Tiled and Naive 2-D Convolution and 1-D Histogram Calculation With Cuda and OpenCL

Drew Afromsky
Email: daa2162@columbia.edu

***Abstract*—2D convolution and 1D histogram calculation was performed in both CUDA and OpenCL. 2D convolution was implemented, taking advantage of both shared memory/tiles and global memory (naive methods). Tiled 2D convolution was performed in CUDA only. For naive 2D convolution, the input to the algorithm is an [M X N] matrix and a [K X K] kernel of odd dimension sizes. The output after convolution remained the same size as the input; zero padding was performed to take into account halo/ghost cells, when the kernel was "acting" on "non-existent" pixels/matrix elements. For the tiled 2D convolution, the kernel size was fixed: [5 x 5] to avoid dealing with dynamic memory allocation. For both methods, a serial implementation of 2D convolution was performed using scipy function (signal.convolve2D). Execution times for 2D convolution CUDA naive, 2D convolution CUDA tiled, and 2D convolution serial were recorded and plotted for comparison. Execution times for 2D convolution in OpenCL were compared to 2D convolution in serial and plotted as well. Matrices for both CUDA and OpenCL were initialized and iteratively increased in size in both dimensions by the same factor. Similarly, for 1D histogram calculation, execution times were recorded in both OpenCL and CUDA, and serial code, and then plotted and compared. One plot was created for CUDA versus serial. A second plot was created for OpenCL versus serial.**

***Index Terms*—CUDA, OpenCL, naive, tiled, convolution, histogram, halo/ghost cell**

I. Introduction

Convolution is a popular array or matrix operation performed in various forms of signal processing, digital recording, image processing, video processing, and computer vision. In these application areas, convolution is often performed as a filter that transforms signals and pixels into more desirable values. Examples include convolution for image blurring and Gaussian filters for sharpening boundaries and edges of objects in images. In high-performance computing, convolution pattern is referred to as stencil computation. Output data elements can be calculated independently of each other, which is a desirable trait for parallel computing. There also exists a substantial level of input data sharing among output data elements with boundary conditions. More specifically, convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements. The weights are defined by a convolutional kernel, also referred to as a mask array/matrix. The size of this kernel for 1D arrays is an array of length 2*n+1, and for 2D arrays [2n+1 x 2n+1], to ensure odd dimensions. Because convolution is defined in terms of neighboring elements, boundary conditions arise for output elements close to the ends of an array or matrix. For handling such boundary condition cases, zero padding is usually performed. Zero padding is essentially filling in missing cells or matrix elements, known as ghost or halo cells, with a zero value.

A histogram is a display of the frequency of data items in successive numerical intervals. The value intervals are plotted along the horizontal axis and the frequency of data items in each interval is represented as the height of a bar. Histograms provide useful summaries of data sets. Histograms of different types of object images, such as faces versus cars, as seen in computer vision tasks, tend to exhibit different shapes. By dividing an image into subareas and analyzing the histograms for these subareas, one can quickly identify interesting subareas of an image that may contain the objects of interest. This is the basis of feature extraction.

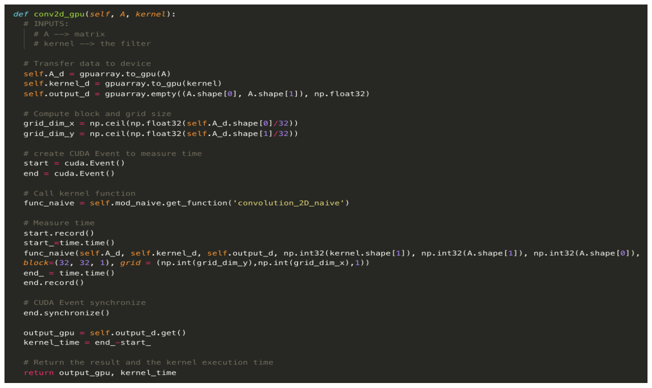
A CUDA device contains several types of memory that can help programmers improve compute-to-global-memory-access-ratio, defined as the number of floating-point calculations performed for each access to the global memory within a region of a program. Global memory can be written and read by the device. Shared memory is on-chip memory. Variables that exist in this type of memory can be accessed at very high-speed in a highly parallel manner. Shared memory locations are allocated to thread blocks; all threads in a block can access shared memory variables allocated to that block. Shared memory is an efficient way for threads to share their input data and intermediate results. The global memory is large, but slow, whereas shared memory is small, but fast. This causes a trade-off in the use of device memories. A common strategy is to partition the data into subsets called tiles, so that each tile fits into the shared memory. We also introduce constant memory, where variables in this type of memory are also visible to all thread blocks. In contrast to global memory, constant memory cannot be changed by threads during kernel execution. Kernel functions also access constant memory variables as global variables, so their pointers don't need to be passed to the kernel as parameters. Constant memory variables are located in DRAM like global memory variables. CUDA-enabled hardware caches the constant memory variables during kernel execution. As a result, when all threads in a warp access the same constant memory variable, as in the case with convolutional kernels, the caches can provide tremendous amount of bandwidth to satisfy the data needs of threads.

