Mobile CNN: A Comparison between the Parallelism of iOS Mobile Device and GHC Machine

Fong An Chang
ECE Department
Carnegie Mellon University
fonganc@andrew.cmu.edu

Po Hao Chou
ECE Department
Carnegie Mellon University
pohaoc@andrew.cmu.edu

1 Summary

We implemented basic units of convolutional neural network using CUDA framework in C++ on GHC machines and Metal framework on iOS device. We constructed models using the units we created. Baseline is implemented in sequential version. We compare our results on three popular models with different complexity and conduct some analysis on the results.

2 Background

With the great success of deep learning in different applications, there is an increasing trend to deploy deep learning model on different devices. For example, if a deep learning model is deployed on a mobile phone, a user can use the phone camera to identity objects directly without Internet connection. However, the limited resources of mobile devices make it challenging to parallelize the deep network prediction. Also, as the network gets deeper, which is the trend in deep learning, is the parallelism design still scalable?

In this project, we plan to implement both METAL SHADING and CUDA parallel version of network forwarding for modules related to image classification, and then conduct a profiling and review on the implementations and devices. Through this analysis, we hope to identify the scalability or limitation of mobile GPU hardware compared to CUDA hardware and their parallel design differences.

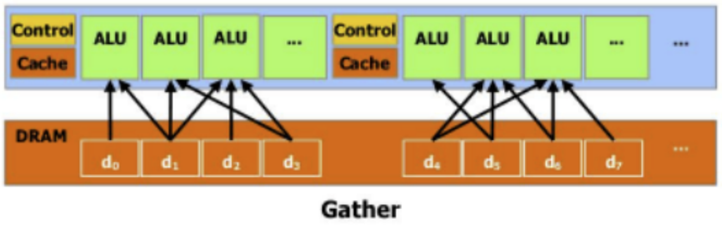
3 Approach

There are many kinds of operation in deep learning and we have implemented the four most popular operation for solving image classification problem. Each operation is implemented in sequential version and parallel version. Moreover, we apply gather pattern instead of scatter pattern in order to make performance better in GPU because we want to avoid atomic or lock operation. The detailed illustration is shown in figure 1.

3.1 Fully Connected Layer

Algorithm 1 shows the pseudo code of naive sequential fully-connected layer, we treat linear layer as a matrix multiplication problem. For an feature vector of dimension NxD_{in}, weighting of dimension D_{in}xD_{out}, bias of dimension D_{out}, we can get output feature vector of dimension of NxD_{out} by matrix multiplication operation and add operation. The time complexity for a computation unit will be $O(ND_{in}D_{out})$.

Just like the technique mentioned in the lecture, we first tried to create a CUDA thread for each position of output feature vector. Each thread will be responsible for do element-wise multiplication along dimension D_{in}. The pseudo code is shown in Algorithm 2. Furthermore, after profiling using nvprof tool, we found that the matrix multiplication is still bounded by memory access. As a result,



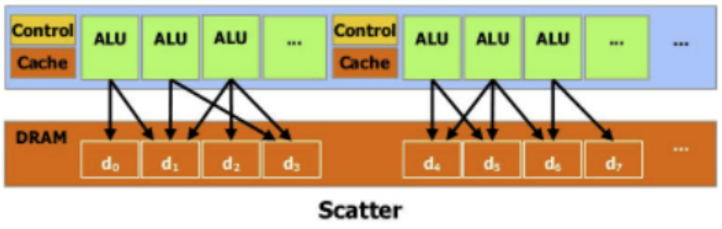


Figure 1: Gather and Scatter Pattern

we modify the code to do blocked matrix multiplication. We first load the required elements from the global memory into local memory and multiply the matrix in a tile fashion nevertheless the performance is worse than the previous one. Thus, we put the local copy arrays to shared memory to reduce the time for memory access so the performance is enhanced. We can see the profiled time in table 1. This final optimized matrix multiplication will also be used to optimize image convolution.

