Accelerating Data Augmentation for Deep Learning with Cuda and OpenCL

Drew Afromsky
Email: daa2162@columbia.edu

***Abstract*—Two different data augmentation techniques were explored in this project. Specifically, image blurring and image transpose were implemented in Cuda and OpenCL. The data augmentation techniques were conducted on two different datasets. Dataset 1 is comprised 100 pathology images, each of size 3100 x 3100. This was considered a small dataset, but with large matrices. Dataset 2 is a large dataset consisting of 50,000 images, each of size 32 x 32.**

Image blurring utilized both shared memory (optimized, with tiles) and global memory (naive methods). The optimized, tiled image blurring technique was performed in Cuda only. For image blurring, an average image blur was executed, with two different kernel sizes: 3 x 3 for dataset 1 and 11 x 11 for dataset 2 (this also avoids dealing with dynamic memory allocation). For the naive blur kernel, the input to the algorithm is a matrix of size 3100 x 3100 (an image from dataset 1) or a matrix of size 32 x 32 (an image from dataset 2). The output after blurring 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. A serial implementation of image average blur was performed using an openCV function (cv2.blur)

Similarly, image transpose was conducted solely using naive methods in both Cuda and OpenCL on both datasets. These methods were compared to a serial implementation using a numpy function (numpy.transpose).

Execution times for image blur in Cuda (naive), image blur Cuda (tiled), and image blur (serial) were recorded. Execution times for image blur in OpenCL (naive) were also recorded.

Four different plots were created:

Plot 1 demonstrates the four different execution times (Cuda Naive, Cuda Tiled, OpenCL, and Serial) for image blur on dataset 1 (100 pathology images of size 3100 x 3100). Plot 2 demonstrates the four different execution times (Cuda Naive, Cuda Tiled, OpenCL, and Serial) for image blur on dataset 2 (50,000 natural images of size 32 x 32). Plot 3 demonstrates the four different execution times (Cuda Naive, Cuda Tiled, OpenCL, and Serial) for image transpose on dataset 1. (100 pathology images of size 3100 x 3100). Plot 4 demonstrates the four different execution times (Cuda Naive, Cuda Tiled, OpenCL, and Serial) for image transpose on dataset 2. (50,000 natural images of size 32 x 32).

***Index Terms*—Cuda, OpenCL, blur, transpose, naive, tiled, kernel, halo/ghost cell**

I. Introduction

Image augmentation is a popular technique to represent the same image in a different form. For example, a common image augmentation technique is flipping vertically or horizontally, where the image appears to be "mirrored" or "flipped" along an axis after this augmentation is completed. More specifically, image augmentation is regularly used to increase the size of a dataset that is later used for training algorithms utilizing deep learning. In computer vision applications using deep learning, these algorithms tend to be data hungry, requiring many example images to learn from. Data or image augmentation is an effective way to increase the size of a dataset while maintaining the same kind or type of image examples the algorithm learns from. Unfortunately, many of these augmentation techniques are done on the CPU, which is proven as computationally intensive and time consuming. Consider an image dataset with 5000 images each of size 3100 x3100. A single augmentation technique applied to all images would increase the size of this dataset by 5000, with a new dataset containing augmented images and a total of 10,000 images. Now consider applying 5 augmentation techniques. That brings the total dataset size to 30,000 images. Some augmentation techniques can be executed quickly, on the order of milliseconds per image and sometimes quicker. Now imagine, it takes 240 seconds (4 minutes) to conduct a single data augmentation technique on a single image. If we wish to apply that to every image in a dataset of size 5,000, that would take a minimum of 2,0000 minutes (33+ hours) not taking into account the read/write operations to save the images. This process can be grueling and time consuming, often leading to deep learning pipelines that are unnecessarily long. This presents a ripe opportunity to apply parallel processing to various image augmentation techniques. Output data elements can be calculated independently of each other, which is a desirable trait for parallel computing.

In this project, image blurring and image transpose were explored, where the blurring operation 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 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 image blur 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 pixel value.

The significance of transposing a matrix is mainly to represent linear transformations such as rotation and scaling. By taking the transpose of a matrix that represents a linear transformation, properties of the transformation can be revealed. If the transpose of a matrix and the original matrix are equivalent, for example, then the transformation is a symmetric transformation and the matrix is symmetric.

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 image blur kernels, the caches can provide tremendous amount of bandwidth to satisfy the data needs of threads.

