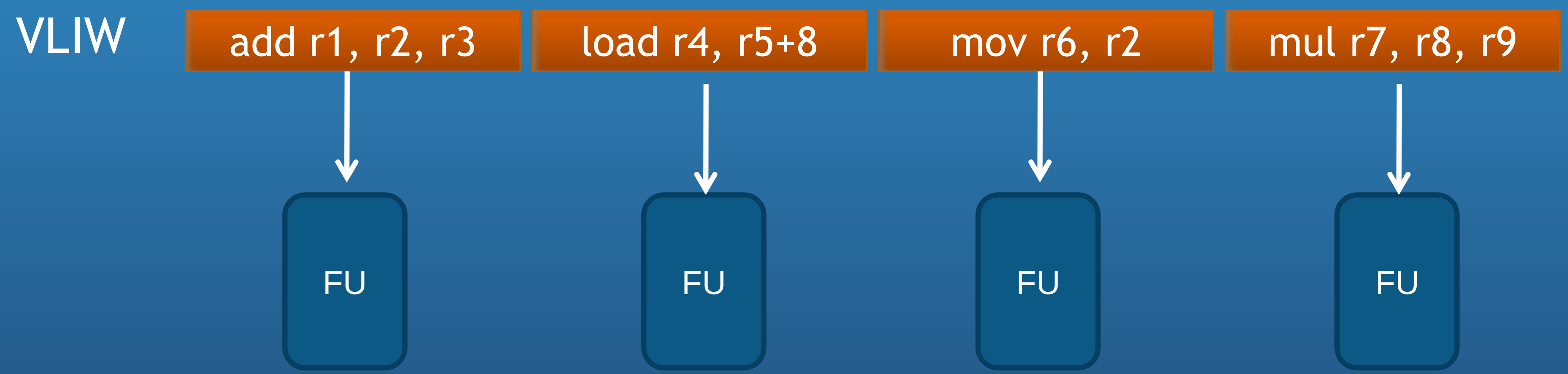
# Vector Hardware and OpenCL Images

Matt Sellitto
Dana Schaa
Northeastern University
NUCAR

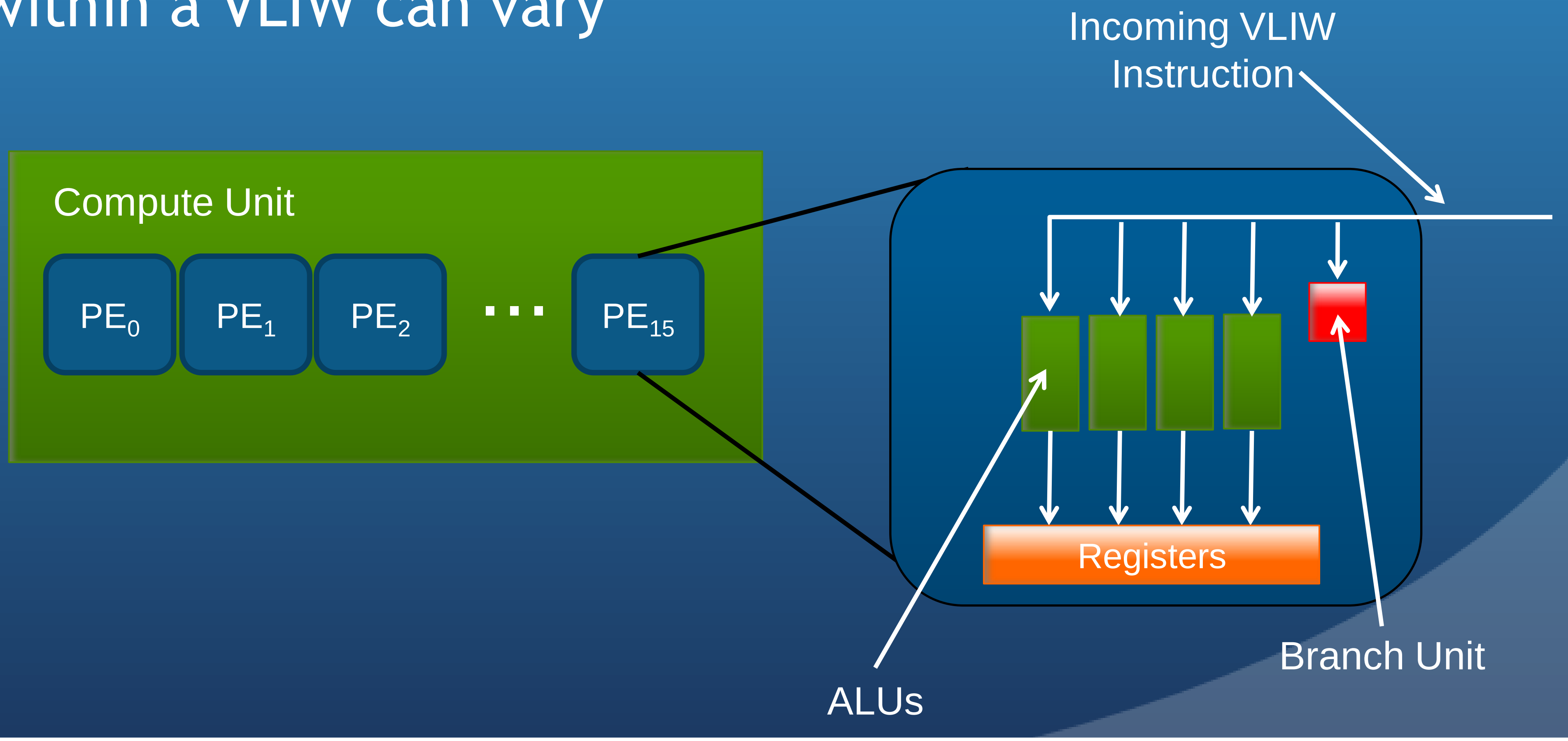
# Very Long Instruction Word

- At compile time, multiple instructions are combined into a single (long) instruction
  - As many execution units as width of VLIW
  - Takes advantage of ILP without complex hardware



# Vector Hardware

- AMD "Cayman" hardware (e.g., Radeon 6970)
- Each PE executes a 4-way VLIW instruction
  - The compiler can pack up to 4 instructions to be executed at a time
  - The same VLIW is executed by all PEs, but the instructions within a VLIW can vary



# Vector Hardware

- For complete utilization, there must be enough instruction level parallelism
- Compiler cannot always find enough instructions to pack into a VLIW
  - Data dependencies
  - Conditional statements
  - etc.
- If vector data types are used, compiler will be much more likely to find instructions to pack

# Vector Datatypes

- Data is expressed as a vector by adding a suffix to the type
  - float4: vector of four floating-point elements
  - int2: vector of two integer elements
- Elements of the vector are accessed using XYZW notation

```
float4 data = {0, 0, 0, 0};
data.x = 5.0;  // access individual element
data *= 2.0;   // apply to all elements
```