For the Metal Shading version, we also implement both Naive Algorithm and Algorithm 2 for matrix multiplication problem. And we assign the dimensions of thread blocks and threads just the same as the CUDA version. However, in our experiment, we choose iPhone 11 as the iOS device to evaluate, and in iPhone 11, the maximum number of threads in a thread group is 1024, which is also the same as the CUDA hardware in GHC machine. The detail would be introduce in the evaluation section.

The profiled time on iPhone 11 are shown in the Table 2. However, we observe that the Tiling optimization runs slower than the naive version. Here we try to set a default value in the device memory and remove the synchronizations in the loop. The running time of the Tiling version becomes 135.32 ms. Therefore, the trade-off between synchronization and device memory seems not the same as CUDA, or there may be other issues like bank conflicts which we haven't noticed.

Algorithm 1 Sequential Fully-Connected Layer

```
1: for i from 1 to N do
2:     for j from 1 to Dout do
3:         for k from 1 to Din do
4:             $X_{out}[i][j] += X_{in}[i][k] * W[k][j]$
5:         end for
6:     end for
7: end for
8: for j from 1 to Dout do
9:     $X_{out}[i][j] += Bias[j]$
10: end for
```

Algorithm 2 Optimized Fully-Connected Layer

```
1: for k from 1 to Din do
2:     $X_{out}[tidy][tidx] += X_{in}[tidy][k] * W[k][tidx]$
3: end for
4: for j from 1 to Dout do
5:     $X_{out}[tidy][j] += Bias[j]$
6: end for
```

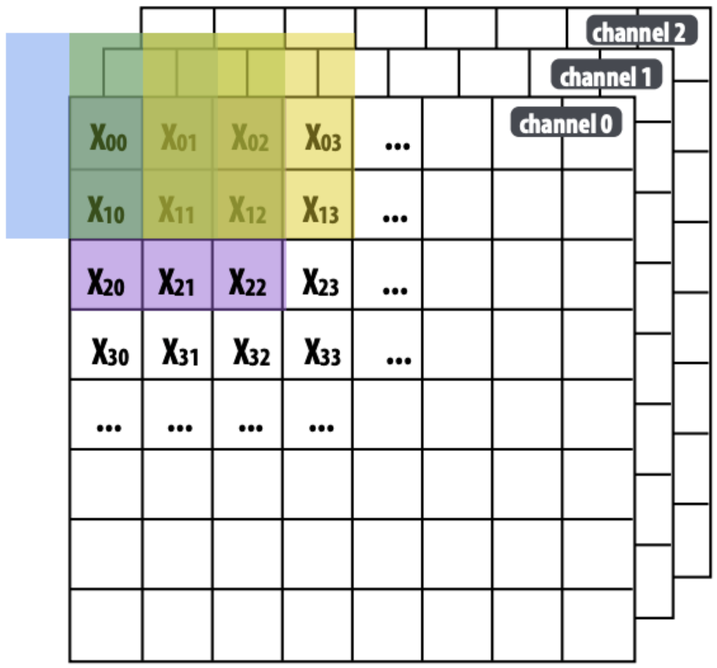


Figure 2: Convolution Operation.

Spec	Naive CUDA Version	Tiling CUDA Version	Speedup
Batch: 128, C$_{in}$: 4096, C_{out} : 4096	5.837 ms	4.791 ms	1.21841

Table 1: Performance comparison between naive CUDA version and tiling CUDA version on GHC

Spec	Naive Metal Version	Tiling Metal Version	Speedup
Batch: 128, C$_{in}$: 4096, C_{out} : 4096	141.7414 ms	188.7117 ms	0.7511

Table 2: Performance comparison between naive Metal version and tiling Metal version on iPhone 11

3.2 2-D Convolution Layer

The next operation we implemented is convolution operation. 3 shows the pseudo code naive sequential version of convolution operation and figure **??** shows how convolution works. We can see that this is a very expensive operation because it requires seven for loops to get the correct value. We also found that the data is independent and this operation can be easily optimized using SIMD. Therefore, we created CUDA threads on pixel position of output feature maps. The threads can be created in three dimension x, y and z, however, we are dealing with four dimension. Thus, assign thread along the x dimension to handle batch dimension, y dimension to handle channel dimension and z dimension to handle image width and height dimension. Each thread is responsible for collecting the required elements to calculate the correct value. Yet, we tested the Conv2D operation in Pytorch and we found that the performance is much greater than we expected. Thus, we surveyed for the reasons what made this difference. Finally, we found that convolution operation can be solved by matrix multiplication if the image and the weight are reordered. The figure 3 shows how it works. Thanks to modern compute units which is optimized for matrix multiplication problem so the convolution performance can be enhanced a lot. We profiled the difference of the performance between naive CUDA version and image to column CUDA version and the results are shown in Table 3.

