UNIT-5

IMAGE CONVOLUTION

- Convolution is a commonly used algorithm that modifies the

value of each pixel in an image by using information from neighbouring pixels.

- A convolution kernel, or filter, describes how each pixel will be influenced by its neighbours.
- Example: a blurring filter will take the weighted average of neighboring pixels so that large differences between pixel values are reduced.
- By using the same source image and changing only the filter, one can produce effects such as sharpening, blurring, edge enhancing, and embossing

IMAGE CONVOLUTION

- Convolution algorithms works by iterating over each pixel in the source image.
- For each source pixel, the filter is centered over the pixel, and the values of the filter multiply the pixel values that they overlay
- A sum of the products is then taken to produce a new pixel value.

IMAGE CONVOLUTION

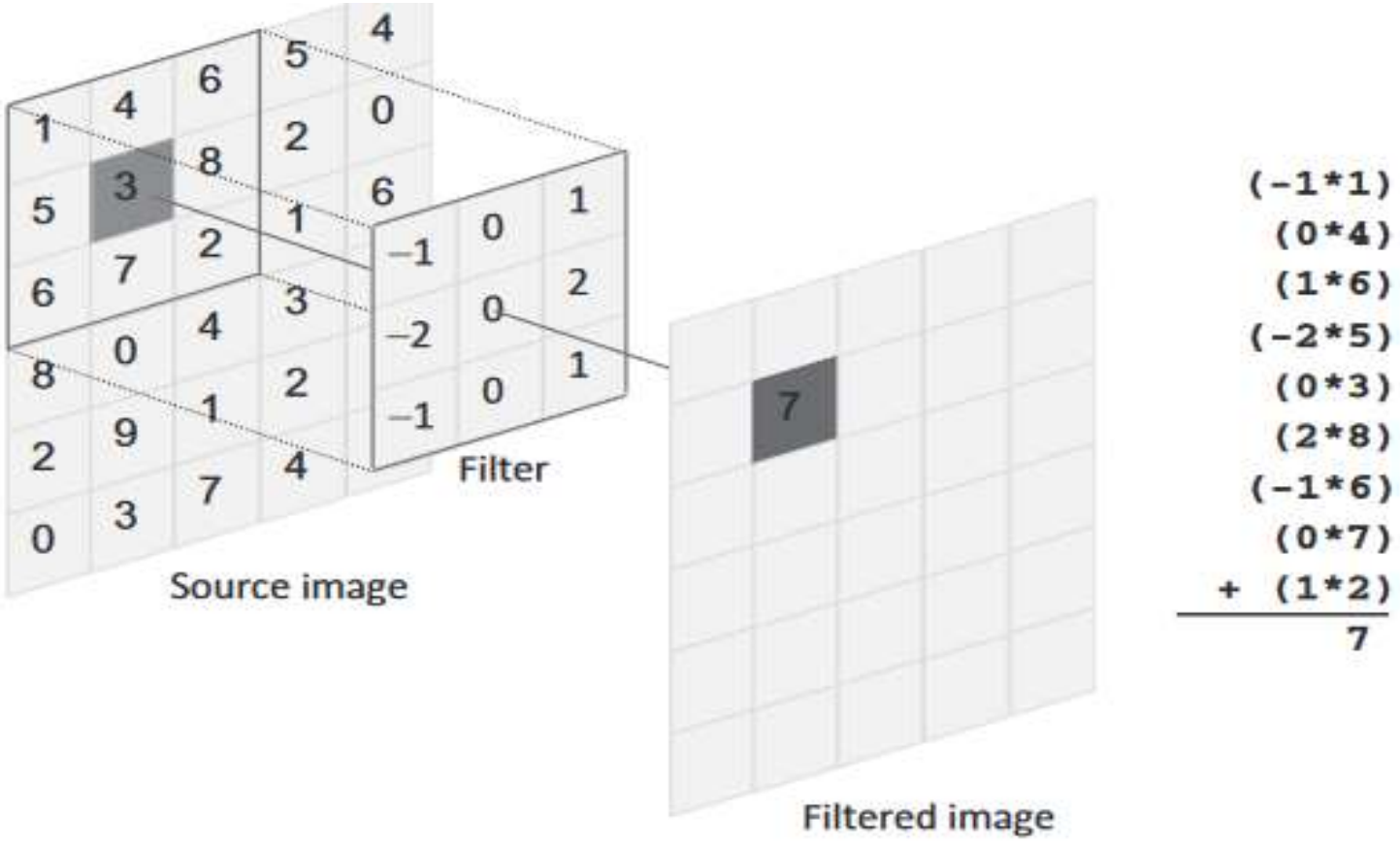

IMAGE CONVOLUTION

- a serial convolution in C/C++. The two outer loops iterate over pixels in the source image, selecting the next source pixel.

