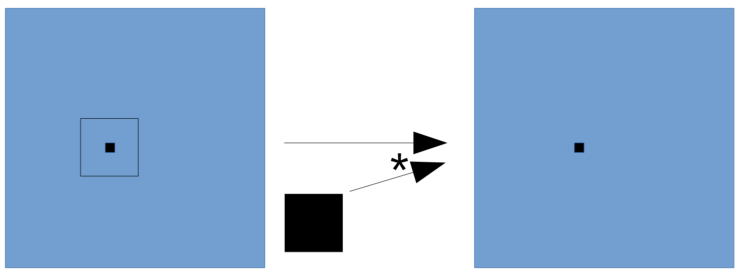
Convolution Blur Filter in a COMPUTE Shader

Filter

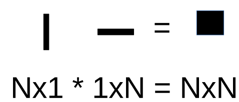
Input: A 2D image, and an $n \times n$ kernel of weights (which sum to one)
Output: A 2D image, where each output pixel is the weighted average of a corresponding $n \times n$ square of input pixels times the respective kernel weights.

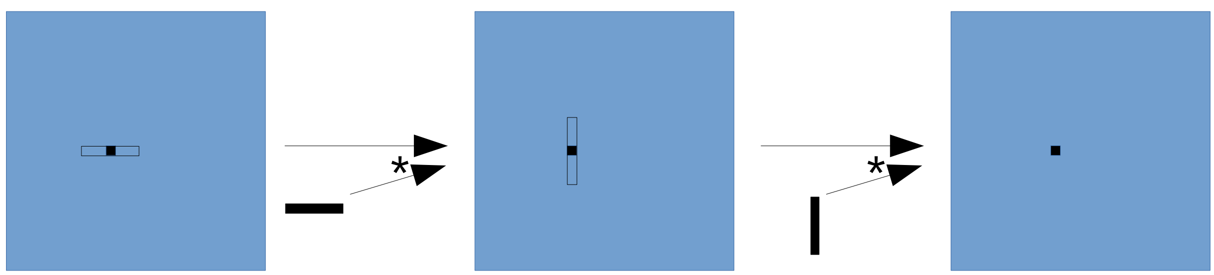
What we want: (This is $O(n^2)$)



What we implement: (This is equivalent but only $O(2n)$.)
Write kernel as product of row and column weights

Perform filter in two steps, one horizontal, one vertical





Building the kernel weights

Build the weight array from a Gaussian bell curve. The true bell curve has infinite tails approaching zero asymptotically. We clamp the range of values to avoid ineffective near-zero weights.

- The width of the kernel is $2w+1$ where the half-width is w .
- The values of the weights are calculated with

  $$e^{-\frac{1}{2}\left(\frac{i}{s}\right)^2} \text{ for i in range } -w \cdots w$$

  where $s=w/2$ controls the width of the bell curve to match the number of desired weights.
- Beware: The range of weights is best considered symmetrically as $-w \cdots w$ whereas the storage in an array must index $0 \cdots 2w$.
- Normalize the array of weights to sum to one.

Efficient filtering in a compute shader

See page 16 of presentation:
http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Efficient%20Compute%20Shader%20Programming.pps

Idea
- **128 threads:** (arranged conceptually into a 128x1 row)
  - will read 128+2w pixels into shared memory
    - one pixel each thread, plus one extra for first 2w threads
  - will compute (and write) 128 output pixels
    - each as a sum of $2w+1$ weights times $2w+1$ pixels

Application will create, use, and dispatch (i.e., run) the shader in thread-groups (tiles) which cover the full image:
- Tile thread groups (tiles) will be blocks of size 128×1
- The dispatch will issue $width/128\times height$ thread groups.

Shader steps:
- Inputs (uniform variables):
  - src, dst images
  - w: half-size of kernel
  - weights: array of $2w+1$ floats

- Declare thread group to be 128×1
- Declare thread-group-shared-memory **v[128+2*w+1]** floats
  - actually must be constant size: **v[128+<largest filter size>]**
- Compute **gpos** as the position of the output pixel,
  - and the center of the $2w+1$ input pixels
- Compute local-index **i** within the thread group
- Every thread reads and stores one pixel from src image into shared array
  - **v[i]=imageLoad(src, gpos+ivec2(-w,0))**
- Some threads (say the first 2w of them) load an extra pixel out beyond 128
  - **v[i+128]=imageLoad(src, gpos+ivec2(128-w,0))**

- Force synchronization between the filling of shared memory (above)
  - and its use (below).

- Compute sum of **weights[0 … 2w]** times corresponding pixels **v[i … i+2w]**
- Store sum at **gpos** in **dst** image
  - **imageStore(dst, gpos, sum)**

Compute Shaders

Application code:

Create compute shader

Same as other shaders,
but use **GL_COMPUTE_SHADER** in **glCreateShader** call
Cannot coexist with other shaders in a shader program

CPU invokes computer shader enough times to tile an image:

```
glUseProgram(programID)
// Set all uniform and image variables
glDispatchCompute(W/128, H, 1) // Tiles WxH image with groups sized 128x1
glUseProgram(0)
```

Send block of weights to shader (as a uniform block)

```
glGenBuffers(1, &blockID) // Generates block
bindpoint = ?; // Start at zero, increment for other blocks

loc = glGetUniformBlockIndex(programID, “blurKernel”)
glUniformBlockBinding(programID, loc, bindpoint)

glBindBuffer(GL_UNIFORM_BUFFER, blockID)
glBindBufferBase(GL_UNIFORM_BUFFER, bindpoint, blockID)
glBufferData(GL_UNIFORM_BUFFER, #bytes, data, GL_STATIC_DRAW)
```

Send two textures (input and output) to the shader as an image2Ds

```
imageUnit = ?;  //  Perhaps 0 for input image and 1 for output image

loc = glGetUniformLocation(programID, “...name...”)  //  Perhaps “src” and “dst”.
glBindImageTexture(imageUnit, textureID, 0, GL_FALSE, 0, GL_READ_ONLY, GL_RGBA32F)
glUniform1i(loc, imageunit)
// Change GL_READ_ONLY to GL_WRITE_ONLY for output image
// Note: GL_RGBA32F means 4 channels (RGBA) of 32 bit floats.
```

Shader code

```
#version 430  // Version of OpenGL with COMPUTE shader support
layout (local_size_x = 128, local_size_y = 1, local_size_z = 1) in; // Declares thread group size

uniform blurKernel {float weights[101]; }; // Declares a uniform block

layout (rgba32f) uniform readonly  image2D src; // src image as 4 channel 32bit float readonly
layout (rgba32f) uniform writeonly image2D dst; // dst image as 4 channel 32bit float writeonly

shared float v[128+101]; // Variable shared with other threads in the 128x1 thread group

void main() {
   ...
   ivec2 gpos = ivec2(gl_GlobalInvocationID.xy); // Combo of groupID, groupSize and localID

   uint i = gl_LocalInvocationID.x; // Local thread id in the 128x1 thread groups128x1

   v[i] = imageLoad(src, gpos+...);  // read an image pixel at an ivec2(.,.) position
   if (i<2*w) v[i+128] = imageLoad(src, gpos+...);  // read extra 2*w pixels

   barrier(); // Wait for all threads to catch up before reading v[]
   ...

   imageStore(dst, gpos, ...); // Write to destination image
```