Identifier Animalia

High Performance Computing

By Rahil Mehta / Ardalan Ahanchi / Andrew Nelson - CSS 535 A

Abstract

Convolutional neural networks are increasingly used for image classification and computer vision. There has been prior work by researchers running neural networks on a GPU. This project explores the performance and accuracy impact of parallelizing the matrix operations involved in training a neural network and running it on GPU. The GPU implementation of the matrix operations was significantly faster than the CPU version. The accuracy was not as high as expected. Future work could include parallelization of image manipulation and increasing the number of hidden layers.

Overview

The quantity of image data that is available is continually increasing. GPUs may be helpful for image processing due to their support for massive parallelism and SIMD calls. The goal of this project was to implement a machine learning algorithm for detecting dogs in images using OpenCV. Dog images were gathered from OpenImages (Open Images V6, 2020) and then used to help train our algorithm. After the algorithm has been trained, we will test our algorithm and compare a CPU and GPU implementation. The success of each implementation was measured by [insert here]. The results and analysis from this comparison is detailed in this report.

Background and Related Work

Scientists are beginning to use computer vision for animal detection to help accelerate the process of observing animal patterns and behaviors. Companies such as Microsoft are raising awareness of threatened species and are using computer vision technology to help (Snow Leopard Trust, 2019; Marek Rogala, 2019). Snow leopards are a threatened species and are difficult to track in the wild [1]. Researchers supported by the Snow Leopard Trust collected 200,000 to 300,000 images using camera trap studies but they had to manually look through the data to identify the animals. A team of Microsoft engineers developed a classifier using deep neural networks to distinguish between leopards and other animals. Norrouzadeh et al. from the University of Wyoming used deep convolutional neural networks to identify and describe the behaviors of 48 species [3]. They used the Serengeti Snapshot database containing 3.2 million images. The system is used to identify the species, count the number of animals present, and identify the activity the animals are engaged in, such as resting or eating. This task can be challenging even for humans. Deep learning requires labeled data for training and significant computing power. Their algorithm identified animals with 93.8% accuracy [3]. When only the images where the system is confident are considered, the accuracy is 99.3% [3].

Convolutional neural networks (CNN) and support vector machines (SVM) are techniques used for image classification. A convolutional neural network is similar to a standard multilayer perceptron (MLP) with the addition of convolutional layers and pooling [4]. A convolutional layer applies a filter to the input image by computing the dot product [4]. Pooling reduces the size of the input image to reduce the computational cost [4]. A support vector machine determines a hyperplane separating two classes [4]. SVMs were designed for binary classification but can be used for multilabel classification using the one-versus-all technique.

Previous research has been done on parallelizing the training of neural networks. It is time-consuming to train a neural network because of the large number of epochs and weight updates required [5]. Poli and Saito implemented parallel facial recognition on GPU with CUDA [6]. They used a Neocognitron Neural Network with a high performance computing architecture. A neurocognitron has several stages of layers organized in two-dimensional matrices known as cellular plains [6]. Poli and Saito used an NVIDIA GeForce 8800 GTX GPU. The images were sourced from two university databases. They achieved 98% accuracy on one of the databases and obtained a significant speedup [6]. Researchers from universities in Merida, Mexico used CUDA and CUBLAS to parallelize parts of a backpropagation algorithm [5]. They used CUBLAS for matrix and vector operations [5]. They achieved a 63x speedup using an NVIDIA Tesla C1060 compared to a CPU [5]. The neural network was tested on cancer and mushroom data [5].

Environment Specifications

The results detailed in this report were generated using the hardware specified below. The source code used for this project was configured using CMAKE to help pull in the necessary dependencies and help create solutions that would work on each team member's development environment. The results gathered in this report was obtained with the build specified below.

GPU Specifications

Name	NVIDIA RTX 2070 MaxQ
Architecture	Turing
Compute Capability	7.5
CUDA Cores	2304
Approximate Total Memory	8 GB

Number of SMs	36
Number of Tensor Cores	288
Number of RT Cores	36
L1 Cache Size (Per SM)	64 KB
L2 Cache Size	4 MB

Computer Specifications

Manufacturer	HP
Operating System	Linux Mint 19.3 (Linux 5.3.0)
RAM Size	32 GB
SATA SSD Storage	1TB
M.2 SSD Storage	512 GB

CPU Specifications

Type	Intel Core i7-9750h
Number of Cores	6
Number of Potential Cores	12 (logical cores using hyper-threading)
Frequency	2.6 GHz - 4.5 GHz
L1 Cache Size	64 KB (Instruction + Data)
L2 Cache Size	1.5 MB
L3 Cache Size	12 MB
Thermal Design Power	45 Watts

Build Specifications

IDE/Editor	Gedit, Visual Studio Community
Platform Target	x64
Compiler	NVCC + GCC (GNU Compiler Collection)
Configuration	Release

Compilation

As mentioned earlier, CMAKE was utilized to help pull in the necessary project dependencies and create solutions that are capable of running on each team member's development environment.

For parsing images and loading images, a separate tool was developed to generate what the team refers to as "resource files". Each resource file is a text file that contains a of pairs. The first element in the pair represents the path to the OpenImage "label" file which contains the coordinates of the image's boundary area, and the second element in the pair is the path to the image itself. Due to the use of these resource files, each team member was required to set up environment variables as specified in the AddingProjectEnvironmentVariable.pdf document.

For the image loading and manipulation, OpenCV must also be installed on the PC. For Windows users, the path to the OpenCV libraries must be added to the system's path. Linux users can utilize a package manager. Lastly, the latest CUDA toolkit must be installed in order to run the appropriate kernel and GPU commands.

With all of the steps described above completed, we were able to use CMAKE to configure and generate our solutions. For Windows users, CMAKE generates a visual studio solution from which the project can be run and further developed. For Linux users, multiple executable files are created that can then be used for development.

Approach

After researching the problem, the team decided to focus most of the performance improvements in a multi-layer perceptron (MLP) algorithm. To provide the most flexibility, we also decided on writing the algorithm from scratch. However, to support this algorithm, steps to gather and format data (image data) must also be done manually. For image detection,

OpenCV can be used as a vehicle to help perform the necessary image transformations to format the data as needed for the rest of the software.

Additionally, we suspect that we can improve the MLP algorithm performance by adding at least one convolutional layer as an input to the MLP algorithm. By adding the convolution layer, it can help improve the performance of the MLP algorithm due to a reduced input. For these reasons, we plan on first manipulating the raw images first utilizing OpenCV and then pass the data into the convolution layer which will then be used as the input to the MLP algorithm.

