

A wireframe landscape rendered in white lines on a black background. The foreground features a grid of squares that recede into the distance. In the background, there are jagged, mountain-like shapes. The sky is filled with numerous small, white dots representing stars.

GPU Computing

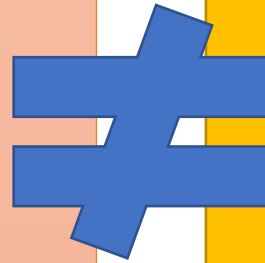
OpenGL - Compute Shader

Graphics vs Compute?



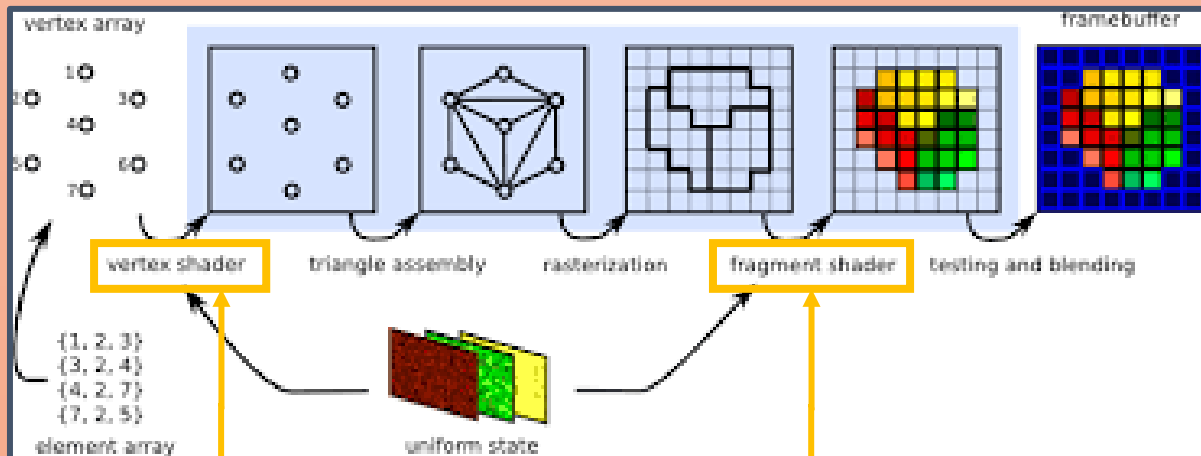
Pipeline graphique

Automatic threads management



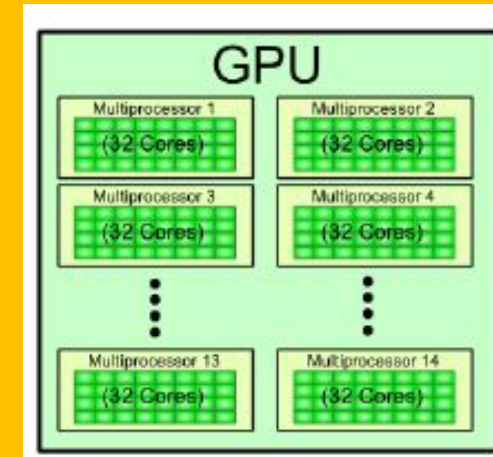
GPU Computing

Manual threads management
(user-defined blocks of threads)



Vertex shader

Fragment shader



INPUTS
(generic)

Buffers (array)
Images (textures)

Compute shader

OUTPUTS
(generic)

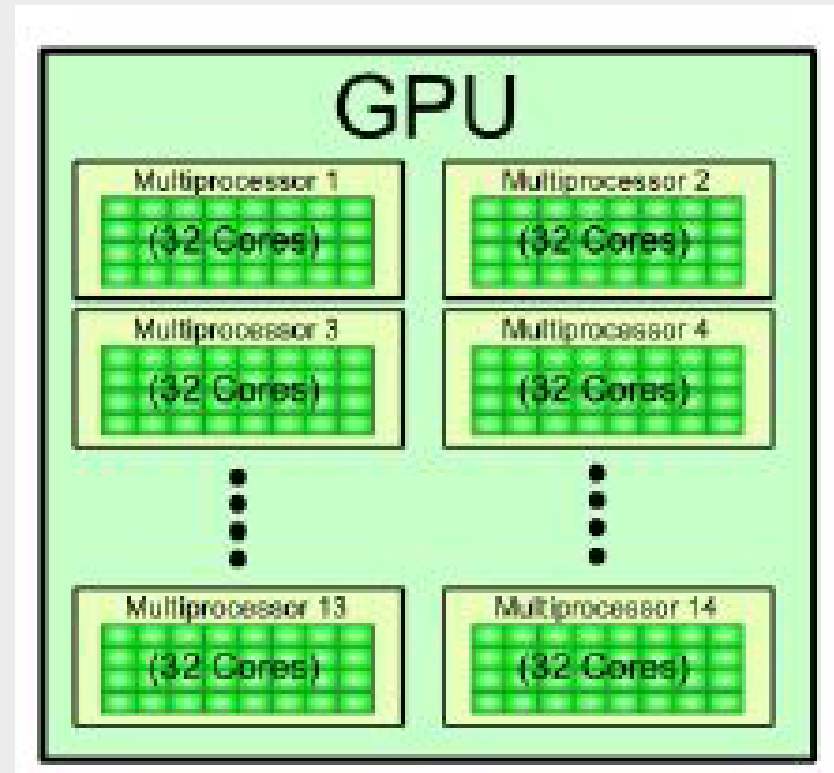
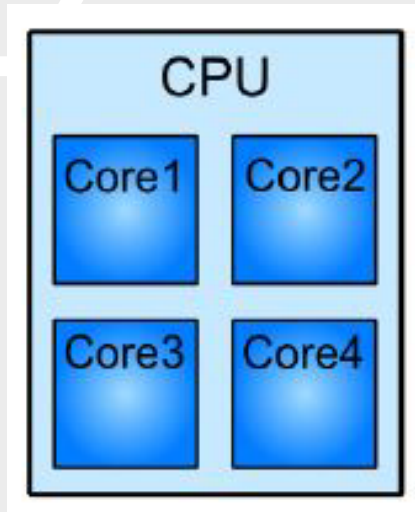
Buffers (array)
Images (textures)

