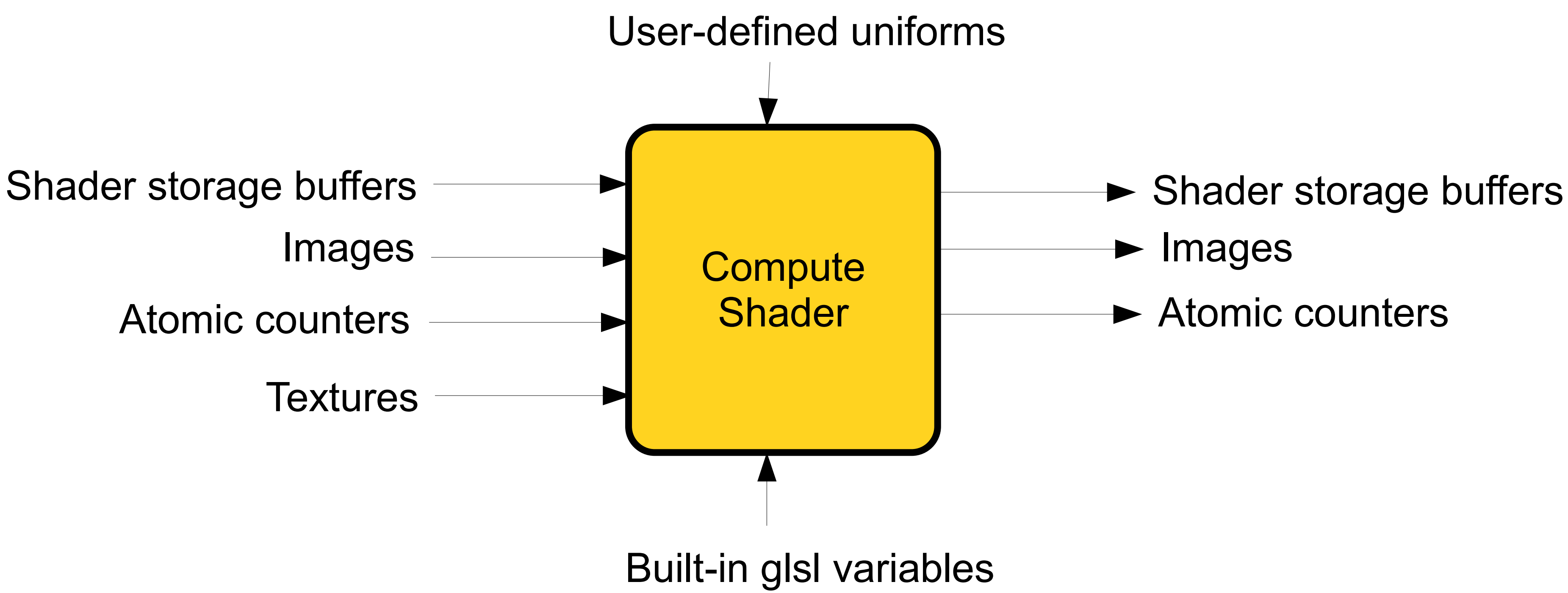
Compute Shader

- Pipeline
- Creation
- Workgroups and Dispatch
- Communication and Synchronization
- Applications

Compute Shader Pipeline

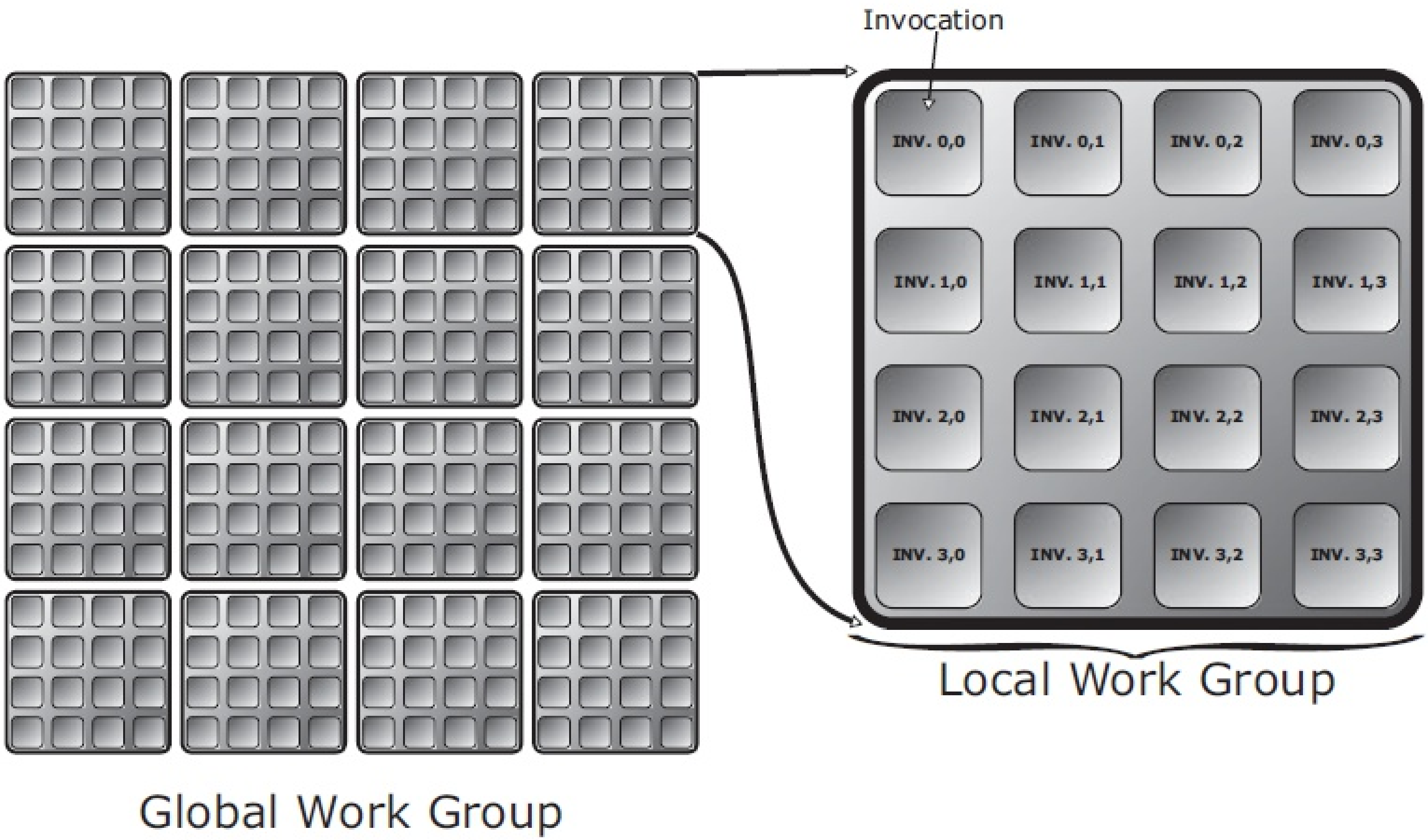
- Separate from graphics pipeline
- Its own special single-stage pipeline
- Useful for general-purpose computation, but not as full-featured as CUDA or OpenCL.