We have parallelized the mathematical operations used by the MLP. The required operations are matrix multiplication, matrix addition, matrix subtraction, ReLu, inverse ReLu, sigmoid, and sigmoid prime. Each of these operations has its own kernel. Each thread is responsible for calculating a single element of the output. Matrix multiplication was optimized with tiling and shared memory. It was designed to work on arbitrary-sized matrices. There is a class for hybrid operations where the CPU or GPU operation is used depending on the array size.

Challenges

From an experience perspective, our team's challenge had to deal with our combined experience with computer vision, convolution neural networks, and MLP. Some of the team members have had experience with these topics, while others had none at all. This resulted in a steep learning curve for all team members.

Technically, our project had limitations as to what pieces of the software can actually be parallelized. The training portion of the MLP algorithm is extremely difficult to parallelize due to the nature of how the algorithm functions. Since the training algorithm updates weights after loading in each data element, each subsequent element becomes dependent on the training results of the previous element. A challenge we faced at the end of the project was integrating the image processing, convolution, and multi-layer perceptron elements of the program. The image manipulation and convolution components pass their outputs to the next phase of the algorithm, so the output must be in the expected format.

Design and Implementation

For this report, the focus on performance improvement is on the MLP algorithm. A sequential implementation was written first, followed by a parallel implementation. As mentioned earlier, in order for the MLP algorithm to function properly, the steps to manipulate the images and add them as part of a convolution layer are required. For that reason, the system's design consists of three primary components - image loading and manipulation, a convolution layer,

and the MLP algorithm. Once the image data is loaded and manipulated into the specific format, the convolution layer can then be created. The convolution layer then serves as the input to the MLP algorithm.

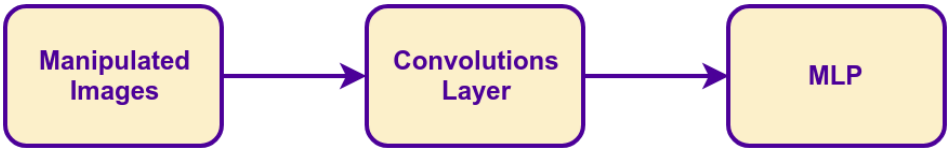

Figure 1. Software Data Flow

Image Loading and Manipulation

Before any other parts of the algorithm are to run, images must be loaded into the software and used to train the algorithm's model. All of the image loading and manipulation was performed using built-in OpenCV calls. For the training algorithm to function as expected, each image needed to be a flattened float matrix which represents normalized values of a grayscale image. Ideally, each image should only contain the content for which to train the algorithm (i.e. just the bounding area around a dog). Lastly, the algorithm requires that each image is of the same size. For that reason, after cropping all of the images, each image needed to be resized. The size was determined by calculating the average width and the average height amongst all of the cropped images. Overall, each image went through these transformation steps:

1. Convert to grayscale
2. Crop to boundary box
3. Resize image to average height and width

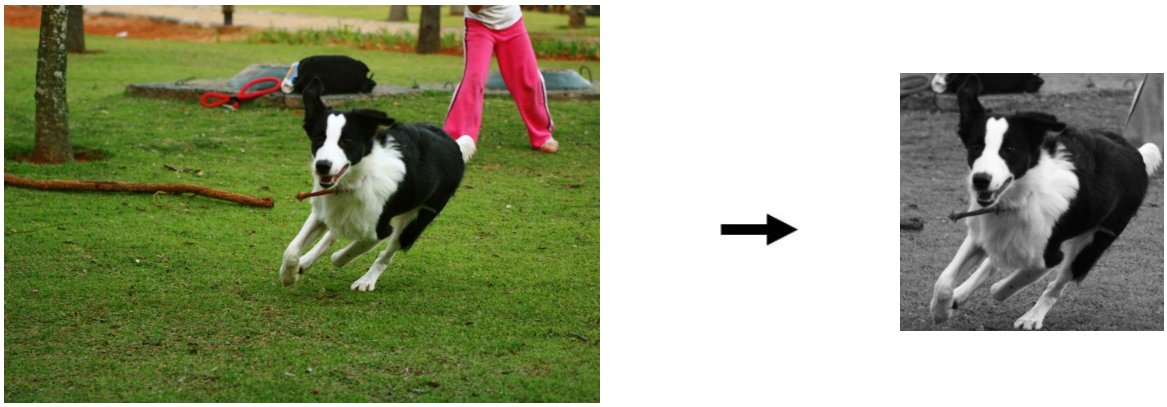

Figure 2. Image Transformation Example

Custom image handler classes were created to take the responsibility of loading and manipulating images. The class provided multiple utility methods, but ultimately one method

could be used to perform the different manipulations. The primary method chaining the image manipulation calls and the use of the methods can be seen in the code snippets shown below.

```
/*
 *  Wrapper method which parses the resource file and performs all the neccessary
 *  transforms on the image for other algorithms used in the program. Uses the specified
 *  dimension size when applying the resize step.
 *  @param rows - number of rows (height) for the desired image size
 *  @param cols - number of columns (width) for the desired image size
 *  @returns - OpenCV images that have been cropped, grayscaled, and resized
 */
std::vector<cv::Mat> ImageHandler::applyTransforms(unsigned int rows, unsigned int cols)
{
    std::vector<cv::Mat> resultImages;
    std::vector<OpenImage> openImages = parseImages();
    if (!openImages.empty())
    {
        std::vector<cv::Mat> boundaryImages = applyBoundaryTransform(openImages);
        if (!boundaryImages.empty())
        {
            cv::Size desiredSize(cols, rows);
            std::vector<cv::Mat> resizedImages = resizeImages(boundaryImages, desiredSize);
            if (!resizedImages.empty())
            {
                resultImages = resizedImages;
            }
        }
    }
    return resultImages;
}
```

Figure 3. Image Manipulation Wrapper

```
ImageHandler dogHandler(projectDir, oiDogResourceFile, NUM_IMAGES);
ImageHandler testHandler(projectDir, oiTestResourceFile, NUM_IMAGES);

std::vector<cv::Mat> transformedDogImages = dogHandler.applyTransforms();
std::vector<cv::Mat> testImages = testHandler.parseRawImagesFromResource();
std::vector<cv::Mat> rawDogImages = dogHandler.getRawImages();
```

Figure 4. ImageHandler initialization

Convolution Layer

We have implemented an experimental convolution layer though it is not the main focus of this report. It has been implemented from scratch without the use of any external libraries. The layer provides convolution and maxpooling operations. A Gaussian kernel is used for convolution. The convolution multiplies the image's grayscale picture values by the kernel.

Maxpooling is used to reduce the input dimensions of the input images. The output of the convolution layer is the input to the MLP.
The following code snippet shows how the pixels of an image are multiplied by the kernel or filter.

```
// index of input signal, used for checking boundary
rowIndex = i + (kCenterY - mm);
colIndex = j + (kCenterX - nn);

// ignore input samples which are out of bound
if (rowIndex >= 0 && rowIndex < images.rows() && colIndex >= 0 && colIndex < images.cols())
    sum += images.data[images.cols() * rowIndex + colIndex] * kernel[kernelSizeX * mm + nn];
```

Figure 5. Multiplication code from the convolution function.