II. Methods

A. Block Diagram

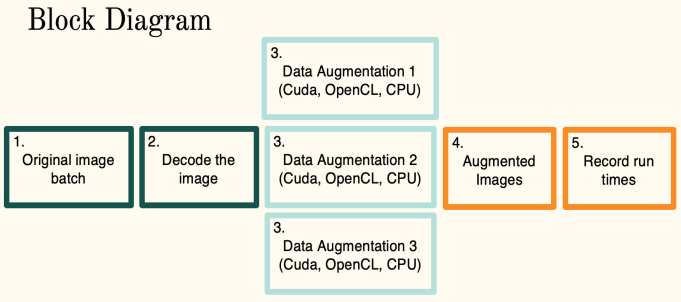


Fig. 1. Block Diagram for Image Augmentation with OpenCL and Cuda

B. Naive Image Blur Cuda and OpenCL

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

(1.) Initialize the input matrix with size of the dataset of interest (either 3100 x 3100, or 32 x 32) blur_kernel = [K X K], where K=2*n+1 and n 1 for dataset 2 (32 x 32 images) and 5 for dataset 1 (3100 x 3100 images). After blurring, the output matrix/image retains the same shape as the input matrix. (2.) Create the kernel code for image blurring utilizing simple indexing for each thread and computing the weighted sum of pixels/matrix elements of the input array with those of the blur kernel; global/device memory is used here only. (3.) Move input and expected output matrices to device memory; matrices here refer to a single image matrix. (4.) Compute block and grid size based on the shape of the input matrices and the optimal block size (32 ,32, 1) for 2D. (5.) Call kernel function to perform image blurring (6.) Measure run-time and return the output - a matrix resulting from the convolution of input matrix and blur kernel.

These steps were performed for both datasets.

For naive image blurring and tiled blurring, convolutional kernel size was kept constant.

C. Optimized/Tiled Image Blurring Cuda

The optimized/tiled 2D image blur 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 image blurring 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

D. Naive Image Transpose Cuda and OpenCL

We take a 2D matrix of any size and transpose it, which means the rows become the columns and the columns become the rows.

The steps taken to produce a transpoed image are similar to those described for image blurring.

(1.) Declare host variables a. Input matrix, shapes of this matrix, and a zeros matrix that will be the shape of the desired output and filled with the values calculated on the device (2.) Allocate memory on the device/GPU for the input matrix and the output matrix (3.) Create the kernel code/function b. We have 32x32 threads per block, and block size is 32x32, and grid size is calculated according to the size of the input array in the x-dimension divided by 32 and the same is done in the y-dimension. (4.) Similar as before, we launch and call the kernel code declaring the block and grid size, after transferring input data from the host (CPU) to the device (GPU). (5.) Call kernel function to perform image transpose (6.) To retrieve our desired output, the transposed matrix, we must then transfer the data on the GPU back to the CPU. (7.) Measure run-time and return the output - a matrix resulting from the transpose of the input matrix.