II. Methods

A. *Naive 2D Convolution PyCuda and PyOpenCL*

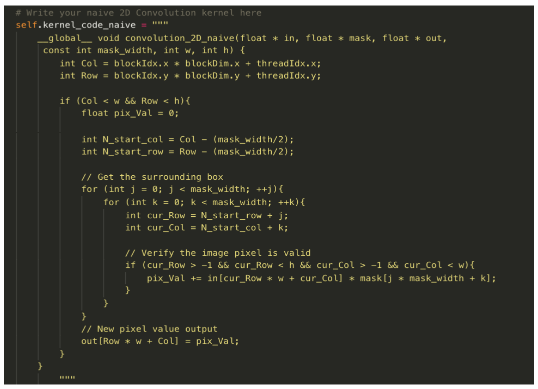
Unlike an optimized/tiled method for 2D convolution, a naive method does not take advantage of shared memory and tiling. The process for computing convolution, of two matrices (input matrix and convolutional kernel), using a naive approach can be broken down into the following steps:

(1.) initialize the input matrix of arbitrary size following the format input_matrix = [M X N] and convolutional_kernel = [K X K], where K=2*n+1 and n is an integer. After convolution, the output matrix retains the same shape as the input matrix. (2.) create the kernel code for 2D convolution utilizing simple indexing for each thread and computing the weighted sum of pixels/matrix elements of the input array with those of the convolutional kernel; global/device memory is used here only (3.) move input and expected output matrices to device memory, (4.) compute block and grid size (5.) call kernel function to perform 2D naive convolution, (6.) measure run-time and return the output - a matrix resulting from the convolution of input matrix and convolutional kernel. These steps were performed for matrices of various sizes created by iteratively increasing the dimension of the initialized matrix in both dimensions. For naive 2D convolution, kernel size was also iteratively changing in size. Convolutional kernel size was kept constant for 2D tiled convolution.



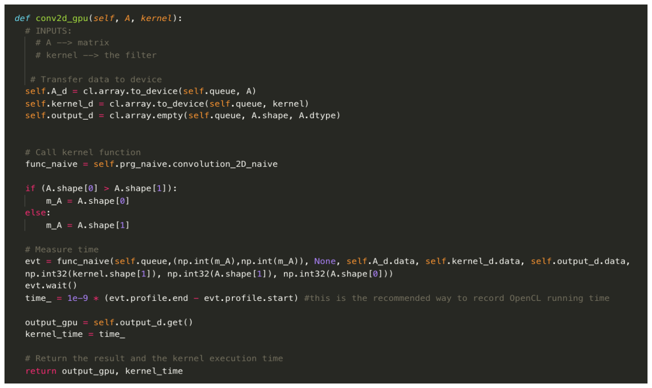
```python
def conv2d_gpu(self, A, kernel):
  # INPUTS:
    # A --> matrix
    # kernel --> the filter

  # Transfer data to device
  self.A_d = gpuarray.to_gpu(A)
  self.kernel_d = gpuarray.to_gpu(kernel)
  self.output_d = gpuarray.empty((A.shape[0], A.shape[1]), np.float32)

  # Compute block and grid size
  grid_dim_x = np.ceil(np.float32(self.A_d.shape[0]/32))
  grid_dim_y = np.ceil(np.float32(self.A_d.shape[1]/32))

  # create CUDA Event to measure time
  start = cuda.Event()
  end = cuda.Event()

  # Call kernel function
  func_naive = self.mod_naive.get_function('convolution_2D_naive')

  # Measure time
  start.record()
  start_=time.time()
  func_naive(self.A_d, self.kernel_d, self.output_d, np.int32(kernel.shape[1]), np.int32(A.shape[1]), np.int32(A.shape[0]),
  block=(32, 32, 1), grid = (np.int(grid_dim_y),np.int(grid_dim_x),1))
  end_ = time.time()
  end.record()

  # CUDA Event synchronize
  end.synchronize()

  output_gpu = self.output_d.get()
  kernel_time = end_-start_

  # Return the result and the kernel execution time
  return output_gpu, kernel_time
```

