SUCCESS) return -1;
```

```
const char *KernelSource = "\n" \
"__kernel void hello(              \n" \
"   __global char* a,          \n" \
"   __global char* b,          \n" \
"   __global char* c,          \n" \
"   const unsigned int count)  \n" \
"{                             \n" \
"   int i = get_global_id(0);  \n" \
"   if(i < count)              \n" \
"       c[i] = a[i] + b[i];    \n" \
"}                             \n" \
"\n";
```

Most programs load kernels from files

7-8: Get the data in there

```
// Create space for data and copy a and b to device
// (note that we could also use clEnqueueWriteBuffer to upload)
input = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
sizeof(char) * DATA_SIZE, a, NULL);
input2 = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
sizeof(char) * DATA_SIZE, b, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(char) * DATA_SIZE,
NULL, NULL);
if (!input || !output) return -1;

// Send data
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &input2);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
if (err != CL_SUCCESS) return -1;
```

9-10: Run kernel, wait for completion

```
// Run kernel!
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global,
&local, 0, NULL, NULL);

if (err != CL_SUCCESS) return -1;

clFinish(commands);
```

11-12: Read back data, release

```
// Read result
err = clEnqueueReadBuffer( commands, output, CL_TRUE, 0, sizeof(char) * count,
c, 0, NULL, NULL );
if (err != CL_SUCCESS) return -1;

//  Print result
printf("%s\n", c);

// Clean up
clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);
```

”Platform” vs ”device”

Platform = an OpenCL implementation

Device = a chip which the platform supports

Language freedom... sort of

+ Very easy to call from any language! Anything that can call into a C API can call OpenCL!

+ Based on C99. Similar to CUDA.

- Kernel code is only C-style (although a specific implementation may choose to support more). C++ in 2.2.

Performance

Investigations report remarkably small differences

Our research on FFT so far has CUDA up to 2x faster

Very hard to compare, due to multiple OpenCL implementations

Some report CUDA to be better on NVidia platforms... some report a draw even there.

Our experience: Usually very close!

Conclusions on OpenCL

Don’t fear the complex setup phase! The rest is similar to CUDA.

Performance tend to be on par with CUDA or almost.

Speciality: heterogenous systems!

Compute shaders

The future of GPU computing or a late rip-off of Direct Compute?

Compute shaders

Previously a Microsoft concept, Direct Compute

Also in OpenGL since OpenGL 4.3

Why is this important?

Why use that instead of CUDA or OpenCL?

+ Better integration with OpenGL

+ No extra installation!

+ Easier to configure than OpenCL

+ Not NVidia specific like CUDA

+ If you know GLSL, Compute Shaders are (fairly) easy!

Not only plus...

- Some new concepts

- Not part of the main graphics pipeline like fragment shaders

- Some vendors lagging behind or has own solution (Apple)

Compute shaders run alone, not compiled together with others.

Image removed

Image removed

So how do I use it?

Compiled like other shaders!

Trivial change from the usual shader loader/compilers from graphics programs, just compile as GL_COMPUTE_SHADER.

Easy:

- Uniforms work as usual
- Textures work as usual

A bit different

No longer not one thread per fragment (output pixel)

Thereby: No thread specific output

Shader Storage Buffer Objects (SSBO):

General buffer type for arbitrary data

Can be declared as an array of structures

Read and written freely by Compute Shaders!

How do I upload input data?

Upload to SSBO:

```
glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, size, ptr,
GL_STATIC_DRAW);
```

How does the shader know?

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, id,
ssbo);
```

```
layout(std430, binding = id, buffer x {type y[];};
```

Access data in the shader

Set number of threads per block:

layout(local_size_x = width, local_size_y = height)

Thread number:

gl_GlobalInvocation
gl_LocalInvocation

```
void main()
{
  buffer[gl_GlobalInvocation.x] =
    - buffer[gl_GlobalInvocation.x];
}
```

Execute kernel

glUseProgram(program);

glDispatchCompute(sizex, sizey, sizez);

The arguments to glDispatchProgram set the number of blocks / workgroups. The number of threads (work items) per block are set by the shader.

Getting output data

```
glBindBuffer(GL_SHADER_STORAGE, ssbo);
ptr = (int *) glMapBuffer(GL_SHADER_STORAGE,
GL_READ_ONLY);
```

Then read from ptr[i]

```
glUnmapBuffer(GL_SHADER_STORAGE);
```

Complete main program:

```
int main(int argc, char **argv)
{
  glutInit (&argc, argv);
  glutCreateWindow("TEST1");

// Load and compile the compute shader
  GLuint p =loadShader("cs.csh");

  GLuint ssbo; //Shader Storage Buffer Object

  // Some data
  int buf[16] = {1, 2, -3, 4, 5, -6, 7, 8, 9,
                                    10, 11, 12, 13, 14, 15, 16};
  int *ptr;

// Create buffer, upload data
  glGenBuffers(1, &ssbo);
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
  glBufferData(GL_SHADER_STORAGE_BUFFER,
      16 * sizeof(int), &buf, GL_STATIC_DRAW);

// Tell it where the input goes!
// "5" matches "layuot" in the shader.

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
         5, ssbo);

// Get rolling!
    glDispatchCompute(16, 1, 1);

// Get data back!
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
  ptr = (int *)glMapBuffer(
         GL_SHADER_STORAGE_BUFFER,
         GL_READ_ONLY);
  for (int i=0; i < 16; i++)
  {
    printf("%d\n", ptr[i]);
  }
}
```