III. Results

A. Image Blurring and Image Transpose)

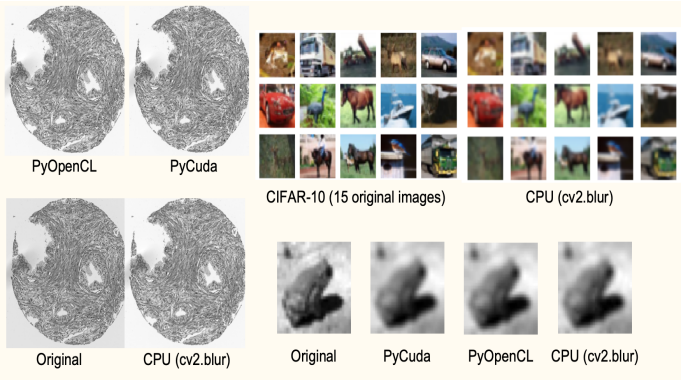


Fig. 2. Image Blurring with Cuda, OpenCL, and OpenCV

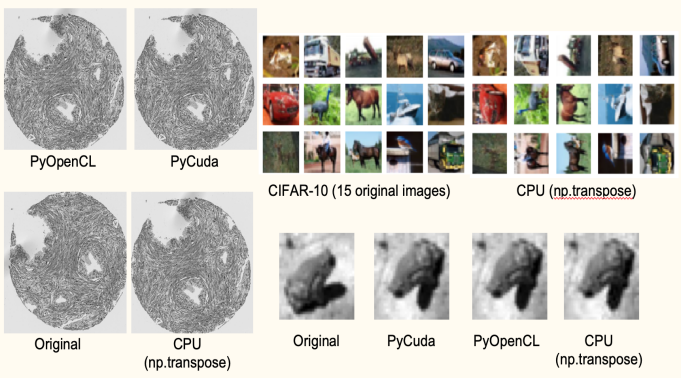


Fig. 3. Image Transpose with Cuda, OpenCL, and Numpy Transpose

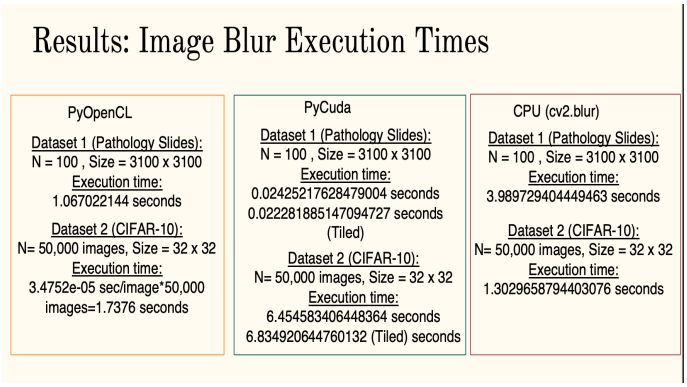


Fig. 4. Image Blur Execution Times for Dataset 1 and Dataset 2

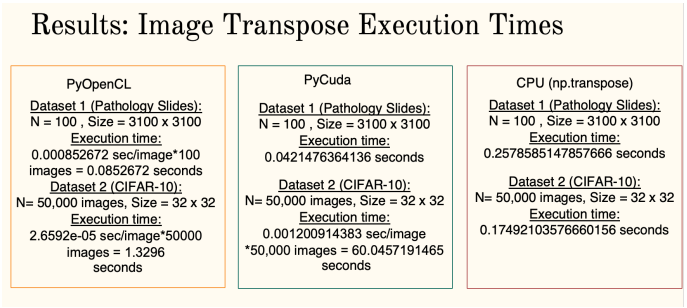


Fig. 5. Image Transpose Execution Times for Dataset 1 and Dataset 2

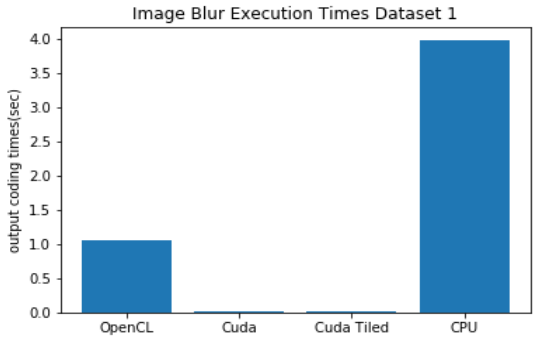


Fig. 6. Bar Plot for Image Blurring on Dataset 1

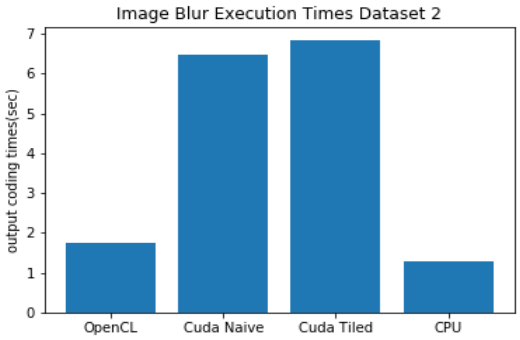


Fig. 7. Bar Plot for Image Blurring on Dataset 2

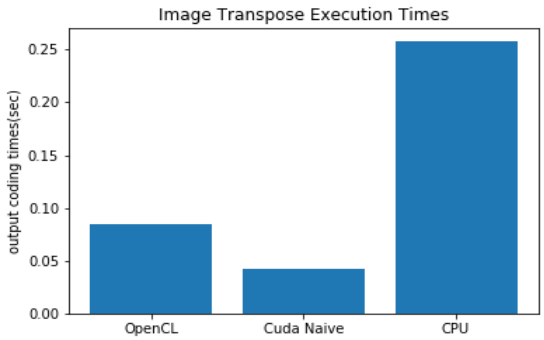


Fig. 8. Bar Plot for Image Transpose on Dataset 1

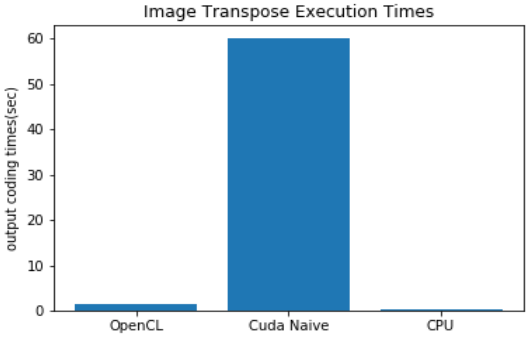


Fig. 9. Bar Plot for Image Transpose on Dataset 2

IV. Discussion

In regards to figures 6 and 7, we see that tiled method for image blurring using PyCuda was the fastest at executing the task for dataset 1. However, for dataset 2, we see that cpu was the fastest. I believe that for larger images and smaller batches Cuda performs better, whereas for smaller images and larger batches, execution time increases with Cuda. OpenCL seems to remain steady regardless of batch size and image size.

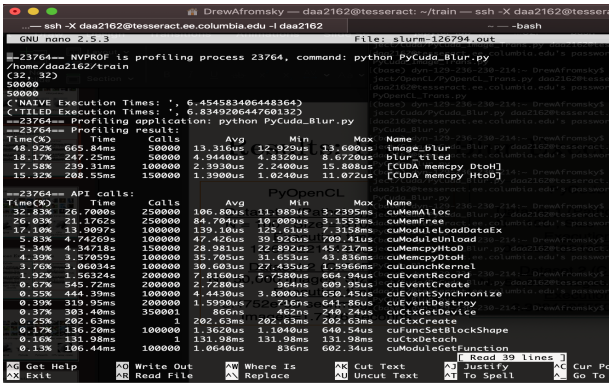