MLP

Algorithm

We have implemented a basic multi-layer perceptron with one input layer, a variable number of hidden layers, and one output layer. All the layers can have an arbitrary number of neurons. Backpropagation is used to compute gradients and adjust the weights. The weights and biases are adjusted based on the learning rate. Iteratively adjusting the weights and biases improves the accuracy as the neural network is trained. We decided to implement this from scratch in order to control which parts we would like to parallelize.

Backward and forward propagation heavily relies on the use of matrix multiplication to help calculate the new weights and biases. While parallelizing the propagations is not possible given the nature of the algorithm (each layer can only be calculated when the previous layer has finished it's calculations), the operations that each layer computes can be parallelized to calculate the values from all the neurons in that layer in parallel. The following code snippet shows the forward propagation in the neural network.

```
//Go through the layers and compute the layers (up to the output).
for(size_t i = 0; i < this->_num_layers - 1; i++) {
    Mat layer;
    layer = this->_ops->mult(this->_layers[i], this->_weights[i]); //Multiply the weights.
    layer = this->_ops->add(layer, this->_biases[i]);              //Add the biases.
    this->_ops->sigmoid(layer);                                    //Apply sigmoid.

    //Assign the layer build to the correct index.
    this->_layers[i + 1] = layer;
}
```

Figure 6. Forward propagation as part for training

Operations

The MLP abstracts the mathematical operations using an operations interface. During the creation of a new network, the operation interface utilized is specified. This allows the program to switch between CPU and GPU execution in run-time. By creating the *Ops* interface, all the heavyweight operations were separated to be implemented to run on various hardware. The following figure demonstrates how the MLP class accesses the operations interface.

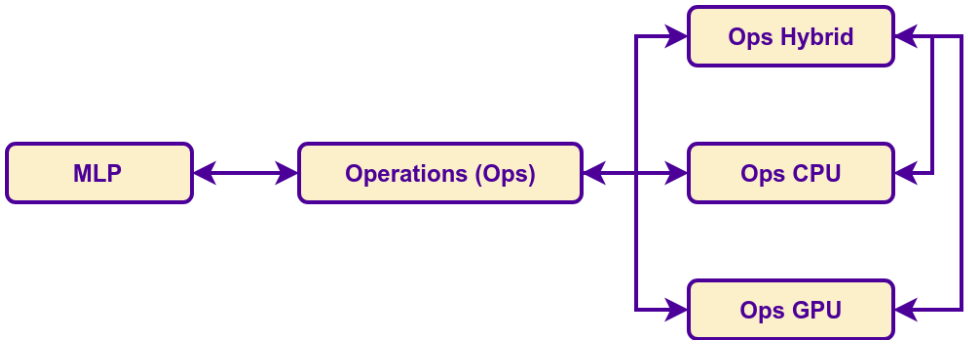

Figure 7. The Operations interface abstracts CPU, GPU, and Hybrid implementations.

A class that implements the *Ops* interface is guaranteed to provide the following functions for all matrix sizes (including vectors). All these operations are used extensively in multi-layer perceptrons:

- Matrix Addition (Used when adding biases)
- Matrix Subtraction (Used for gradients and error)
- Matrix Element-by-Element Multiplication (Used for gradients)
- Matrix Multiplication (Used in forward propagation, and backpropagation)
- Matrix Scaling (Used to apply a learning rate)
- Sigmoid (Used as an Activation function in forward propagation)
- Sigmoid Prime (Used as an Activation function in backpropagation)
- ReLu (Used as an Activation function in forward propagation)
- ReLu Prime (Used as an Activation function in backpropagation)

These operations are the only possibility in the MLP algorithm for parallelization, in this implementation all of the mentioned functions were parallelized using CUDA. The results were also compared to the CPU implementation for various matrix sizes.

A third hybrid Ops class was written to provide a combined CPU and GPU class. The hybrid approach is optimized to choose the CPU or GPU operation in run-time based on the problem

size. The sizes used as trigger points were based on extensive tests (the test methodology and results are discussed in the next sections).

Matrix

A matrix class was implemented which provides easy to access functions. Additionally the memory layout is flat, which allows easy transfer to GPU. However, what makes this implementation unique is that it does reference counting on the data of the matrix. The data is allocated on the heap, and by reference counting, a matrix object can be passed around without needing a separate malloc. This prevents memory leaks in the long-run since the memory is deallocated when the object falls out of scope. The following figure demonstrates the memory layout of this reference counted matrix.

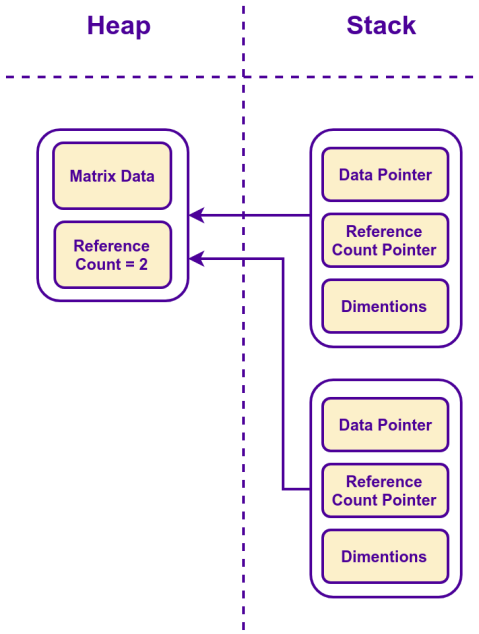

Figure 8. The implemented mat object's memory layout. The stack objects can be passed by value without deep copying the memory.

Optimizations

The Ops interface was fully implemented for parallel execution on the GPU. All the functions are optimized to maximize SM occupancy, and utilize GPU resources. In this section the two primary groups of optimizations are discussed.

Matrix Multiplication

Matrix multiplication is the only operation (from the *Ops* interface) which has a potential for memory re-use. Thus, there are many memory based-optimizations implemented in this section. The optimizations include tiling, shared memory usage, constant-memory usage, and

explicit register usage. The following figure demonstrates some of the optimizations for the matrix multiplication kernel.

```
 98 __global__ void mult_kernel(type* a_gpu, type* b_gpu, type* c_gpu) {
 99     //Holds each tile with TILE_SIDE x TILE_SIDE size for A and B matrices.
100     __shared__ type a_tile[TILE * TILE];
101     __shared__ type b_tile[TILE * TILE];
102
103     //Seperately stored for caching, and simplifying the complex indexing.
104     int tx = threadIdx.x;
105     int ty = threadIdx.y;
106
107     //Find the row and column for the output matrix (C).
108     int row = blockIdx.x * TILE + tx;
109     int col = blockIdx.y * TILE + ty;
110
111     //Used to find C at the end of calculations within the block.
112     type sum_c = 0;
113
114     //Go through every tile (in a single direction).
115     for(size_t t = 0; t < ceil((a_dims[1] / (type) TILE)) ; t++)
```

Figure 9. Demonstrating some of the optimizations for the multiplication kernel.