CUDA

Let's look at CUDA to get context

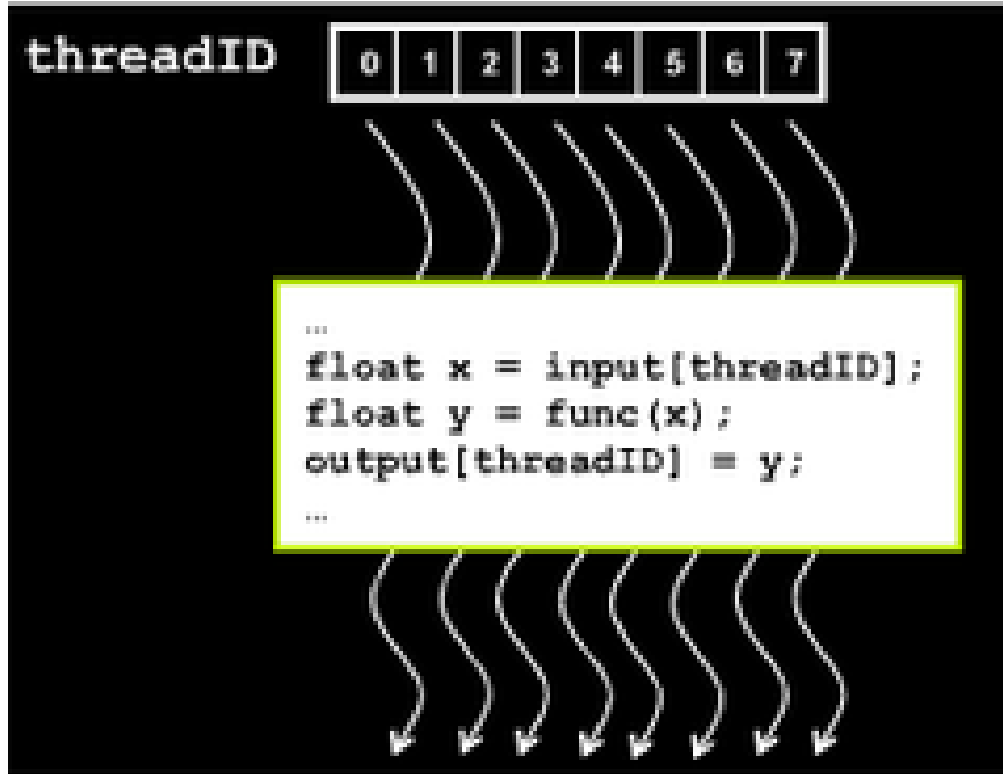
GPU Computing

CUDA (Compute Unified Device Architecture)



GPU Computing in CUDA

User launches “kernel” on GPU, executed by threads in parallel (equivalent to “thread function” on CPU for Python, C++, ...)

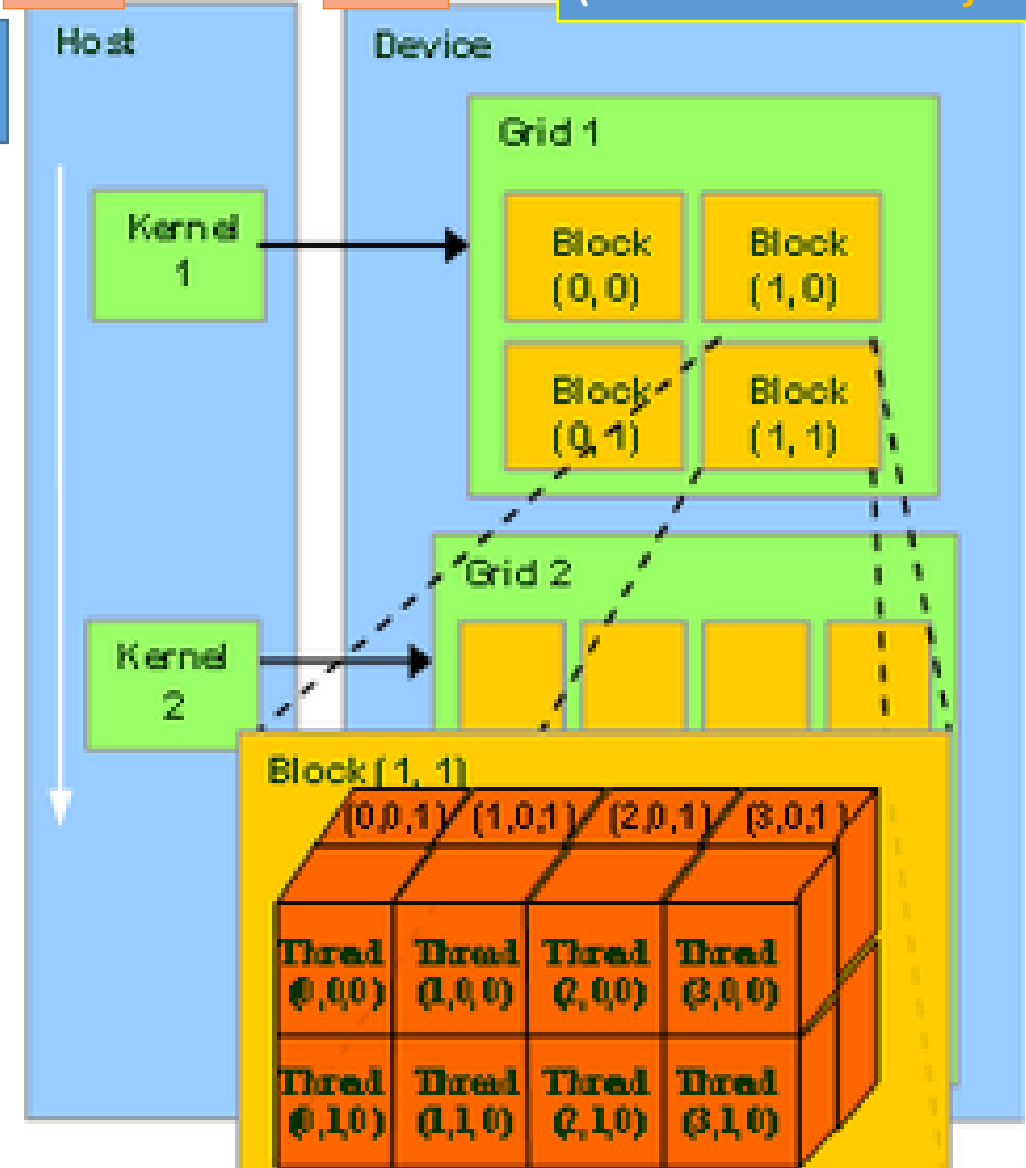


As in Python, C++, ... input and output data is read and written according to a thread index. Here thread index is 3D, one can read any location, and write to any location.

CPU

GPU

Manual threads management
(user-defined *blocks of threads*)



CUDA : example of a grid of block of threads

Grid:
4 blocks of 4 threads

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Figure 5.1 A two-dimensional arrangement of a collection of blocks and threads

CUDA provides:

- “3D block dimension” (blockDim)
- “3D block index” (blockIdx)
- “3D thread index” (threadIdx)

`int tid = threadIdx.x + blockIdx.x * blockDim.x;`

=> same concepts in other API such as OpenCL, OpenGL, etc...

CUDA : basic example – introduction – main idea

3.2.1 HELLO, WORLD!

```
#include "../common/book.h"

int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```


CUDA : basic example – introduction – main idea

3.2.2 A KERNEL CALL

Now we will build upon our example with some code that should look more foreign than our plain-vanilla “Hello, World!” program.

