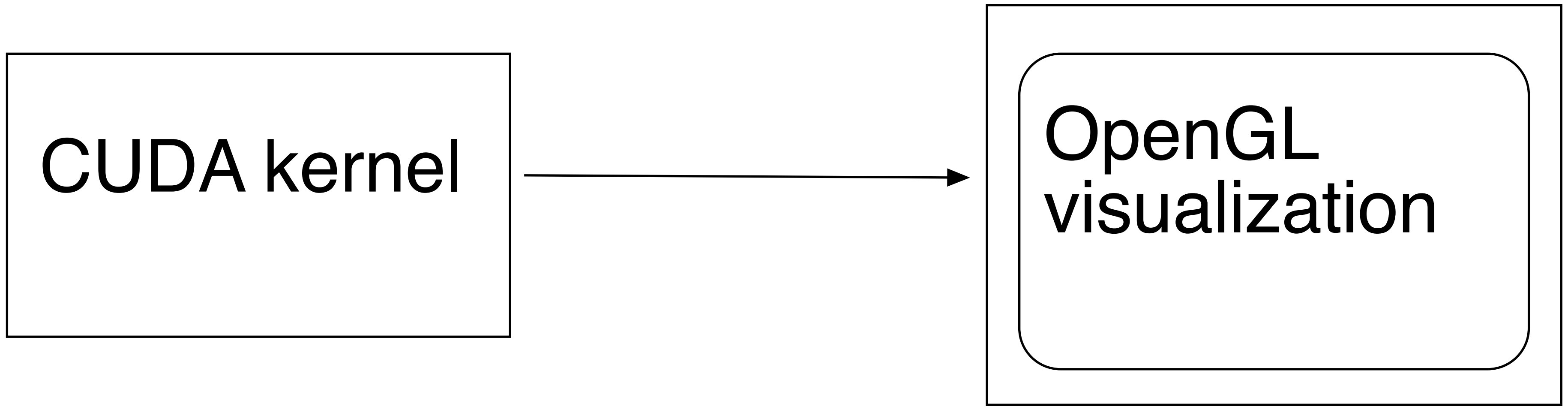
# CUDA-OpenGL Interoperability

## Visualize results with OpenGL

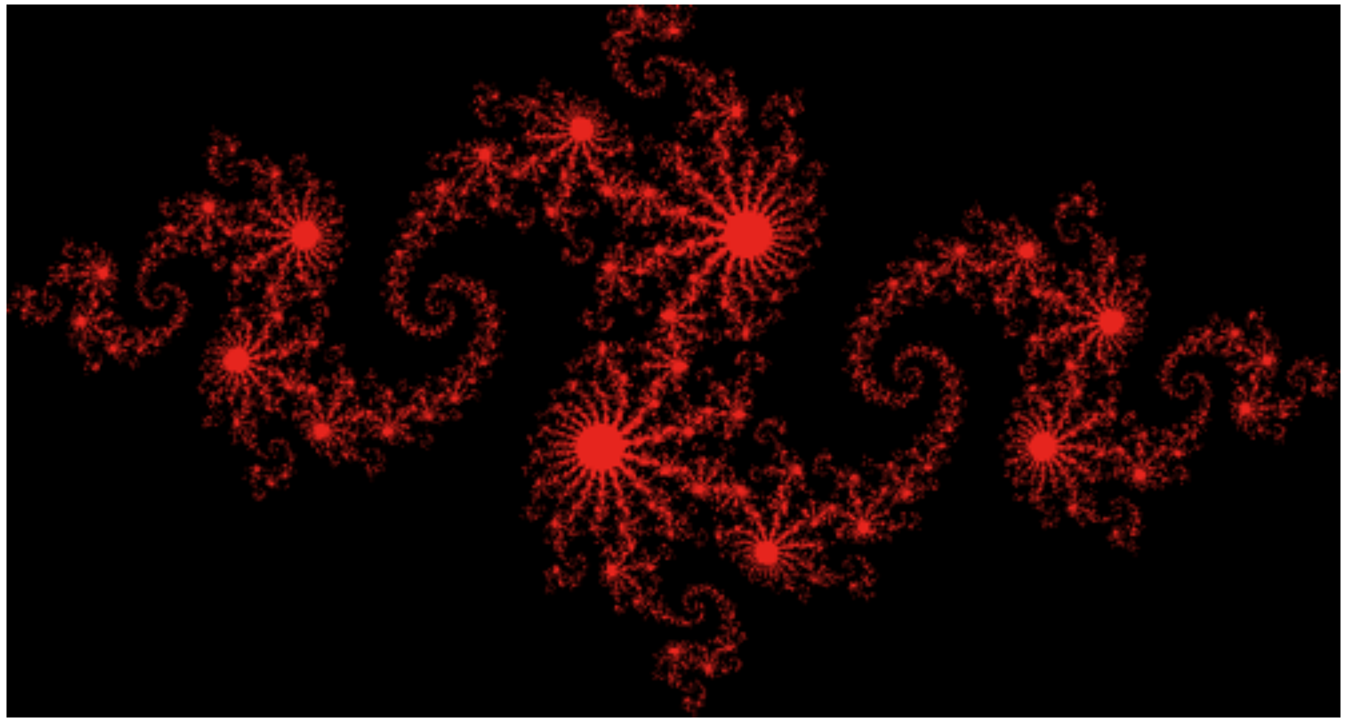