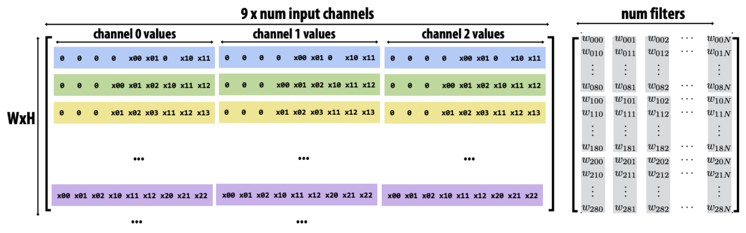


Figure 3: Optimized Convolution Operation.

Algorithm 3 2-D Convolution Layer

```
1:  for j from 0 to Cout do
2:      $temp = Bias[j]$
3:      for i from 0 to N do
4:          for k from 0 to Cin do
5:              for l from 0 to M do
6:                  for m from 0 to N do
7:                      for n from 0 to K do
8:                          for o from 0 to K do
9:                              $temp += X\_in[i][k][l+n][m+o] * W[k][j][n][o]$
10:                         end for
11:                     end for
12:                 end for
13:             end for
14:         end for
15:     end for
16:     $output[i][j][l][m] = temp$
17: end for
```

Spec	Naive CUDA Version	Im2Col CUDA Version	Speedup
Image Batch: 10, C$_i$: 3, W : 227, H : 227 Weight C$_i$: 3, C_o : 96, K : 11, $Stride$: 4	6.186 ms	2.670 ms	2.3168

Table 3: Performance comparison between naive CUDA version and im2col CUDA version on GHC

For the Metal version, we also implement both Algorithms. Especially for the im2col algorithm, we show the result of naive version matrix multiplication instead of the tiling version here because we found that the performance of naive version is better, which was stated in the Fully Connected Layer.

The result of Metal is shown in Table 4. We observe that the im2col version runs slower than the naive version in this Spec. However, if we use 1 as batch size instead of 10, the im2col runs a lot faster. The reason is that in our current im2col algorithm, we iterate each batch sequentially and for each batch, we call the GPU kernel to do im2col and matrix multiplication. Therefore, when batch size is large, the im2col version may not utilize all the GPU resource compared with the naive version. Also, it will have more overhead from kernel calls. However, for scenario like prediction, which usually use 1 as the batch size, we can see that im2col performs better, and the prediction is a more common use case for a mobile device.

Otherwise, it should also be possible to integrate the whole batch iteration, im2col and matrix multiplication into one kernel function. This may help the Metal im2col version performs better even if the batch size is large. If there are cases we plan do train on our mobile device, this can be optimized further.

3.3 Max Pooling Layer

The next operation we implemented is max pooling operation. 3 shows the pseudo code naive sequential version of convolution operation. Pooling operation serves as a down-sample operation in

Spec	Naive Metal Version	Im2Col Metal Version	Speedup
Image Batch: 10, C$_i$ $: 3, W : 227, H : 227$ Weight C$_i$ $: 3, C_o : 96, K : 11, Stride : 4$	162.263 ms	199.41 ms	0.8137
Image Batch: 1, C$_i$ $: 3, W : 227, H : 227$ Weight C$_i$ $: 3, C_o : 96, K : 11, Stride : 4$	35.249 ms	19.591 ms	1.8

Table 4: Performance comparison between naive and im2col Metal version on iPhone 11

convolutional neural network and it is very helpful for the reason that it can reduce the computational complexity. From the pseudo code, we can see that it requires six loops to get the correct value. This operation is also a data independent operation so it can be easily optimized using SIMD. We also created CUDA threads on pixel position of output feature maps. We assigned thread along the x dimension to handle batch dimension, y dimension to handle channel dimension and z dimension to handle image width and height dimension. Each thread is responsible for collecting the required elements to calculate the correct value. With the help of CUDA threads. we optimize the max pooling algorithm from $O(NC_{in}WHK^2)$ to $O(K^2)$ per computer unit.