In the figure above, Lines 100-101 demonstrate the usage of shared memory with tiling. Lines 108-109 store the row and column number for improved register usage, and line 115 demonstrates the storage of matrix dimensions (a_dims) inside constant memory. The tile size used is a constant set at compilation time. Based on previous tests (for Program 2), the chosen tile size was 8x8 (best relative performance). This algorithm works on all arbitrary sized matrices and even vectors. However, there are performance penalties for tiling when using it to multiply vectors.

Linear Functions

The linear functions can not re-use data since they access every variable only once, and set it once. To optimize these functions, the focus was to run as many threads as possible at once while maximizing occupancy. To do this, 256 threads per block were used (since it is a multiple of warps, and it doesn't limit occupancy). The number of blocks were determined based on the number of calculations required, and the maximum number of blocks available based on the compute capability of the GPU.

If a single kernel could not compute all the results completely, the kernel will run as many times as required with the re-calculated number of threads and blocks. The kernel might not be able to cover every element if the matrices are too large, or if there are remainders (when the number of matrix elements is not a multiple of the total number of threads). In such cases, an offset will be stored to allow execution of the kernel again from where the last kernel left-off.

This approach will eliminate the need to do padding, or conditionals (which are expensive). The following figure demonstrates the loop which is used to auto-calculate linear kernel parameters (utilized for every function except matrix multiplication).

```
583     //Go through and calculate the kernel untill we finish the calculation.
584     while(num_calcs != 0) {
585         //Find the number of threads needed (based on how many calculations we have left).
586         size_t num_threads = (num_calcs > max_threads ? max_threads : num_calcs);
587 
588         //Calculate the number of blocks required.
589         size_t num_blocks = std::floor(num_calcs / num_threads);
590 
591         //Check if the number of blocks is more than maximum supported on the architecture.
592         if(num_blocks > max_blocks)
593             num_blocks = max_blocks;
594 
595         //Save the kernel offset in constant memory.
596         size_t kernel_offset_host[1] = { (output_size - num_calcs) };
597         cudaMemcpyToSymbol(kernel_offset, kernel_offset_host, sizeof(size_t));
598 
599         //Perform the kernel call based on the opcode.
600         this->kernel_call(opcode, num_blocks, num_threads, a_gpu, b_gpu, c_gpu);
601 
602         //Reduce the number of calculations left based on the amount calculated.
603         num_calcs -= (num_blocks * num_threads);
604     }
```

Figure 10. A loop used for calling linear kernels with the correct number of threads and blocks.

In addition to optimizing the kernel parameters, minimal memory optimizations were done wherever possible. For instance, the kernel offset mentioned previously is stored in constant memory. Another example of constant memory usage is storing the constant value used for scaling the matrix (will be set in run-time before the kernel's execution). The following code snippet demonstrates the constant values used throughout this GPU implementation.

```
32 /** The offset index used for each naive kernel (since the kernel exectues many times). */
33 __constant__ size_t kernel_offset[1];
34 
35 /** The constant value which is used for the scale kernel. */
36 __constant__ type scale_number[1];
37 
38 //Thes values are only used for matrix multiplication.
39 __constant__ size_t a_dims[2];  /**< Number of rows [0], and cols [1] in first matrix. */
40 __constant__ size_t b_dims[2];  /**< Number of rows [0], and cols [1] in second matrix. */
41 __constant__ size_t c_dims[2];  /**< Number of rows [0], and cols [1] in output matrix. */
```

Figure 11. Usage of constant memory for frequently used variables to allow improved caching, and very fast memory access.

Performance Analysis and Results

This section lists and examines the performance of the parallelized operations (which are used in this application), rather than the accuracy of the deep learning model. The goal was to provide the potential of performance improvements to future machine learning projects using an abstracted operations layer which parallelizes all possible heavyweight operations.

Initially, A CPU implementation for each operation was created. The CPU implementation is purely sequential and acts as the team's "naive" algorithm. This implementation serves as a baseline of comparison for the other methods.

Additionally, all the operations were implemented in parallel on the GPU. We planned to compare the CPU version with the full GPU implementation to examine the parallelisation potential of each operation (with different problem sizes). Initially, we tested our MLP algorithm with standard linear functions, an unexpected decrease in performance was noticed when using GPU. The probable reason for this slowdown was the memory allocation time and memory transfer time for the GPU far exceeds the amount of time it takes to perform that operation sequentially on the host. To accomodate this issue, the hybrid implementation was introduced.

This hybrid implementation is capable of determining whether to target the CPU or the GPU for a certain operation given the size of the problem. To help determine whether to run the problem on the CPU or the GPU, performance markers were logged for each implementation across a series of data points. Those markers were then used to create "triggers" for the hybrid implementation. A trigger acts as the data point at which the hybrid algorithm starts executing the problem on the GPU rather than the CPU. The comparisons of each algorithm for matrix operations and vector operations can be seen below.

Methodology

To test the effectiveness of the implementations, each operation was tested with various problem sizes (matrix and vector sizes), and then compared between the CPU, GPU, and Hybrid implementations. A total of 2700 data points were gathered in combination. The data was then parsed and filtered to create each plot.

Vector Operations

Our team was a bit surprised after gathering the initial results for the vector operations. Overall, the vector operations appeared to out-perform the GPU implementation by a significant margin. Our suspicion for this behavior is due the vector operations being linear operations. In addition, the time it takes to allocate memory and transfer memory to the GPU significantly impacts the overall performance, especially for our smaller data sets. Because of this, our hybrid implementation is running most of the operations on the CPU which ultimately results in similar performance measures to the CPU. The data plots for the vector operations can be seen below