```
#include <iostream>
```

```
__global__ void kernel( void ) {  
}
```

CUDA “kernel” executed on GPU
(equivalent to “thread function” in Python, C++...)

```
int main( void ) {  
    kernel<<<1,1>>>();  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

User launch kernel on GPU,
given a grid of blocks of threads
(this is “asynchronous” with CPU)

CUDA "kernel" executed on GPU

```
#include <iostream>
#include "book.h"
```

```
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}
```

Allocation Array on GPU

```
int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
```

Launch kernel on GPU

```
add<<<1,1>>>( 2, 7, dev_c );
```

Retrieve GPU data on PC

```
HANDLE_ERROR( cudaMemcpy( &c,
                          dev_c,
                          sizeof(int),
                          cudaMemcpyDeviceToHost ) );
```

copy direction:
GPU to CPU

Free memory on GPU

```
printf( "2 + 7 = %d\n", c );
cudaFree( dev_c );

return 0;
}
```

A wireframe landscape rendered in white lines on a black background. The foreground features a grid of squares that recedes into the distance. In the background, there are jagged, mountain-like shapes. The sky is filled with numerous small, white dots representing stars.

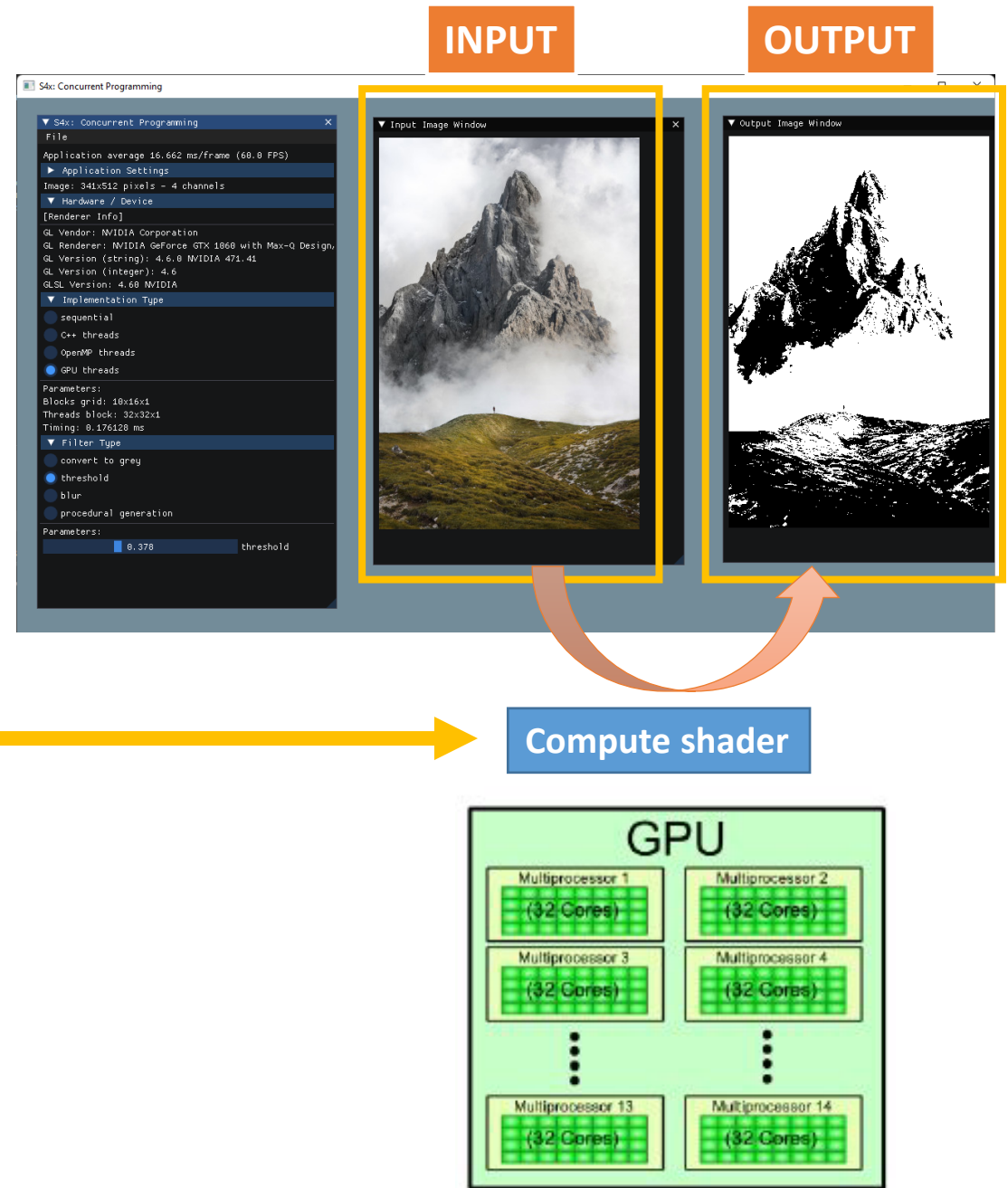
OpenGL

Compute Shader

OpenGL version ≥ 4.5

Les étapes principales

- Initialization
 - create shader
- Image loading
 - load image
 - create Input texture
 - create Output texture
- User action [ex : click bouton]
 - set current shader : `glUseProgram()`
 - launch kernel : `glDispatchCompute()`
 - save image
- Rendering
 - Display textures (images)