- At each source pixel, the filter is applied to the neighboring pixels.

- The filter can try to access pixels that are out-of-bounds. To handle this situation, we provide four explicit checks within the innermost loop to set the out-of-bounds coordinate to the nearest border pixel

IMAGE CONVOLUTION

```
1  /* Iterate over the rows of the source image */
2  for (int i = 0; i < rows; i++)
3  {
4     /* Iterate over the columns of the source image */
5     for (int j = 0; j < cols; j++)
6     {
7        /* Reset sum for new source pixel */
8        int sum = 0;
9
10       /* Apply the filter to the neighborhood */
11       for (int k = -halfFilterWidth; k <= halfFilterWidth; k++)
12       {
13          for (int l = -halfFilterWidth; l <= halfFilterWidth; l++)
14          {
15             /* Indices used to access the image */
16             int r = i+k;
17             int c = j+l;
18
```

IMAGE CONVOLUTION

```
19              /* Handle out-of-bounds locations by clamping to
20               * the border pixel */
21              r = (r < 0) ? 0 : r;
22              c = (c < 0) ? 0 : c;
23              r = (r >= rows) ? rows-1 : r;
24              c = (c >= cols) ? cols-1 : c;
25
26              sum += Image[r][c] *
27                     Filter[k+halfFilterWidth][l+halfFilterWidth];
28            }
29          }
30
31          /* Write the new pixel value */
32          outputImage[i][j] = sum;
33        }
34    }
```

IMAGE CONVOLUTION

- The OpenCL implementation of the convolution kernel is fairly straightforward, and is written similarly to the C version.
- In the OpenCL version, we create one work-item per output pixel, using parallelism to remove the two outer loops.
- The task of each work-item is to execute the two innermost loops, which perform the filtering operation.
- Reads from the source image must be performed using an OpenCL construct that is specific to the data type.

IMAGE CONVOLUTION

- The full OpenCL kernel is shown below:

```
1   __kernel
2   void convolution(
3               __read_only image2d_t inputImage,
4              __write_only image2d_t outputImage,
5                           int rows,
6                           int cols,
7                  __constant float* filter,
8                           int filterWidth,
9                       sampler_t sampler)
10  {
11      /* Store each work-item's unique row and column */
12      int column = get_global_id(0);
13      int row = get_global_id(1);
```

IMAGE CONVOLUTION

```
14
15      /* Half the width of the filter is needed for indexing
16       * memory later */
17      int halfWidth = (int)(filterWidth/2);
18
19      /* All accesses to images return data as four-element vectors
20       * (i.e., float4), although only the    x    component will contain
21       * meaningful data in this code */
22      float4 sum = {0.0f, 0.0f, 0.0f, 0.0f};
23
24      /* Iterator for the filter */
25      int filterIdx = 0;
26
27      /* Each work-item iterates around its local area on the basis of
            the
```

IMAGE CONVOLUTION

```
28          * size of the filter */
29         int2 coords; // Coordinates for accessing the image
30
31         /* Iterate the filter rows */
32         for(int i = -halfWidth; i <= halfWidth; i++)
33         {
34             coords.y = row + i;
35             /* Iterate over the filter columns */
36             for(int j = -halfWidth; j <= halfWidth; j++)
37             {
38                 coords.x = column + j;
39
40                 /* Read a pixel from the image. A single-channel image
41                  * stores the pixel in the    x    coordinate of the returned
42                  * vector. */
43                    float4 pixel;
44                 pixel = read_imagef(inputImage, sampler, coords);
45                 sum.x += pixel.x * filter[filterIdx++];
46             }
47         }
48
49         /* Copy the data to the output image */
50         coords.x = column;
51         coords.y = row;
52         write_imagef(outputImage, coords, sum);
53     }
```

IMAGE CONVOLUTION

- Accesses to an image always return a four-element vector (one per channel).
- we will declare both pixel (the value returned by the image access) and sum (resultant data that is copied to the output image) as type float4.
- We then use the x-component when accumulating the filtered pixel value

IMAGE CONVOLUTION

- The convolution filter is a perfect candidate for constant memory in this example because all work-items access the same element each iteration.
- Simply adding the keyword _ _constant in the signature of the function (Line 7) places the filter in constant memory

PrefixSum

- The Prefix Sum algorithm is used to calculate the cumulative sum of elements in an input array.
- Given an input array A of n elements, the Prefix Sum algorithm produces an output array B, where each element B[i] is the sum of all elements in A up to and including A[i].

PrefixSum

- There are two main variants of the Prefix Sum algorithm: **exclusive** and **inclusive**
- In an **exclusive** Prefix Sum, the element at index i in the output array is the sum of all elements in the input array up to, but not including, index i.
- In an **inclusive** Prefix Sum, the element at index i in the output array is the sum of all elements in the input array up to and including index i.