Creation

```
GLuint shader, program;
static const GLchar* source[] =
{
    "#version 430 core "
    " "
    "// Input layout qualifier declaring a 16 x 16 (x 1) local"
    "// workgroup size "
    "layout (local_size_x = 16, local_size_y = 16) in;"
    " "
    "void main(void) "
    "{"
    "  // Do something here."
    "}"
};
//or read shader code from a file

shader = glCreateShader(GL_COMPUTE_SHADER);
glShaderSource(shader, 1, source, NULL);
glCompileShader(shader);
program = glCreateProgram();
glAttachShader(program, shader);
glLinkProgram(program);
```

Workgroups and Dispatch

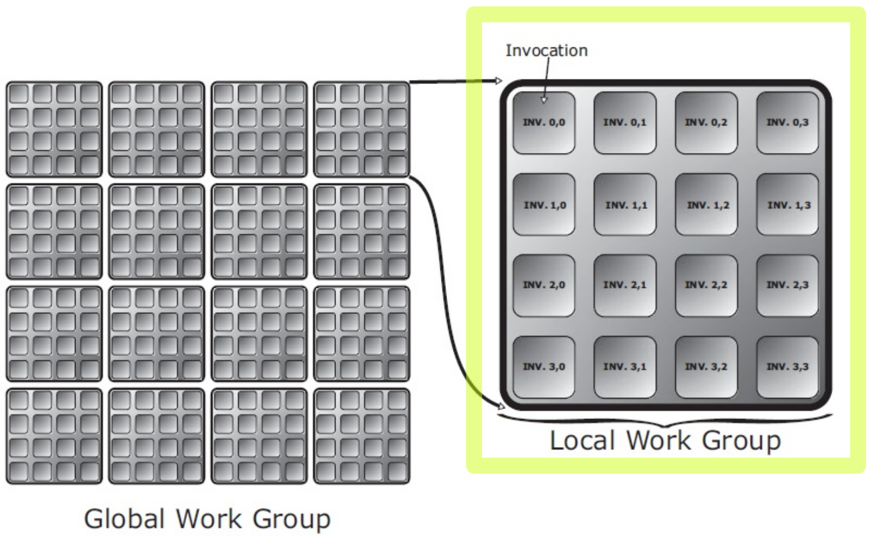


- Compute shaders operate on a grid-based work group structure
  - Compute shader will execute once for each "invocation"

Local Workgroup Size

- Declared in shader code
- Can be 1D, 2D or 3D

```
#version 430 core
// Input layout qualifier declaring a 4 x 4 (x 1) local
// workgroup size
layout (local_size_x = 4, local_size_y = 4) in;
void main(void)
{
// Do something.
}
```



Local vs Global

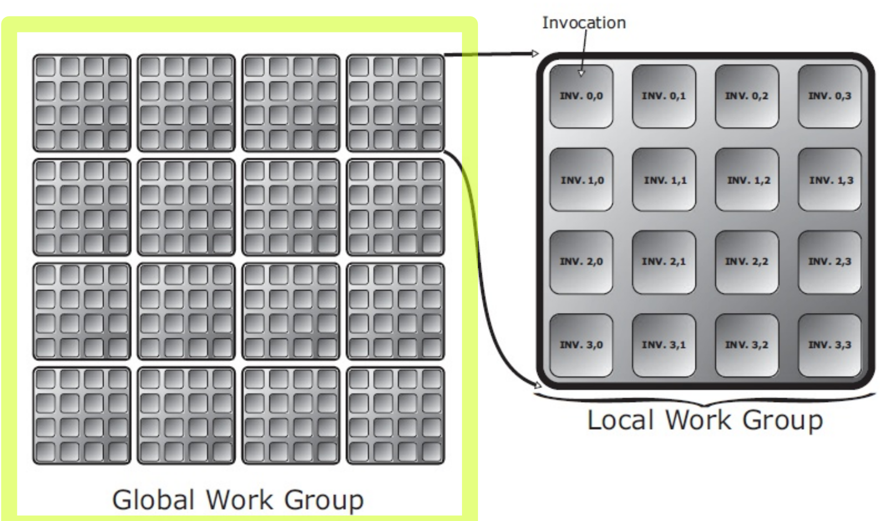
- Why the distinction?
  - Invocations in the local workgroup can access **shared memory**
  - Invocations in the same local workgroup will **run in parallel** if possible