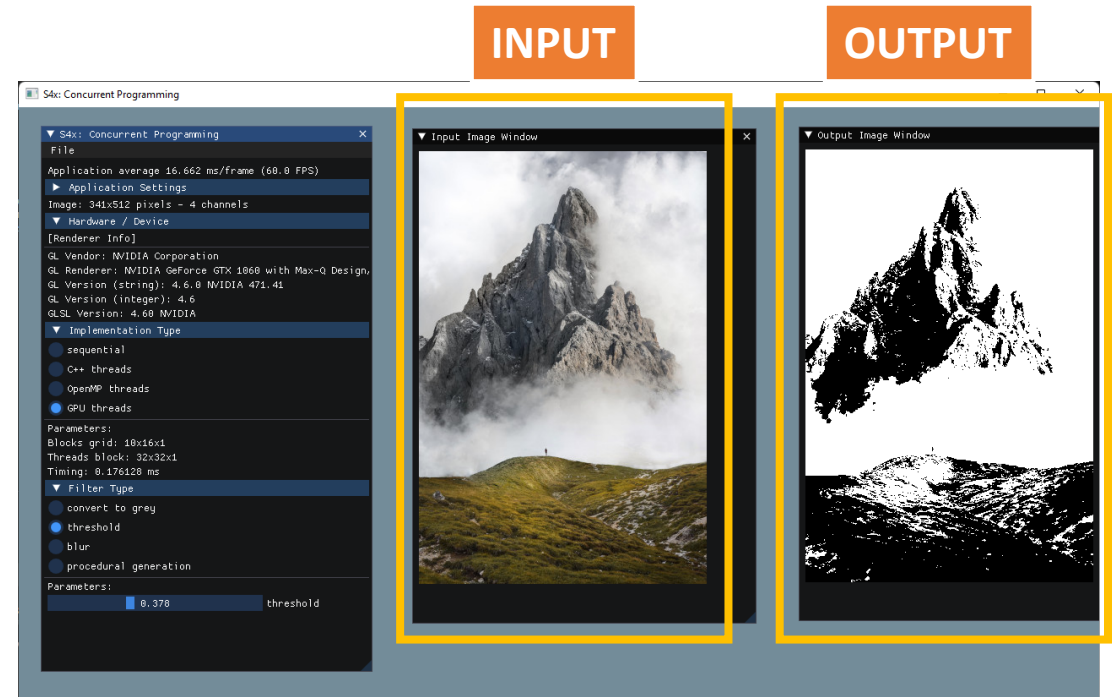


Detailed code



IMAGE LOADING & Texture creations

- Initialization
 - create shader
- Image loading
 - load image
 - create Input texture
 - create Output texture
- User action [ex : click bouton]
 - set current shader : `glUseProgram()`
 - launch kernel : `glDispatchCompute()`
 - save image
- Rendering
 - Display textures (images)



Read Image (STB library)

```
CAlgorithm* algorithm = new CAlgorithm();  
algorithm->readImage( "mountain.jpg" )
```



GLuint texture;

```
createInputTexture( texture, width, height,  
                    algorithm->getInputData() )
```

```
66  /***** texture: OpenGL texture ID that will be created *****/  
67  * Create input texture  
68  *****/  
69  bool createInputTexture( GLuint& texture, const unsigned int width, const unsigned int height, const std::vector< std::uint8_t >&  
70  {  
71      // Create texture  
72      glCreateTextures( GL_TEXTURE_2D, 1, &texture );  
73      // - texture parameters  
74      glTextureParameteri( texture, GL_TEXTURE_BASE_LEVEL, 0 );  
75      glTextureParameteri( texture, GL_TEXTURE_MAX_LEVEL, 0 );  
76      // - allocate memory  
77      const GLsizei levels = 1;  
78      const GLenum internalFormat = GL_RGBA8;  
79      glTextureStorage2D( texture, levels, internalFormat, width, height );  
80      // - fill data  
81      const GLenum format = GL_RGBA;  
82      const GLenum type = GL_UNSIGNED_BYTE;  
83      glTextureSubImage2D( texture, 0/*level*/, 0/*xoffset*/, 0/*yoffset*/, width, height, format, type, data.data() );  
84  
85      return true;  
86  }  
87
```

data: vector of uint8 read by STB library when loading an image.

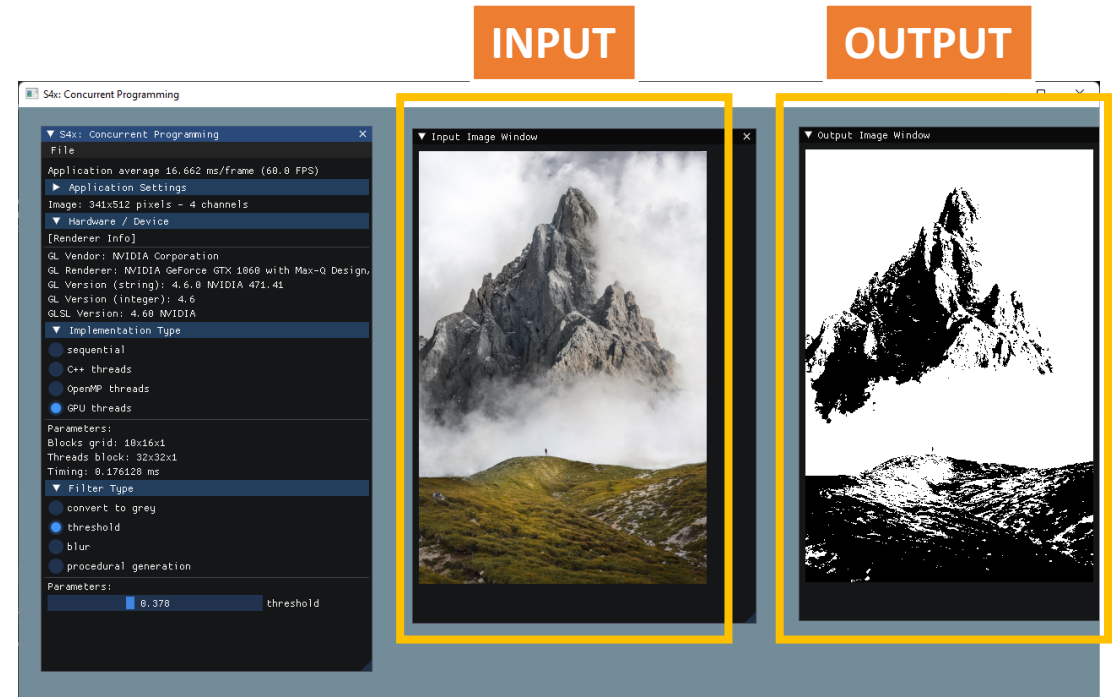
data: copy data to GPU memory


```
GLuint outputTexture;  
createOutputTexture( outputTexture, width, height )
```

```
88  /***** texture: OpenGL texture ID that will be created *****/  
89  * Create output texture  
90  *****/  
91  bool createOutputTexture( GLuint& texture, const unsigned int width, const unsigned int height )  
92  {  
93      // Create texture  
94      glCreateTextures( GL_TEXTURE_2D, 1, &texture );  
95      // - texture parameters  
96      glTextureParameteri( texture, GL_TEXTURE_BASE_LEVEL, 0 );  
97      glTextureParameteri( texture, GL_TEXTURE_MAX_LEVEL, 0 );  
98      // - allocate memory  
99      const GLsizei levels = 1;  
100     const GLenum internalFormat = GL_RGBA8;  
101     glTextureStorage2D( texture, levels, internalFormat, width, height );  
102     // - fill data  
103     const GLenum format = GL_RGBA;  
104     const GLenum type = GL_UNSIGNED_BYTE;  
105     std::uint32_t initValue = 0; // 4 * 8 = 32  
106     glClearTexImage( texture, /*level*/0, format, type, &initValue );  
107  
108     return true;  
109 }  
110
```