Fig. 1. PyCuda Naive 2D Convolution Key Steps



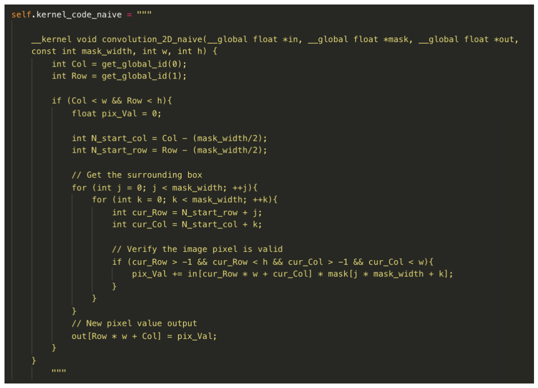
```python
# Write your naive 2D Convolution kernel here
self.kernel_code_naive = """
    __global__ void convolution_2D_naive(float * in, float * mask, float * out,
     const int mask_width, int w, int h) {
        int Col = blockIdx.x * blockDim.x + threadIdx.x;
        int Row = blockIdx.y * blockDim.y + threadIdx.y;

        if (Col < w && Row < h){
            float pix_Val = 0;

            int N_start_col = Col - (mask_width/2);
            int N_start_row = Row - (mask_width/2);

            // Get the surrounding box
            for (int j = 0; j < mask_width; ++j){
                for (int k = 0; k < mask_width; ++k){
                    int cur_Row = N_start_row + j;
                    int cur_Col = N_start_col + k;

                    // Verify the image pixel is valid
                    if (cur_Row > -1 && cur_Row < h && cur_Col > -1 && cur_Col < w){
                        pix_Val += in[cur_Row * w + cur_Col] * mask[j * mask_width + k];
                    }
                }
            }
            // New pixel value output
            out[Row * w + Col] = pix_Val;
        }
    }
        """
```

Fig. 2. PyCuda Naive 2D Convolution Kernel Code



```python
def conv2d_gpu(self, A, kernel):
  # INPUTS:
    # A --> matrix
    # kernel --> the filter

   # Transfer data to device
  self.A_d = cl.array.to_device(self.queue, A)
  self.kernel_d = cl.array.to_device(self.queue, kernel)
  self.output_d = cl.array.empty(self.queue, A.shape, A.dtype)


  # Call kernel function
  func_naive = self.prg_naive.convolution_2D_naive

  if (A.shape[0] > A.shape[1]):
      m_A = A.shape[0]
  else:
      m_A = A.shape[1]

  # Measure time
  evt = func_naive(self.queue,(np.int(m_A),np.int(m_A)), None, self.A_d.data, self.kernel_d.data, self.output_d.data,
  np.int32(kernel.shape[1]), np.int32(A.shape[1]), np.int32(A.shape[0]))
  evt.wait()
  time_ = 1e-9 * (evt.profile.end - evt.profile.start) #this is the recommended way to record OpenCL running time

  output_gpu = self.output_d.get()
  kernel_time = time_

  # Return the result and the kernel execution time
  return output_gpu, kernel_time
```

Fig. 3. PyOpenCL Naive 2D Convolution Key Steps



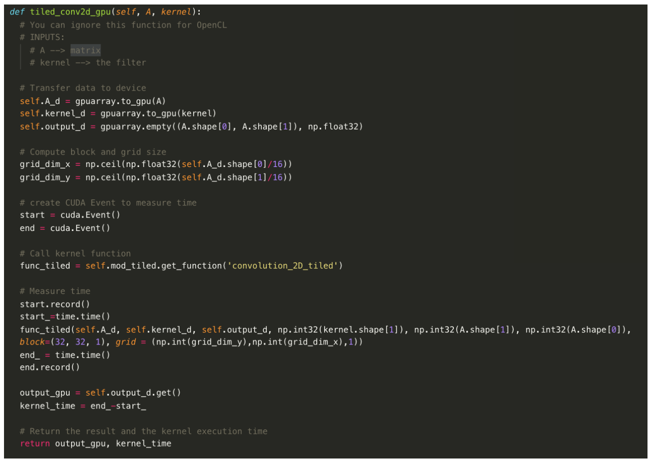
```python
self.kernel_code_naive = """

    __kernel void convolution_2D_naive(__global float *in, __global float *mask, __global float *out,
    const int mask_width, int w, int h) {
        int Col = get_global_id(0);
        int Row = get_global_id(1);

        if (Col < w && Row < h){
            float pix_Val = 0;

            int N_start_col = Col - (mask_width/2);
            int N_start_row = Row - (mask_width/2);

            // Get the surrounding box
            for (int j = 0; j < mask_width; ++j){
                for (int k = 0; k < mask_width; ++k){
                    int cur_Row = N_start_row + j;
                    int cur_Col = N_start_col + k;

                    // Verify the image pixel is valid
                    if (cur_Row > -1 && cur_Row < h && cur_Col > -1 && cur_Col < w){
                        pix_Val += in[cur_Row * w + cur_Col] * mask[j * mask_width + k];
                    }
                }
            }
            // New pixel value output
            out[Row * w + Col] = pix_Val;
        }
    }
        """
```