Global Workgroup Size

- Specified when the shader is dispatched

```
void glDispatchCompute(
    GLuint num_groups_x,
    GLuint num_groups_y,
    GLuint num_groups_z);
```

  - Call glUseProgram(...) **before** glDispatchCompute



Knowing where you are

- Compute shader will execute once for each “invocation”
- Built-in glsl variables
  - `const uvec3 gl_WorkGroupSize; : local workgroup size`
    - equal to uvec3(local_size_x, local_size_y, local_size_z)
  - `in uvec3 gl_NumWorkGroups;    : global workgroup size`
    - same as paramaters to glDispatchCompute(...)
  - `in uvec3 gl_LocalInvocationID;:` index of the current invocation within the local workgroup
  - `in uvec3 gl_WorkGroupID;:` index of current local workgroup within the global workgroup
  - `in uvec3 gl_GlobalInvocationID; :` index of the current invocation within the global workgroup
  - `in uint gl_LocalInvocationIndex;` : a flattened 1D index computed from local invocation ID and local workgroup size
    - Useful for indexing into a shared 1D array

Communication and Synchronization

- **Shared** variables
  - Variables shared among the local work group
  - Compute shader can read or write
  - **Faster** access than images or shader storage buffers
  - Size is limited.
    - Query max size with glGetIntegerv(GL_MAX_COMPUTE_SHARED_MEMORY_SIZE, ...)
- Synchronization
  - barrier()
  - groupMemoryBarrier()

Shared variables

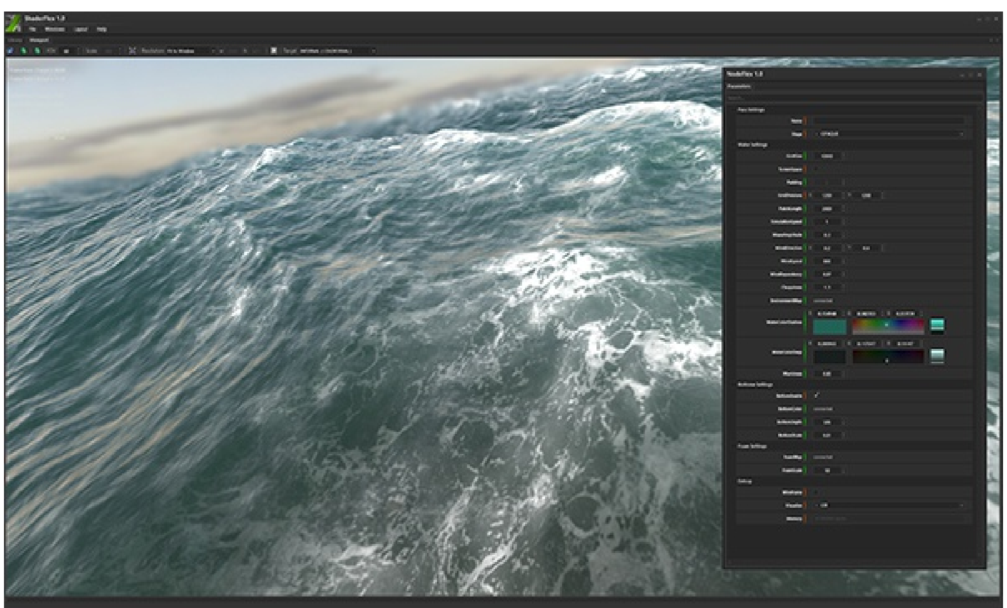
- Declared using the **shared** qualifier
  - `shared uint foo;`
  - `shared vec4 bar[128];`
  -
```
shared struct baz_struct
{
    vec4 a_vec;
    int an_int;
    ivec2 array_of_int[27];
} baz[42];
```

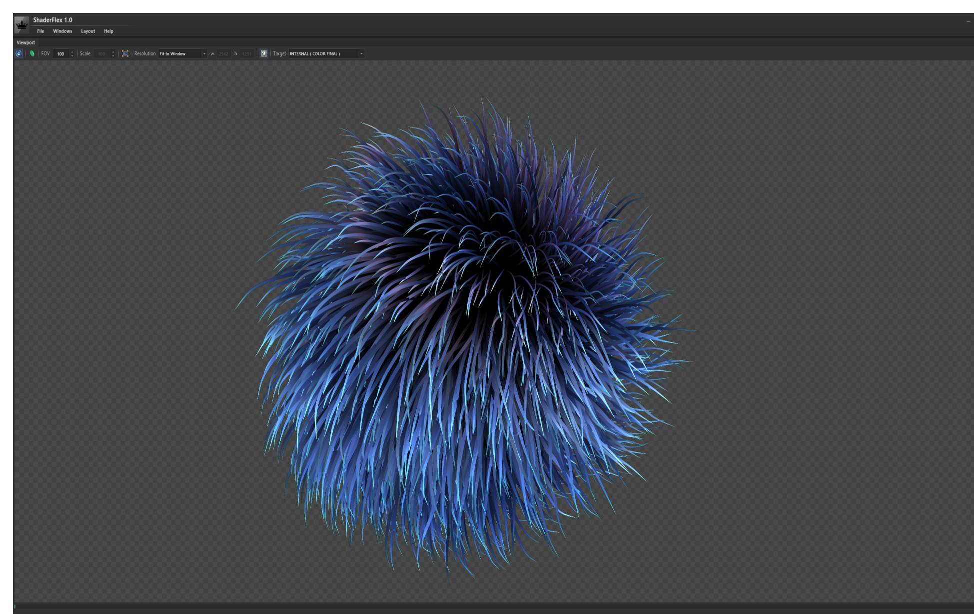
Synchronization

- **barrier()**
  - All invocations within a single local workgroup will finish **all prior commands** before continuing.
  - Ensures that values written by one invocation (to images, shared variables, etc) can be safely read by others
- **groupMemoryBarrier()**
  - Ensures **all memory write operations** within the local workgroup <u>that have been started are finished</u> before continuing

