# Compute shaders

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

# Compute shaders

**Previously a Microsoft concept, Direct Compute**

**Now also in OpenGL, new kind of shader since the recent OpenGL 4.3**

**Kind of ”Bleeding edge”; even today since 4.3 is not fully universal**

# 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...

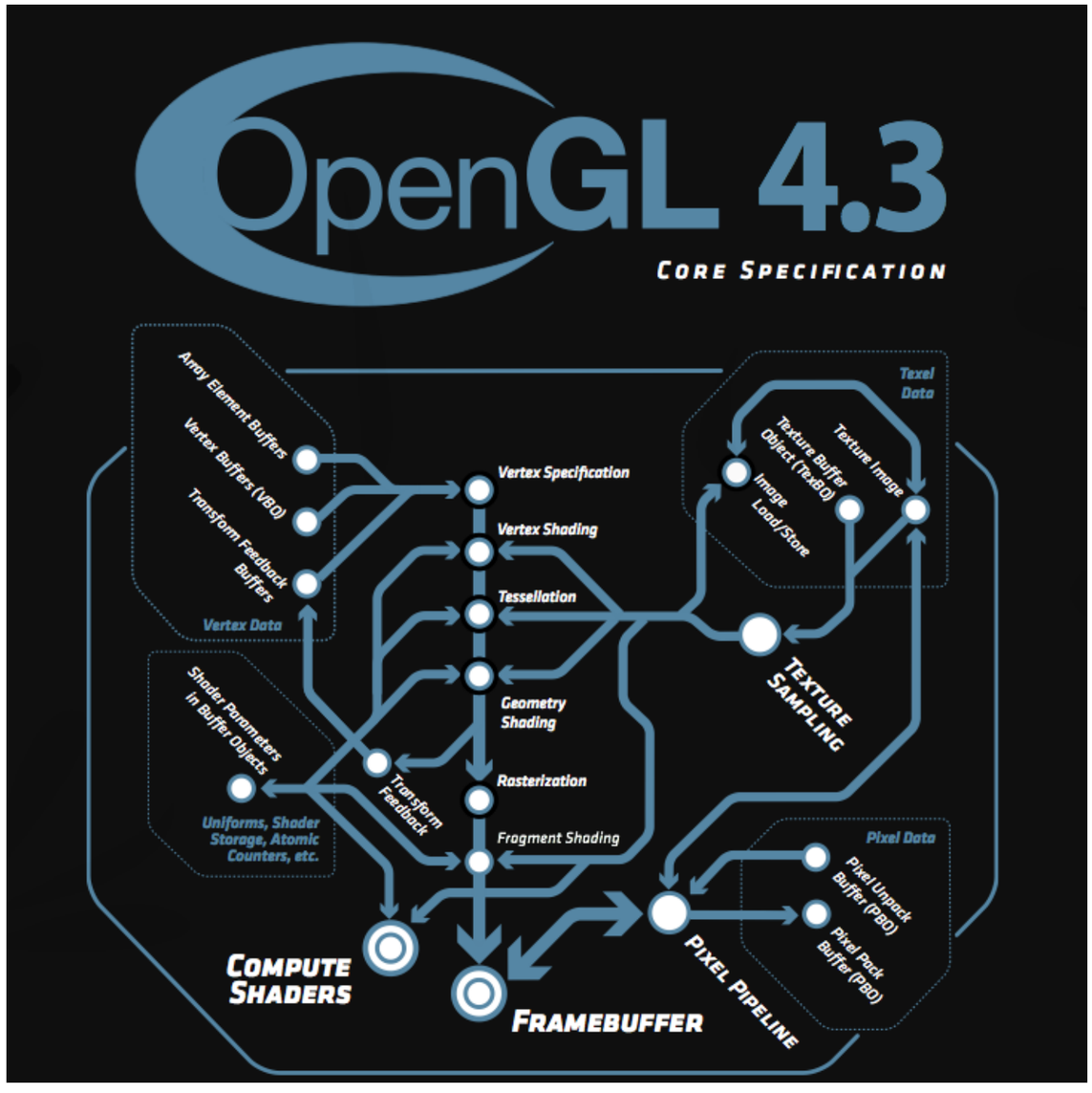
**- Modest hardware demands: Kepler + 4.3**