RENDERING IMAGES

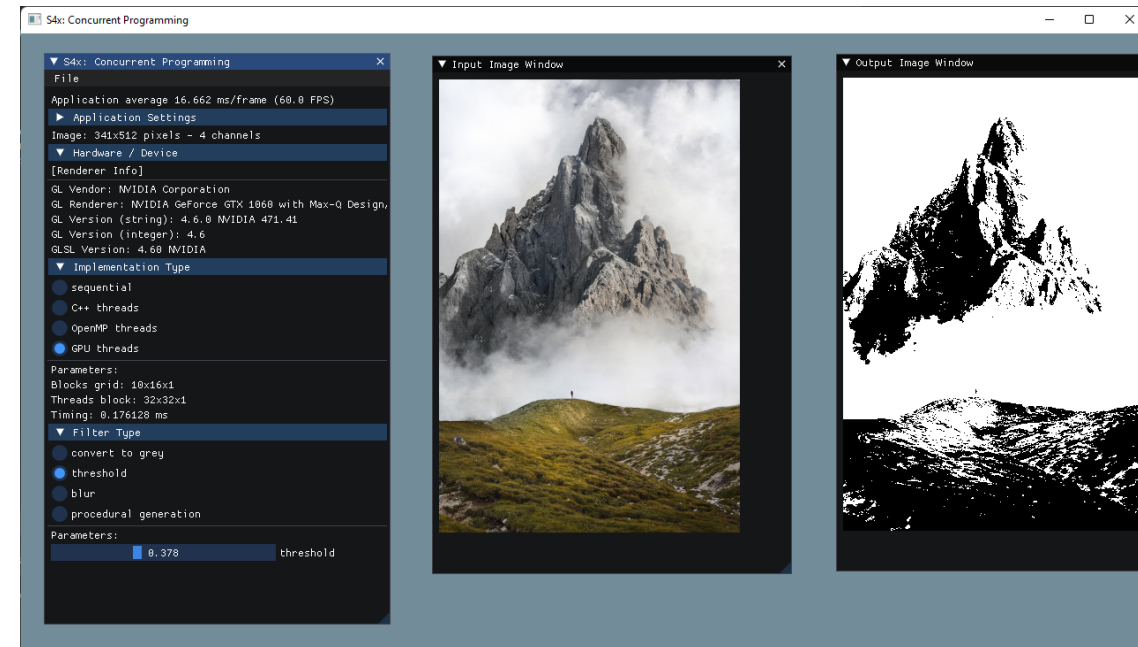
- Initialization
 - create shader
- Image loading
 - load image
 - create Input texture
 - create Output texture
- User action [ex : click bouton]
 - set current shader : `glUseProgram()`
 - launch kernel : `glDispatchCompute()`
 - save image
- Rendering
 - Display textures (images)



ImGui

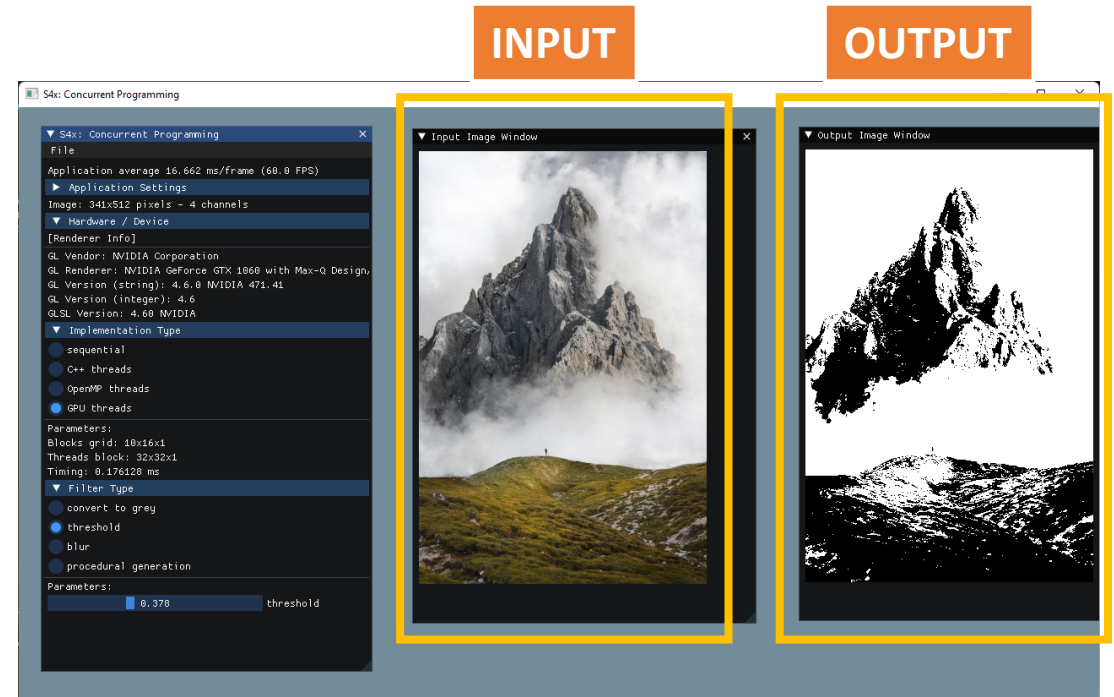
automatic display of images (textures)

```
bool show_input_image_window = true;
bool show_output_image_window = true;
if ( show_input_image_window )
{
    ImGui::Begin( "Input Image Window", &show_input_image_window );
    ImGui::Image( (void*)(mInputTexture),
        ImVec2( static_cast< float >( algorithm->getWidth() ),
            static_cast< float >( algorithm->getHeight() ) ) );
    ImGui::End();
}
if ( show_output_image_window )
{
    ImGui::Begin( "Output Image Window", &show_output_image_window );
    ImGui::Image( (void*)(mOutputTexture),
        ImVec2( static_cast< float >( algorithm->getWidth() ),
            static_cast< float >( algorithm->getHeight() ) ) );
    ImGui::End();
}
```



CREATING SHADER

- Initialization
 - create shader
- Image loading
 - load image
 - create Input texture
 - create Output texture
- User action [ex : click bouton]
 - set current shader : `glUseProgram()`
 - launch kernel : `glDispatchCompute()`
 - save image
- Rendering
 - Display textures (images)



```
GLuint shaderProgram_convertToGrey;  
const std::string convertToGrey_shaderSource = "void main( )\n..."  
createShader( shaderProgram_convertToGrey, convertToGrey_shaderSource )
```

shaderProgram: OpenGL shader program ID that will be created

shaderSource: your shader source code (*main()*...),
as in shadertoy. It will be compiled and linked
as a program in C, C++...

```
35  /*****  
36  * Create a shader  
37  *****/  
38  bool createShader( GLuint& shaderProgram, const std::string& shaderSource )  
39  {  
40      // Create shader  
41      GLuint shader_handle = glCreateShader( GL_COMPUTE_SHADER );  
42  
43      // Set shader source code  
44      const GLchar* compute_shader_with_version[ 1 ] = { shaderSource.c_str() };  
45      glShaderSource( shader_handle, 1, compute_shader_with_version, NULL );  
46  
47      // Compile shader  
48      glCompileShader( shader_handle );  
49  
50      // Check compilation status  
51      ...  
52  
53      // Create shader program  
54      shaderProgram = glCreateProgram();  
55  
56      // Link shader  
57      glAttachShader( shaderProgram, shader_handle );  
58      glLinkProgram( shaderProgram );  
59  
60      // Check linking status  
61      ...  
62  
63      return true;  
64  }
```

```
const std::string write_shaderSource = "#version 460\n\nlayout(..."
```