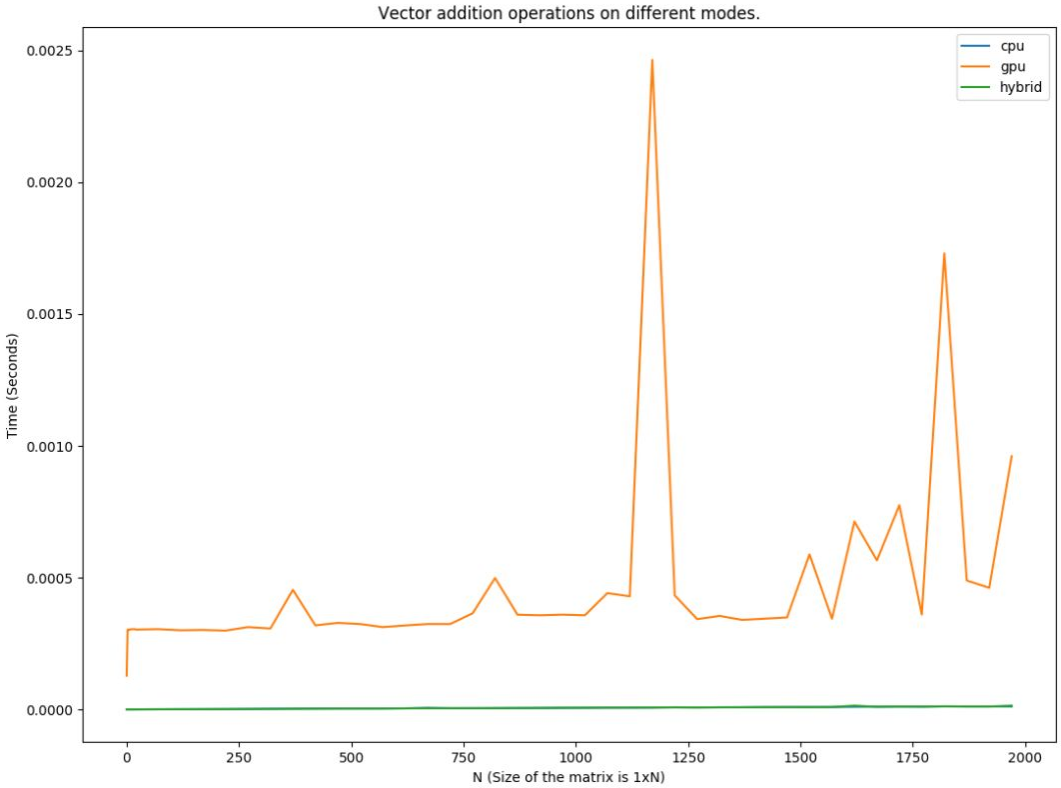

Figure 12. Performance Results for Vector Addition for Each Implementation

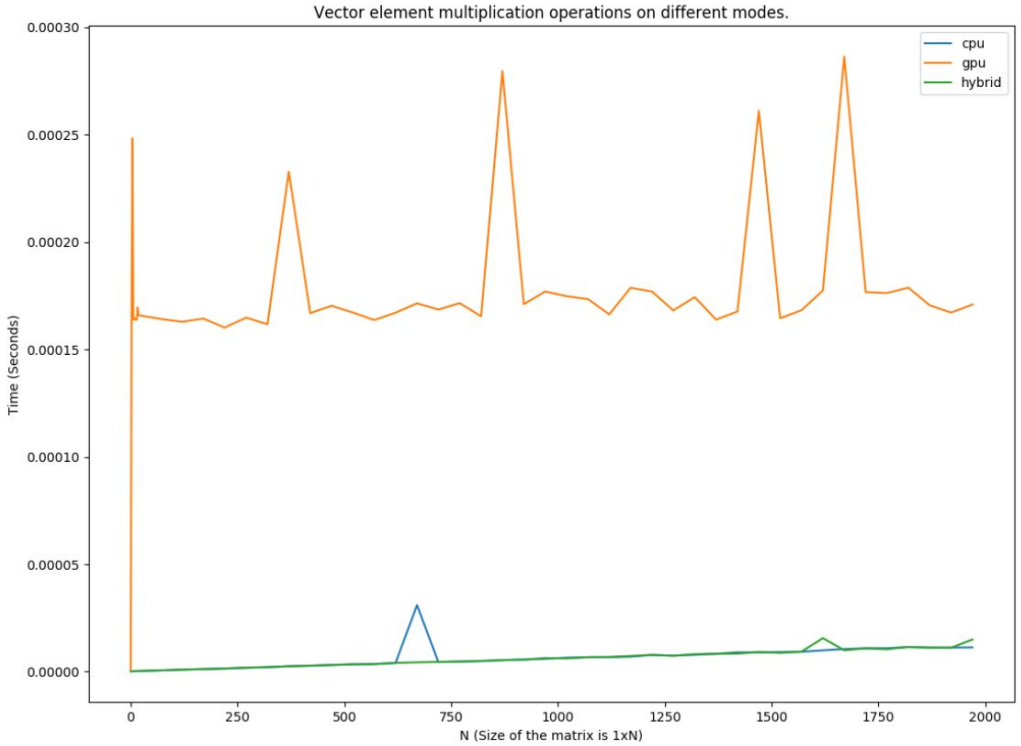

Figure 13. Performance Results for Vector Element Multiplication for Each Implementation