# CUDA and graphics

**Simplest way: Pass output from CUDA, typically to an OpenGL texture.**

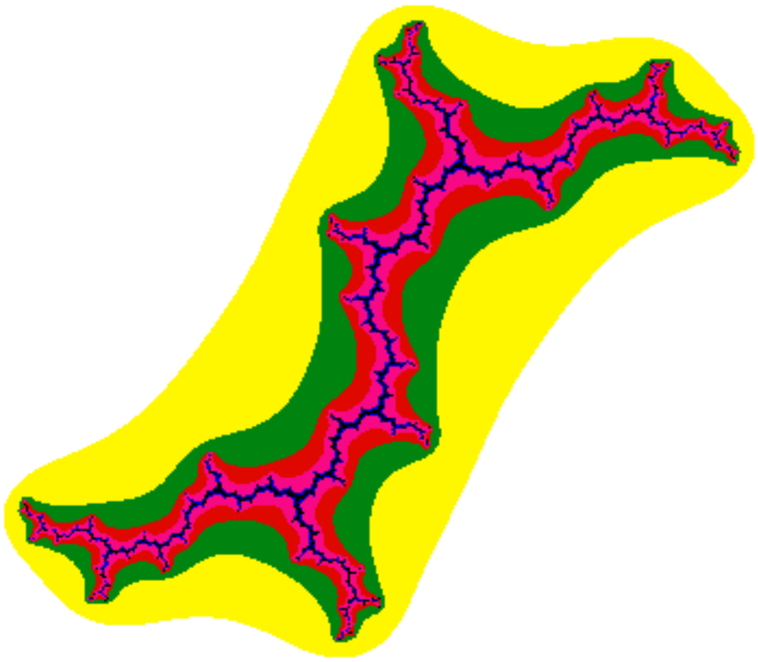
**Example: Julia set, Lab 4 Mandelbrot, ray caster...**