# Vector Datatypes

- In OpenCL, an array of floats is specified as float4 by setting datatype in the kernel signature
- Copy example
  - Each work item copies 4 elements from input array to output array

```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```

# Vector Datatypes

- Vector operations are ideal for data transfers as well
- Comparison of vector to scalar transfer on Radeon 5870 GPU

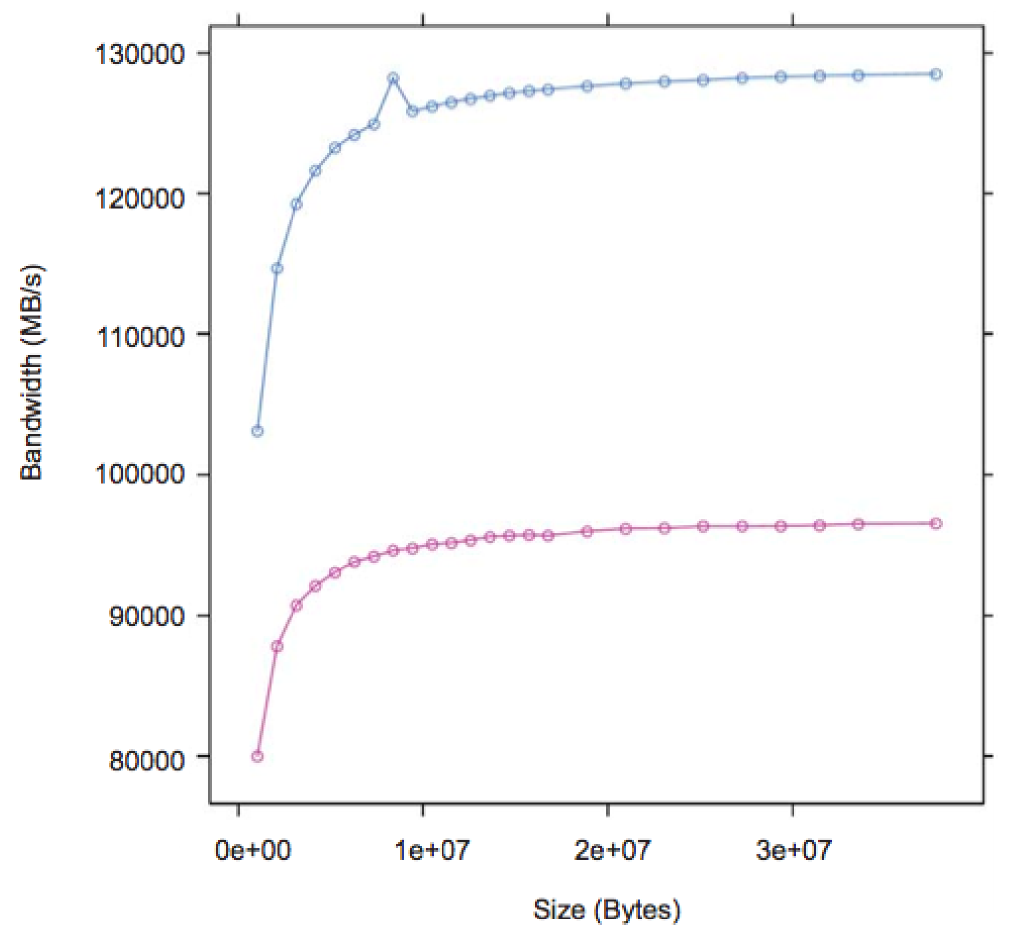



**Figure 4.4 Two Kernels: One Using float4 (blue), the Other float1 (red)**

# Vector Datatypes

- Implication of vector data types
  - Each work item computes multiple results (not always the case)
- What if algorithm isn't suited for vector hardware?
  - Use scalar data types, rely on compiler for packing
- Why vector hardware?  Graphics!
  - Images are commonly represented as RGBA values

# OpenCL Images

- Buffers are used to store 1D data (similar to arrays in C)
  - Data is stored contiguously in memory
  - Addressable using pointer arithmetic

    ```
    A[i] = B[i] + C[i]
    ```

  - Data can be scalar, vector, or user-defined structure
- Images are multidimensional, opaque data types
  - Data is accessed indirectly
    - Physical layout in memory is unknown
    - Coordinates, etc are passed to lookup function which returns data from desired location
  - Data types and formats are predefined

# OpenCL Images

- Why use images?
  - GPUs automatically cache image data
    - 2D or 3D spatial caching (based on image dimensions)
  - Automatic bounds checking and handling of out of bound accesses
    - Return 0, clamp to nearest border pixel, etc
    - Very efficient to not check bounds between multiple accesses!
  - Automatic hardware-based interpolation between pixels

# OpenCL Images