PrefixSum

- Consider the exclusive Prefix Sum algorithm.
- The algorithm can be efficiently parallelized using OpenCL to take advantage of the massive parallelism offered by GPUs.

PrefixSum

- Algorithm: Exclusive Prefix Sum using OpenCL

1. Input: A - Input array of size n.
2. Output: B - Output array of size n.
3. Create two OpenCL buffers to hold the input and output arrays A and B.
4. Load the OpenCL kernel code to perform the Prefix Sum calculation.
5. Set up the OpenCL context, device, and command queue.
6. Enqueue the input data from the host (CPU) to the device (GPU).
7. Set the kernel arguments (input and output buffers) for the Prefix Sum kernel.
8. Launch the OpenCL kernel to perform the Prefix Sum calculation in parallel.
9. Enqueue the result (output array B) from the device to the host.
10. Release OpenCL resources.

PrefixSum

- #include <CL/cl.hpp>
- #include <vector>

- #define N 256 // Total number of elements (should be a power of 2)
- #define WORKGROUP_SIZE 64 // Workgroup size (usually a power of 2)

- int main() {
- // Initialize input data (example)
- std::vector<int> A(N);
- // Populate A with data...

PrefixSum

- // Create OpenCL context, device, and command queue (error checking omitted for simplicity)
- cl::Context context;
- cl::Device device;
- cl::CommandQueue queue;

- // Create buffers for input and output data
- cl::Buffer bufferA(context, CL_MEM_READ_ONLY, N * sizeof(int));
- cl::Buffer bufferB(context, CL_MEM_WRITE_ONLY, N * sizeof(int));

PrefixSum

- / Load the OpenCL kernel code from file (PrefixSum.cl)

- // Read the kernel source code from a file
- std::ifstream kernelFile("PrefixSum.cl");
- std::string kernelSource(std::istreambuf_iterator<char>(kernelFile), (std::istreambuf_iterator<char>()));
- cl::Program::Sources sources(1, std::make_pair(kernelSource.c_str(), kernelSource.length()));

- // Create the program from the source code
- cl::Program program(context, sources);

- // Build the program for the selected device
- program.build({ device });

PrefixSum

- // Create the kernel
- cl::Kernel kernel(program, "prefixSum");

- // Enqueue input data to the device
- queue.enqueueWriteBuffer(bufferA, CL_TRUE, 0, N * sizeof(int), A.data());

- // Set kernel arguments
- kernel.setArg(0, bufferA);
- kernel.setArg(1, bufferB);
- kernel.setArg(2, N);

- // Define the global and local work sizes
- cl::NDRange globalSize(N);
- cl::NDRange localSize(WORKGROUP_SIZE);

PrefixSum

```
// Launch the kernel
    queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSize, localSize);


    // Enqueue the result from device to host
    std::vector<int> B(N);
    queue.enqueueReadBuffer(bufferB, CL_TRUE, 0, N * sizeof(int), B.data());


    // Output the result
    for (int i = 0; i < N; i++) {
        std::cout << "B[" << i << "] = " << B[i] << std::endl;
    }
    return 0;
}
```

PrefixSum

- __kernel void prefixSum(__global const int* A, __global int* B, const int n) {
- // Local memory for each workgroup
- __local int temp[N]; // N should be a power of 2, usually set to the workgroup size

- int gid = get_global_id(0);
- int lid = get_local_id(0);
- int lsize = get_local_size(0);

- // Load data from global memory to local memory
- temp[lid] = A[gid];

PrefixSum

```
// Perform reduction in local memory
    for (int stride = 1; stride < lsize; stride *= 2) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if (lid >= stride) {
            temp[lid] += temp[lid - stride];
        }
    }

    // Perform post-reduction in local memory
    barrier(CLK_LOCAL_MEM_FENCE);

    // Write the result to global memory
    B[gid] = (gid == 0) ? 0 : temp[lid - 1];
}
```

- Host Code (C/C++):

Matrix Multiplication

- Matrix multiplication is a computationally intensive task that can be effectively parallelized using OpenCL.
- The algorithm involves breaking down the matrices into smaller work units and distributing the computation across available processing units (e.g., GPU cores) to achieve parallelism.
- The standard matrix multiplication for matrices A (size MxK) and B (size KxN) to produce matrix C (size MxN) can be described as follows:

- for i = 0 to M

    for j = 0 to N

    C[i][j] = 0

        for k = 0 to K

        C[i][j] += A[i][k] * B[k][j].

