# ECSE 420
# Lab 2: CUDA Convolution and Matrix Inversion

Alex Hale
260672475
Thomas Hillyer
260680811

November 4th, 2019

# Convolution

Convolution is an operation often used in image processing to apply a filter across an image. For example, a box filter could be used to blur an image, or an edge detector filter could be used to detect vertical or horizontal edges in the image.

Instead of creating this solution in C, it was written in Python within a Google Colab notebook. This allowed for remote work instead of working exclusively within the lab (since Colab offers free use of GPUs), as well as easing the data transfer between host and device. Python's Numba package compiles Python code into an efficient format, using the same CUDA kernel function format found in C. The square bracket syntax is the same as the <<<>>> syntax in C CUDA.

```
convolution[numBlocks, threadsPerBlock](originalImg, resultImg, convDim , w)
```

The convolution kernel involves two distinct steps. First, the mapping of each thread (on each block) must be mapped to an individual pixel. The thread's unique ID is generated in the first line of the snippet below. Then, if the unique ID is larger than the height of the image, the index "wraps around" to increase the horizontal value of the pixel location.

```
@cuda.jit
def convolution(originalImg, resultImg, convDim, weightMatrix):
  x = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
  y = x // len(resultImg[0])
  x %= len(resultImg)
```

Next, the convolution operation must be performed around the identified pixel. Following the formula in the lab description, iteration is performed between 0 and the size of the weight matrix in two dimensions to access the appropriate dimension of square around the target pixel. At each iteration, the value of each of the three colours (R, G, B) is multiplied by the appropriate value in the weight matrix and added to the sum of the appropriate colour at the target pixel. After the sum is complete, the value is clamped if it exceeds [0, 255].

```
for convX in range(convDim):
  for convY in range(convDim):
    for z in [0, 1, 2]:
      resultImg[x, y, z] += originalImg[x + convX - 1][y + convY - 1][z] * weightMatrix[convX][convY]
```

The convolution kernel results in a properly convolved image for all three sizes of weight matrices, matching the sample result images provided. The images are displayed below, as well as in the attached iPython notebook and in the attached individual files.

The MSE is higher than expected given how similar the images look visually. This is likely due to some sort of import error on the expected_result images.

| Weight Matrix Size | Mean Square Error |
|---|---|
| 7x7 | 1876 |
| 5x5 | 1564 |
| 3x3 | 680 |

3x3 Convolution Matrix

5x5 Convolution Matrix

7x7 Convolution Matrix

# Matrix Inversion and Linear System Solution

The inverse of matrix A can be found by finding a matrix *X* such that *AX* = *I*, where I is the identity matrix of the same dimensions as matrix *A*. The task given in this lab is a more general version of the problem, where given matrix A and vector b, vector X must be found such that AX=b. After finding the matrix X, the result can be confirmed by performing the multiplication AX and confirming that the result is equivalent to b.

Since the project specifications were relaxed to remove the parallel linear system solution requirement, the 10x10 X matrix is found using the linalg.solve function built into NumPy. With that covered, the only remaining requirement was a parallel matrix multiplication algorithm.

```
@cuda.jit
# 2D matrix A multiplied by 1D vector X to produce 1D result
def multiply(A, X, result, length):
  x = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if x < length:
    for i in range(length):
      result[x] += A[x][i] * X[i]
```

**Matrix Multiplication Kernel Function**

The matrix multiplication kernel takes as input a 2D matrix A, a 1D vector X, a 1D placeholder vector in which the result is to be placed, and the length of the input. The height of the matrix A must match the length of the vector X. Note that this kernel was made specifically for the multiplication of a matrix by a vector, and was intentionally not generalized for multiplying a matrix by another matrix.