Fig. 4. PyOpenCL Naive 2D Convolution Kernel Code

B. *Optimized/Tiled 2D Convolution PyCuda*

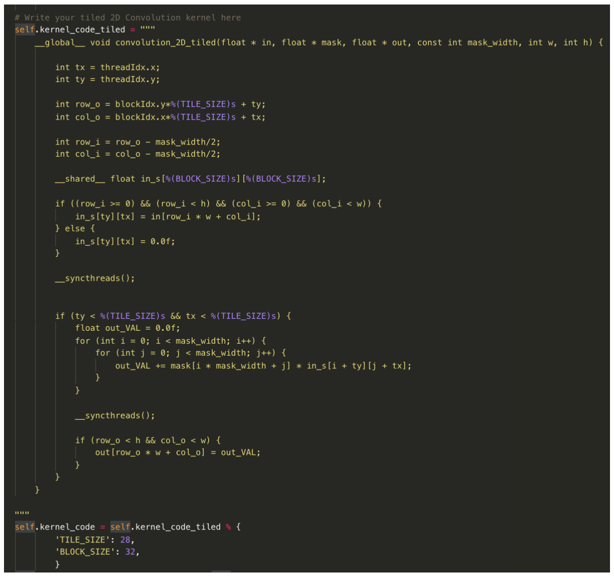
The optimized/tiled 2D convolution was conducted in a similar manner to the naive method, however, the kernel code incorporates tiles and shared memory. The benefit for using shared memory and tiling is that through tiling we are able to partition the data that will be processed by threads and with utilizing shared memory (on-chip memory) we can access these variables quickly. With each tile fitting into the shared memory, all the threads in that tile access these memory variables and we avoid global memory accesses (larger, but slower access). Therefore, steps 1-6 above were also followed for the tiled 2D convolution method. The kernel code differs, however, in that shared memory is declared and shared memory locations are allocated to tiles of thread blocks.

The kernel code follows the following structure: (1.) Each thread of the kernel first calculates the y and x indices of its output element - col_o and row_o variables of the kernel, (2.) each thread then calculates the y and x indices of the input element it is to load into the shared memory by subtracting mask_width/2 from row_o and col_o and assigning the results to row_i and col_i, (3.) declare shared memory variables, where all threads in a block have access to the elements loaded into these memory variables (one pair for each block); loading input tiles into shared memory, (4.) check if the y and x indices of the input tiles are within range of the input; if not, the input element it is attempting to load is a ghost element and a 0.0 value should be placed into shared memory, (5.) compute the output value using input elements in shared memory, (6.) ensure all threads have finished loading the tiles of input matrices into shared memory matrix variables before any can move forward (using __syncthreads), (7.) ensure that only the threads whose indices are smaller than tile_size should participate in the calculation of output pixels, (8.) ensure all threads have finished using the input matrix elements in the shared memory before any move on to the next iteration and load elements from next set of tiles, avoiding loading elements too early and corrupting input values for other threads, (9.) all threads whose output elements are in the valid range write their result values into their respective output elements



```python
def tiled_conv2d_gpu(self, A, kernel):
    # You can ignore this function for OpenCL
    # INPUTS:
        # A --> matrix
        # kernel --> the filter

    # Transfer data to device
    self.A_d = gpuarray.to_gpu(A)
    self.kernel_d = gpuarray.to_gpu(kernel)
    self.output_d = gpuarray.empty((A.shape[0], A.shape[1]), np.float32)

    # Compute block and grid size
    grid_dim_x = np.ceil(np.float32(self.A_d.shape[0]/16))
    grid_dim_y = np.ceil(np.float32(self.A_d.shape[1]/16))

    # create CUDA Event to measure time
    start = cuda.Event()
    end = cuda.Event()

    # Call kernel function
    func_tiled = self.mod_tiled.get_function('convolution_2D_tiled')

    # Measure time
    start.record()
    start_=time.time()
    func_tiled(self.A_d, self.kernel_d, self.output_d, np.int32(kernel.shape[1]), np.int32(A.shape[1]), np.int32(A.shape[0]),
    block=(32, 32, 1), grid = (np.int(grid_dim_y),np.int(grid_dim_x),1))
    end_ = time.time()
    end.record()

    output_gpu = self.output_d.get()
    kernel_time = end_-start_

    # Return the result and the kernel execution time
    return output_gpu, kernel_time
```