**Good for visualizing results. Better methods exist, without having to move data to CPU and back.**



# The Julia set

$$z_{k+1} = z_k^2 + \lambda$$



**Julia set for**
**$\lambda = (0, 1) = 0 + j$**

**Start with position in complex space.**

**Apply complex function recursively**

**Inspect distance to origin**

**Perfectly parallel algorothm**

# CUDA-OpenGL Interoperability

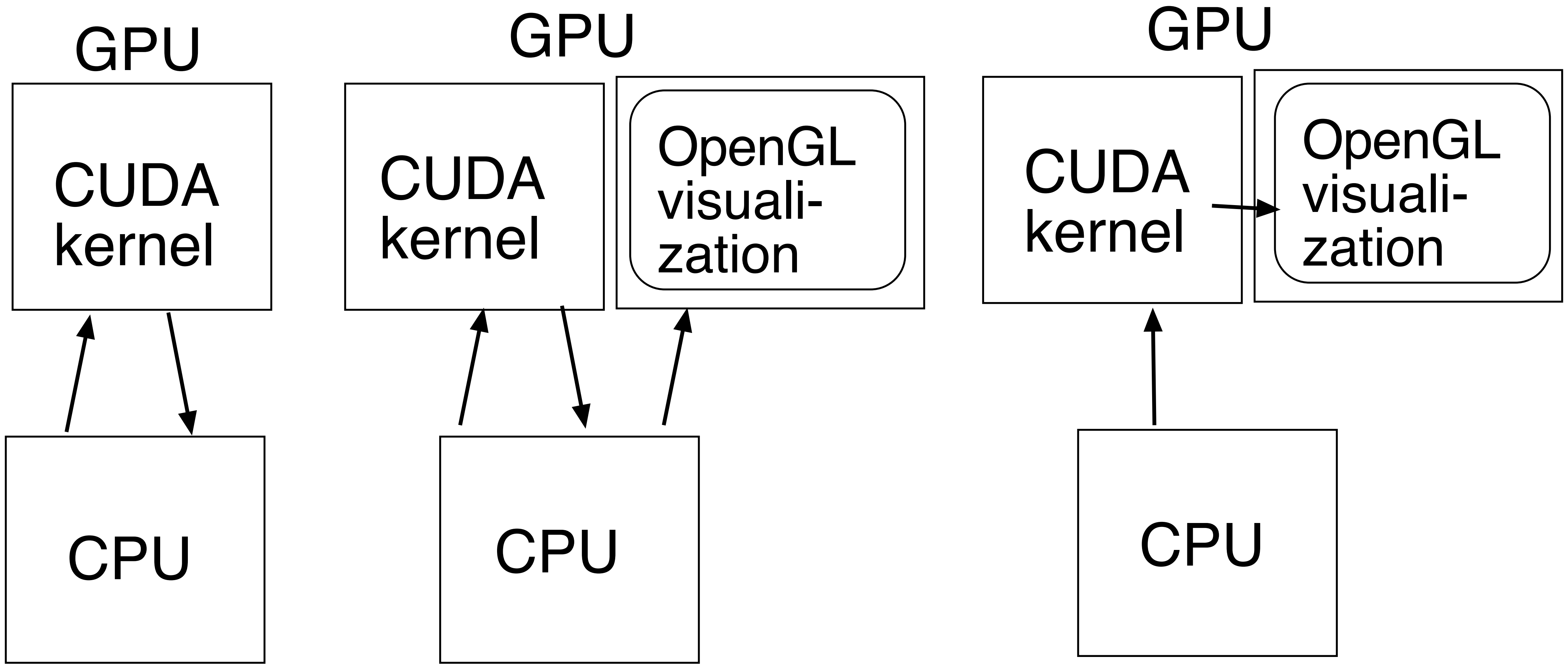
- Integrate for better performance!
- Possible to visualize without leaving GPU

An output which is not the CPU

**No visualization**

**Simple visualization**

**Visualization with OpenGL interoperability**



# Steps for interoperability

- Decide what data CUDA will process
- Allocate with OpenGL
- Register with CUDA
- Map buffer to get CUDA pointer
- Pass pointer to CUDA kernel
- Release pointer
- Use result in OpenGL graphics

- **Allocate with OpenGL**
- **Register with CUDA**

Allocate VBO (vertex buffer)

```
glGenBuffers(1, &positionsVBO);
glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
unsigned int size = NUM_VERTS * 4 * sizeof(float);
glBufferData(GL_ARRAY_BUFFER, size, NULL, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Register with CUDA

