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!

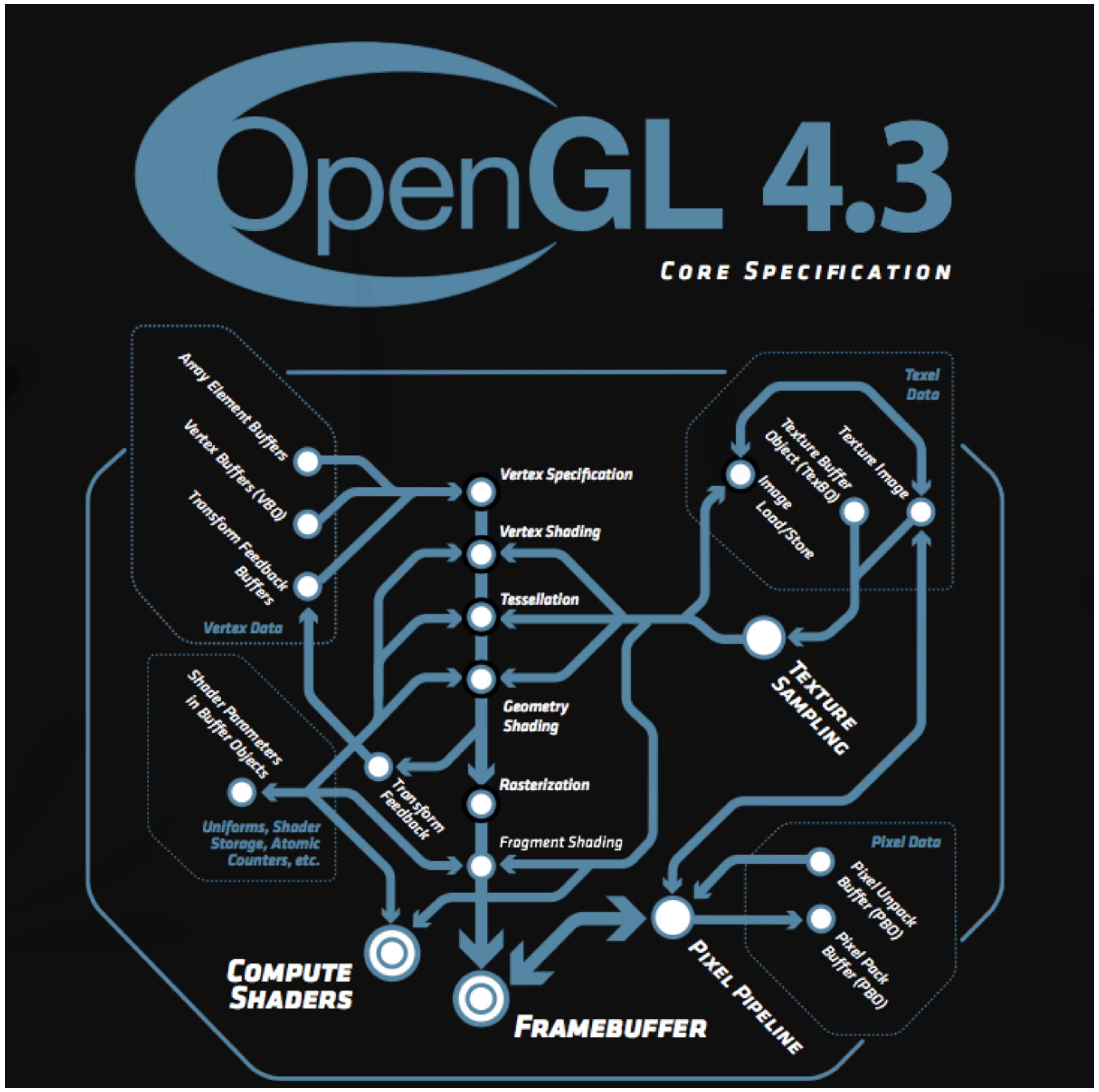
Not only plus...