```
cl_mem    clCreateImage2D (cl_context context,
                           cl_mem_flags flags,
                           const cl_image_format *image_format,
                           size_t image_width,
                           size_t image_height,
                           size_t image_row_pitch,
                           void *host_ptr,
                           cl_int *errcode_ret)
```

- 2D or 3D images can be created
  - Similar to buffer creation except height and width is specified
  - Pitch is optionally given (to optimize for specific hardware)
  - Image format must be supplied (next slide)
- Images are based on RGBA graphics format
  - Most explicit example of OpenCL bending towards GPUs instead of the other way around

# Image formats

```
typedef struct _cl_image_format {
     cl_channel_order    image_channel_order;
     cl_channel_type     image_channel_data_type;
} cl_image_format;
```

- Format descriptor defines image order and data type
- Order is the data layout (based on RGBA/vector type)
  - `CL_RBGA, CL_R, CL_RG, etc`
  - When working with non-RGBA data, only vector width is important
- Data type defines the type and size of each element in the vector
  - `CL_SIGNED_INT32, CL_FLOAT, etc.`

# Transferring Images

- An array on the host is written to an image on the device using clEnqueueWriteImage()
- Images are read using clEnqueueReadImage()
- Similar to clEnqueue{Read|Write}Buffer except
  - Instead of offset, origin is provided
  - Instead of number of bytes to access, a dimensions for a region are provided
  - Pitch is also provided if used when creating image

# Reading Images

- On the device, images are accessed using read_image<type>
  - read_imagef() for floating point data
  - read_imagei() for integer data
- A pointer to the image, the coordinates to access, and information about how to read the image (called a *sampler*) are all provided
- Regardless of how many channels are used (CL_R = 1 channel, CL_RGBA = 4 channels), the function to read data from an image returns a 4-element vector

```
float4 read_imagef (image2d_t image,
                    sampler_t sampler,
                    int2 coord)
```

# Samplers

- Consist of three options describing how data should be accessed

1. Normalized coordinates
   - Should the data be treated as coordinates from 0 to width-1 (FALSE), or normalized between 0.0 and 1.0 (TRUE)
2. Address mode
   - What to do if data access is out of bounds (repeat border pixel, return 0, etc.). Very useful (avoids conditional checks)
3. Filter mode
   - Pick the nearest pixel, or linearly interpolate between pixels

```
const sampler_t  samplerA = CLK_NORMALIZED_COORDS_TRUE |
                            CLK_ADDRESS_REPEAT         |
                            CLK_FILTER_NEAREST;
```

# Image Example (Host code)

```
// host array
float *A = (float*)malloc(sizeof(float)*16);

// Image format (single channel floats)
cl_image_format imgFmt = {CL_R, CL_FLOAT};

// Create image (4 rows by 4 cols)
cl_mem imgA = clCreateImage2D(.., imgFmt, 4, 4, ...)

// Copy image to device
float[3] origin = {0,0,0};
float[3] region = {4,4,1};
clEnqueueWriteImage(..., imgA, ..., origin, region, A, ...);
```

# Image Example (Kernel code)

```
const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
                          CLK_ADDRESS_CLAMP_TO_EDGE   |
                          CLK_FILTER_NEAREST;

__kernel
void imgCopy(__read_only image2d_t input,
             ...
{

  int2 coords;
  coords.x = get_global_id(0);
  coords.y = get_global_id(1);

  float4 data = read_imagef(input, sampler, coords);