Fig. 5. PyCuda Tiled 2D Convolution Key Steps



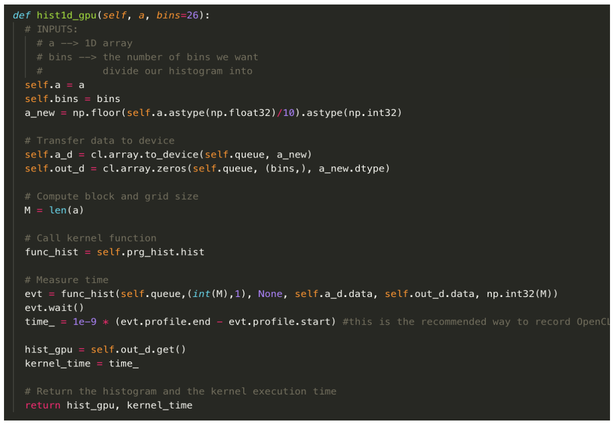
```python
# Write your tiled 2D Convolution kernel here
self.kernel_code_tiled = """
    __global__ void convolution_2D_tiled(float * in, float * mask, float * out, const int mask_width, int w, int h) {

        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int row_o = blockIdx.y*%(TILE_SIZE)s + ty;
        int col_o = blockIdx.x*%(TILE_SIZE)s + tx;

        int row_i = row_o - mask_width/2;
        int col_i = col_o - mask_width/2;

        __shared__ float in_s[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];

        if ((row_i >= 0) && (row_i < h) && (col_i >= 0) && (col_i < w)) {
            in_s[ty][tx] = in[row_i * w + col_i];
        } else {
            in_s[ty][tx] = 0.0f;
        }

        __syncthreads();


        if (ty < %(TILE_SIZE)s && tx < %(TILE_SIZE)s) {
            float out_VAL = 0.0f;
            for (int i = 0; i < mask_width; i++) {
                for (int j = 0; j < mask_width; j++) {
                    out_VAL += mask[i * mask_width + j] * in_s[i + ty][j + tx];
                }
            }

            __syncthreads();

            if (row_o < h && col_o < w) {
                out[row_o * w + col_o] = out_VAL;
            }
        }
    }

"""
self.kernel_code = self.kernel_code_tiled % {
        'TILE_SIZE': 28,
        'BLOCK_SIZE': 32,
        }
```

Fig. 6. PyCuda Tiled 2D Convolution Kernel Code

C. *1D Histogram Calculation PyCuda and PyOpenCL*

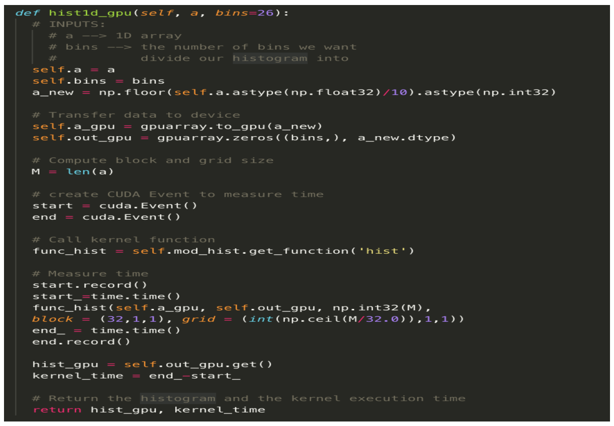
1D histogram calculation can be broken down into the following steps:

(1.) initialize the input array of length N and containing randomly sampled integers between 0 and 255. (2.) create the kernel code for 1D histogram calculation (3.) move input and expected output matrices to device memory, (4.) compute block and grid size (5.) call kernel function to perform 1D histogram calculation, (6.) measure run-time and return the output - a histogram of the input array. These steps were performed for arrays of various sizes created by iteratively increasing the dimension of the initialized array. Bin size of 26 was kept constant.



```python
def hist1d_gpu(self, a, bins=26):
    # INPUTS:
        # a --> 1D array
        # bins --> the number of bins we want
        #          divide our histogram into
    self.a = a
    self.bins = bins
    a_new = np.floor(self.a.astype(np.float32)/10).astype(np.int32)

    # Transfer data to device
    self.a_d = cl.array.to_device(self.queue, a_new)
    self.out_d = cl.array.zeros(self.queue, (bins,), a_new.dtype)

    # Compute block and grid size
    M = len(a)

    # Call kernel function
    func_hist = self.prg_hist.hist

    # Measure time
    evt = func_hist(self.queue,(int(M),1), None, self.a_d.data, self.out_d.data, np.int32(M))
    evt.wait()
    time_ = 1e-9 * (evt.profile.end - evt.profile.start) #this is the recommended way to record OpenCL

    hist_gpu = self.out_d.get()
    kernel_time = time_

    # Return the histogram and the kernel execution time
    return hist_gpu, kernel_time
```