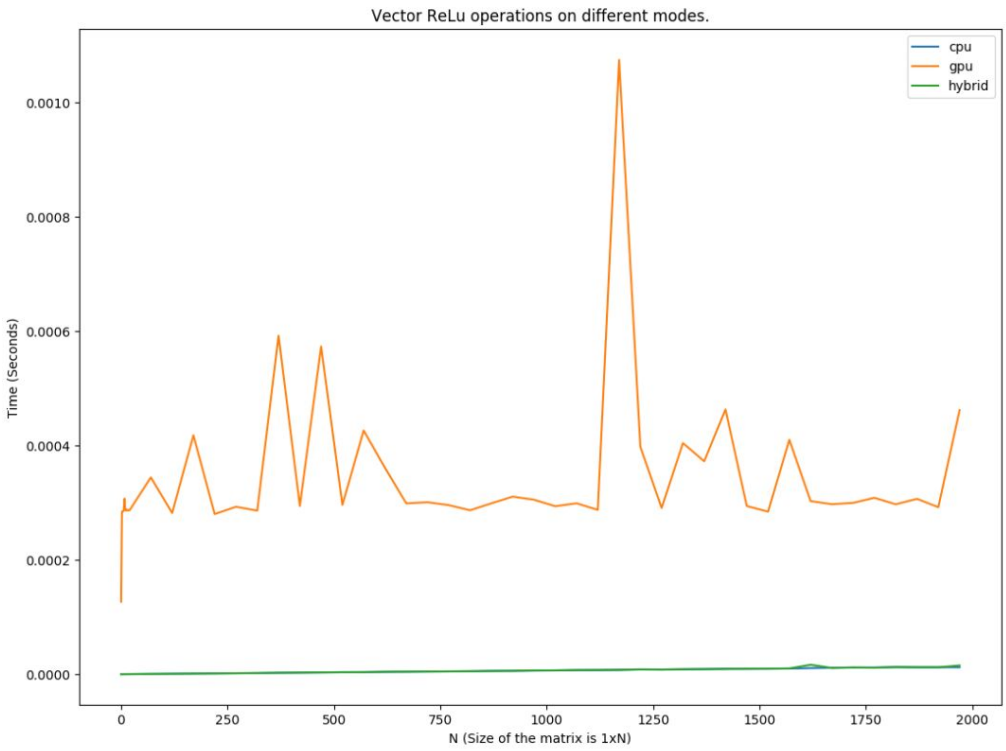

Figure 14. Performance Results for Vector ReLu for Each Implementation

Matrix Operations

Unlike our vector operations, our matrix operations performed significantly better with the GPU and hybrid implementations over the CPU implementation overall. This resulted in our hybrid implementation running more operations on the GPU. In comparison to the vector operations, the CPU performance could not out-perform the GPU/hybrid implementations when the problem asks for much more data. However, we did see one exception to this behavior with our matrix addition operations. We noticed that the CPU implementation started to outperform the GPU implementation with larger data sets. We suspect that other applications running at the time of this test may have affected the results by using the GPU for the resource. The data plots for the matrix operations can be seen below.

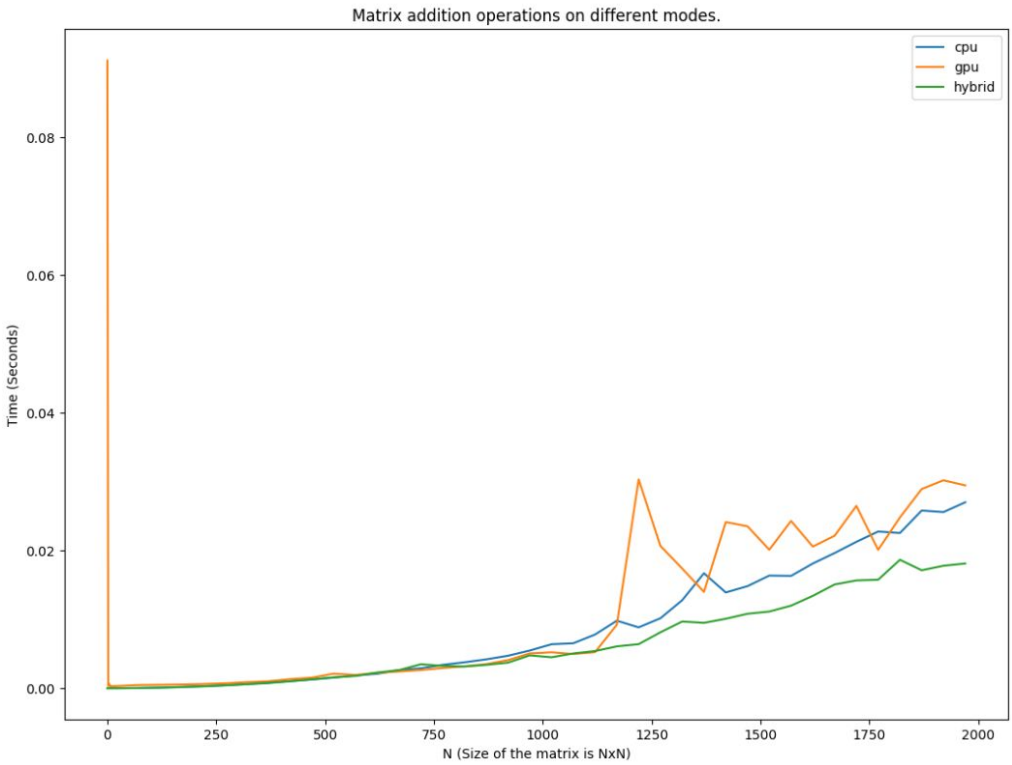

Figure 15. Performance Results for Matrix Addition for Each Implementation

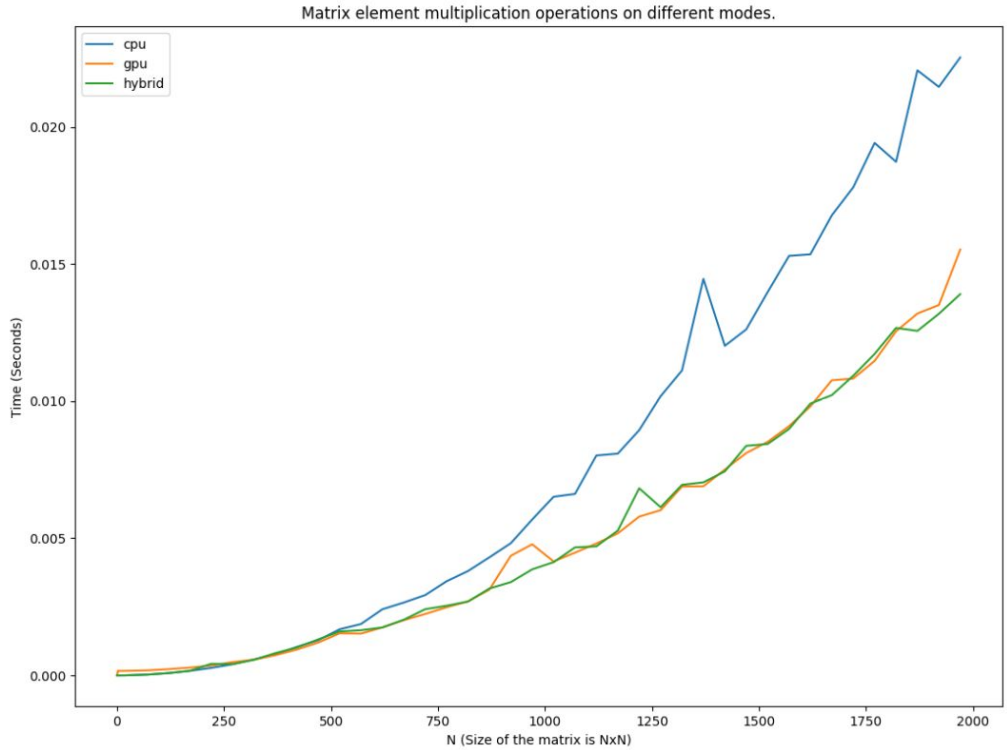

Figure 16. Performance Results for Matrix Element Multiplication for Each Implementation