  ...
}
```

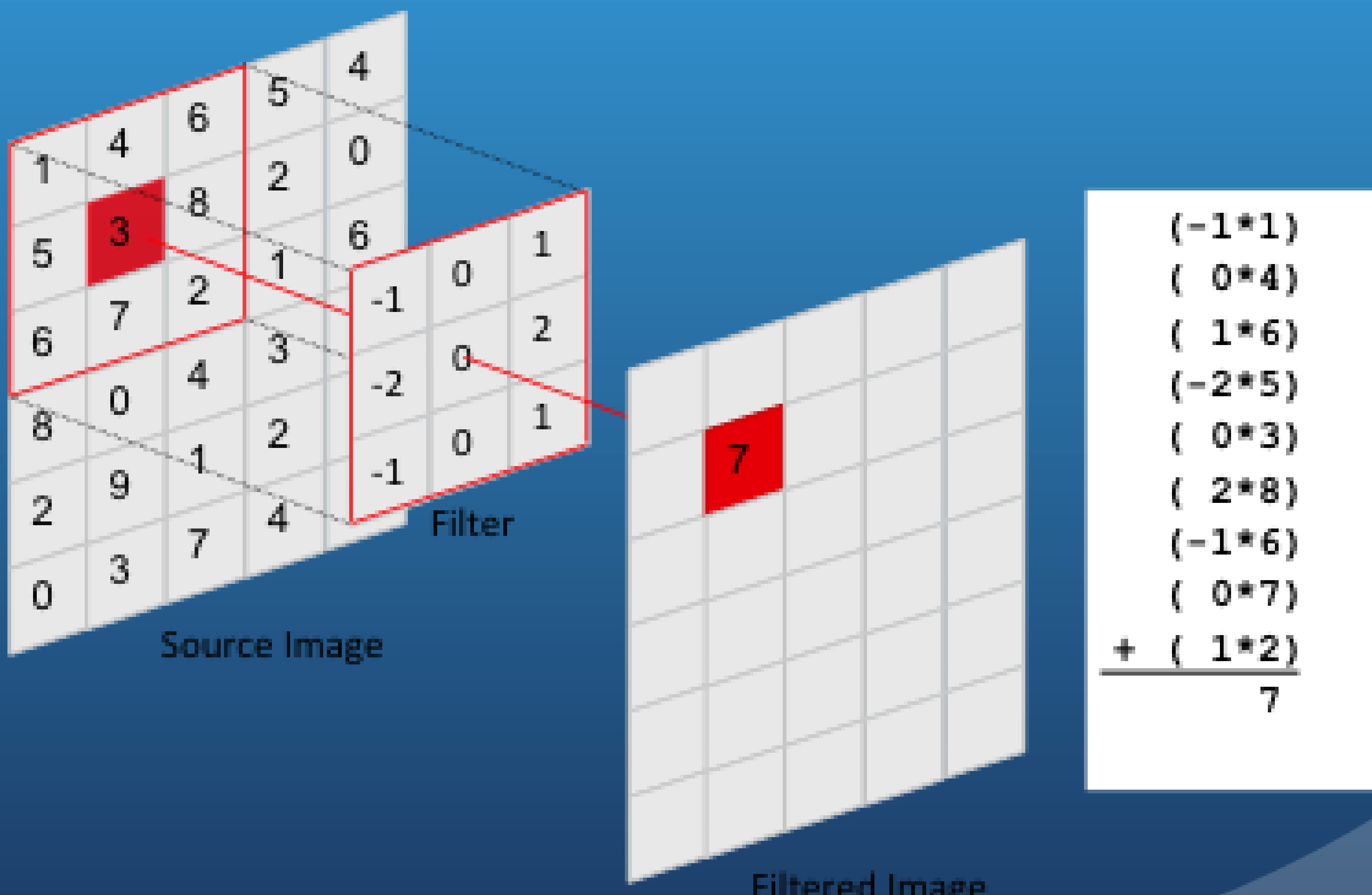
# Writing Images

```
void write_imagef (image2d_t image,
                   int2 coord,
                   float4 color)
```

- Writing to an image requires a 4-element vector, *color*, that matches the image format defined for the image
- The coordinates must be valid (in bounds) and non-normalized

# Example: Convolution

- Convolution processes an image by weighting pixels in a neighborhood