Applications

- Simulation
- Image processing

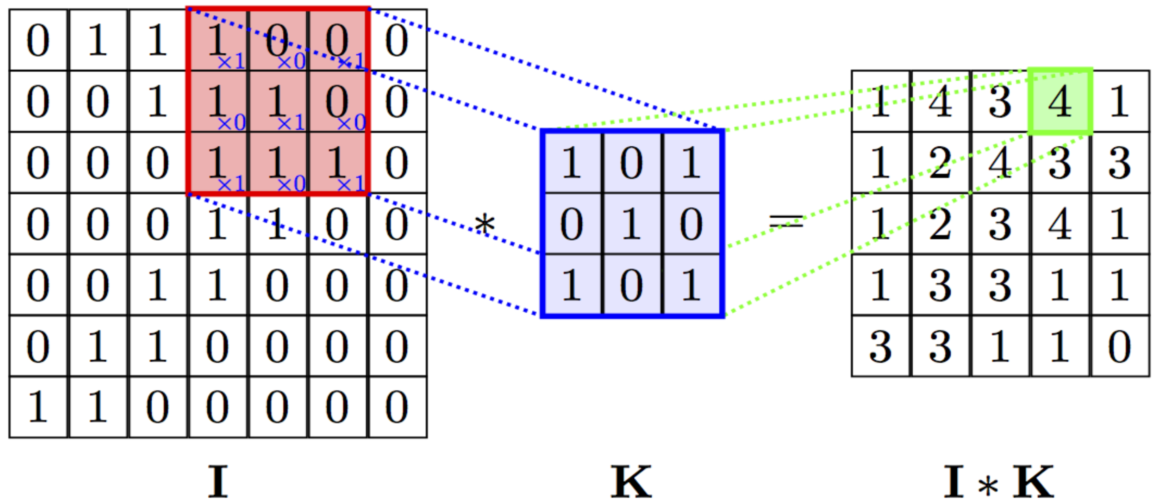




Particle simulation

```
#version 430

layout( std140, binding=4 ) buffer Pos
{
    vec4 Positions[ ];
};
layout( std140, binding=5 ) buffer Vel
{
    vec4 Velocities[ ];
};

layout( local_size_x = 128, local_size_y = 1, local_size_z = 1 ) in;

const vec3 G = vec3( 0., -9.8, 0. );
const float DT = 0.1;

void main()
{
    uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case

    vec3 p = Positions[ gid ].xyz;
    vec3 v = Velocities[ gid ].xyz;
    p = p + v*DT + .5*DT*DT*G;
    v = v + G*DT;
    Positions[ gid ].xyz = p;
    Velocities[ gid ].xyz = v;
}
```

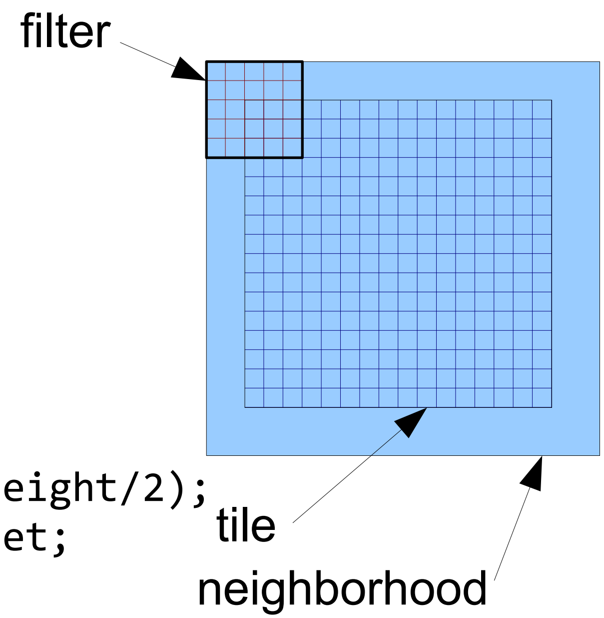
Image Convolution (Blurring) Overview



- Simplified overview of code presented in "OpenGL 4.3 and Beyond" by Mark Kilgard
  - Phase 1: Write image block into shared memory
    - Since reads overlap it will be faster to have image data in shared memory
  - Phase 2: Read from neighborhood of shared memory and compute average
  - Phase 3: Write to output image

Image convolution : Phase 1

Why prefetch into shared memory?
Naive implementation does many redundant reads. Shared memory is faster, probably cached more efficiently.




```
const int tileWidth = 16, tileHeight = 16
const int filterWidth = 5, filterHeight = 5;
const ivec2 tileSize = ivec2 (tileWidth,tileHeight);
const ivec2 filterOffset = ivec2 (filterWidth/2,filterHeight/2);
const ivec2 neighborhoodSize = tileSize + 2* filterOffset;

// Declare the input and output images.
layout(binding=0,rgba8) uniform image2D input_image;
layout(binding=1,rgba8) uniform image2D output_image;

uniform float weight = 1.0/(filterHeight*filterWidth); // for mean filter

layout(local_size_x=TILE_WIDTH,local_size_y=TILE_HEIGHT) in;

shared vec4 pixel[NEIGHBORHOOD_HEIGHT][NEIGHBORHOOD_WIDTH];
```