For the Metal version, we use the same parallel algorithm as the CUDA version.

Algorithm 4 Max Pooling Layer

```
1: for i from 0 to N do
2:     for j from 0 to Cin do
3:         for k from 0 to W do
4:             for l from 0 to H do
5:                 for m from 0 to K do
6:                     for n from 0 to K do
7:                         $temp = max(X\_in[i][j][k+m][l+n], temp)$
8:                     end for
9:                 end for
10:                $output[i][j][k][l] = temp$
11:            end for
12:        end for
13:    end for
14: end for
```

3.4 ReLU Layer

The next operation we implemented is ReLU operation. 5 shows the pseudo code sequential version of ReLU operation. We need to traverse the 4-D tensor using four loops. Our GPU implementation is very similar to the previous we did. We assign threads in x dimension to deal with batch dimension of tensor, threads in y dimension to deal with channel dimension tensor and threads in z dimension to deal with width and height dimension. As algorithm 6 shows, we optimize the ReLU algorithm from $O(NCWH)$ to $O(1)$ per computer unit.

Algorithm 5 ReLU Layer

```
1: for i from 0 to N do
2:     for j from 0 to C do
3:         for k from 0 to W do
4:             for l from 0 to H do
5:
6:                 $X_{out}[i][j][k][l] = X_{in}[i][j][k][l]$
7:                 if $X_{in}[i][j][k][l] < 0$ then
8:                     $X_{out}[i][j][k][l] = 0$
9:                 end if
10:            end for
11:        end for
12:    end for
13: end for
```

Algorithm 6 ReLU Layer

```
1: $X_{out}[i][j][k][l] = X_{in}[i][j][k][l]$
2: if $X_{in}[i][j][k][l] < 0$ then
3:     $X_{out}[i][j][k][l] = 0$
4: end if
```

4 Results

4.1 Environments

In the experiment, we choose Gates & Hillman Centers (GHC) machines to evaluate our CUDA speedup. The GPU device is GeForce RTX 2080. The CUDA Driver Version is 11.4, and Runtime version is 10.2. The CUDA Capability version is 7.5. Its maximum number of threads per block is 1024.

As for iOS mobile device, we choose iPhone 11 to evaluate our Metal speedup. Its GPU family is Apple 6 [1]. Its maximum number of threads per block [2] is also 1024.

4.2 Experiment Setting

We evaluated our works on three different models LeNet-5, VGG-16 and AlexNet with different complexity level. For the CUDA hardware, we will compare our handcrafted GPU algorithm with both the CPU algorithm and PyTorch CUDA version. For the Metal hardware, we will compare our handcrafted GPU algorithm with the CPU algorithm. Finally, we will compare the speedup of CUDA version and the speedup of Metal version.

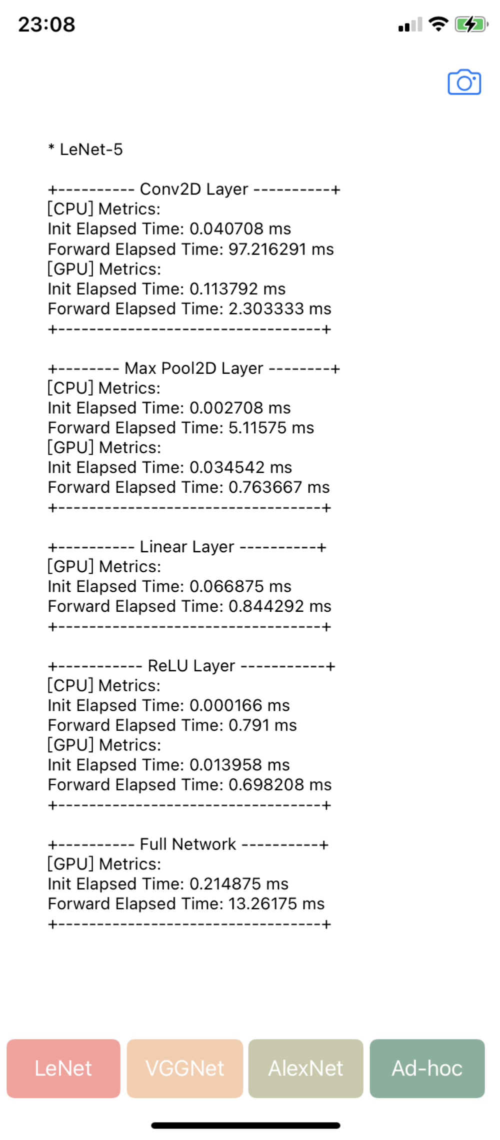
For Metal, we implement a benchmark App to display the metrics. One of the view is shown as in the 4.

