OpenCL Examples

Prof. Yan Luo

Department of Electrical and Computer Engineering
University of Massachusetts Lowell

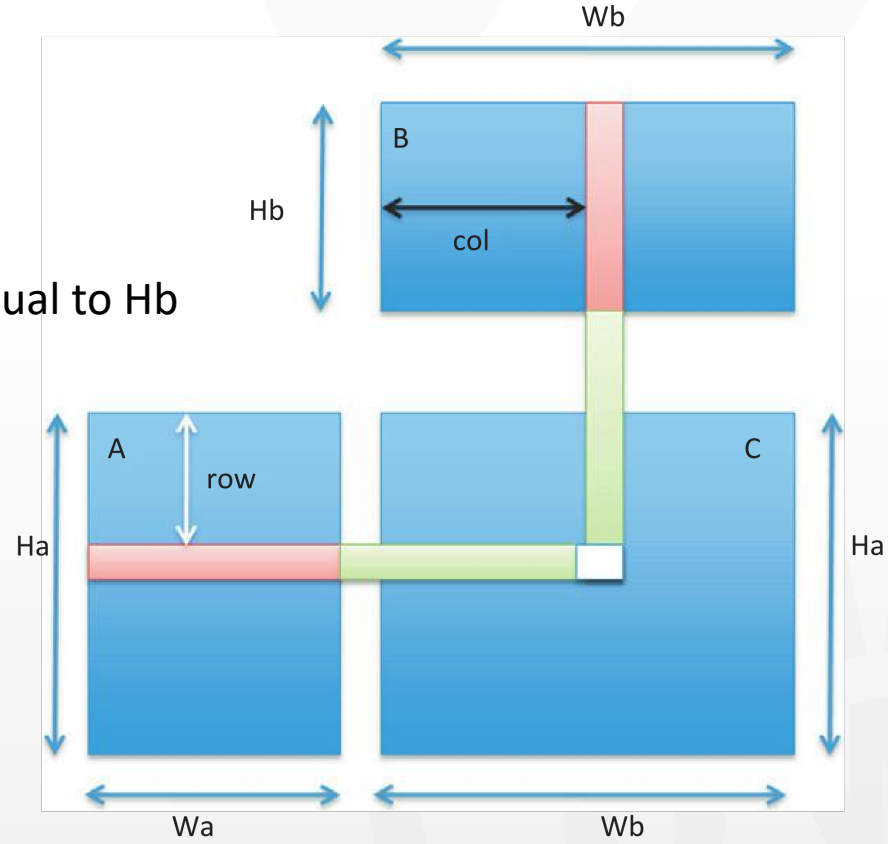
Outline

- Matrix Multiplication
- Image Rotation
- Image Convolution
- Demonstrations

Illustration of Matrix Multiplication

A is Wa x Ha
B is Wb x Hb
Note: Wa must equal to Hb

C is Wb x Ha

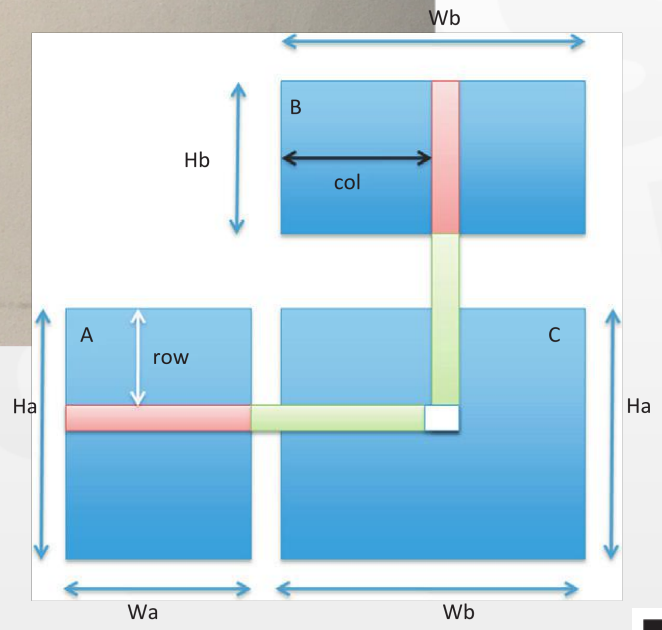


A * B = C

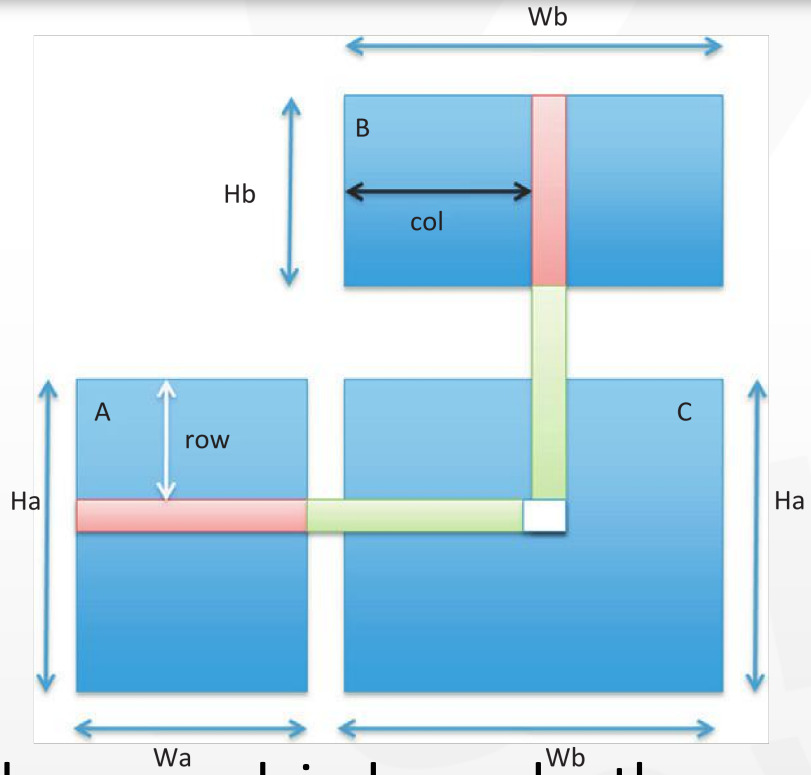
Matrix Multiplication – in C

Nested loops

```
// Iterate over the rows of Matrix A
for(int i = 0; i < heightA; i++) {
    // Iterate over the columns of Matrix B
    for(int j = 0; j < widthB; j++) {
        C[i][j] = 0;
        // Multiply and accumulate the values in the current row