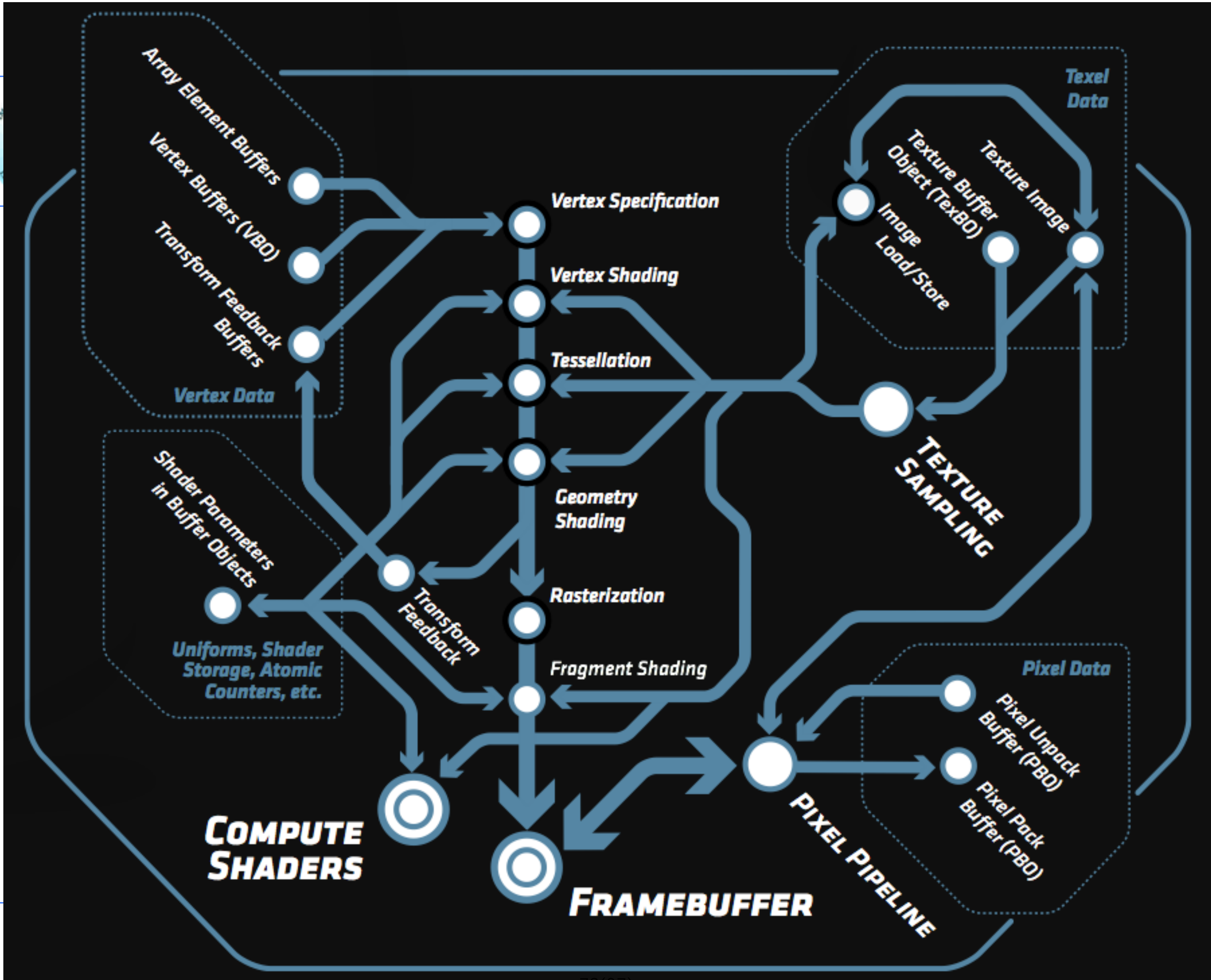
- Some new concepts

- Not part of the main graphics pipeline like fragment shaders

- Some vendors (Apple) lagging behind

Compute shaders run alone, not compiled together with others.


OpenGL 4.3
Core Specification
Array Element Buffers
Vertex Buffers (VBO)
Transform Feedback Buffers
Vertex Data
Vertex Specification
Vertex Shading
Tessellation
Geometry Shading
Rasterization
Fragment Shading
Texel Data
Texture Buffer Object (TexBO)
Texture Image
Image Load/Store
Texture Sampling
Shader Parameters in Buffer Objects
Uniforms, Shader Storage, Atomic Counters, etc.