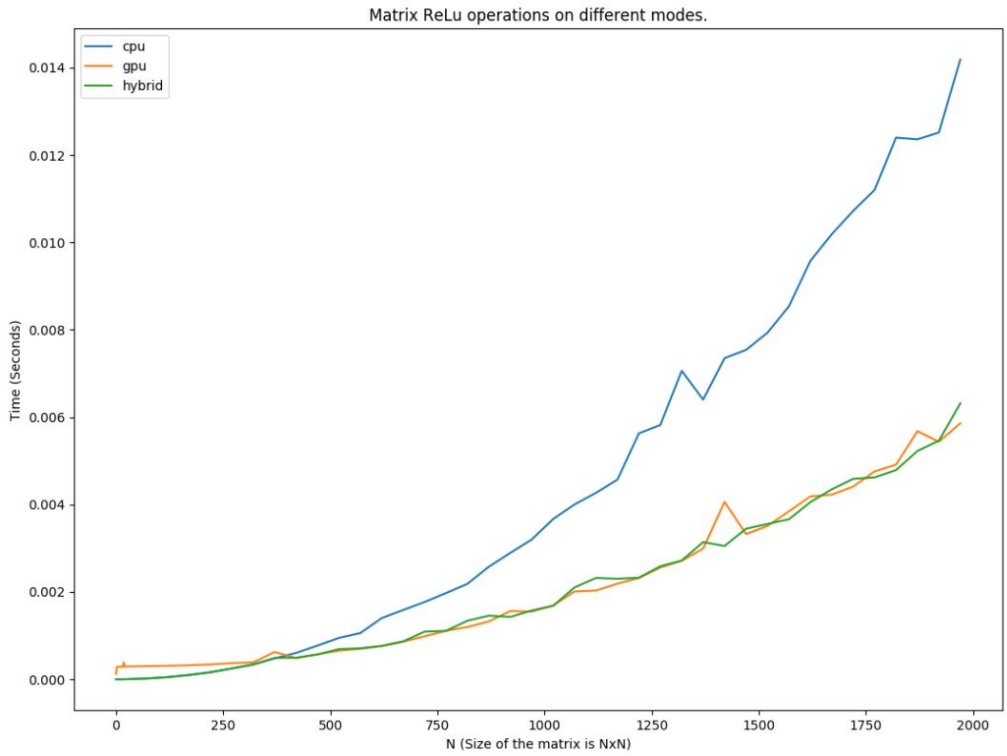

Figure 17. Performance Results for Matrix ReLu for Each Implementation

The profiler results are shown and described below.

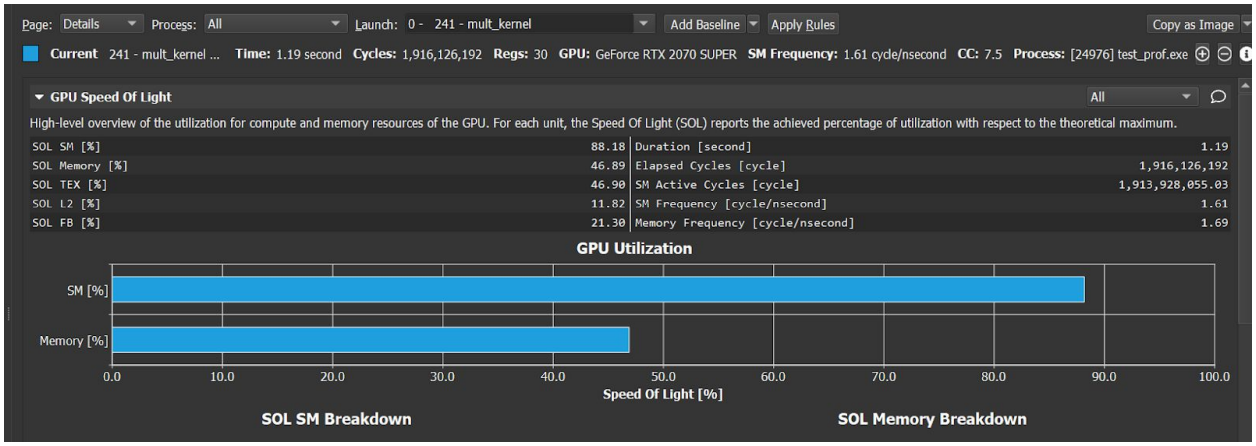

Figure 18. Profiler results for matrix multiplication

The profiler results show that the matrix multiplication operation called while training the neural network is computationally expensive. Each pixel value of the images has to be operated on and the thousands of floats require a significant amount of memory. Only 12% of the L2 cache is used, suggesting a potential optimization.

Deep Learning Tests

Methodology

To test the Deep learning algorithm's accuracy, a few tests were developed. Initially, two models were trained to predict the XOR function's results, and the results for a random linear function. The XOR function is a simple linear function which is often used for testing machine learning predictions (it's a starting point). After training the models, the program will run for multiple times and gatherers results for accuracy of the model. Additionally, the images were fed to the MLP algorithm to create a model, and some unknown images were passed to the model for prediction. The number of correct predictions were recorded to examine the model. The timing results were gathered for the whole execution to compare different operations classes.

Accuracy Results

The XOR function proved to be 100% accurate in every execution. And the custom linear function with 3 inputs (predicting the results for the arbitrary function $5x + 2y + z > 4$) was able to achieve over 90% of accuracy during execution. The following figure demonstrates the execution output of the aforementioned tests.

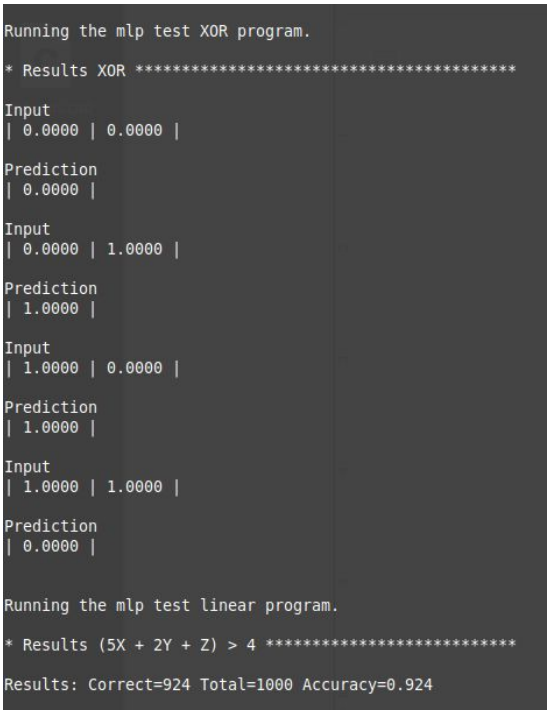

Figure 19. The results (input vs. predicted output) for models trained in the XOR function, and the custom linear function. The second results measure the accuracy of testing 1000 random inputs and comparing it to the correct (calculated) results.

The model trained for image detection had a detection correction accuracy of approximately 65% accuracy for images. Although the accuracy was not as high as the team had hoped to achieve, the value was large enough for us to move forward to focus on algorithm performance. The reason for the low accuracy is the small number of layers used, and the lack of experience in optimizing deep learning parameters (number of layers and neurons). Many deep learning algorithms heavily optimize the number of layers. The following figure demonstrates the image classification results based on 800 images of dogs and other objects. 640 of the images were used for training the model, and the rest were used to test it.