Simple Compute Shader:

```
#version 430
#define width 16
#define height 16

// Compute shader invocations in each work group

layout(std430, binding = 5) buffer bbs {int bs[];};

layout(local_size_x=width, local_size_y=height) in;

//Kernel Program
void main()
{
  int i = int(gl_LocalInvocationID.x * 2);
  bs[gl_LocalInvocationID.x] = -bs[gl_LocalInvocationID.x];
}
```

Note: Too many threads for data (16*16*16)

List of variables for identifying thread location in computation:

gl_NumWorkGroups
gl_WorkGroupID
gl_WorkGroupSize
gl_LocalInvocationID
gl_GlobalInvocaionID
gl_LocalInvocationIndex

All are 3-dimensional except the last, which is a convenience integer:

gl_LocalInvocationIndex= gl_LocalInvocationID.z * gl_WorkGroupSize.x * gl_WorkGroupSize.y + gl_LocalInvocationID.y * gl_WorkGroupSize.x + gl_LocalInvocationID.x

Example with shared memory:

```
#version 450
#extension GL_ARB_compute_shader : enable
#define width 16
#define height 1

// Compute shader invocations in each work group

layout(std430, binding = 7) buffer outbuf {float c[];};
layout(std430, binding = 5) buffer bufc {float a[];};
layout(local_size_x=width, local_size_y=height) in;
```

```
ingemar@Trixie:~/Dokument/maxa$ ./maxa
Vendor: Intel Open Source Technology Center
Renderer: Mesa DRI Intel(R) HD Graphics 4400 (HSW GT2)
Version: 4.5 (Core Profile) Mesa 21.0.3
GLSL: 4.50
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
79 79 79 79 79 79 79 79 79 79 79 79 79 79 79 79
95 95 95 95 95 95 95 95 95 95 95 95 95 95 95 95
111 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111
127 127 127 127 127 127 127 127 127 127 127 127 127 127 127 127
```

```
//Kernel Program
void main()
{
    shared float sa[16];
    sa[gl_LocalInvocationID.x] =
a[gl_GlobalInvocationID.x];
    // synchronize
        barrier();

        float maxa = 0;
        for (int i = 0; i < 16; i++)
        {
            maxa = max(maxa, sa[i]);
        }

    c[gl_GlobalInvocationID.x] = maxa;
}
```

Textures as in- and output data

Integration with OpenGL's pipeline

An important strength with Compute Shaders: Better integration with OpenGL.

Integration through texture

1) Create a texture with OpenGL

2) Bind it as floating-point texture

3) Send the texture unit to the compute shader

4) glDispatchCompute to run

5) Use imageStore in the shader to write the texture

6) Use the texture as desired

OpenGL Compute Shaders supported for NVidia and AMD since the start. Later also supported in

GLES 3.1 (embedded systems!)

MESA for Intel GPUs (Haswell)

but still not on Macs...

Are Compute Shaders an alternative?

- Portable between GPUs and OSes
- Easy installation, as easy as with fragment shaders
- Steep hardware demands less and less a problem

All advantages?

Let's not forget Direct Compute

- Its own shader language (HLSL)
- Microsoft only
- Similar to OpenCL in setup. A bit messy?
- Close to graphics code

	Portable	Features	Install	Code
CUDA	Weak	Great	Weak	Great
OpenCL	Great	Good	Weak	OK
GLSL Fragment shaders	Great	Weak	Great	Messy
GLSL Compute shaders	Great	Good	Great	OK
DC Compute shaders	Weak	Good	Great	OK

But how about the performance???

Some comparisons

One old project: CUDA vs GLSL vs OpenCL,
compared with a mass-spring system

One recent project: Multiple platforms,
compared with similar FFT implementation

Mass-spring system

by Marco Fratarcangeli

Part of my GPU computing PhD course many years ago.