Image convolution : Phase 1

- Fill the shared memory array pixel[][]
  - Each invocation fills a small part

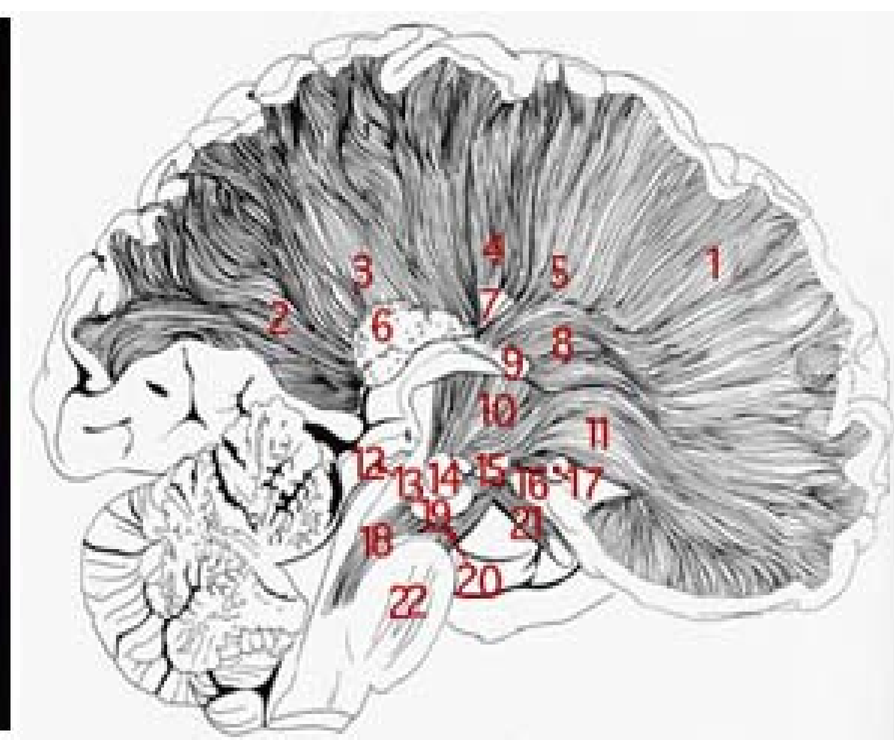
```
void main ()
{
const ivec2 tile_xy = ivec2(gl_WorkGroupID);
const ivec2 thread_xy = ivec2(gl_LocalInvocationID);
const ivec2 pixel_xy = tile_xy*tileSize + thread_xy;
const uint x = thread_xy.x;
const uint y = thread_xy.y;

for(int j=0; j<neighborhoodSize.y; j += tileHeight)
{
    for (int i=0; i<neighborhoodSize.x i=0; i += tileWidth)
    {
        const ivec2 read_at = pixel_xy+ ivec2(i,j); //TODO: clamp at image
boundary
        pixel[y+j][x+i] = imageLoad(input_image, read_at);
    }
}
barrier(); //don't proceed until shared memory array is completely written
```

Image convolution : Phases 2 and 3

```
// Phase 2: Compute local averages
vec4 result = vec4(0);
for(int j=0; j<filterHeight; j++)
{
	for(int i=0; i<filterWidth; i++)
	{
		result += pixel[y+j][x+i] * weight;
	}
}

// Phase 3: Store result to output image.
imageStore(output_image, pixel_xy, result);
```

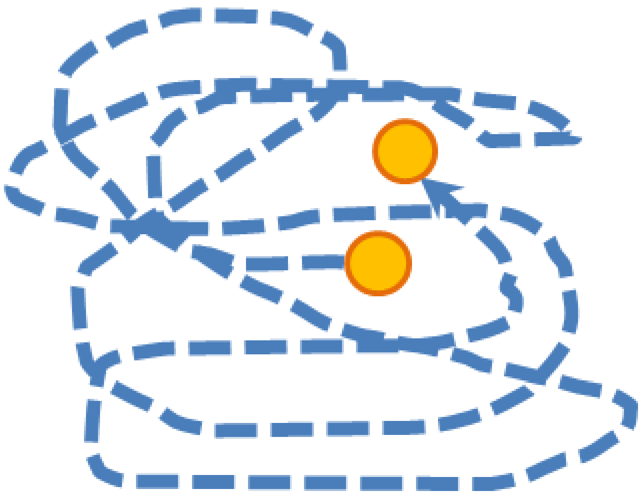
A medical imaging application

- Water diffuses within biological systems
  - Restricted by certain materials
  - Especially in white matter fiber bundles