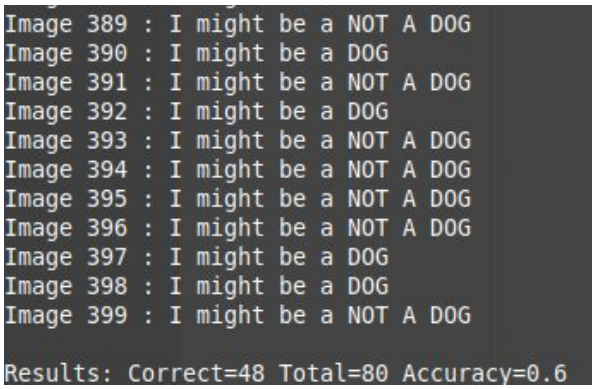

Figure 20. Image classification accuracy results for the trained model.

Performance

The XOR and the linear function were not helpful for measuring performance results for the parallel implementation. In both functions, the input sizes are very small (2 input elements for XOR, and 3 for the custom function). In such cases, all matrix operations are performed on extremely small matrices. In such cases, the GPU version was about 600% slower compared with the sequential. As mentioned previously, this is due to the transfer, and allocation overheads for such matrices. The following figure demonstrates how the different Ops implementations compare in total execution time.

Figure 21. Image classification performance using different ops implementations (Lower bars equal better performance).

As demonstrated in the figure, the performance was improved significantly using the hybrid implementation. However, as the input data gets larger, the potential for performance improvements also increases. In machine learning applications (or applications in general) with larger input data, and multi-dimensional input data, the performance gains will be much more significant.

External Dependencies

The project used the following tools and software in addition to our own source code to help our team achieve our goal.

OpenImages

OpenImages is a compilation of images provided by Google which is part of a challenge that aims to challenge software professionals and encourage the research of computer vision (Open Images V6 2020). OpenImages not only provides the image data itself, but it also includes certain information regarding the image. As part of the OpenImages training set, a text file is provided for each image which contains the boundary area of a specific item within the image. The image data and image information was used in this project to help train our algorithm. For this project, images from V5 were used since at the start of the project V6 was not yet available.

OIDv4_Toolkit

Although OpenImages is helpful in obtaining images, gathering specific images based on a keyword wasn't straightforward. OIDv4_Toolkit is a free to use and open source python script that was used to help gather specific data from OpenImages (OIDv4Toolkit 2020). Using this tool, we were able to pull our target image data for training, but also miscellaneous images for testing.

OpenCV

OpenCV is an open source library for computer vision and image processing. We used the C++ interface as our project is implemented in C++. OpenCV provides simple library functions for reading images into Mat objects. An image can be stored as RGB or grayscale pixels.

MatplotLib

The output data from the operations test program were formatted in a standardized way. Thus, the data is feeded into a Python program which parses the information using Regex patterns. The MatplotLib library was used to create all the plots in this document.

Other Potential Improvements

An improvement that could be made to the help optimize the entire process is during the image loading and manipulation. After loading in a large number of images, the time it took to load and manipulate the images became noticeable. This problem almost became embarrassingly parallel due each image being stored in a vector. In theory, it'd be easy to have each thread perform a single image manipulation method on each index of the image vector. Implementing this idea wouldn't necessarily improve any of the operations mentioned earlier as part of the MLP process, but would help reduce the software runtime.

Other future work could include using more convolutional layers and filters to improve accuracy, which is currently low. Additionally, adding convolution layers would help reduce the input into the MLP algorithm which should reduce the general number of calculations that theMLP would have to compute.

Lastly, considering using OpenMP may also be considered to provide more parallized methods.

Usage

File Hierarchy

- ./build/: Includes all the build executables, this is the directory used by CMake for output of the builds.
- ./images/open-images: This directory is included as part of source, but not included as part of the submission due to the amount of images included. This directory contains all of the images used in the project. Includes both the training images and the test images. The content can be viewed here: https://github.com/ArdalanAhanchi/Cuda_Animal_Detection/tree/master/images/open-images
- ./include/: Contains all the header files.
- ./results/: Contains the output results (text), plots, and the profiler results.
- ./src/: Contains all the source code.
- AddingProjectEnvironmentVariable.pdf: Document to help explain steps needed to set up the proper environment variable referenced in the project.
- CMakeLists.txt: The build file which links the libraries, and creates multiple executables.
- Plot.py: The parsing, and plotting program written to gather results from the test_ops executable, and create 18 plots. The stdout of test_ops should be piped into this script, and the stderr of it should be piped into /dev/null.

Guide

The following commands are used to compile the program on linux compatible systems:

1. To create a makefile: cmake .
2. To compile: make
3. To run: ./build/test_*

The following steps instruct how to use CMake to generate the proper visual studio solution:

1. Install CMake for windows
2. Target the source directory as the input
3. Specify a target output directory
4. Click “Configure”
5. Click “Generate”
6. Open the “Cuda_Animal_Detection” visual studio solution to run and build the code.

References

1. Snow Leopard Trust (2019), How AI Helps us Understand & Protect Snow Leopards https://www.snowleopard.org/how-ai-helps-us-understand-protect-snow-leopards/

2. Marek Rogala (2019), Recognizing Animals in Photos: Building an AI model for Object Recognition https://appsilon.com/object-recognition-transfer-learning/

3. Mohammad Sadegh Norouzzadeh, Anh Nguyen, Margaret Kosmala, Alexandra Swanson, Meredith S. Palmer, Craig Packer, and Jeff Clune (2018), Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning https://www.pnas.org/content/115/25/E5716

4. A. Agarap, "An Architecture Combining Convolutional Neural Network (CNN) and Support Vector Machine (SVM) for Image Classification", *arXiv*, 2017. [Accessed 20 March 2020].
5. X. Sierra-Canto, F. Madera-Ramirez and V. Uc-Cetina, "Parallel Training of a Back-Propagation Neural Network Using CUDA," 2010 Ninth International Conference on Machine Learning and Applications, Washington, DC, 2010, pp. 307-312.

6. G. Poli and J. Hiroki Saito, "Parallel Face Recognition Processing using Neocognitron Neural Network and GPU with CUDA High Performance Architecture", *ResearchGate*, 2020. [Online].

7. Open Images Dataset Stats V5", Storage.googleapis.com, 2020. [Online]. Available: https://storage.googleapis.com/openimages/web/factsfigures.html.

8. OIDv4_ToolKit, GitHub, 2020. [Online]. Available: https://github.com/EscVM/OIDv4_ToolKit.