Published in "Game Engine Gems 2"

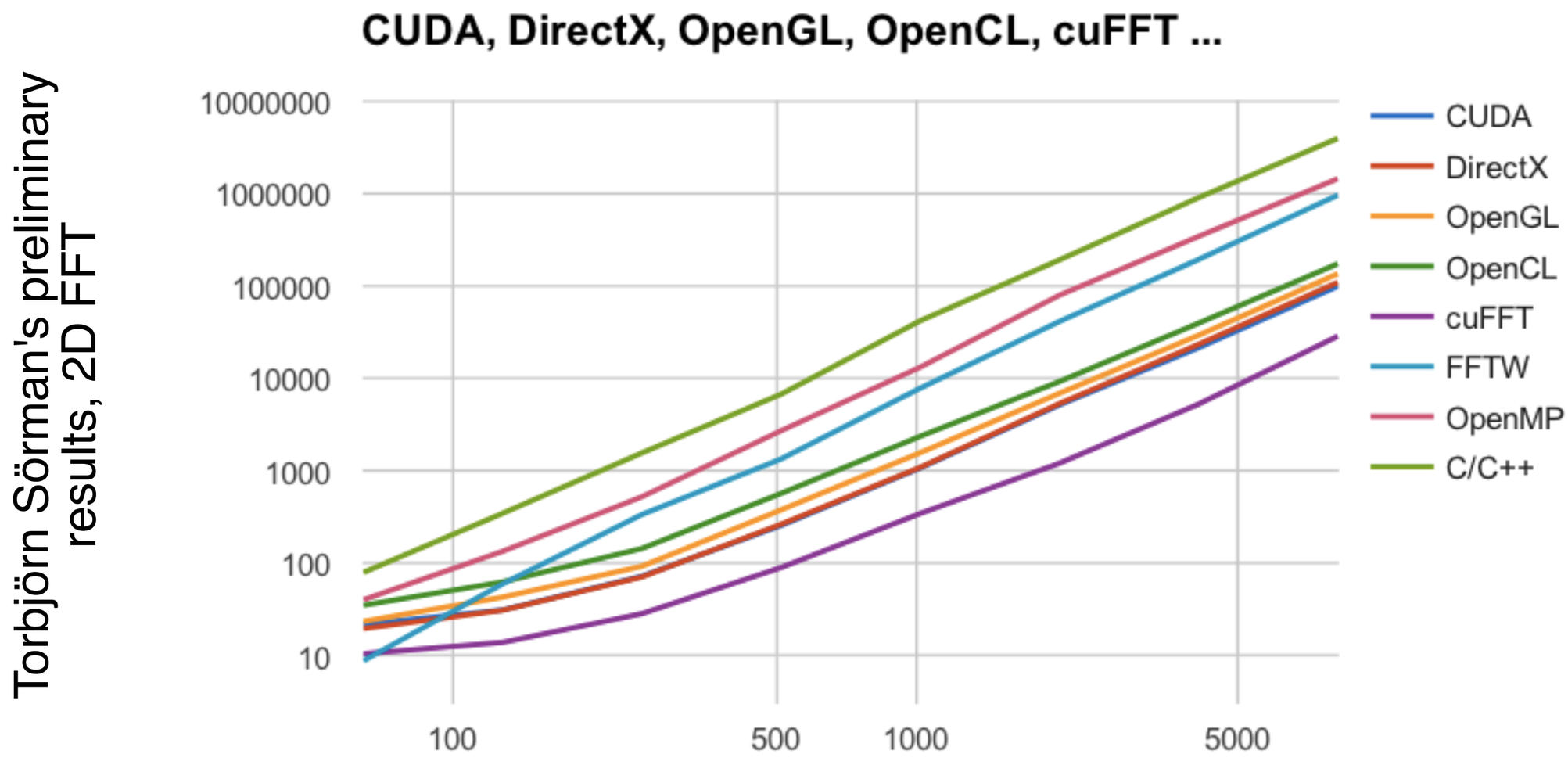
Result: CUDA and GLSL almost the same, OpenCL noticably behind.

"FFT everywhere" project

by Torbjörn Sörman

Mode recent diploma thesis project.

Some interesting results.


CUDA, DirectX, OpenGL, OpenCL, cuFFT ...
Torbjörn Sörman's preliminary results, 2D FFT
10000000
1000000
100000
10000
1000
100
10
100
500
1000
5000
CUDA
DirectX
OpenGL
OpenCL
cuFFT
FFTW
OpenMP
C/C++

Torbjörn Sörman's results

• cuFFT so much faster that it is scary...
• Torbjörn's own GPU implementations much faster than CPU versions
• On NVidia, CUDA and Direct Compute significantly faster than OpenGL Compute Shaders and OpenCL
• On AMD, Direct Compute, OpenCL and OpenGL Compute Shaders ran side-by-side

Lots of if's and but's... but two clear conclusions:

• Hard optimization (cuFFT and FFTW) pays, and not just by a little!
• OpenCL and Compute Shaders very close - basically the same?

The new OpenGL - also the new open parallel computing platform?

Will it step in and take over?

- Cross-platform
- Built for both graphics and general-purpose computations
- Not suitable beginner's platform

Image removed

So how do I do GPU computing with Vulkan?

Simple: Uses GLSL Compute Shaders!

All I said about Compute Shaders are true for Vulkan, except that the host looks different!

GPU computing conclusions

The desktop supercomputer

Fast changing area

Great performance for big problems that fit the architecture

Good performance for many other problems