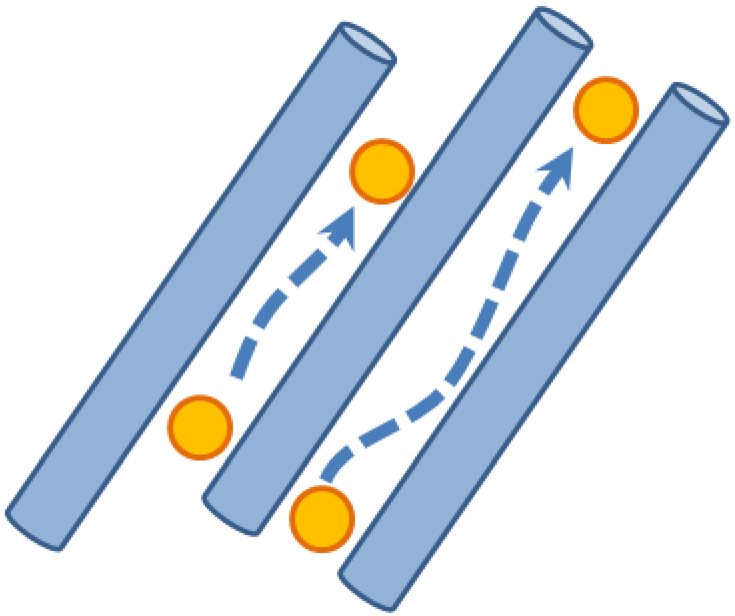


Introduction

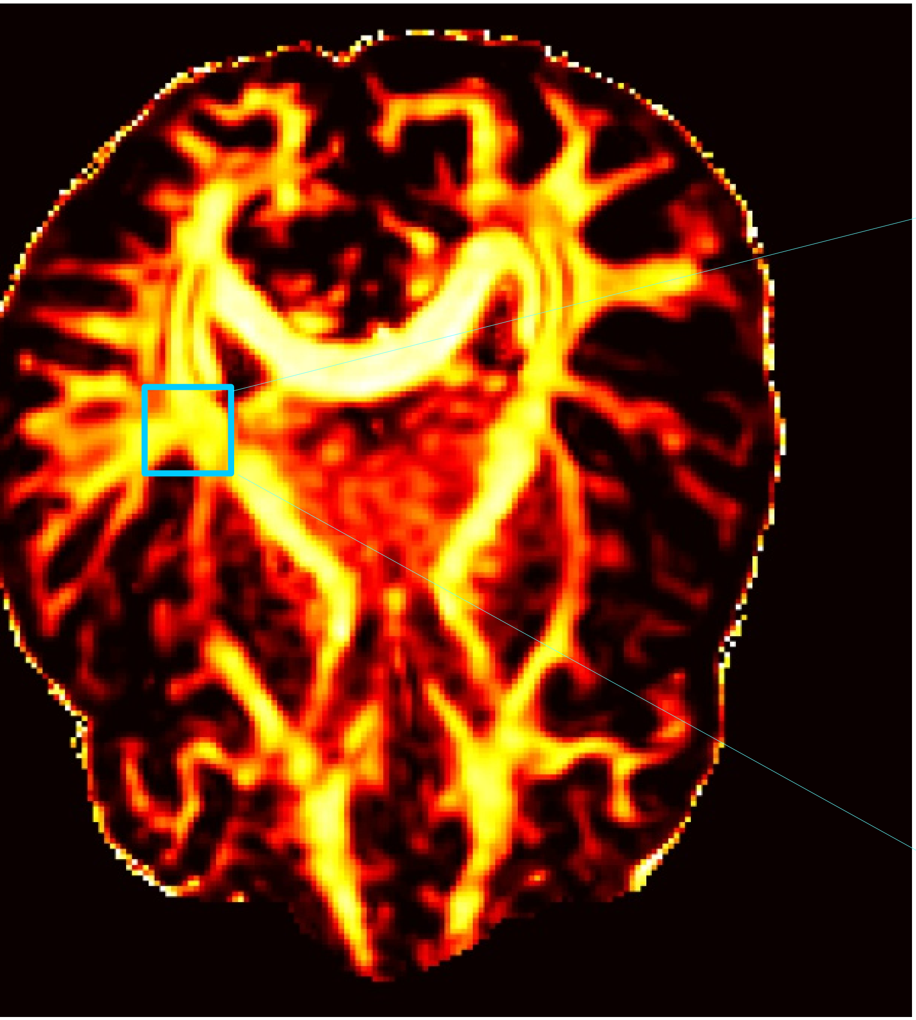
- <u>Diffusion Tensor-MRI</u>
  - An MRI modality for indirectly measuring structural connectivity