        // of A and column of B
        for(int k = 0; k < widthA; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```



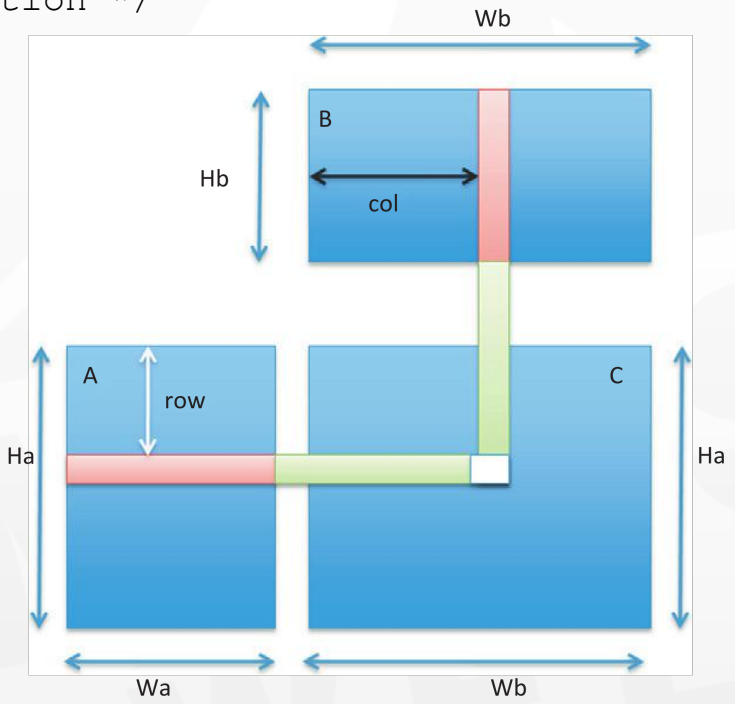
Illustration



- Two outer for-loops work independently
- a separate work item can be created for each output element of the matrix
- Two outer for-loops are mapped to the 2-dimensional range of work items

Matrix Multiplication in OpenCL

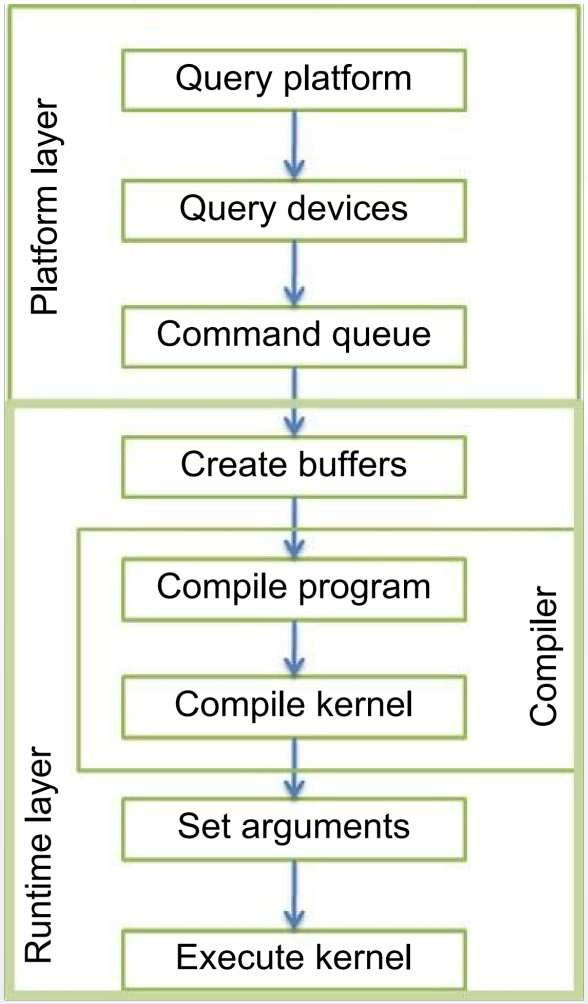
```
/* widthA=heightB for valid matrix multiplication */
__kernel void simpleMultiply(
    __global float *outputC,
    int widthA,
    int heightA,
    int widthB,
    int heightB,
    __global float *inputA,
    __global float *inputB){