Fig. 7. PyOpenCL 1D Histogram Key Steps



```python
def hist1d_gpu(self, a, bins=26):
    # INPUTS:
        # a --> 1D array
        # bins --> the number of bins we want
        #          divide our histogram into
    self.a = a
    self.bins = bins
    a_new = np.floor(self.a.astype(np.float32)/10).astype(np.int32)

    # Transfer data to device
    self.a_gpu = gpuarray.to_gpu(a_new)
    self.out_gpu = gpuarray.zeros((bins,), a_new.dtype)

    # Compute block and grid size
    M = len(a)

    # create CUDA Event to measure time
    start = cuda.Event()
    end = cuda.Event()

    # Call kernel function
    func_hist = self.mod_hist.get_function('hist')

    # Measure time
    start.record()
    start_=time.time()
    func_hist(self.a_gpu, self.out_gpu, np.int32(M),
    block = (32,1,1), grid = (int(np.ceil(M/32.0)),1,1))
    end_ = time.time()
    end.record()

    hist_gpu = self.out_gpu.get()
    kernel_time = end_-start_

    # Return the histogram and the kernel execution time
    return hist_gpu, kernel_time
```

Fig. 8. PyOpenCL 1D Histogram Key Steps

III. Results

A. *Run-Time Comparisons, Code Equality, Profiling (PyCuda 2D Convolution)*

The size of the input matrices in each dimension were iteratively increased to produce a new matrix during 2D convolution (both naive and tiled). For each iteration, the CPU, PyCuda kernel, and PyOpenCl kernel execution times for 2D convolution were recorded. PyCuda tiled and naive

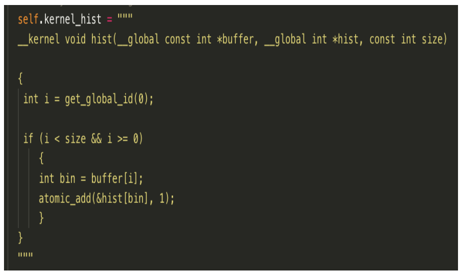

```
self.kernel_hist = """
__kernel void hist(__global const int *buffer, __global int *hist, const int size)

{
 int i = get_global_id(0);

 if (i < size && i >= 0)
    {
    int bin = buffer[i];
    atomic_add(&hist[bin], 1);
    }
}
"""
```

Fig. 9. PyOpenCL 1D Histogram Kernel Code

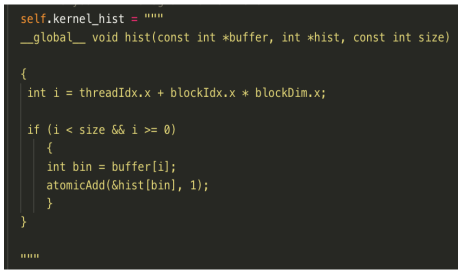

```
self.kernel_hist = """
__global__ void hist(const int *buffer, int *hist, const int size)

{
 int i = threadIdx.x + blockIdx.x * blockDim.x;

 if (i < size && i >= 0)
    {
    int bin = buffer[i];
    atomicAdd(&hist[bin], 1);
    }
}

"""
```

Fig. 10. PyCuda 1D Histogram Kernel Code

2D convoution kernel execution times were compared to CPU/serial code execution times. PyOpenCL and 2D convolution kernel execution times were compared to CPU/serial code execution times. For consistency between PyOpenCL and PyCuda, the figures below were produced using the same size input matrices beginning with dimension [20 x 25] iterating each dimension by a factor 1 to 39. The horizontal axis is the number of elements in the newly produced matrix during each iteration. The vertical axes are the run times in seconds. From the plots, we see that both the CUDA naive and tiled methods run faster than OpenCL, and naive and tiled methods for both OpenCL and CUDA run much faster than serial/CPU code. Similarly, for 1D histogram, the array was iteratively increased.

Figure 15 is a screenshot of the profiling for PyCuda. The Visual Profiler is a graphical profiling tool that displays a timeline of CPU and GPU activity, and that includes an automated analysis engine to identify optimization opportunities.

IV. Discussion

2D convolution and 1D histogram calculation was successfully implemented using three different methods: 1. CPU/Serial, 2. PyCuda (naive + tiled), and 3. PyOpenCL (naive). Tiled 2D convolution was only performed in PyCuda. From the results, it is clear that CPU histogram operation and 2D convolution are much slower compared to PyCuda and PyOpenCL. The naive PyCuda method execution time was slower than the tiled/optimized PyCuda execution time. This is to be expected, since reducing global memory accesses and utilizing tiled matrix multiplication with shared memory accesses should speed up kernel execution time. The execution time for PyOpenCL naive was much faster than CPU and slower than both the naive and tiled/optimized versions in PyCuda. Similarly for 1D histogram calculation, CUDA appeared to outperform OpenCL and both outperformed serial/CPU implementation.