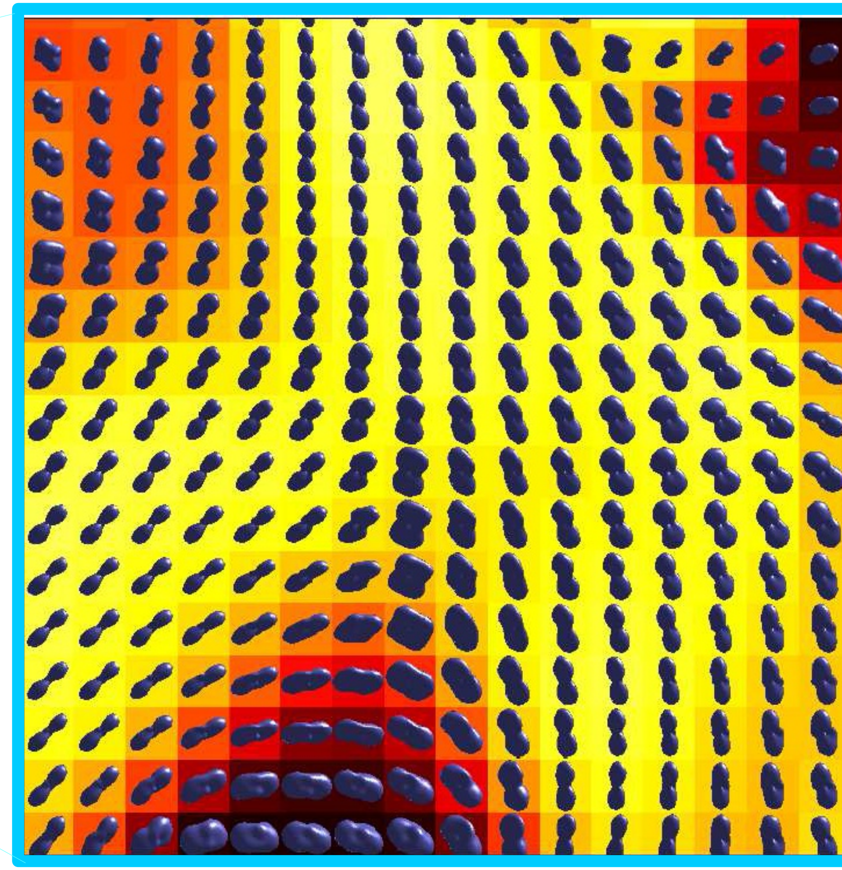
Free diffusion



Restricted diffusion

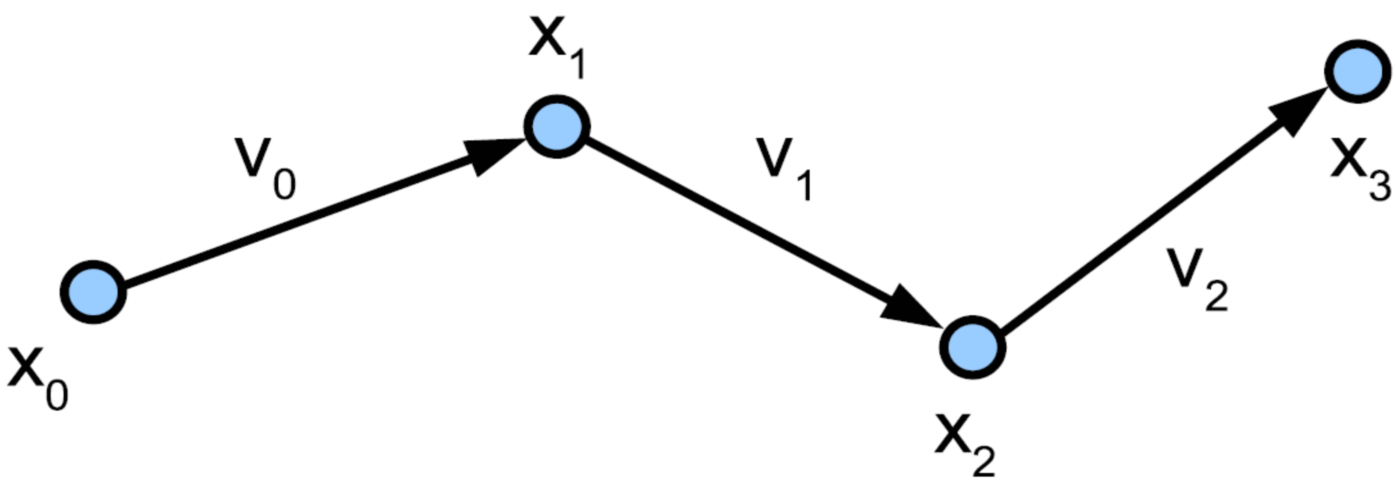


White matter map



Fiber orientation distributions

Tracing fibers by following water molecules

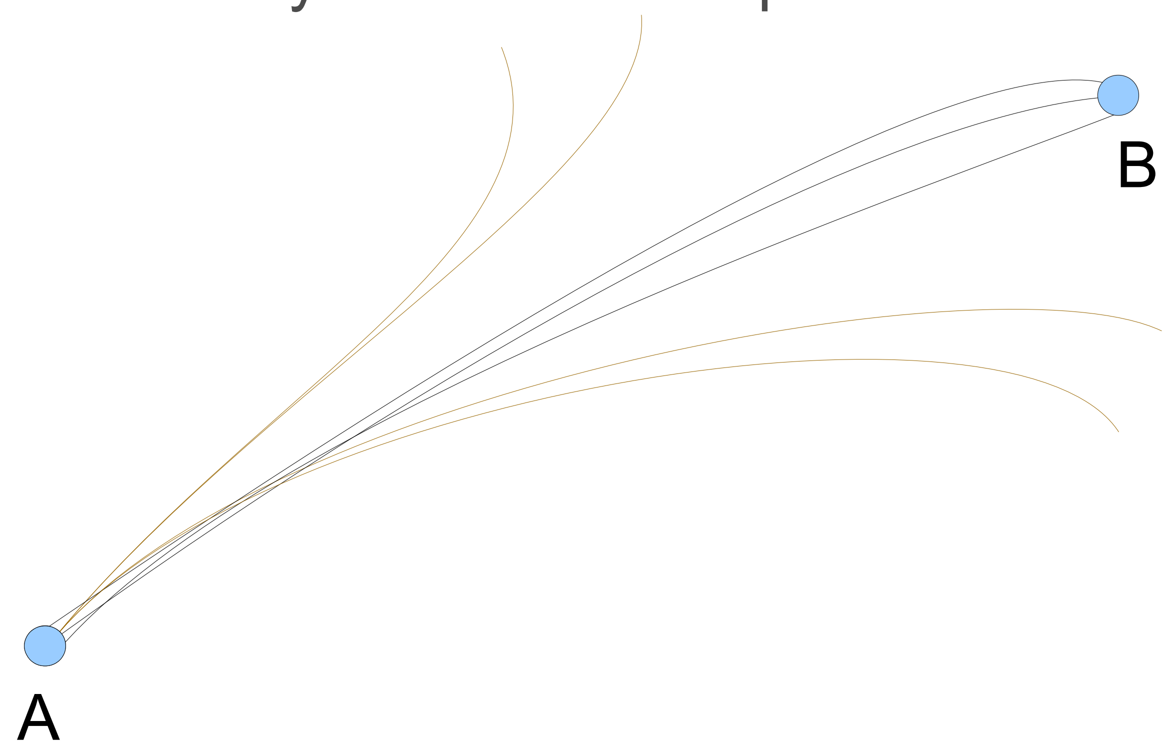


1. Draw sample displacement, v(x), from distribution
2. Update position, x
3. Goto step 1

Just like a particle system.