  - The matrix of the weights is a *filter*



# Convolution: Algorithm

- In OpenCL, outer two loops map to work items

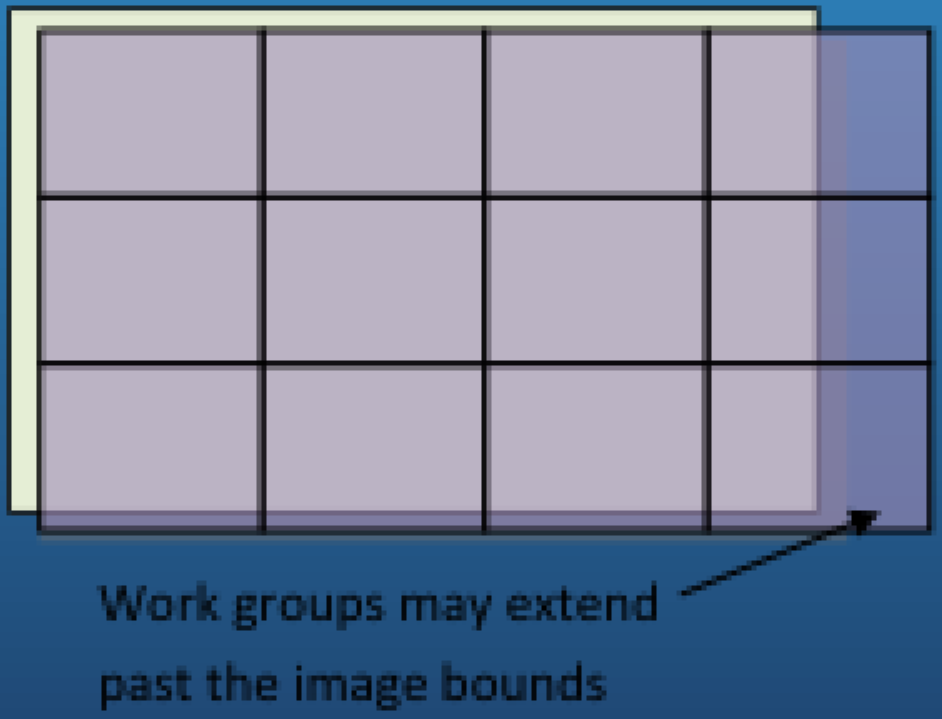
```
// hfw == half filter width
// Iterate over the rows of the source image
for(int i = 0; i < rows; i++) {

   // Iterate over the columns of the source image
   for(int j = 0; j < cols; j++) {
      sum = 0; // Reset sum for new source pixel