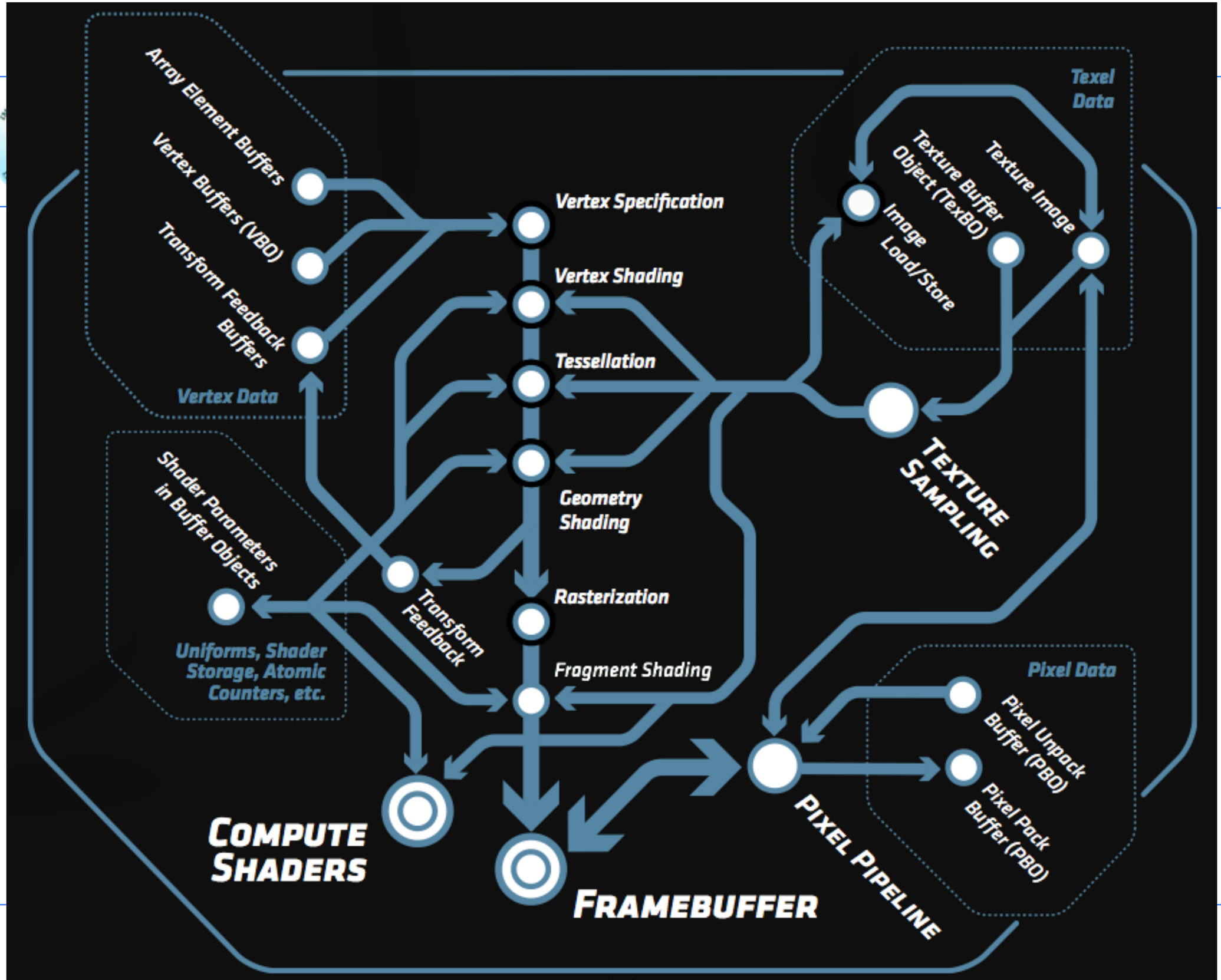
**- Some new concepts**

**- 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
Shader Parameters in Buffer Objects
Uniforms, Shader Storage, Atomic Counters, etc.
Transform Feedback
Texel Data
Texture Buffer Object (TexBO)
Texture Image
Image Load/Store
TEXTURE SAMPLING
Pixel Data
Pixel Unpack Buffer (PBO)
Pixel Pack Buffer (PBO)
PIXEL PIPELINE
COMPUTE SHADERS
FRAMEBUFFER


