18-645: Term Project - LeNet-5 Convolution Neural Network Optimization

Name - Yu Chen(cyu2), Yifei Gong(yifeig), Guangbo Li(guangbol)

May 9, 2017

1 Introducing LeNet-5

LeNet is one of the first Convolutional Neural Networks(CNN) that has been widely applied in the industry. With incredible classification accuracy, it is used by several banks to recognize hand-written digits. This network was introduced in a paper by Yann LeCun et.al [1] and is seen as the "hello world" equivalent of Deep Learning for image classification and laid the groundwork for later research. For our term project, we decided to optimize the simple, yet structurally representative LeNet instead of the more complicated AlexNet that we originally proposed in our abstract. We believe that by optimizing LeNet, we can still learn the knowledge and experience that working on AlexNet would provide us. Our goal is to optimize the network by using OpenMP and CUDA platform respectively, in order to better understand the techniques we learned from the course.

1.1 Architecture of LeNet-5

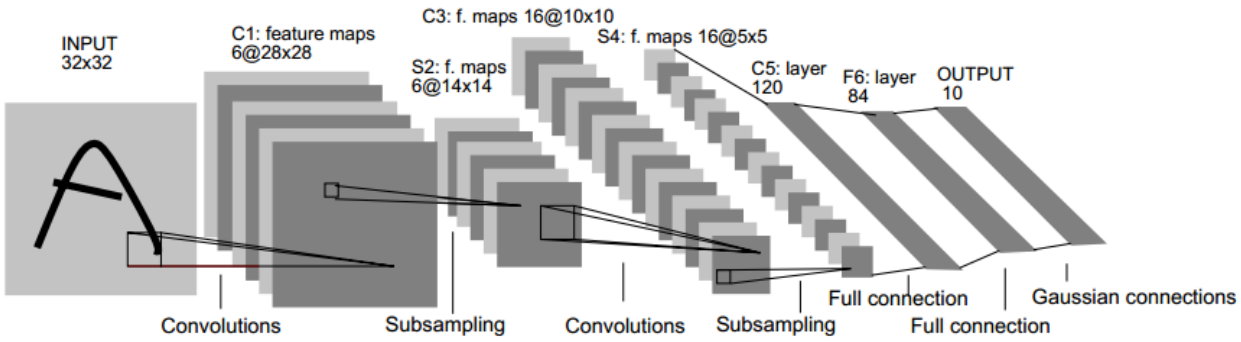

Figure 1: Architecture of LeNet-5

LeNet-5 consists of 7 levels and can process 32-by-32 images that contain hand-written digits. Among 7 layers, there are 2 convolution layers, 2 subsampling layers, 2 full connection layers and a Gaussian connection layer. It introduced a pioneering architecture known as convolutional neural network. Typical convolutional neural networks consist of 3 types of layers: convolutional layers, max-pooling layers and fully-connected layers.

Convolutional layers consist of a rectangular grid of neurons. It requires that the previous layer also be a rectangular grid of neurons. Each neuron takes inputs from a rectangular section of the previous layer; the weights for this rectangular section are the same for each neuron in the convolutional layer. Thus, the convolutional layer is just an image convolution of the previous layer, where the weights specify the convolution filter.

Max-pooling layers takes small rectangular blocks from the convolutional layer and subsamples it to produce a single output from that block. There are several ways to do this pooling, such as taking the average or the maximum, or a learned linear combination of the neurons

in the block. Our pooling layers will always be max-pooling layers; that is, they take the maximum of the block they are pooling.

Fully-connected layers handle the high-level reasoning in the neural network. A fully connected layer takes all neurons in the previous layer (be it fully connected, pooling, or convolutional) and connects it to every single neuron it has. Fully connected layers are not spatially located anymore (you can visualize them as one-dimensional), so there can be no convolutional layers after a fully connected layer.

1.2 Literature Survey

Here we will introduce related research on accelerating convolutional neural networks. Because of the wide application of LeNet-5, researchers has been trying to accelerate this network with various techniques. P.Yu.Izotov et.al [2] introduced an CUDA-accelerated implementation of LeNet-5. In the paper, a modified algorithm that utilizes parallel computing units is presented and achieved significant acceleration. Another paper by Max Jaderberg et.al [3] presents that by exploiting cross-channel or filter redundancy to construct a low rank basis of filters that are rank-1 in the spatial domain, drastic speedup could be achieved on convolutional layers. This method is also architecture agnostic which can can be easily applied to existing CPU and GPU convolutional frameworks for tuneable speedup performance.