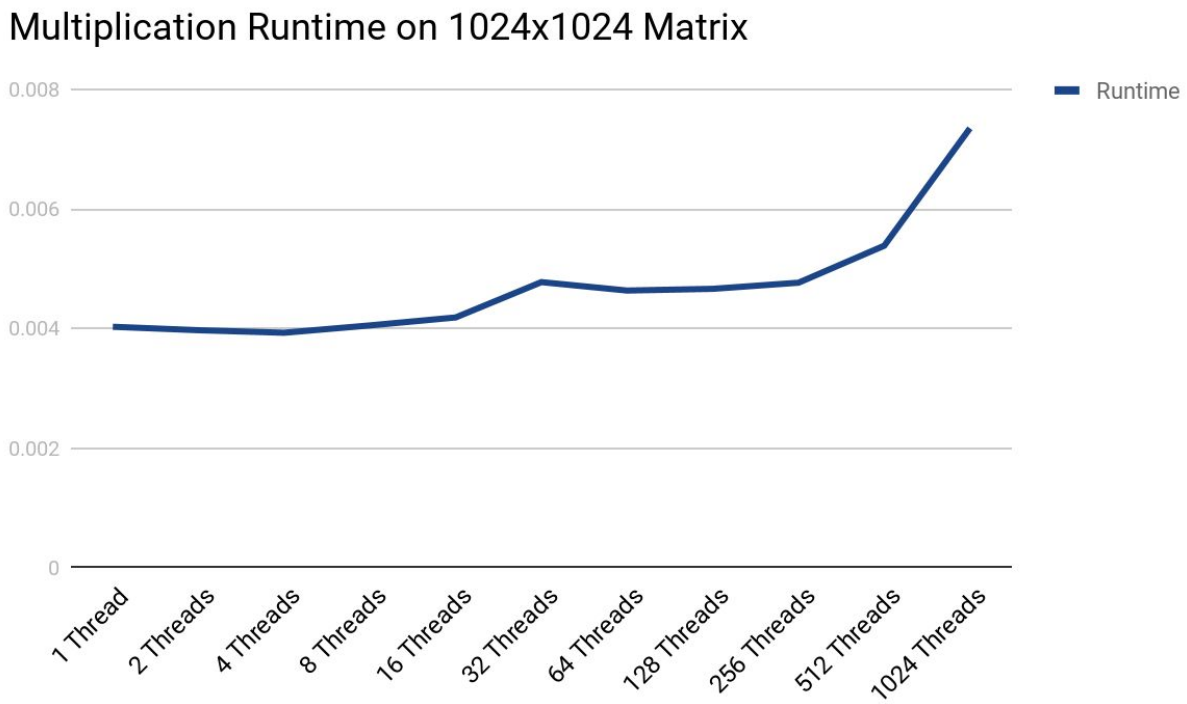
First, the unique index for this block and thread are calculated and stored in variable *x*. If the calculated *x* is within the height of the matrix A (and equivalently, within the length of the vector X), an iteration is performed simultaneously over the selected row of A and over each cell of X. At each cell, the value at A[row][i] is multiplied by X[i] and summed to the running total, producing the output at result[row]. The ability to perform this operation in one cell of the output at a time is what makes the matrix multiplication operation highly parallelizable.

Performance Measurements

To measure performance, the multiplication operation was performed for the A matrix of size 1024x1024, with the number of threads per block varying between 1 and 1024. The number of blocks used was 10. The multiplication at each number of threads was repeated 1000 times to gain an accurate average runtime of the operation.

As observed in the plot of the performance below, near-constant performance was observed for values of threads per block between 1 and 16. Once the number of threads per block increases beyond this range, the performance begins to degrade. This is because the overhead of switching between threads begins to outweigh the performance benefit of parallelizing the matrix multiplication operation.

With 10 blocks, the optimal number of threads per block would be 103, because it would be close to exactly one thread operating on each cell of the matrix. However, this result is not quite observed, because the performance at lower than 103 threads is better than the performance at 128 threads. The exponential increase in runtime toward 1024 threads per block is an expected result, because there are far more threads than the number of cells required.