GLSL : OpenGL shading language
(subset of C language with extensions)

Example of a shader that writes data to an output

```
const std::string computeShader_TEST_WRITE = R"(
#version 460
layout( local_size_x = 32, local_size_y = 32 ) in;
layout( rgba8, binding = 1 ) uniform writeonly image2D uOutputImage;

void main()
{
    ivec2 imSize = imageSize( uOutputImage );
    ivec2 gid = ivec2( gl_GlobalInvocationID.xy );
    if ( gid.x < imSize.x && gid.y < imSize.y )
    {
        imageStore( uOutputImage, gid/*coordinate*/, vec4( 1.0, 0.0, 0.0, 1.0 )/*data*/ );
    }
};
)";
```

1st line: GLSL version (same as OpenGL version)

3D size (x,y,z) of each block of threads
Here: 2D blocks of 32x32 threads=1024
(check GPU hardware limite, prefers multiple of 32)

data type: rgba 8 bits

binding slot ID 1 (check max number = 8?)
(need to bind same ID on CPU)

output image

GLSL has equivalent to CUDA 3D "blockDim", "BlockIdx" and "ThreadIdx". And also has gl_GlobalInvocationID which is the global linearized 1D index in data.

write to output image


```
const std::string read_write_shaderSource = "#version 460\n\nlayout(..."
```

```
const std::string computeShader_TEST_READ_WRITE = R"(|
```

```
#version 460
```

Example of a shader that reads data from an input, and writes data to an output

```
layout( local_size_x = 32, local_size_y = 32 ) in;
```

```
layout( rgba8, binding = 0 ) uniform readonly image2D uInputImage;
```

```
layout( rgba8, binding = 1 ) uniform writeonly image2D uOutputImage;
```

binding slots ID 0 and 1 (check max number = 8?)
(need to bind same IDs on CPU)

```
void main()
```

```
{
```

```
    ivec2 imSize = imageSize( uOutputImage );
```

```
    ivec2 gid = ivec2( gl_GlobalInvocationID.xy );
```

```
    if ( gid.x < imSize.x && gid.y < imSize.y )
```

```
    {
```

read from input image

```
        vec4 color = imageLoad( uInputImage, gid );
```

```
        imageStore( uOutputImage, gid/*coordinate*/, vec4( color.rgb, 1.0 )/*data*/ );
```

```
    }
```

write to output image

```
};
```

```
)";
```

Sending Uniforms

Example of a shader that reads data from an input, and writes data to an output, and use a “uniform” to customize your code (ex : radius of region to blur used for blur filtering)

```
const std::string computeShader_blur = R"(
    #version 460

    layout( local_size_x = 32, local_size_y = 32 ) in;

    layout( rgba8, binding = 0 ) uniform readonly image2D uInputImage;
    layout( rgba8, binding = 1 ) uniform writeonly image2D uOutputImage;

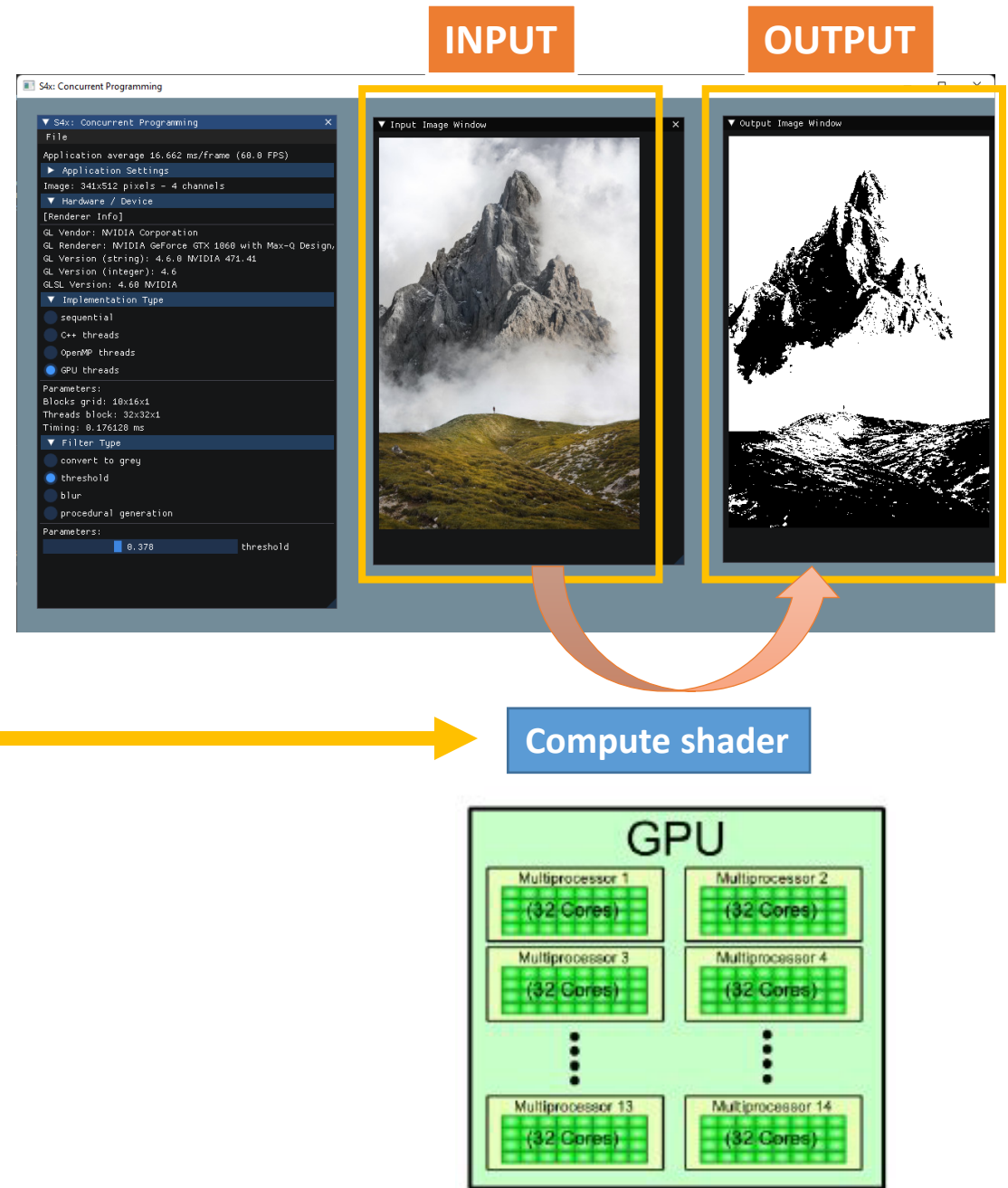
    layout ( location = 0 ) uniform int uFilterRadius;

    void main()
    {
        ivec2 imSize = imageSize( uOutputImage );
        ivec2 gid = ivec2( gl_GlobalInvocationID.xy );
        if ( gid.x < imSize.x && gid.y < imSize.y )
        {
            vec3 meanColor = vec3( 0.0 );
```

binding slots ID 0 (check max number = 1024?)
(need to bind same ID on CPU when sending value)