Array Element Buffers
Vertex Buffers (VBO)
Transform Feedback Buffers
Vertex Data
Vertex Specification
Vertex Shading
Tessellation
Geometry Shading
Rasterization
Transform Feedback
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.
Pixel Data
Pixel Unpack Buffer (PBO)
Pixel Pack Buffer (PBO)
COMPUTE SHADERS
PIXEL PIPELINE
FRAMEBUFFER

# 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)

**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 a few years ago.**

**Published in "Game Engine Gems 2"**

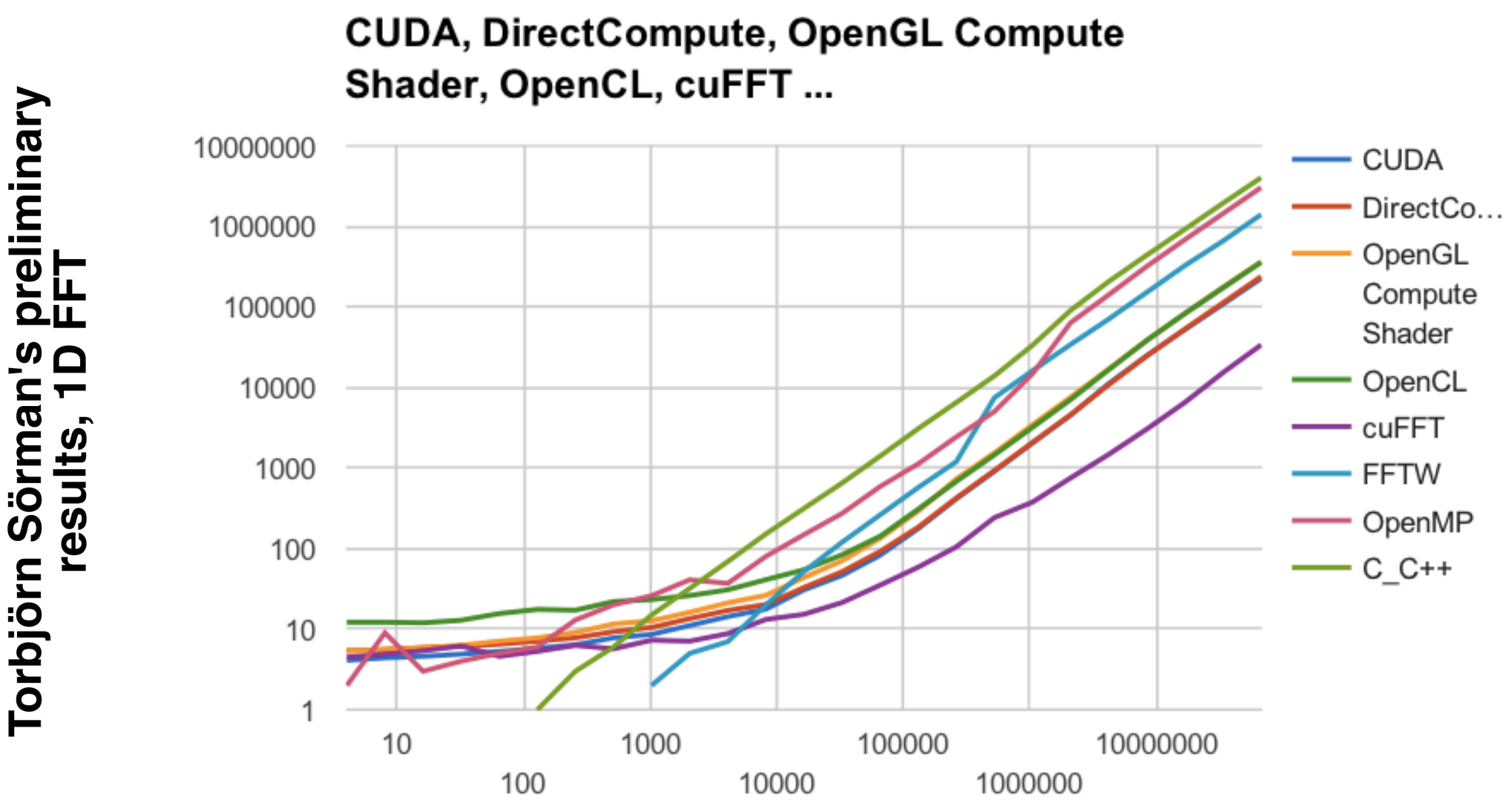
**Result: CUDA and GLSL almost the same, OpenCL noticably behind.**