    /* get global position in Y direction */
    int row = get_global_id (1);
    /* get global position in X direction */
    int col = get_global_id (0);
    float sum = 0.0f;
    /* calculate result of one element of Matrix C */
    for (int i=0; i<widthA; i++) {
        sum += inputA[row*widthA + i] * inputB[i*widthB + col];
    }
    outputC[row*widthB + col] = sum;
}
```



Things are Still Missing Here

- The kernel is a data-parallel kernel, we still need:
  1) Setup the "environment" for computation
  2) Create context, command queue
  3) Have input data ready in the device memory for computation
  4) Dispatch kernel
  5) Collect results by reading from device memory

Programming Steps



1. Setup Environment

Declare a context, choose device type, create context and command queue

```
/* Get Platform and Device Info */
clGetPlatformIDs(1, NULL, &platformCount);
platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) *
platformCount);
clGetPlatformIDs(platformCount, platforms, NULL);
// we only use platform 0, even if there are more plantforms
// Query the available OpenCL device.
ret = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_DEFAULT, 1,
&device_id, &ret_num_devices);
ret = clGetDeviceInfo(device_id, CL_DEVICE_NAME, DEVICE_NAME_LEN,
dev_name, NULL);
printf("device name= %s\n", dev_name);

 /* Create OpenCL context */
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
/* Create Command Queue */
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

2. Declare Buffers and Move Data

```
 /* We assume A, B, C are float arrays which
   have been declared and initialized */
/* allocate space for Matrix A on the device */
cl_mem bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY,
wA*hA*sizeof(float), NULL, &ret);
/* copy Matrix A to the device */
clEnqueueWriteBuffer(command_queue, bufferA, CL_TRUE, 0,
 wA*hA*sizeof(float), (void *)A, 0, NULL, NULL);
/* allocate space for Matrix B on the device */
cl_mem bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY,
wB*hB*sizeof(float), NULL, &ret);
/* copy Matrix B to the device */
clEnqueueWriteBuffer(command_queue, bufferB, CL_TRUE, 0,
wB*hB*sizeof(float), (void *)B, 0, NULL, NULL);
/* allocate space for Matrix C on the device */
cl_mem bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
wC*hC*sizeof(float), NULL, &ret);
```

```
cl_mem clCreateBuffer (cl_context context,
                       cl_mem_flags flags,
                       size_t size,
                       void *host_ptr,
                       cl_int *errcode_ret)
```

cl_mem_flags	Description
CL_MEM_READ_WRITE	This flag specifies that the memory object will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	This flags specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with CL_MEM_WRITE_ONLY inside a kernel is undefined.
CL_MEM_READ_ONLY	This flag specifies that the memory object is a read-only memory object when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined.
CL_MEM_USE_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by *host_ptr* as the storage bits for the memory object. OpenCL implementations are allowed to cache the buffer contents pointed to by *host_ptr* in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same *host_ptr* or overlapping host regions is considered to be undefined.
CL_MEM_ALLOC_HOST_PTR	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.
CL_MEM_COPY_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by *host_ptr*. CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive. CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.