Connectivity estimation

- Generate N fibers starting at point A
  - N = millions
- Connectivity of A to B is $n_{A,B}$ / N
  - Initialize $n_{A,B}$ to 0
  - Increment every time a fiber steps into voxel B

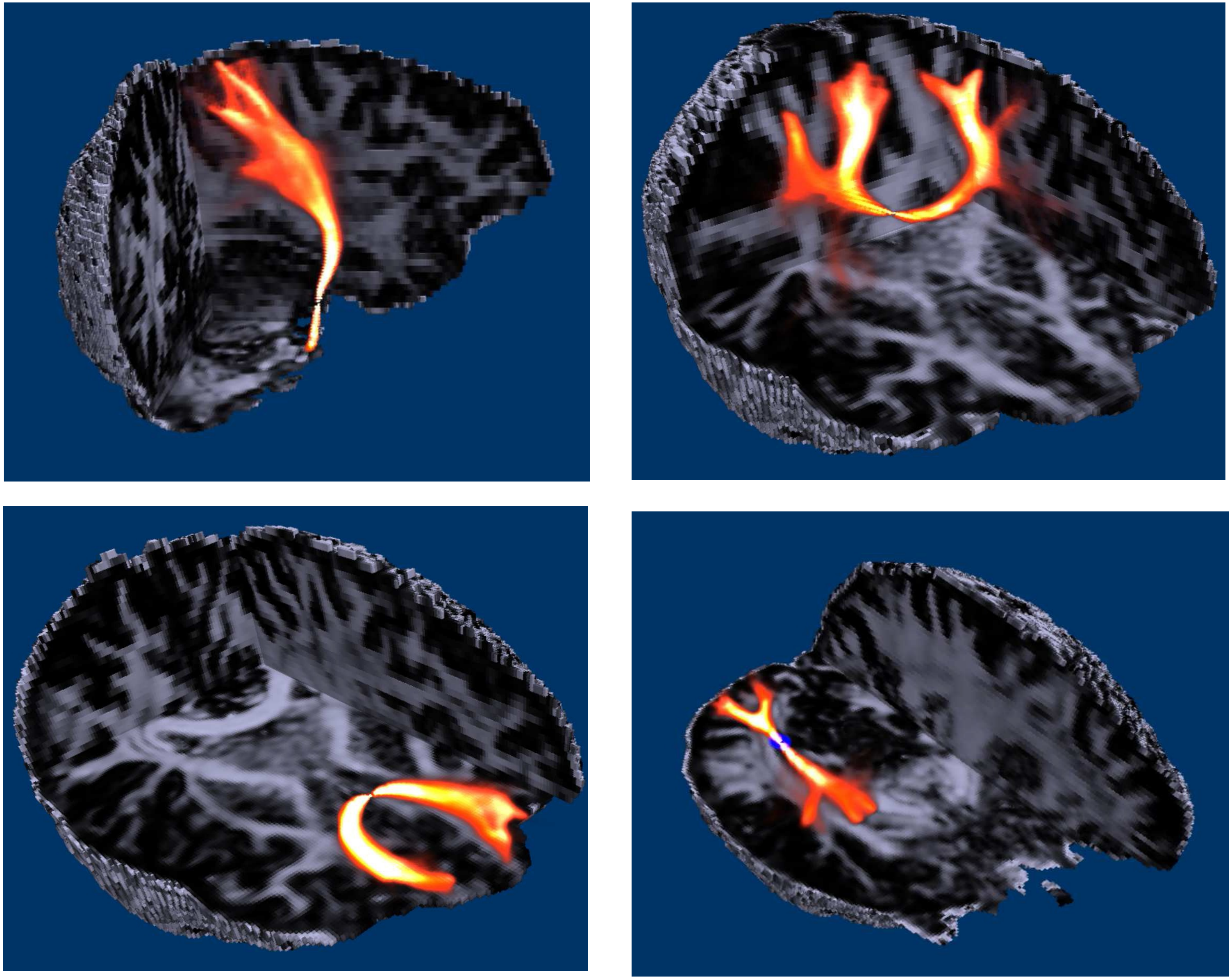


Compute shader outline

```
ivec2 ix = ivec2(gl_GlobalInvocationID.xy);
//load current positions and velocities from images
vec4 x0 = imageLoad(fiberXTex, ix);
vec4 v0 = imageLoad(fiberVTex, ix);

//update position and velocity
vec4 v1 = draw_from_distribution(x0, v0);
vec4 x1 = x0 + v1;

// reinitialize if outside image or outside white matter
if(reinit_needed(x1))
{
  x1.xyz = seed_pos;
  v1.xyz = vec3(0.0);
  atomicCounterIncrement(N); //increment total fiber count
}

//write back new positions
imageStore(particlePosTex, ix, x1);
imageStore(particleVelTex, ix, v1);

// accumulate connectivity if we stepped into a new voxel
ivec3 x0_vox = ivec3(round(worldToConnectVox*x0.xyz));
ivec3 x1_vox = ivec3(round(worldToConnectVox*x1.xyz));
if(x0_vox != x1_vox)
{
 imageAtomicAdd(connectivityTex, x1_vox, 1);
}
```

Results

- <u>Raycast anatomy and connectivity</u>



Compute shader wrap-up

- Useful for general purpose computation within a graphics application
  - Animation, simulation, image processing
- Can read buffers, images, textures, uniforms
- Can write buffers, images
- Not as flexible or fully-featured as other GPGPU solutions
  - If you need to factorize a matrix or solve a linear system it is time to move on to CUDA or OpenCL
- Beware of read/write conflicts and use synchronization when appropriate
- Benchmark to find optimal local workgroup size for your hardware