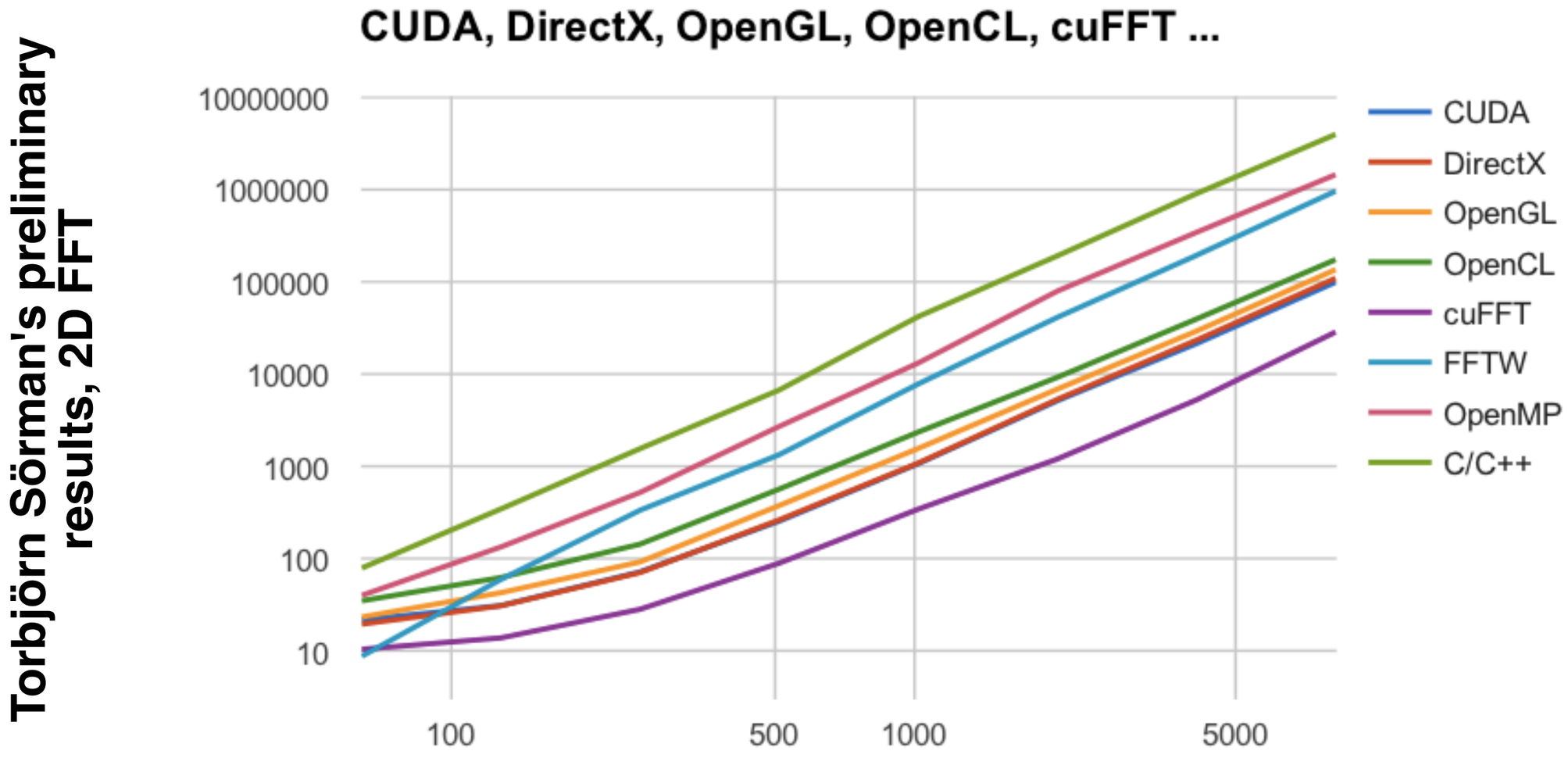
# "FFT everywhere" project

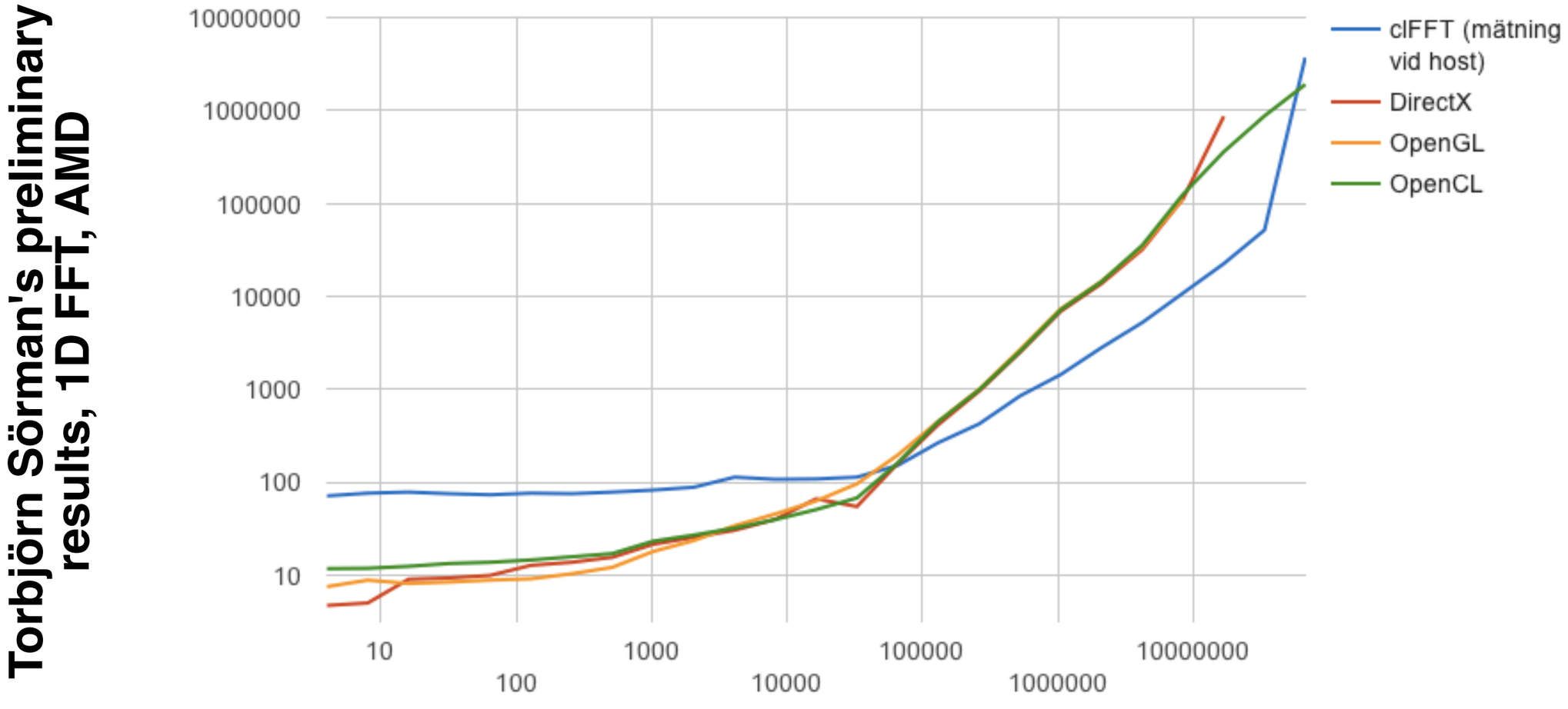
**by Torbjörn Sörman**

**Recent diploma thesis project.**

**Some interesting results.**


Torbjörn Sörman's preliminary results, 1D FFT
CUDA, DirectCompute, OpenGL Compute Shader, OpenCL, cuFFT ...
10000000
1000000
100000
10000
1000
100
10
1
10
100
1000
10000
100000
1000000
10000000
CUDA
DirectCo…
OpenGL Compute Shader
OpenCL
cuFFT
FFTW
OpenMP
C_C++