Besides, many researchers focus on acceleration by using reconfigurable architecture. Masakazu Tanomoto et.al [4] introduced an acceleration method for convolutional neural networks based on Coarse Grained Reconfigurable Architecture(CGRA) for low-power embedded systems in order to fit specific application scenarios.Another acceleration method presented by Abdulrahman Alhamali et.al [5] focuses on using FPGA-accelerated Hadoop cluster for deep learning computations. The hardware prototype they presented combines the advantages of both cluster computing and FPGA-based hardware acceleration.

2 Optimization Process

In this section, we will introduce the whole optimization process and analyze them respectively. The first is about OpenMP optimization and the other part is abut CUDA optimization. We analyze the whole process of training and testing. The original CPU-version code are downloaded from a open-source project on Github [6].

In the training process, the CNN is updated by batch training. In the batch training, a batch of data is used to train the same LeNet-5(with the same parameter). After this, the LeNet-5 is updated by the batch training delta. So we can use parallel technique during batch training. In every batch training, there is four parts: input data normalization, forward propagation, use softmax to get the predict result and error, back propagation to get updated delta parameter. So we can use parallel to speed up these four parts respectively.

In the process of testing, we can predict the test data at the same time, this is the most impressive optimization. Then in testing, it also has the same operations the same as training including input data normalization and propagation forward. So this part will be the same as the training process.

2.1 Optimization using OpenMP

In this subsection, we will talk about the optimization process of LeNet-5 by OpenMP with different optimization method and in every step of training and testing.

2.1.1 Optimization on batching processing

a) Optimization Goal

- Improve the speed in the process batching training, as the LeNet-5 is the same for the batch data, so the training process is independently.

b) Optimization Process

- Use the unroll in the for loop in fetch batch data to reduce time in memory data translation. In the TrainBatch process, utilizing parallel for to speed up the independent part. Be careful about the updated delta, we try it with critical and atomic, we find critical is faster than atomic. In the LeNet-5 parameter updated process, we use another parallel to speed up the process.

The code relevant to batch processing showing below:

```
1  void training(LeNet5 *lenet, image *train_data, uint8 *train_label, int ...
       batch_size, int total_size)
2  {
3      #pragma unroll(10)
4      for (int i = 0, percent = 0; i ≤ total_size - batch_size; i += ...
           batch_size)
5      {
6          TrainBatch(lenet, train_data + i, train_label + i, batch_size);
7          if (i * 100 / total_size > percent)
8              printf("batchsize:%d\ttrain:%2d%%\n", batch_size, percent = i ...
                   * 100 / total_size);
9      }
10 }
11
12 void TrainBatch(LeNet5 *lenet, image *inputs, uint8 *labels, int batchSize)
13 {
14     double buffer[GETCOUNT(LeNet5)] = { 0 };
15     int i = 0;
16     int num =  (sizeof(LeNet5)/sizeof(double));
17
18     #pragma omp parallel for
19     for (i = 0; i < batchSize; ++i)
20     {
21         Feature features = { 0 };
22         Feature errors = { 0 };
23         LeNet5  ∆s = { 0 };
24         load_input(&features, inputs[i]);
25         forward(lenet, &features, relu);
26         load_target(&features, &errors, labels[i]);
27         backward(lenet, &∆s, &errors, &features, relugrad);
28         #pragma omp critical
29         for (int j = 0; j < num; ++j)
30             buffer[j] += ((double *)&∆s)[j];
31     }
32     double k = ALPHA / batchSize;
33
34     #pragma omp parallel for
35     FOREACH(i, num)
36         ((double *)lenet)[i] += k * buffer[i];
37 }
```

2.1.2 Optimization on input data normalization

a) Optimization Goal

- Use parallel for and reduction to improve the normalization speed.

b) Optimization Process

- In the process of normalization, we calculate the standard difference and mean of and input image(28x28), then smooth every input data with standard difference and mean. Beside the 28x28 image is padding with zero boundary. So the output is 32x32 normalized data.