Transform Feedback
Pixel Data
Pixel Unpack Buffer (PBO)
Pixel Pack Buffer (PBO)
Compute Shaders
Framebuffer
Pixel Pipeline



Array Element Buffers
Vertex Buffers (VBO)
Transform Feedback Buffers
Vertex Data
Vertex Specification
Vertex Shading
Tessellation
Geometry Shading
Rasterization
Fragment Shading
Texel Data
Texture Buffer Object (TexBO)
Texture Image
Image Load/Store
Texture Sampling
Shader Parameters in Buffer Objects
Uniforms, Shader Storage, Atomic Counters, etc.
Transform Feedback
Compute Shaders
Framebuffer
Pixel Pipeline
Pixel Data
Pixel Unpack Buffer (PBO)
Pixel Pack Buffer (PBO)

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:

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

```
#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];
}
```

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;
}
```

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
- 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.

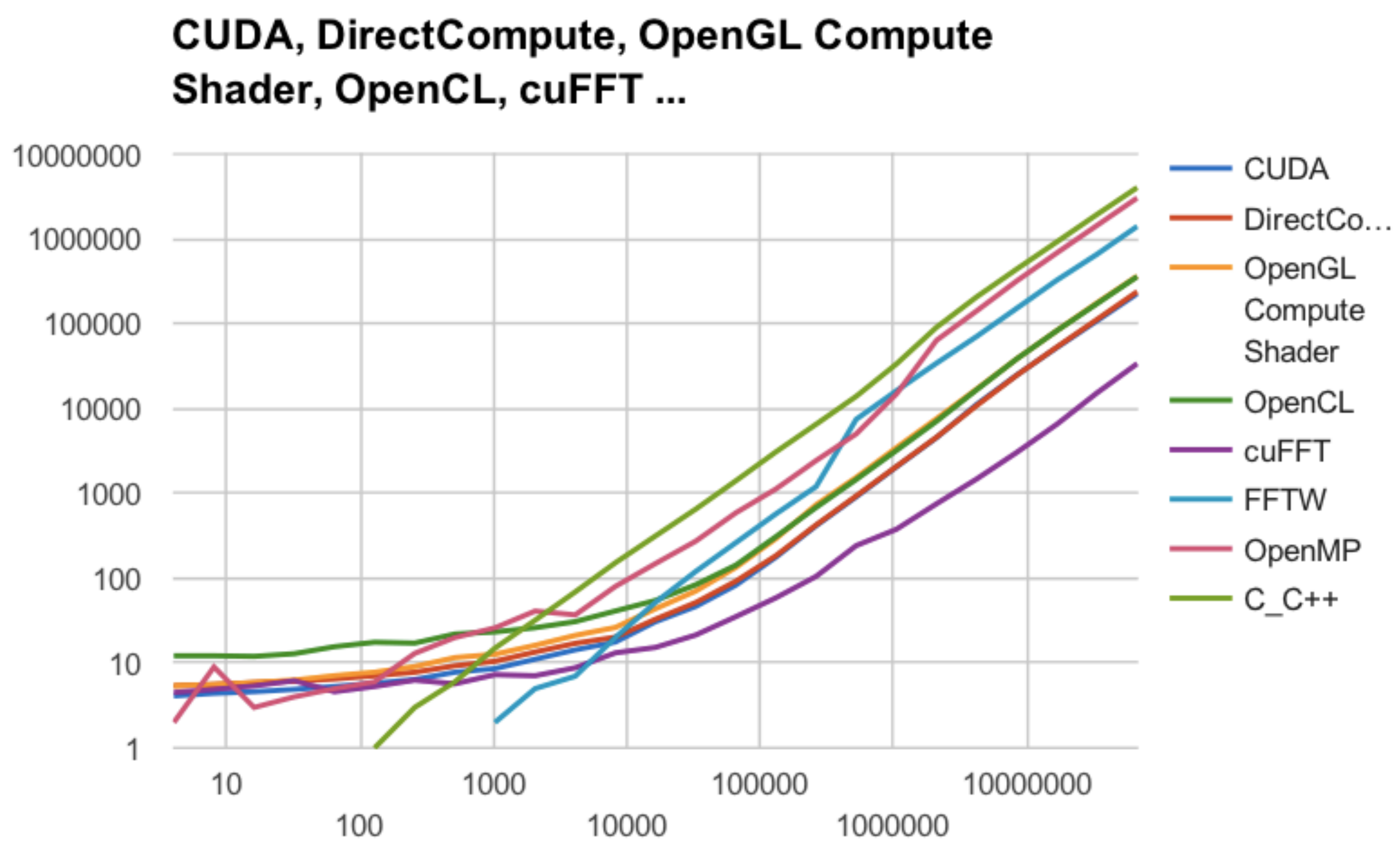
"FFT everywhere" project

by Torbjörn Sörman

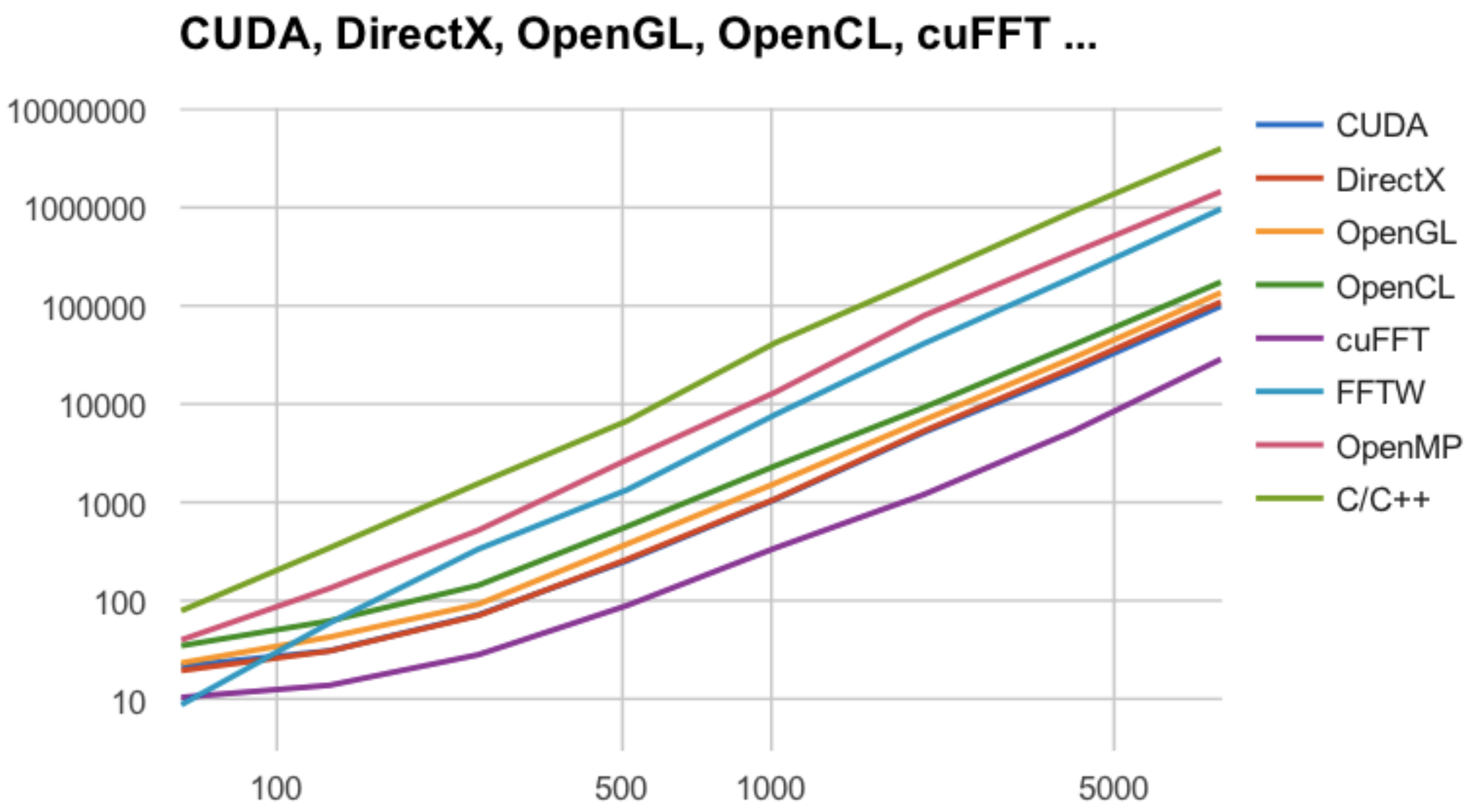
Recent diploma thesis project.