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 preliminary results, 1D FFT, AMD
10000000
1000000
100000
10000
1000
100
10
10
100
1000
10000
100000
1000000
10000000
clFFT (mätning vid host)
DirectX
OpenGL
OpenCL

# 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

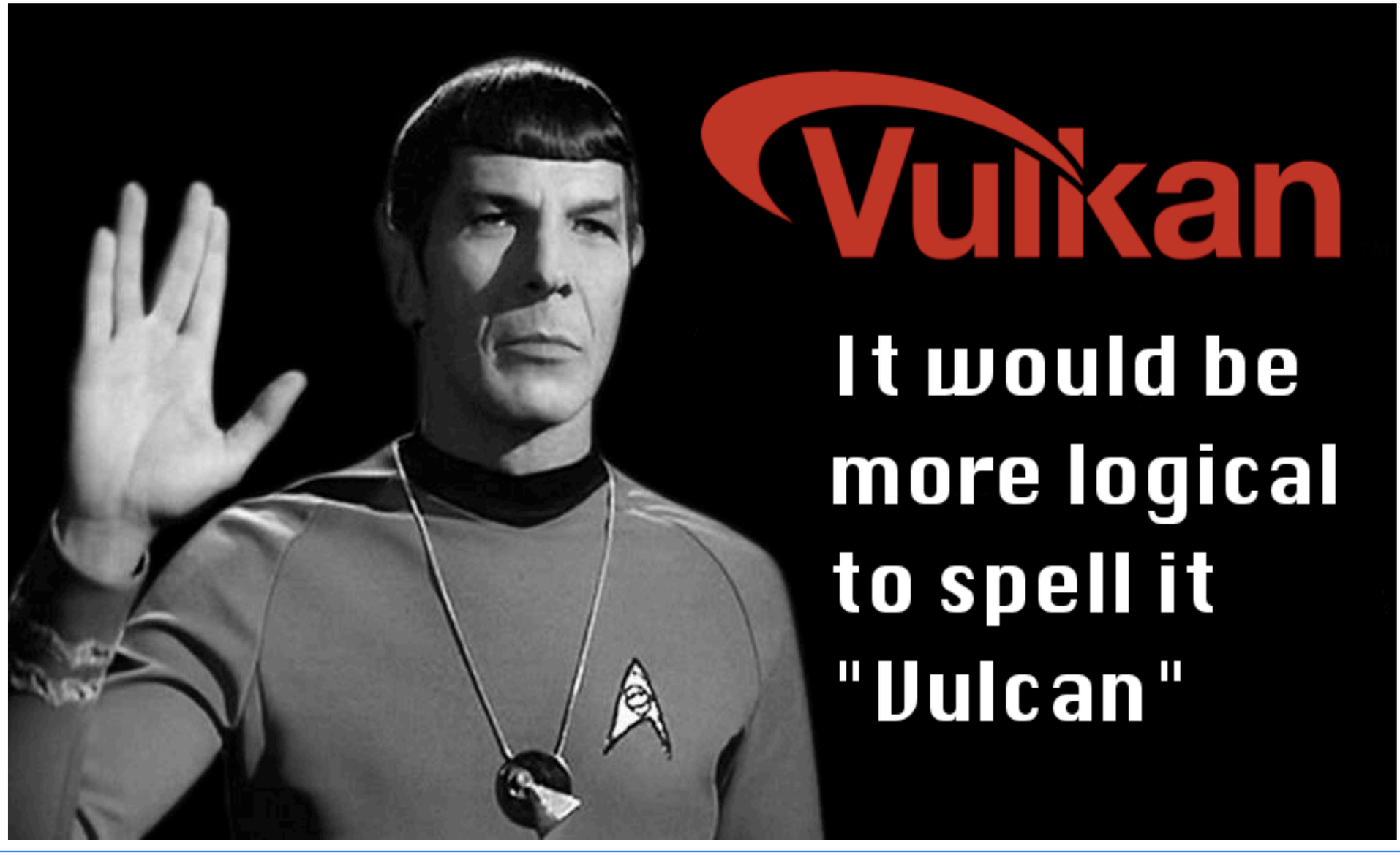
Lots of if's and but's... but two clear conclusions:

- 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