3. Runtime Kernel Compilation*

Runtime compilation (or loading binary for FPGA)*

```
// We assume that the program source is stored in the variable
// 'source' and is NULL terminated
cl_program myprog = clCreateProgramWithSource (
    ctx,
    1,
    (const char**)&source,
    NULL,
    &ciErrNum);

// Compile the program. Passing NULL for the 'device_list'
// argument targets all devices in the context
ciErrNum = clBuildProgram(myprog, 0, NULL, NULL, NULL, NULL);

// Create the kernel
cl_kernel mykernel = clCreateKernel(
    myprog,
    "simpleMultiply",
    &ciErrNum);
```

4. Run the Kernel Program

```
// Set the kernel arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem), (void *)&d_C);
clSetKernelArg(mykernel, 1, sizeof(cl_int), (void *)&wA);
clSetKernelArg(mykernel, 2, sizeof(cl_int), (void *)&hA);
clSetKernelArg(mykernel, 3, sizeof(cl_int), (void *)&wB);
clSetKernelArg(mykernel, 4, sizeof(cl_int), (void *)&hB);
clSetKernelArg(mykernel, 5, sizeof(cl_mem), (void *)&d_A);
clSetKernelArg(mykernel, 6, sizeof(cl_mem), (void *)&d_B);

// Set local and global workgroup sizes
//We assume the matrix dimensions are divisible by 16
size_t localws[2] = {16,16} ;
size_t globalws[2] = {wC, hC};

// Execute the kernel
ciErrNum = clEnqueueNDRangeKernel(
    myqueue,
    mykernel,
    2,
    NULL,
    globalws,
    localws,
    0,
    NULL,
    NULL);
```

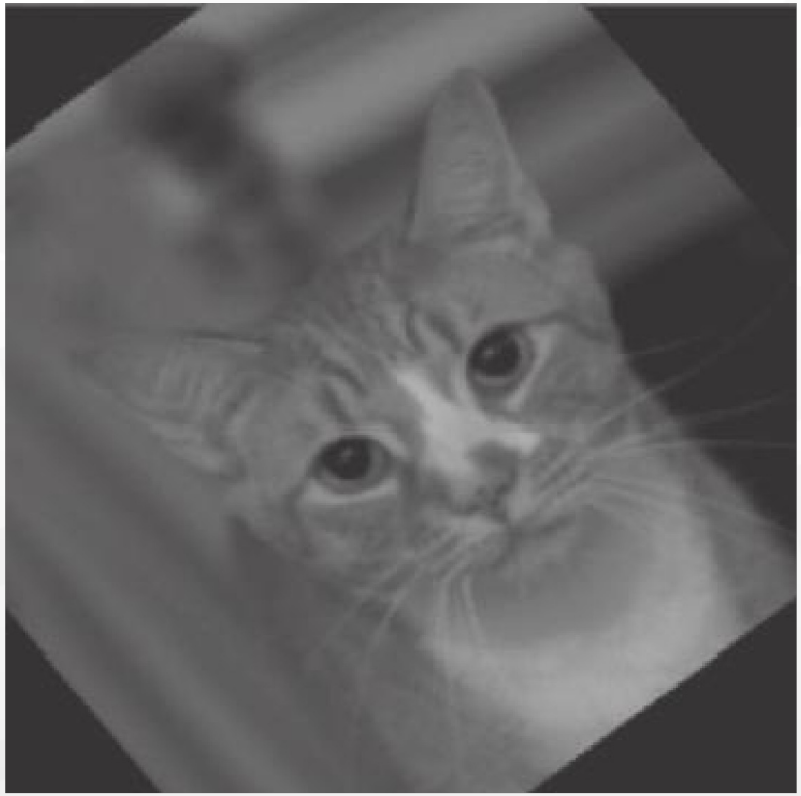
5. Obtain Results

```
// Read the output data back to the host
ciErrNum = clEnqueueReadBuffer(
    myqueue,
    d_C,
    CL_TRUE,
    0,
    wC*hC*sizeof(float),
    (void *)C,
    0,
    NULL,
    NULL);
```