In regards to figures 8 and 9, for dataset 1 we see that cuda outperforms both cpu and openCL on image transpose, where the images are large, but the batches are small. However, Cuda seems to take longer on larger batch sizes, and smaller matrices. Cpu outperforms both openCL and Cuda for these images/batches.

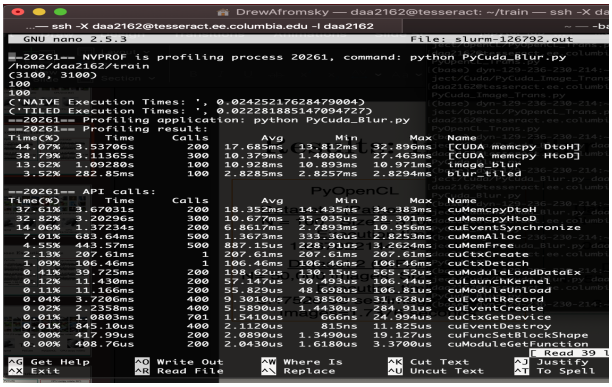
In general, we see that Cuda executes the quickest on larger matrices and smaller batches. OpenCL seems to work best with larger batch sizes and smaller image sizes. CPU performs the best on image transpose, when the batch size is large and images are small.

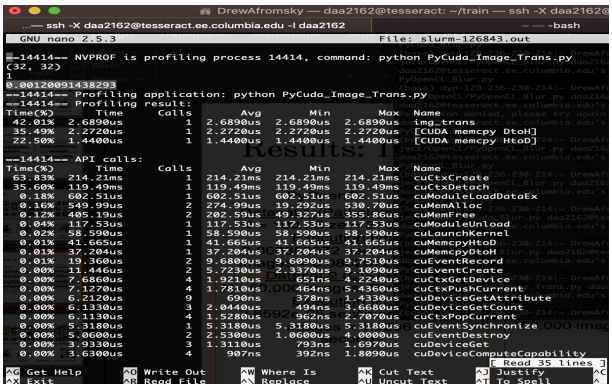
It would have been interesting to also compare these methods for large batch sizes with large matrices and small batch sizes and small matrices. Additonally, if given more time, it would have been intersting to compare to data augmentations tools publicly available such as albumentations, NVIDIA's DALI, and openCV's GPU-compatible framework.

I believe the tiled image blurring may have some issues in terms of ability to work with larger matrices, since some of the output images appeared to be partial images. Also, an attempt to generate the Cuda code for image rotation was done, but not completed and no image outputs were obtained.

V. Appendix







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.