Some interesting results.

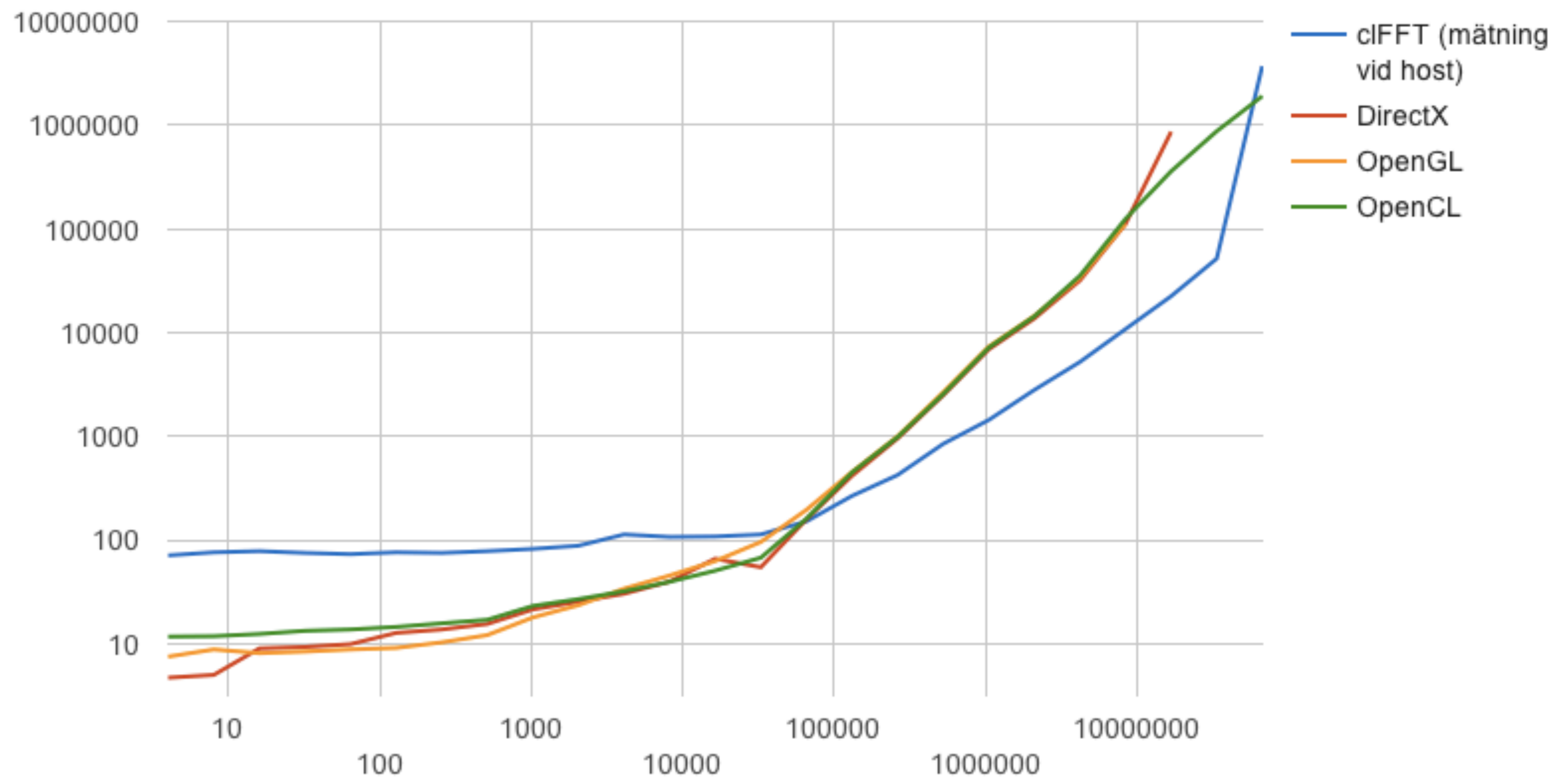
Torbjörn Sörman's preliminary results, 1D FFT



Torbjörn Sörman's preliminary results, 2D FFT



Torbjörn Sörman's preliminary results, 1D FFT, AMD



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