Demonstrations

Image Rotation

Original image



After rotation of 45°

Math behind Rotation

The coordinates of a point (x_1, y_1) when rotated by an angle θ around (x_0, y_0) become (x_2, y_2), as shown by the following equation:

$$x_2 = \cos(\theta) * (x_1 - x_0) + \sin(\theta) * (y_1 - y_0)$$
$$y_2 = -\sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0)$$

By rotating the image about the origin (0, 0), we get

$$x_2 = \cos(\theta) * (x_1) + \sin(\theta) * (y_1)$$
$$y_2 = -\sin(\theta) * (x_1) + \cos(\theta) * (y_1)$$

Each new x and y coordinate of a pixel can be calculated independently

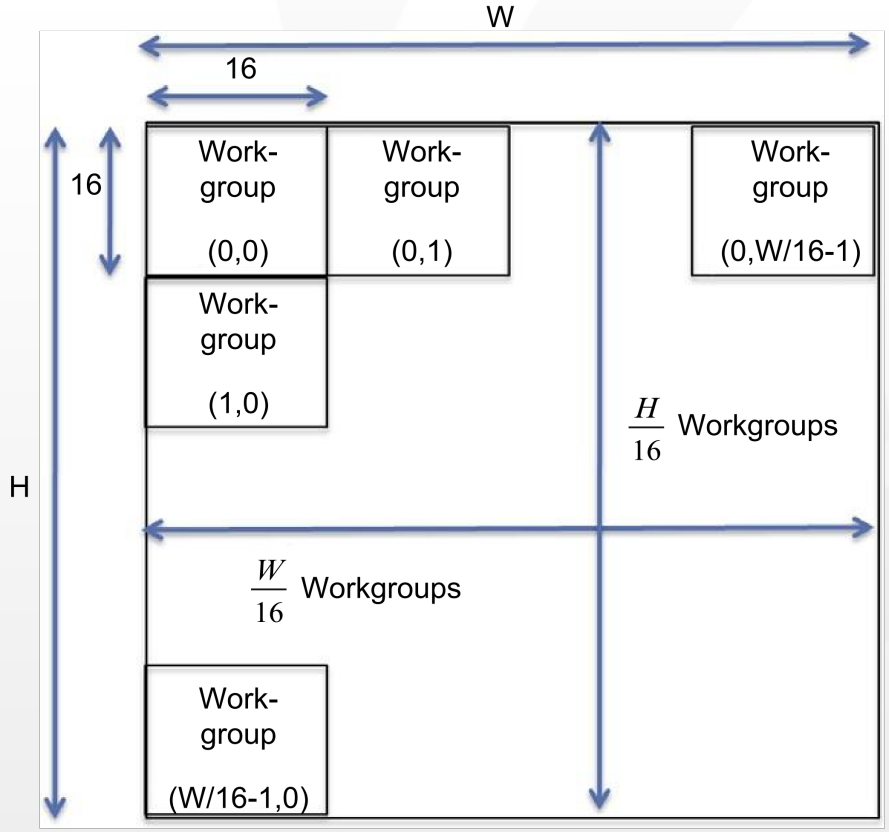
Each work item will calculate the new position of a single pixel
Each work item can obtain the location of pixel using its global ID

Input Image Workgroup Configuration

"Input Decomposition"

Each work item will calculate the new position of a single pixel

Each work item can obtain the location of pixel using its global ID



Input image workgroup configuration

Implementation in OpenCL

```
__kernel void img_rotate(
    __global float* dest_data, __global float* src_data,
    int W, int H,                          //Image Dimensions
    float sinTheta, float cosTheta ) //Rotation Parameters
{
    //Work-item gets its index within index space
    const int ix = get_global_id(0);
    const int iy = get_global_id(1);

    //Calculate location of data to move into (ix,iy)
    //Output decomposition as mentioned
    float xpos = ((float)ix)*cosTheta + ((float)iy)*sinTheta;
    float ypos = -1.0*((float)ix)*sinTheta + ((float)iy)*cosTheta;

    //Bound Checking
    if(((int)xpos>=0) && ((int)xpos< W) &&
       ((int)ypos>=0) && ((int)ypos< H))
       {
       // Read (ix,iy) src_data and store at (xpos,ypos) in
       // dest_data
       // In this case, because we rotating about the origin
       // and there is no translation, we know that (xpos,ypos)
       // will be unique for each input (ix,iy) and so each
       // work-item can write its results independently

       dest_data[(int)ypos*W+(int)xpos]= src_data[iy*W+ix];
    }
}
```