LAUNCHING GPU KERNEL

- Initialization
 - create shader
- Image loading
 - load image
 - create Input texture
 - create Output texture
- User action [ex : click bouton]
 - set current shader : `glUseProgram()`
 - launch kernel : `glDispatchCompute()`
 - save image
- Rendering
 - Display textures (images)



Launching a kernel to write to an output

Compute shader code extract

```
layout( rgba8, binding = 1 ) uniform writeonly image2D uOutputImage;
```

binding slot ID 1 (check max number = 8?)
(need to bind same IDs on CPU and GPU)

```
// - kernel  
glUseProgram( shaderProgram );
```

```
// - texture(s)  
// - output image  
glBindImageTexture( 1/*unit*/, mOutputTexture, 0/*level*/, GL_FALSE/*layered*/, 0/*layer*/, GL_WRITE_ONLY, GL_RGBA8 );
```

```
// Kernel management (grid of block)  
// - block  
const int blockSizeX = 32; // TODO: benchmark with 8x8, 16x16 and 32x32  
const int blockSizeY = 32;  
// - grid  
const GLuint gridSizeX = std::max( ( width + blockSizeX - 1 ) / blockSizeX, 1 );  
const GLuint gridSizeY = std::max( ( height + blockSizeY - 1 ) / blockSizeY, 1 );
```

```
// - launch kernel  
glDispatchCompute( gridSizeX, gridSizeY, 1 );
```

Launch kernel with a given grid of blocks of threads
Ex : here it computes the number of blocks to cover all the image with 1 thread per pixel.

Launching a kernel to read and write to an output

```
// - kernel
glUseProgram( shaderProgram );

// - texture(s)
// - input image
glBindImageTexture( 0/*unit*/, mInputTexture, 0/*level*/, GL_FALSE/*layered*/, 0/*layer*/, GL_READ_ONLY, GL_RGBA8 );
// - output image
glBindImageTexture( 1/*unit*/, mOutputTexture, 0/*level*/, GL_FALSE/*layered*/, 0/*layer*/, GL_WRITE_ONLY, GL_RGBA8 );

// Kernel management (grid of block)
// - block
const int blockSizeX = 32; // TODO: benchmark with 8x8, 16x16 and 32x32
const int blockSizeY = 32;
// - grid
const GLuint gridSizeX = std::max( ( width + blockSizeX - 1 ) / blockSizeX, 1 );
const GLuint gridSizeY = std::max( ( height + blockSizeY - 1 ) / blockSizeY, 1 );

// - launch kernel
glDispatchCompute( gridSizeX, gridSizeY, 1 );
```

Launching a kernel to read and write to an output,
and send some uniforms.

```
// - kernel  
glUseProgram( shaderProgram );
```

```
// - texture(s)  
// - input image  
glBindImageTexture( 0/*unit*/, mInputTexture, 0/*level*/, GL_FALSE/*layered*/, 0/*layer*/, GL_READ_ONLY, GL_RGBA8 );  
// - output image  
glBindImageTexture( 1/*unit*/, mOutputTexture, 0/*level*/, GL_FALSE/*layered*/, 0/*layer*/, GL_WRITE_ONLY, GL_RGBA8 );
```

```
// - uniforms  
glUniformf( 0, threshold );  
glUniformli( 1, filterMaskSize );
```

```
// Kernel management (grid of block)  
// - block  
const int blockSizeX = 32; // TODO: benchmark with 8x8, 16x16 and 32x32  
const int blockSizeY = 32;  
// - grid  
const GLuint gridSizeX = std::max( ( width + blockSizeX - 1 ) / blockSizeX, 1 );  
const GLuint gridSizeY = std::max( ( height + blockSizeY - 1 ) / blockSizeY, 1 );  
  
// - launch kernel  
glDispatchCompute( gridSizeX, gridSizeY, 1 );
```

Compute shader code extract (example)

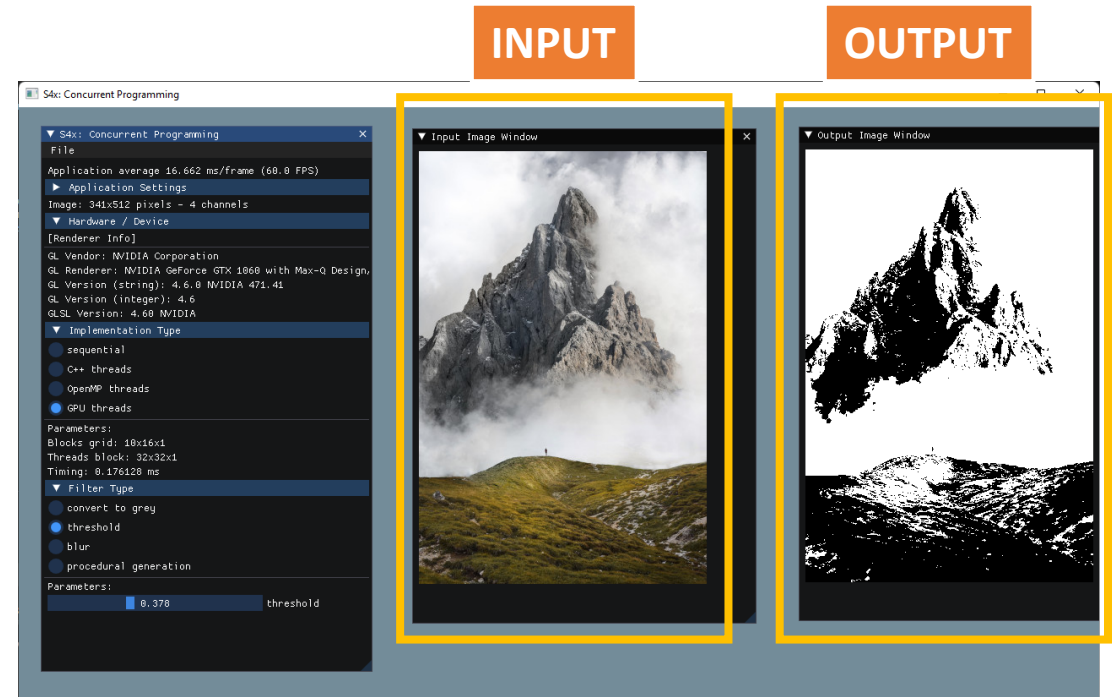
```
layout ( location = 0 ) uniform int uFilterRadius;
```

binding slots ID 0 and 1 (check max number = 1024?)
(need to bind same IDs on CPU when sending value)

Send "uniform" values from CPU to GPU,
select same binding slots as in shader code (here 0 and 1).