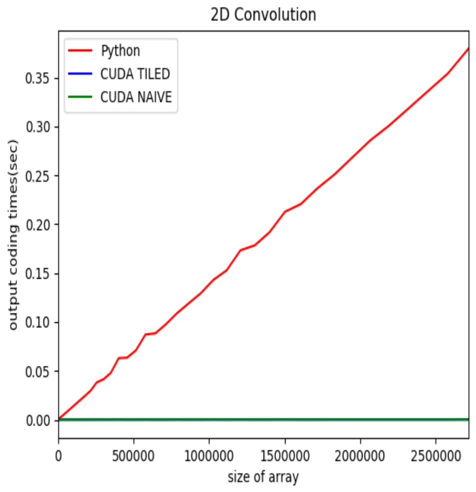


Fig. 11. PyCuda Execution Times for CPU, Naive, and Tiled 2D Convolution

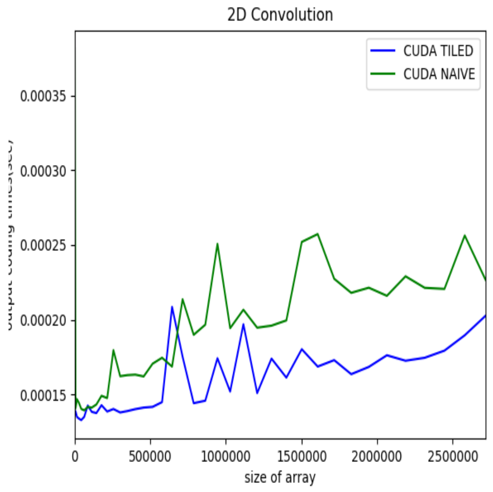


Fig. 12. Closer Look at PyCuda Execution Times for Naive and Tiled 2D Convolution ONLY

References

[1] David B. Kirk and Wen-mei W. Hwu. 2016. Programming Massively Parallel Processors, Third Edition: A Hands-On Approach (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