The complete code (C++ binding example)

Step 1: Set Up Environment

```
// Discover platforms
cl::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

// Create a context with the first platform
cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
   (cl_context_properties)(platforms[0])(), 0};

// Create a context using this platform for a GPU type device
cl::Context context(CL_DEVICE_TYPE_ALL, cps);

// Get device list from the context
cl::vector<cl::Device> devices =
   context.getInfo<CL_CONTEXT_DEVICES>();

// Create a command queue on the first device
cl::CommandQueue queue = cl::CommandQueue(context,
   devices[0], 0);
```

Step 2: Declare Buffers and Move Data

```
// Create buffers for the input and output data ("W" and "H"
// are the width and height of the image, respectively)
cl::Buffer d_ip = cl::Buffer(context, CL_MEM_READ_ONLY,
   W*H* sizeof(float));
cl::Buffer d_op = cl::Buffer(context, CL_MEM_WRITE_ONLY,
   W*H* sizeof(float));

// Copy the input data to the device (assume that the input
// image is the array "ip")
queue.enqueueWriteBuffer(d_ip, CL_TRUE, 0, W*H*
   sizeof(float), ip);
```

Step 3: Runtime Kernel Compilation

```
// Read in the program source
std::ifstream sourceFileName("img_rotate_kernel.cl");

std::string sourceFile(
     std::istreambuf_iterator<char>(sourceFileName),
     (std::istreambuf_iterator<char>()));

cl::Program::Sources rotn_source(1,
     std::make_pair(sourceFile.c_str(),
                    sourceFile.length()+1));

// Create the program
cl::Program rotn_program(context, rotn_source);

// Build the program
rotn_program.build(devices);

// Create the kernel
cl::Kernel rotn_kernel(rotn_program, "img_rotate");
```

Step 4: Run the Program

```
// The angle of rotation is theta
float cos_theta = cos(theta);
float sin_theta = sin(theta);

// Set the kernel arguments
rotn_kernel.setArg(0, d_op);
rotn_kernel.setArg(1, d_ip);
rotn_kernel.setArg(2, W);
rotn_kernel.setArg(3, H);
rotn_kernel.setArg(4, cos_theta);
rotn_kernel.setArg(5, sin_theta);

// Set the size of the NDRange and workgroups
cl::NDRange globalws(W,H);
cl::NDRange localws(16,16);

// Run the kernel
queue.enqueueNDRangeKernel(rotn_kernel, cl::NullRange,
   globalws, localws);
```