4.3 CUDA Version Speedup

The Tables 5, 6 and 7 show the performance we enhanced and the speedup compared with the baseline. Observing the table, we can see that Pytorch does not have a good speedup on the operation which requires less computation. However, for those complicated operation like convolution operation, Pytorch really has a good speedup. For our work, for smaller model like LeNet-5, we got less speedup compared with the complicated model like VGG11 since there are less parallelizable works.

[1]https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf!

[2]Thread block is called thread group in Metal!



```
* LeNet-5

+---------- Conv2D Layer ----------+
[CPU] Metrics:
Init Elapsed Time: 0.040708 ms
Forward Elapsed Time: 97.216291 ms
[GPU] Metrics:
Init Elapsed Time: 0.113792 ms
Forward Elapsed Time: 2.303333 ms
+----------------------------------+

+-------- Max Pool2D Layer --------+
[CPU] Metrics:
Init Elapsed Time: 0.002708 ms
Forward Elapsed Time: 5.11575 ms
[GPU] Metrics:
Init Elapsed Time: 0.034542 ms
Forward Elapsed Time: 0.763667 ms
+----------------------------------+

+---------- Linear Layer ----------+
[GPU] Metrics:
Init Elapsed Time: 0.066875 ms
Forward Elapsed Time: 0.844292 ms
+----------------------------------+

+----------- ReLU Layer -----------+
[CPU] Metrics:
Init Elapsed Time: 0.000166 ms
Forward Elapsed Time: 0.791 ms
[GPU] Metrics:
Init Elapsed Time: 0.013958 ms
Forward Elapsed Time: 0.698208 ms
+----------------------------------+

+---------- Full Network ----------+
[GPU] Metrics:
Init Elapsed Time: 0.214875 ms
Forward Elapsed Time: 13.26175 ms
+----------------------------------+
```

Figure 4: One of the views in our benchmark App

4.4 iOS Metal Version Speedup

The Tables 8, 9 and 10 show the performance we enhanced and the speedup compared with the baseline. The GPU version here refers to the im2col with naive matrix multiplication version. We can see that for both Convolution layer and Linear Layer, the speedup is more effective as the network size and Mul-Add Count increase. The correlation is the same as the CUDA experiment.

Besides, fore Metal, we also compare its im2col algorithm and naive convolution algorithm on these networks when batch size is 1 to verify our previous observation. The results are shown in Table 11. We can see that when the convolution layer is large, it is more effective to apply im2col. However, when the convolution layer is small, the performance is worse. There are two possible reasons here. One is that in our matrix multiplication kernel, we assign 32×32 threads for each thread block, and the matrix are assigned to the threads by block favor. That is, the output height and width should be both greater than 32 to get the most benefit. However, the number of output channels of the Convolution layer in LeNet is 3, which is far smaller than 32. Therefore, the img2col version doesn't utilize the full GPU resource, and performs worse than the naive algorithm.