Matrix Multiplication

1. Include necessary headers and define the matrix dimensions (M, K, N) and the corresponding memory sizes.
2. Define the matrices A, B, and C, and allocate memory for them on the host (CPU).
3. Create an OpenCL context and command queue to interact with the device (e.g., GPU).
4. Create OpenCL memory objects (buffers) to hold matrices A, B, and C on the device.
5. copy matrices A and B from host memory to device memory.
6. Load and build the OpenCL kernel from a string representation.
7. Set kernel arguments to pass the matrices A, B, and C to the kernel function.
8. Define global and local work sizes to specify how the computation is distributed among work items (threads).
9. Enqueue the kernel for execution on the device.
10. Read the result matrix C from the device back to the host.
11. Clean up OpenCL resources.

Matrix Multiplication

```
// Include necessary headers and define the matrix dimensions (M, K, N) and the corresponding memory sizes
#include <CL/cl.h>
#define M 1024
#define K 1024
#define N 1024

//Define the matrices A, B, and C, and allocate memory for them on the host (CPU)
float A[M * K], B[K * N], C[M * N];
```

Matrix Multiplication

```
//Create an OpenCL context and command queue to interact with the device (e.g., GPU).
cl_context context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
cl_command_queue queue = clCreateCommandQueue(context, device_id, 0, &err);


//Create OpenCL memory objects (buffers) to hold matrices A, B, and C on the device.
cl_mem buffer_A = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * M * K, NULL, &err);
cl_mem buffer_B = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * K * N, NULL, &err);
cl_mem buffer_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * M * N, NULL, &err);
```

Matrix Multiplication

```
//copy matrices A and B from host memory to device memory.


clEnqueueWriteBuffer(queue, buffer_A, CL_TRUE, 0, sizeof(float) * M *
                                                  K, A, 0, NULL, NULL);
clEnqueueWriteBuffer(queue, buffer_B, CL_TRUE, 0, sizeof(float) * K *
                                                  N, B, 0, NULL, NULL);
```

Matrix Multiplication

```
//Load and build the OpenCL kernel from a string representation.

const char* kernel_source =
    "__kernel void matrixMultiplication(__global float* A, __global float* B, __global float* C) {"
    "   int i = get_global_id(0);"
    "   int j = get_global_id(1);"
    "   float sum = 0.0;"
    "   for (int k = 0; k < K; ++k) {"
    "       sum += A[i * K + k] * B[k * N + j];"
    "   }"
    "   C[i * N + j] = sum;"
    "}";
cl_program program = clCreateProgramWithSource(context, 1, &kernel_source, NULL, &err);
clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "matrixMultiplication", &err);
```

Matrix Multiplication

//Set kernel arguments to pass the matrices A, B, and C to the kernel function.

clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer_A);

clSetKernelArg(kernel, 1, sizeof(cl_mem), &buffer_B);

clSetKernelArg(kernel, 2, sizeof(cl_mem), &buffer_C);

//Define global and local work sizes to specify how the computation is distributed among work items (threads)

size_t global_size[2] = {M, N};

size_t local_size[2] = {16, 16}; // For example, use a 16x16 workgroup size

Matrix Multiplication

```
//Enqueue the kernel for execution on the device.
clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global_size,
                                                local_size, 0, NULL, NULL);

//Read the result matrix C from the device back to the host.
clEnqueueReadBuffer(queue, buffer_C, CL_TRUE, 0, sizeof(float) * M *
                                                N, C, 0, NULL, NULL);
```

Matrix Multiplication

```
//Clean up OpenCL resources.
clReleaseMemObject(buffer_A);
clReleaseMemObject(buffer_B);
clReleaseMemObject(buffer_C);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

Sparse Matrix

- Implementing a sparse matrix algorithm on a GPU using OpenCL involves several steps:

1. Data Representation:

Choose an appropriate sparse matrix storage format.

Common formats include:

- Compressed Sparse Row (CSR) format
- Compressed Sparse Column (CSC) format
- Coordinate (COO) format
- ELLPACK (ELL) format
- Hybrid formats like HYB (combining CSR and ELL)

Based on the chosen format, organize your sparse matrix data into arrays that can be efficiently processed on the GPU.

Sparse Matrix

2.Kernel Implementation:

- Write an OpenCL kernel that performs the required sparse matrix operations, such as matrix-vector multiplication, matrix-matrix multiplication, and others.
- Take advantage of the data parallelism offered by the GPU to process multiple elements simultaneously.

3. Memory Management:

- Allocate memory for the sparse matrix data on the GPU using OpenCL memory allocation functions.
- Transfer data from the host (CPU) to the GPU using OpenCL memory transfer functions.

Sparse Matrix

4.Execute the Kernel:

- Launch the OpenCL kernel on the GPU, specifying the global and local work sizes based on the problem's characteristics.
- Allow the GPU to perform the computations in parallel.

5.Retrieve Results:

- Once the kernel execution is complete, transfer the results back from the GPU to the host (CPU) memory if needed.