Step 5: Read Result Back to Host

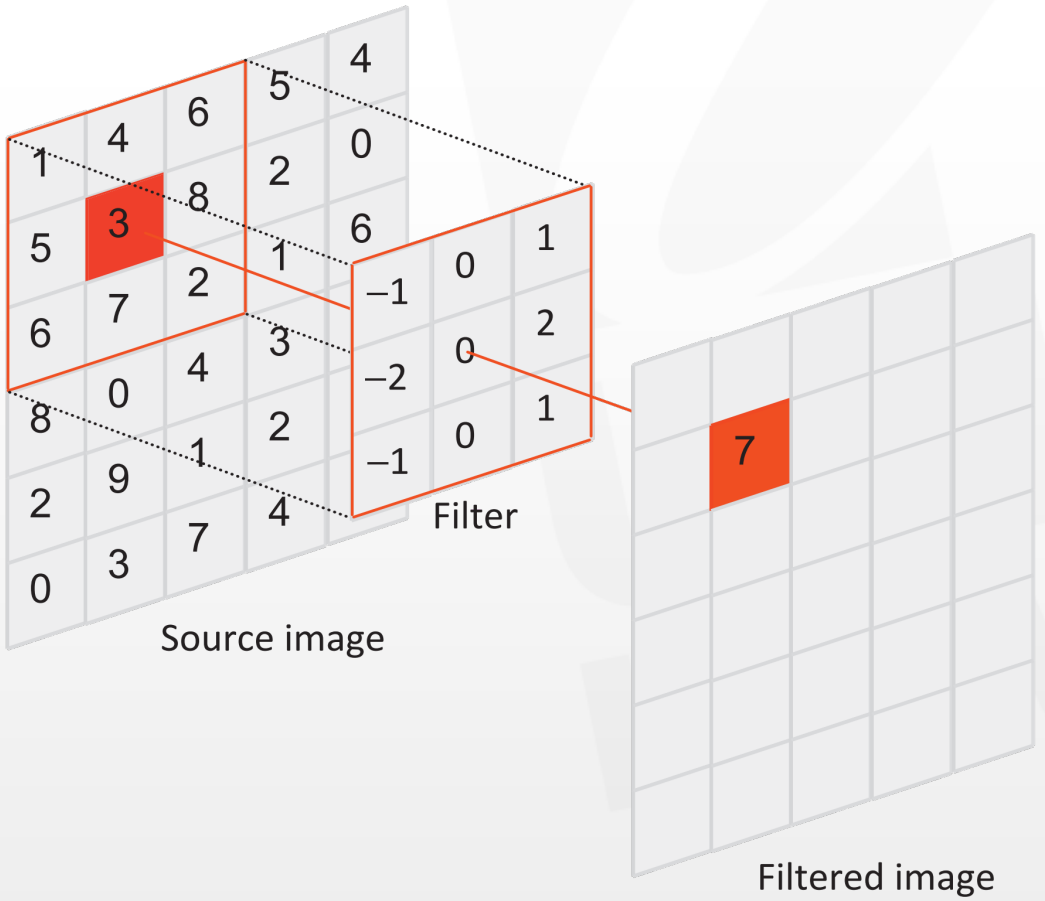
```
// Read the output buffer back to the host
queue.enqueueReadBuffer(d_op, CL_TRUE, 0, W*H*sizeof(float), op
```

Demonstrations

Image Convolution

- Image convolution modifies the value of each pixel in an image using information from neighboring pixels
- Convolution kernel = "filter", defining the influence of neighboring pixels
  - E.g. blurring kernel: take the weighted average of neighboring pixels to reduce the large differences between pixel values

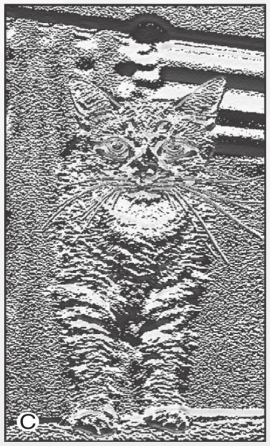
Applying a Convolution Filter to Source Image



Example Filters

original

blurring



Edge detection

Convolution in C

- Outer two loops iterate over the source image, selecting next source pixel
- Inner loop applies the filter to the neighboring pixels

```
// Iterate over the rows of the source image
for(int i = halfFilterWidth; i < rows - halfFilterWidth; i++) {
   // Iterate over the columns of the source image
   for(int j = halfFilterWidth; j < cols - halfFilterWidth; j++) {
      sum = 0; // Reset sum for new source pixel
      // Apply the filter to the neighborhood
      for(int k = - halfFilterWidth; k <= halfFilterWidth; k++) {
         for(int l = - halfFilterWidth; l <= halfFilterWidth; l++) {
            sum += Image[i+k][j+l] *
                   Filter[k+ halfFilterWidth][l+ halfFilterWidth];
         }
      }
      outputImage[i][j] = sum;
   }
}
```

Image Data in OpenCL

- Opaque types that cannot be viewed directly through points in device code
- Multi-dimensional structures
- Not suitable for implementing arbitrary structures
- Intend for access to special function hardware on graphics processors
  - Operations such as transformation of image data based on pixels
  - Such operations would require long instructions sequences with multiple reads
- Image format
  - Channel order: CL_RGB, CL_R, CL_ARGB
  - Channel type: CL_FLOAT, CL_UNORM_SHORT_565, CL_SIGNED_INT32, CL_UNSIGNED_INT8, etc.