Spec	CPU	GPU	Speedup	Pytorch Speedup	Mul-Add Count
Convolution layer Image Batch: 1, C$_i : 1, W : 32, H : 32$ Weight C$_i : 1, C_o : 6, K : 5$	0.707 ms	0.080 ms	8.80620	7.8458	117606
Pooling layer Image Batch: 1, C$_i : 3, W : 28, H : 28$	0.056 ms	0.019 ms	2.86266	0.636	2352
Fully-connected layer Image Batch: 1, C$_i : 120$ Weight C$_i : 120, C_o : 84$	1.942 ms	0.059 ms	32.66462	3.963	10164
ReLU layer Image Batch: 1, C$_i : 3, W : 28, H : 28$	0.053 ms	0.03 ms	1.748	0.693	N/A
Full Network	7.819 ms	0.314 ms	24.9	14.55	606876

Table 5: CUDA Performance evaluation on LeNet-5

Spec	CPU	GPU	Speedup	Pytorch Speedup	Mul-Add Count
Convolution layer Image Batch: 1, C$_i : 3, W : 227, H : 227$ Weight C$_i : 3, C_o : 64, K : 11, Stride : 4$	1729.64 ms	0.621 ms	2787.12	5052	72855552
Pooling layer Image Batch: 1, C$_i : 64, W : 54, H : 54$	0.187 ms	0.021 ms	9.006	1.79	2052864
Fully-connected layer Image Batch: 1, C$_i : 9216$ Weight C$_i : 9216, C_o : 4096$	1378.56 ms	0.849 ms	1623.74	1680.124	37748736
ReLU layer Image Batch: 1, C$_i : 3, W : 55, H : 55$	0.038 ms	0.202 ms	5.30895	0.3709	N/A

Table 6: CUDA Performance evaluation on AlexNet

4.5 Comparison between CUDA Speedup and Metal Speedup

First, Table 5 and Table 8 shows their results on LeNet-5. We can see that the speedup of Metal is large. However, the reason seems that even for a small Convolution layer, the mobile CPU runs very slow. Especially for Fully-Connected layer, the CPU version of GHC machine is slow. Therefore, the speedup is even more obvious than the Metal version.

From AlexNet and VGG-11 (Table 6 v.s. Table 9) (Table 7 v.s. Table 10), we can see that their speedups are getting closer. That is, CUDA speedup is getting better. We thought it may because the GPU resource of CUDA is utilized more as the network size increases, but the mobile hardware resource becomes limited.

Finally, we observe that the running time between CUDA and Metal is getting larger as the network size increases. CUDA GPU is about 30x 40x faster than the Metal GPU when running on LeNet-5 and AlexNet, but 50x faster when running on VGG-11. This may also due to the GPU resource is still rich or not.

5 Conclusions

We designed and implemented an end-to-end network using CUDA framework in C++ language and Metal framework. We compared the speedup using three different complexity model LeNet-5, AlexNet and VGG. Significant speedup is observed on VGG network compared with LeNet-5.

Compared with CUDA, we found that the speedup of Metal device is bounded as the network size gets as large as VGG. One possible reason is the limited GPU resource of mobile devices.

Also, our experiment show that for mobile device, our current im2col implementation performs worse than the naive convolution algorithm when batch size is large. Therefore, for prediction, we will choose im2col implementation. For training, we will switch to the naive convolution. One possible

Spec	CPU	GPU	Speedup	Pytorch Speedup	Mul-Add Count
Convolution layer Image Batch: 1, C$_i : 3, W : 224, H : 224$ Weight C$_i : 3, C_o : 64, K : 3$	1773.301 ms	0.45 ms	3941.66	5304.964	86704192
Pooling layer Image Batch: 1, C$_i : 64, W : 112, H : 112$	0.78 ms	0.048 ms	16.3	7.822529	3211264
Fully-connected layer Image Batch: 1, C$_i : 25088$ Weight C$_i : 25088, C_o : 4096$	3758.24 ms	2.184 ms	1720.8	2591.918	102764544
ReLU layer Image Batch: 1, C$_i : 3, W : 112, H : 112$	0.053 ms	0.03 ms	1.748	10.3646	N/A
Full Network	13447.9 ms	20.498 ms	656.06	2335.9056	15470000000