SAVING OUTPUT IMAGE

- Initialization
 - create shader
- Image loading
 - load image
 - create Input texture
 - create Output texture
- User action [ex : click bouton]
 - set current shader : `glUseProgram()`
 - launch kernel : `glDispatchCompute()`
 - save image
- Rendering
 - Display textures (images)



Copydata from GPU to CPU



Write image (STB library)

```
// Synchronization
// - make sure writing to image has finished before read
glMemoryBarrier( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
glMemoryBarrier( GL_ALL_BARRIER_BITS );

// Retrieve data on host
std::vector< std::uint8_t > outputData( width * height * algorithm->getNbChannels() );
glGetTextureImage( mOutputTexture, 0/*level*/, GL_RGBA, GL_UNSIGNED_BYTE,
    sizeof( std::uint8_t ) * outputData.size(), outputData.data() );

// Save data in image
const std::string databasePath = std::string( "" );
const std::string pptbfName = std::string( "GPU_" ) + std::to_string( imageCounter );
const std::string pptbfFilename = databasePath + pptbfName + std::string( ".jpg" );

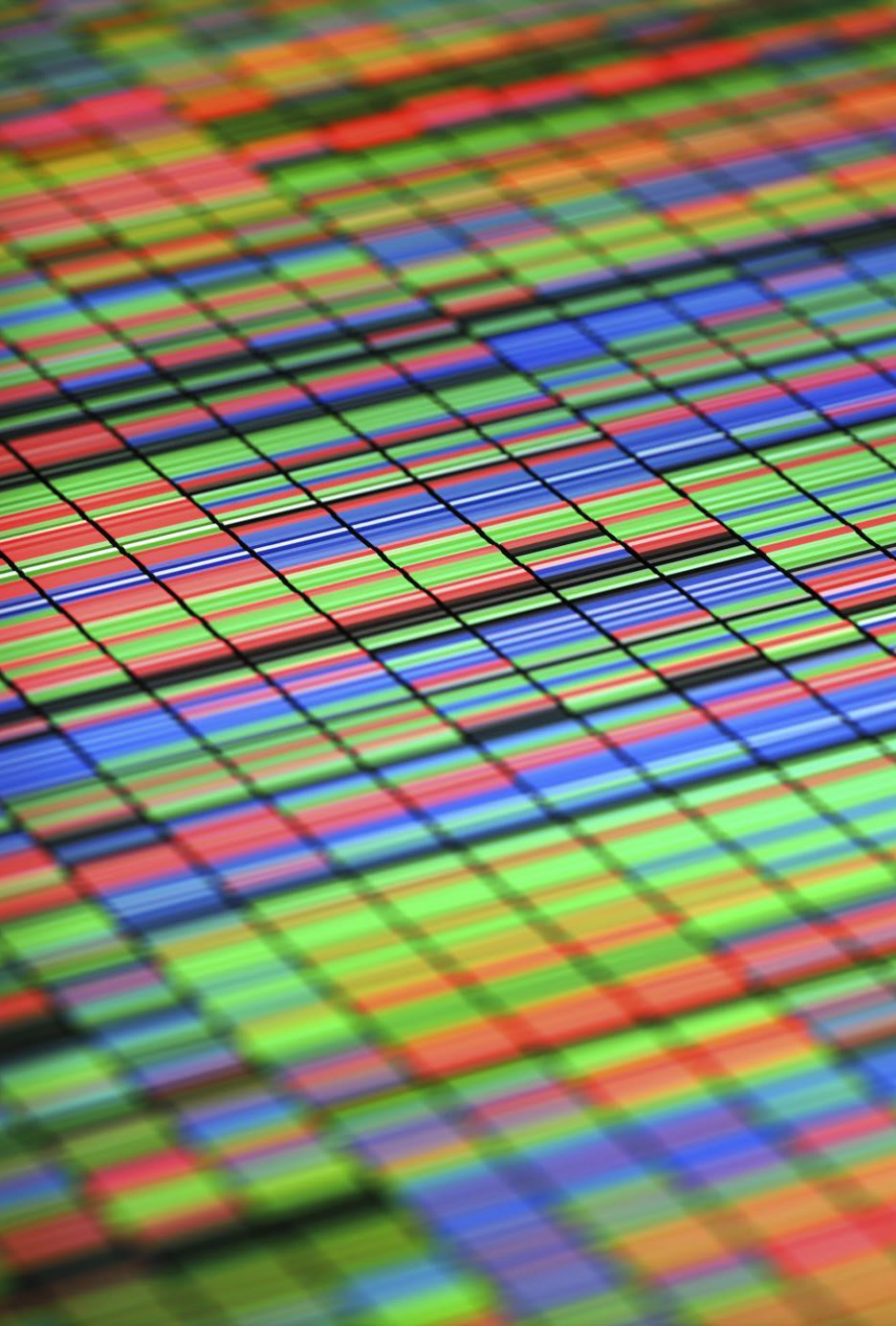
// EXPORT resulting image with STB
const int nbChannels = algorithm->getNbChannels();
const std::string outputFilename = pptbfFilename;
const int quality = 90;
const int status = stbi_write_jpg( outputFilename.c_str(),
    width, height, nbChannels,
    /*u_pptbf.data()*/outputData.data(),
    quality );
```

Not sure to need these 2 lines of code?

It's more when using previously written images in a compute shader code, to be used in an other shader (*kind of cache flush*)

This call to `glGetTextureImage()` is synchronous.

It waits for GPU to finish its work then copy GPU memory to a CPU array.



-
- Application
 - Image Processing

▼ S4x: Concurrent Programming

File

Application average 16.662 ms/frame (60.0 FPS)

► Application Settings

Image: 341x512 pixels - 4 channels

▼ Hardware / Device

[Renderer Info]

GL Vendor: NVIDIA Corporation
GL Renderer: NVIDIA GeForce GTX 1060 with Max-Q Design,
GL Version (string): 4.6.0 NVIDIA 471.41
GL Version (integer): 4.6
GLSL Version: 4.60 NVIDIA

▼ Implementation Type

☐ sequential
☐ C++ threads
☐ OpenMP threads
☒ GPU threads

Parameters:
Blocks grid: 10x16x1
Threads block: 32x32x1
Timing: 0.176128 ms

▼ Filter Type

☐ convert to grey
☒ threshold
☐ blur
☐ procedural generation

Parameters:
 0.378 threshold