```
cudaGraphicsGLRegisterBuffer(&positionsVBO_CUDA, positionsVBO,
cudaGraphicsMapFlagsWriteDiscard);
```

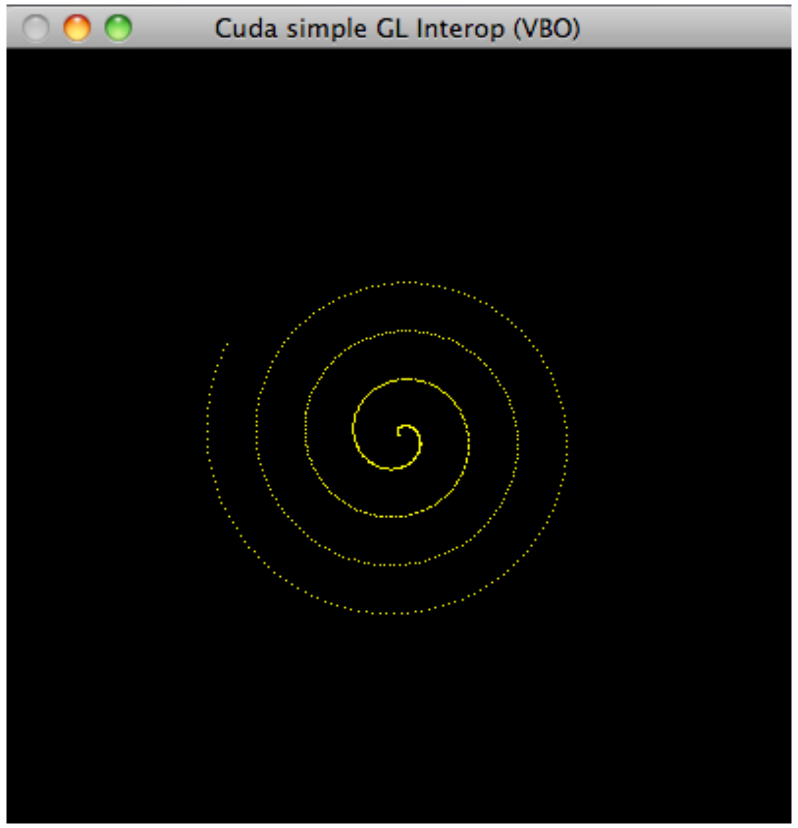
- **Map buffer to get CUDA pointer**
- **Pass pointer to CUDA kernel**
- **Release pointer**

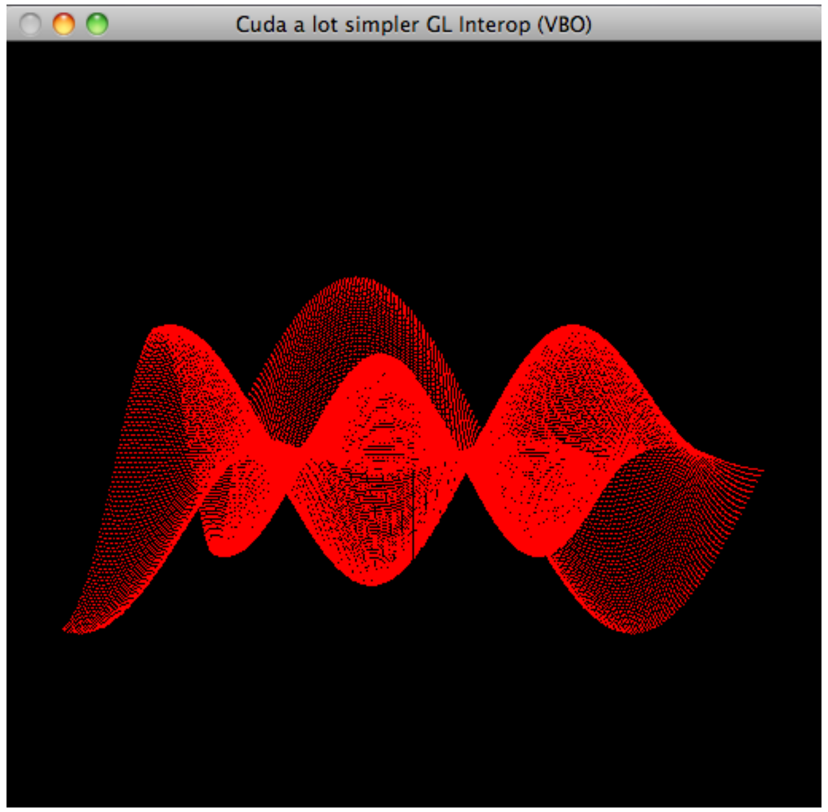
```
cudaGraphicsMapResources(1, &positionsVBO_CUDA, 0);
size_t num_bytes;
cudaGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
positionsVBO_CUDA);printError(NULL, err);

// Execute kernel
dim3 dimBlock(16, 1, 1);
dim3 dimGrid(NUM_VERTS / dimBlock.x, 1, 1);
createVertices<<<dimGrid, dimBlock>>>(positions, anim, NUM_VERTS);

// Unmap buffer object
cudaGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);
```

# Simple CUDA kernel for producing vertices for graphics

```
// CUDA vertex kernel
__global__ void createVertices(float4* positions, float time, unsigned int num)
{
 unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;

 positions[x].w = 1.0;
 positions[x].z = 0.0;
 positions[x].x = 0.5*sin(kVarv * (time + x * 2 * 3.14 / num)) * x/num;
 positions[x].y = 0.5*cos(kVarv * (time + x * 2 * 3.14 / num)) * x/num;
}
```

# Simple examples:





**Just vertices - but you can draw surfaces, compute textures, use any OpenGL effects (light, materials)**

# But should we use CUDA for OpenGL?

**Great for visualizing**

**Faster than going over CPU**

**Slower than plain OpenGL for graphics!**

**and OpenGL has CUDA-like functionality built-in!**
**(Compute Shaders.) (Later lecture)**

# Conclusions

**CUDA can be coupled closer to OpenGL than the simple way we have done before!**

**Moving data back and forth is wastefui, there is performance to gain!**

**Some interesting alternatives exist as well.**