Table 7: CUDA Performance evaluation on VGG-11

Spec	CPU	GPU	Speedup	Mul-Add Count
Convolution layer Image Batch: 1, C$_i : 1, W : 32, H : 32$ Weight C$_i : 1, C_o : 6, K : 5$	100.29 ms	2.4071 ms	41.664	117606
Pooling layer Image Batch: 1, C$_i : 3, W : 28, H : 28$	5.0773 ms	0.7267 ms	6.9868	2352
Fully-connected layer Image Batch: 1, C$_i : 120$ Weight C$_i : 120, C_o : 84$	3.99 ms	0.84396 ms	4.7277	10164
ReLU layer Image Batch: 1, C$_i : 3, W : 28, H : 28$	0.80075 ms	0.695 ms	1.152	N/A
Full Network	325.4914 ms	16.8789 ms	19.284	606876

Table 8: Metal Performance evaluation on LeNet-5

future work is to integrate im2col and matrix multiplication into one kernel call to do parallel on batch size.

6 Future work

We will implement more layers so we can construct more modern deep learning model. Besides, we will think about how to make training part fast using CUDA and Metal framework. Also, for realistic purposes, we will pretrained classification model using Pytorch framework and feed the pretrained weight to our work to see if it is an user-friendly application.

7 Work distribution

We work together for solving anything problems in this project so we think we both contribute 50% for this project.

References

[1] https://www.cs.cmu.edu/afs/cs/academic/class/15418-s22/www/lectures/24_dnn_impl.pdf.

Spec	**CPU**	**GPU**	**Speedup**	**Mul-Add Count**
Convolution layer Image Batch: 1, C$_i$ $: 3, W: 227, H: 227$ Weight C$_i$ $: 3, C_o: 64, K: 11, Stride: 4$	47752.24 ms	18.435 ms	2590.3032	72855552
Pooling layer Image Batch: 1, C$_i$ $: 64, W: 54, H: 54$	38.1542 ms	0.893 ms	42.726	2052864
Fully-connected layer Image Batch: 1, C$_i$ $: 9216$ Weight C$_i$ $: 9216, C_o: 4096$	19927.375 ms	54.292 ms	367.0407	37748736
ReLU layer Image Batch: 1, C$_i$ $: 3, W: 55, H: 55$	3.886458 ms	1.1378 ms	3.41577	N/A

Table 9: Metal Performance evaluation on AlexNet

Spec	**CPU**	**GPU**	**Speedup**	**Mul-Add Count**
Convolution layer Image Batch: 1, C$_i$ $: 3, W: 224, H: 224$ Weight C$_i$ $: 3, C_o: 64, K: 3$	64163.89 ms	23.072958 ms	2780.913	86704192
Pooling layer Image Batch: 1, C$_i$ $: 64, W: 112, H: 112$	81.2929 ms	1.1979 ms	16.3	3211264
Fully-connected layer Image Batch: 1, C$_i$ $: 25088$ Weight C$_i$ $: 25088, C_o: 4096$	309.7696 ms	107.06 ms	1720.8	102764544
ReLU layer Image Batch: 1, C$_i$ $: 3, W: 112, H: 112$	13.2405 ms	1.378 ms	1.748	N/A
Full Network	More than 1 hour	1324.1698 ms	More than 2719	15470000000

Table 10: Metal Performance evaluation on VGG-11

Spec	**Naive**	**im2col**	**Speedup**
Convolution layer Image Batch: 1, C$_i$ $: 1, W: 32, H: 32$ Weight C$_i$ $: 1, C_o: 6, K: 5$	1.326 ms	2.4071 ms	0.55087
LeNet-5 Full Network	15.0539 ms	16.8789 ms	0.89187684
Convolution layer Image Batch: 1, C$_i$ $: 3, W: 227, H: 227$ Weight C$_i$ $: 3, C_o: 64, K: 11, Stride: 4$	35.126667 ms	18.435 ms	1.9054335
Convolution layer Image Batch: 1, C$_i$ $: 3, W: 224, H: 224$ Weight C$_i$ $: 3, C_o: 64, K: 3$	29.787 ms	23.072958 ms	1.291
Full Network	1785.973 ms	1324.1698 ms	1.34875

Table 11: Metal performance comparison between different convolution algorithms (batch size = 1)