# Appendix

*The code is pasted below. Additionally, the code and relevant input and output files are attached alongside this report.*

```
# ECSE 420 - Lab 2
Parallelized convolution, matrix inversion, and matrix multiplication
algorithms.

!apt-get install nvidia-cuda-toolkit
!pip3 install numba

import os
os.environ['NUMBAPRO_LIBDEVICE'] = "/usr/lib/nvidia-cuda-toolkit/libdevice"
os.environ['NUMBAPRO_NVVM'] = "/usr/lib/x86_64-linux-gnu/libnvvm.so"

from numba import cuda
import numpy as np
import cv2
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/drive/')

@cuda.jit
def convolution(originalImg, resultImg, convDim, weightMatrix):
  x = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
  y = x // len(resultImg[0])
  x %= len(resultImg)

  # perform convolution
  for convX in range(convDim):
    for convY in range(convDim):
      for z in [0, 1, 2]:
        resultImg[x, y, z] += originalImg[x + convX - 1][y + convY - 1][z]
* weightMatrix[convX][convY]

  # clamp result to [0, 255]
  for z in [0, 1, 2]:
    if resultImg[x, y, z] < 0:
```

```
      resultImg[x, y, z] = 0
    elif resultImg[x, y, z] > 255:
      resultImg[x, y, z] = 255

# parameter selection
numBlocks = 978
threadsPerBlock = 1000
convDim = 5

inputFile = '/content/drive/My Drive/Colab Notebooks/ECSE420/test.png'
outputFile = '/content/drive/My Drive/Colab Notebooks/ECSE420/conv' +
str(convDim) + 'x' + str(convDim) + '.png'
expectedFile = '/content/drive/My Drive/Colab
Notebooks/ECSE420/test_convolve_' + str(convDim) + 'x' + str(convDim) +
'.png'

# variable setup
if convDim == 3:
  w = np.asarray([[1, 2, -1],
                  [2, 0.25, -2],
                  [1, -2, -1]])
elif convDim == 5:
  w = np.asarray([[0.5, 0.75, 1, -0.75, -0.5],
                  [0.75, 1, 2, -1, -0.75],
                  [1, 2, 0.25, -2, -1],
                  [0.75, 1, -2, -1, -0.75],
                  [0.5, 0.75, -1, -0.75, -0.5]])
elif convDim == 7:
  w = np.asarray([[0.25, 0.3, 0.5, 0.75, -0.5, -0.3, -0.25],
                  [0.3, 0.5, 0.75, 1, -0.75, -0.5, -0.3],
                  [0.5, 0.75, 1, 2, -1, -0.75, -0.5],
                  [0.75, 1, 2, 0.25, -2, -1, -0.75],
                  [0.5, 0.75, 1, -2, -1, -0.75, -0.5],
                  [0.3, 0.5, 0.75, -1, -0.75, -0.5, -0.3],
                  [0.25, 0.3, 0.5, -0.75, -0.5, -0.3, -0.25]])
originalImg = np.float64(cv2.cvtColor(cv2.imread(inputFile),
cv2.COLOR_BGR2RGB))
resultImg = np.float64(np.zeros((originalImg.shape[0] - (convDim - 1),
                                 originalImg.shape[1] - (convDim - 1),
                                 originalImg.shape[2])))

# function call (numba performs host -> device -> host memory transfers)
```

```
convolution[numBlocks, threadsPerBlock](originalImg, resultImg, convDim ,
w)

# save results
cv2.imwrite(outputFile, resultImg)

# display images
plt.figure(figsize=(15,15))
plt.subplot(121), plt.imshow(np.uint8(originalImg))
plt.subplot(122), plt.imshow(np.uint8(resultImg))
plt.show()

# calculate error between result and expected result
expectedResult = cv2.cvtColor(cv2.imread(expectedFile), cv2.COLOR_BGR2RGB)
print("MSE: {}".format(((resultImg - expectedResult)**2).mean(axis=None)))

import time

@cuda.jit
# 2D matrix A multiplied by 1D vector X to produce 1D result
def multiply(A, X, result, length):
  x = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if x < length:
    for i in range(length):
      result[x] += A[x][i] * X[i]

# filepath
filepath = "/content/drive/My Drive/Colab Notebooks/ECSE420/npArrays/"

# parameter selection
numBlocks = 1
threadsPerBlock = 1000    # maximum 1024
convDim = 3
size = 1024

for threadsPerBlock in [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]:
  executionTime = 0
  for i in range(1000):  # execute 100 times to get an average
    # variable setup
    A = np.load(filepath + "A" + str(size) + ".npy")
    b = np.load(filepath + "b" + str(size) + ".npy")
```

```
    if size == 10:
      X = np.linalg.solve(A, b)
    else:
      X = np.load(filepath + "X" + str(size) + ".npy")

    result = np.zeros((X.shape))

    # function call (numba performs host -> device -> host memory
transfers)
    start = time.time()
    multiply[numBlocks, threadsPerBlock](A, X, result, len(result))
    executionTime += time.time() - start

    if np.subtract(result, b).all() != 0:
      # only print if there was a mistake
      print("Multiplication at size {} was NOT successful.".format(size))
  print("Average execution time is {} with {}
threads.".format(executionTime / 1000, threadsPerBlock))
```