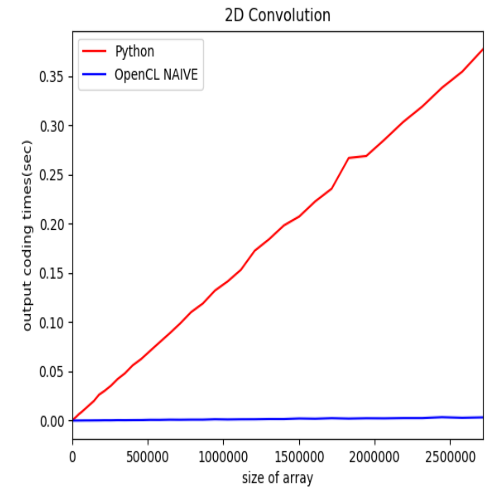


Fig. 13. PyOpenCL Execution Times for CPU and Naive 2D Convolution

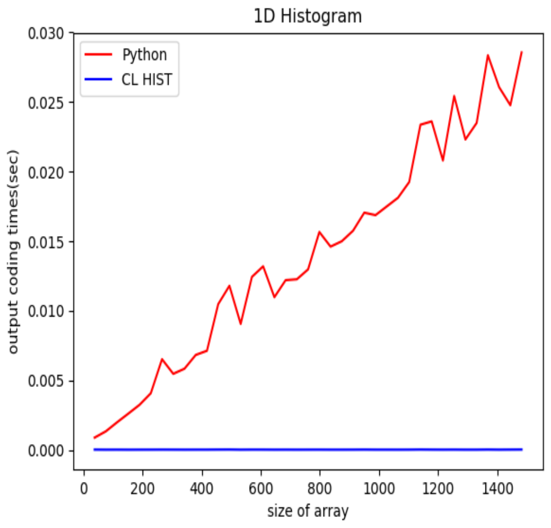


Fig. 16. PyOpenCL 1D Histogram Execution Time

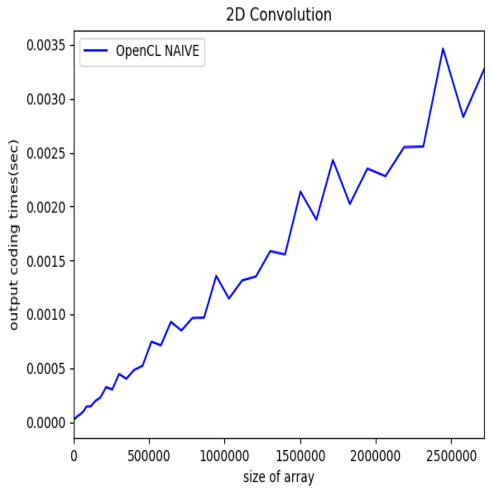


Fig. 14. Closer look at PyOpenCL Execution Time for Naive 2D Convolution

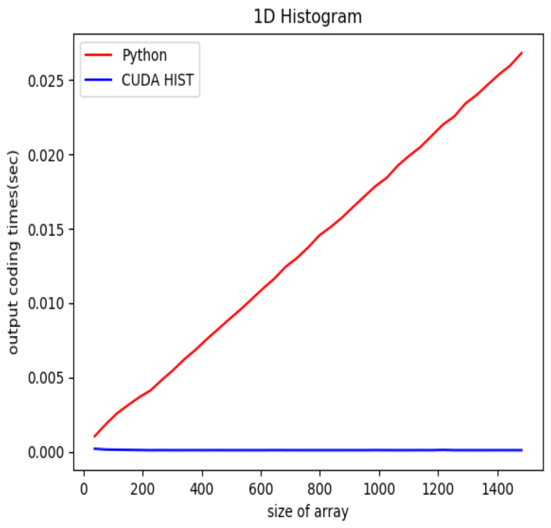


Fig. 17. PyCuda 1D Histogram Execution Time

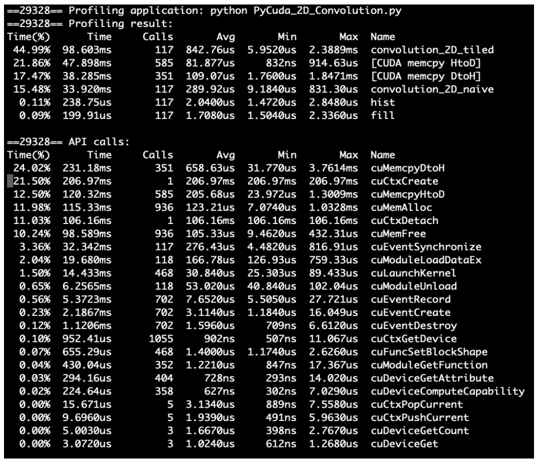

```
==29328== Profiling application: python PyCuda_2D_Convolution.py
==29328== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.99%  98.603ms       117  842.76us  5.9520us  2.3889ms  convolution_2D_tiled
 21.86%  47.898ms       585  81.877us     832ns  914.63us  [CUDA memcpy HtoD]
 17.47%  38.285ms       351  109.07us  1.7600us  1.8471ms  [CUDA memcpy DtoH]
 15.48%  33.920ms       117  289.92us  9.1840us  831.30us  convolution_2D_naive
  0.11%  238.75us       117  2.0400us  1.4720us  2.8480us  hist
  0.09%  199.91us       117  1.7080us  1.5040us  2.3360us  fill

==29328== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 24.02%  231.18ms       351  658.63us  31.770us  3.7614ms  cuMemcpyDtoH
 21.50%  206.97ms         1  206.97ms  206.97ms  206.97ms  cuCtxCreate
 12.50%  120.32ms       585  205.68us  23.972us  1.3009ms  cuMemcpyHtoD
 11.98%  115.33ms       936  123.21us  7.0740us  1.0328ms  cuMemAlloc
 11.03%  106.16ms         1  106.16ms  106.16ms  106.16ms  cuCtxDetach
 10.24%  98.589ms       936  105.33us  9.4620us  432.31us  cuMemFree
  3.36%  32.342ms       117  276.43us  4.4820us  816.91us  cuEventSynchronize
  2.04%  19.680ms       118  166.78us  126.93us  759.33us  cuModuleLoadDataEx
  1.50%  14.433ms       468  30.840us  25.303us  89.433us  cuLaunchKernel
  0.65%  6.2565ms       118  53.020us  40.840us  102.04us  cuModuleUnload
  0.56%  5.3723ms       702  7.6520us  5.5050us  27.721us  cuEventRecord
  0.23%  2.1867ms       702  3.1140us  1.1840us  16.049us  cuEventCreate
  0.12%  1.1206ms       702  1.5960us     709ns  6.6120us  cuEventDestroy
  0.10%  952.41us      1055     902ns     507ns  11.067us  cuCtxGetDevice
  0.07%  655.29us       468  1.4000us  1.1740us  2.6260us  cuFuncSetBlockShape
  0.04%  430.04us       352  1.2210us     847ns  17.367us  cuModuleGetFunction
  0.03%  294.16us       404     728ns     293ns  14.020us  cuDeviceGetAttribute
  0.02%  224.64us       358     627ns     302ns  7.0290us  cuDeviceComputeCapability
  0.00%  15.671us         5  3.1340us     889ns  7.5580us  cuCtxPopCurrent
  0.00%  9.6960us         5  1.9390us     491ns  5.9630us  cuCtxPushCurrent
  0.00%  5.0030us         3  1.6670us     398ns  2.7670us  cuDeviceGetCount
  0.00%  3.0720us         3  1.0240us     612ns  1.2680us  cuDeviceGet
```

Fig. 15. Profiling for Tiled and Naive PyCuda 2D Convolution and 1D Histogram