1. Create Image and Buffer Objects

- Context, command queue, source image, output image, filter, assumed to already been initialized on the host.
- We need to create image objects on the device side

```
// The convolution filter is 7x7
int filterWidth = 7;
int filterSize = filterWidth*filterWidth; // Assume a square kernel

// The image format describes how the data will be stored in memory
cl_image_format format;
format.image_channel_order     = CL_R;      // single channel
format.image_channel_data_type = CL_FLOAT; // float data type

// Create space for the source image on the device
cl_mem bufferSourceImage = clCreateImage2D(
    context,
    0,
    &format,
    width,
    height,
    0,
    NULL,
    NULL);

// Create space for the output image on the device
cl_mem bufferOutputImage = clCreateImage2D(
    context,
    0,
    &format,
    width,
    height,
    0,
    NULL,
    NULL);

// Create space for the 7x7 filter on the device
cl_mem bufferFilter = clCreateBuffer(
    context,
    0,
    filterSize*sizeof(float),
    NULL,
    NULL);
```

2. Write the Input Data

Copy image data and filter data to device

```
// Copy the source image to the device
size_t origin[3] = {0, 0, 0}; // Offset within the image to copy from
size_t region[3] = {width, height, 1}; // Elements to per dimension
clEnqueueWriteImage(
    queue,
    bufferSourceImage,
    CL_FALSE, origin,
    region,
    0,
    0,
    sourceImage,
    0,
    NULL,
    NULL);

// Copy the 7x7 filter to the device
clEnqueueWriteBuffer(
    queue,
    bufferFilter,
    CL_FALSE,
    0,
    filterSize*sizeof(float),
    filter,
    0,
    NULL,
    NULL);
```

OpenCL Sampler

Sampler objects describe how to access an image

```
cl_sampler clCreateSampler (cl_context context,
                            cl_bool normalized_coords,
                            cl_addressing_mode addressing_mode,
                            cl_filter_mode filter_mode,
                            cl_int *errcode_ret)
```

Parameters

context

Must be a valid OpenCL context.

normalized_coords

Determines if the image coordinates specified are normalized (if *normalized_coords* is `CL_TRUE`) or not (if *normalized_coords* is `CL_FALSE`).

addressing_mode

Specifies how out-of-range image coordinates are handled when reading from an image. This can be set to `CL_ADDRESS_REPEAT`, `CL_ADDRESS_CLAMP_TO_EDGE`, `CL_ADDRESS_CLAMP`, and `CL_ADDRESS_NONE`.

filtering_mode

Specifies the type of filter that must be applied when reading an image. This can be `CL_FILTER_NEAREST` or `CL_FILTER_LINEAR`.

3. Create Sampler Object

```
cl_sampler
sampler=clCreateSampler(
context,
CL_FALSE,

CL_ADDRESS_CLAMP_TO_EDGE,



CL_FILTER_NEAREST,

NULL);
```

- We do not use normalized coordinates.
- Any out-of-bounds access return the value on the border of the image
- Access the closest pixel to a coordinates (instead of interpolate between multiple pixels) if the coordinate lies somewhere in between

4. Compile and Execute the Kernel

```
__kernel
void convolution(
    __read_only image2d_t inputImage,
   __write_only image2d_t outputImage,
        __constant float* filter,
                    int filterWidth,
                sampler_t sampler)
{
   /* Store each work-item's unique row and column */
   int column = get_global_id(0);
   int row = get_global_id(1);

   /* Half the width of the filter is needed for indexing
    * memory later */
   int halfWidth = (int)(filterWidth/2);

   /* All accesses to images return data as four-element vector
    * (i.e., float4), although only the 'x' component will contain
    * meaningful data in this code */
   float4 sum = {0.0f, 0.0f, 0.0f, 0.0f};

   /* Iterator for the filter */
   int filterIdx = 0;
```

```
    /* Each work-item iterates around its local area based on the
     * size of the filter */
    int2 coords; // Coordinates for accessing the image

    /* Iterate the filter rows */
    for(int i = -halfWidth; i <= halfWidth; i++)
    {
       coords.y = row + i;
       /* Iterate over the filter columns */
       for(int j = -halfWidth; j <= halfWidth; j++)
       {
          coords.x = column + j;

          /* Read a pixel from the image. A single channel image
           * stores the pixel in the 'x' coordinate of the returned
           * vector. */
             float4 pixel;
          pixel = read_imagef(inputImage, sampler, coords);
          sum.x += pixel.x * filter[filterIdx++];
       }
    }

    /* Copy the data to the output image */
    coords.x = column;
    coords.y = row;
    write_imagef(outputImage, coords, sum);
}
```

5. Read the Result

```
/* Read the output image back to the host */
clEnqueueReadImage(
    queue,
    bufferOutputImage,
    CL_TRUE,
    origin,
    region,
    0,
    0,
    outputImage,
    0,
    NULL,
    NULL);
```

Demonstrations