      // Apply the filter to the neighborhood
      for(int k = -hfw; k <= hfw; k++) {
         for(int l = -hfw; l <= hfw; l++) {
            if(i+k >= 0 && i+k < rows && j+l >= 0 && j+l < cols) {
               sum += Image[i+k][j+l] * Filter[k+hfw][l+hfw];
            }
         }
      }
      outputImage[i][j] = sum;
   }
}
```

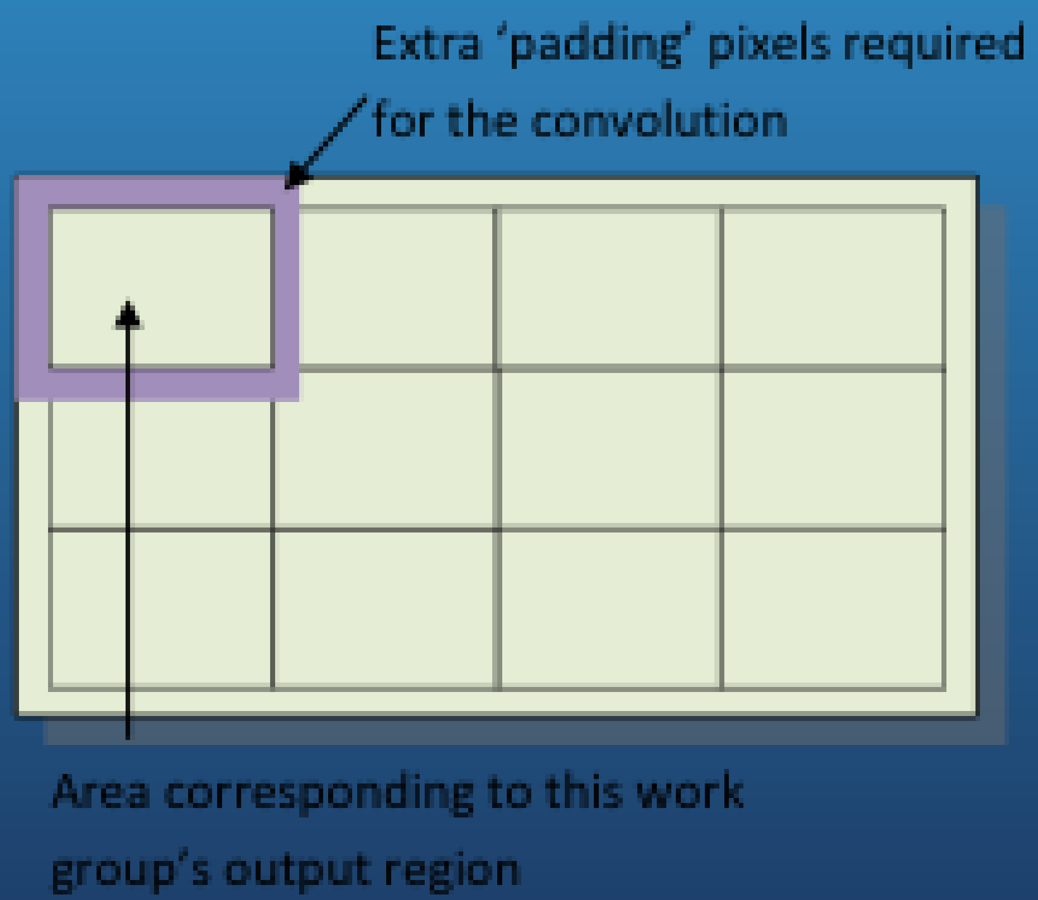
# Convolution: Challenges

- Challenges of convolution
  - Since work group sizes are fixed, there may be more work items created than pixels to be computed
    - We need to ensure each work item is not reading out of bounds



# Convolution: Challenges

- Challenges of convolution
  - The border pixels (half of the filter size) will read out of bounds
    - These either needed to be treated as a special case (requiring conditional checks) or not produce values (information is lost)



# Convolution: Using Buffers

- Buffer implementation
  - Exactly the right data can be manually cached
    - Potentially better performance
    - Requires detailed knowledge of memory architecture
    - Architecture-specific code
    - Error prone
  - Bounds checking must be done using conditional statements
    - Padding can be used to avoid conditional checks
      - Potentially time consuming

# Convolution: Using Images

- Image implementation
  - Automatic bounds checking
    - Return zero or clamp to border pixel
    - Cleaner/fewer lines of code
  - Automatic caching of data
    - Cleaner/fewer lines of code
    - Get good performance for little effort

# Convolution:

- Write an image-based implementation of convolution for OpenCL
- Skeleton code provided
  - Reads in image from file
  - Compares against known result
  - Saves output image to file
- Bonus exercises (using events to measure performance)
  - Try loop unrolling the inner loop in the convolution
  - Try loop unrolling both loops
  - Use mul24 for multiplications inside the kernel

# Convolution: Algorithm

```
i == row
j == col

// Apply the filter to the neighborhood
for(int k = -hfw; k <= hfw; k++) {
    for(int l = -hfw; l <= hfw; l++) {
        sum += Image[i+k][j+l] * Filter[k+hfw][l+hfw];
    }
}

// Write the output value
outputImage[i][j] = sum;
```