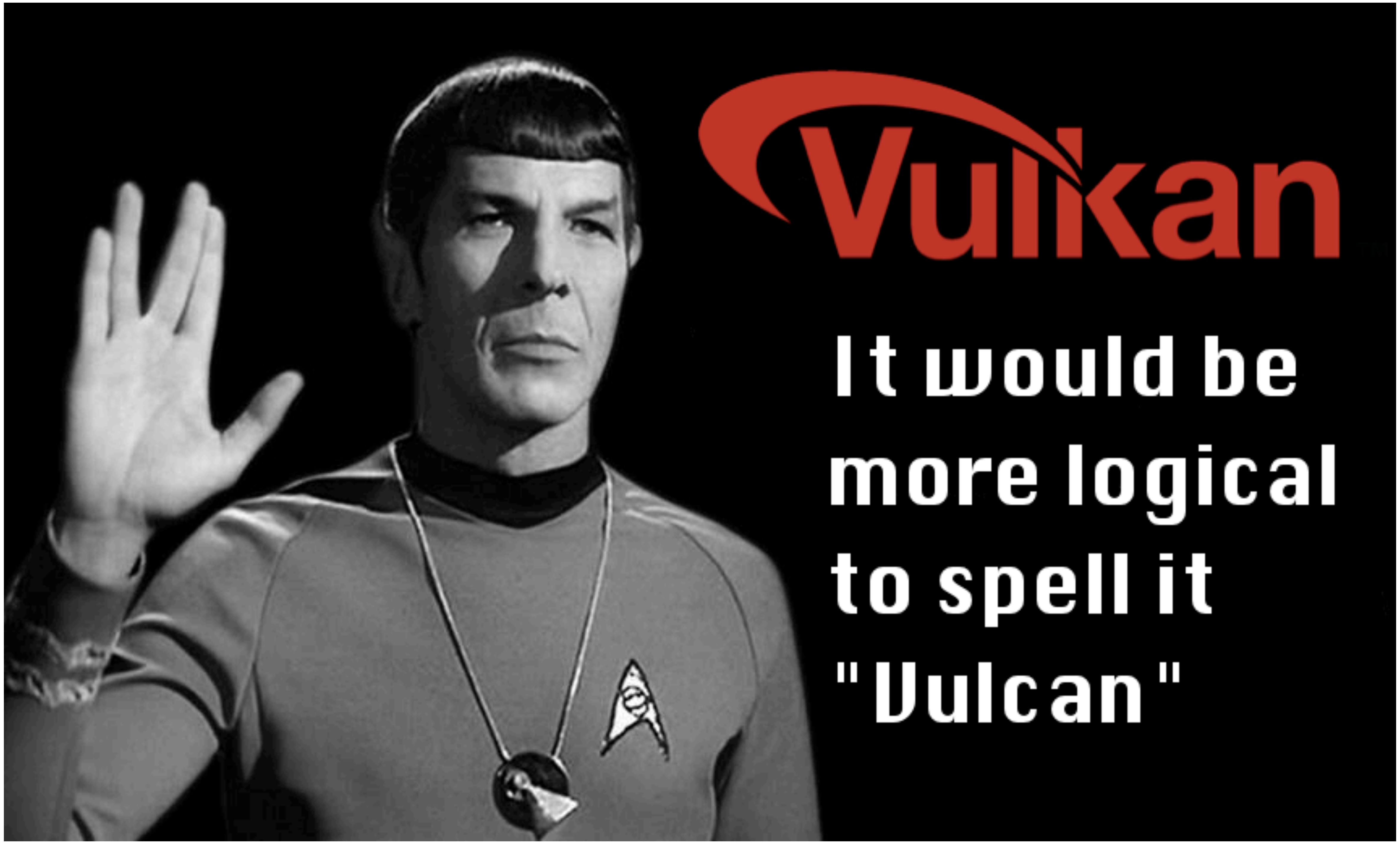
Lots of if's and but's... but two clear conclusions:

• 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**


Vulkan
It would be
more logical
to spell it
"Vulcan"

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