The code for transpose is changed to:

```
static inline void load_input(Feature *features, image input)
{
    double (*layer0)[LENGTH_FEATURE0][LENGTH_FEATURE0] = features->input;

    const long sz = sizeof(image) / sizeof(**input);
    double mean = 0, std = 0;

    #pragma omp parallel for reduction(+:mean) reduction(+:std)
    FOREACH(j, sizeof(image) / sizeof(*input))
        FOREACH(k, sizeof(*input) / sizeof(**input))
        {
            mean = mean + input[j][k];
            std  = std + (input[j][k] * input[j][k]);
        }
    mean /= sz;
    std = sqrt(std / sz - mean*mean);

    #pragma omp parallel for
    FOREACH(j, sizeof(image) / sizeof(*input))
        FOREACH(k, sizeof(*input) / sizeof(**input))
        {
            layer0[0][j + PADDING][k + PADDING] = (input[j][k] - mean) / std;
        }
}
```

2.1.3 Optimization on propagation forward

a) Optimization Goal

- This part has the heaviest computation complexity as back propagation. As this part calculate amount of feature maps with convolution and max subsample pooling. In this part, the calculation for feature map and pooling can be operated at the same time.

b) Optimization Process

- Using parallel for to improve the calculation of feature map and subsampling(pooling). Besides, we use the definite number to take place of some function including GETCOUNT to reduce the cost.

```
1  static void forward(LeNet5 *lenet, Feature *features, ...
       double(*action)(double))
2  {
3      #pragma omp parallel for
4      for (int x = 0; x < 1; ++x)
5          for (int y = 0; y < 6; ++y)
6              CONVOLUTE_VALID(features->input[x], features->layer1[y], ...
                   lenet->weight0_1[x][y]);
7
8      #pragma omp parallel for
9      FOREACH(j, 6)
10         FOREACH(i, 784)
11             ((double *)features->layer1[j])[i] = action(((double ...
                   *)features->layer1[j])[i] + lenet->bias0_1[j]);
12
13     SUBSAMP_MAX_FORWARD(features->layer1, features->layer2);
14
15     #pragma omp parallel for
16     for (int x = 0; x < 6; ++x)
17         for (int y = 0; y < 16; ++y)
18             CONVOLUTE_VALID(features->layer2[x], features->layer3[y], ...
                   lenet->weight2_3[x][y]);
19
20     #pragma omp parallel for
21     FOREACH(j, 16)
22         FOREACH(i, 100)
23             ((double *)features->layer3[j])[i] = action(((double ...
                   *)features->layer3[j])[i] + lenet->bias2_3[j]);
24
25     SUBSAMP_MAX_FORWARD(features->layer3, features->layer4);
26
27     #pragma omp parallel for
28     for (int x = 0; x < 16; ++x)
29         for (int y = 0; y < 120; ++y)
30             CONVOLUTE_VALID(features->layer4[x], features->layer5[y], ...
                   lenet->weight4_5[x][y]);
31
32     #pragma omp parallel for
33     FOREACH(j, 120)
34         FOREACH(i, 1)
35             ((double *)features->layer5[j])[i] = action(((double ...
                   *)features->layer5[j])[i] + lenet->bias4_5[j]);
36
37     DOT_PRODUCT_FORWARD(features->layer5, features->output, ...
           lenet->weight5_6, lenet->bias5_6, action);
38 }
```

2.1.4 Optimization on Softmax

a) Optimization Goal

- Softmax is a kind of method to calculate the score of 10(0 to 9) output score. At the same time, the error is calculated from the score and labels. By utilizing OpenMP technique, we can calculate the 10 output at the same time, as the process is independently.

b) Optimization Process

- Parallel for, reduction and atomic are used to calculate Softmax results and loss. The code is showing below.

```
static inline void softmax(double input[OUTPUT], double loss[OUTPUT], int ...
    label, int count)
{
    double inner = 0;

    #pragma omp parallel for reduction(-:inner)
    for (int i = 0; i < count; ++i)
    {
        double res = 0;
        for (int j = 0; j < count; ++j)
        {
             res += exp(input[j] - input[i]);
        }
        loss[i] = 1. / res;
        #pragma omp atomic
        inner = inner - (loss[i] * loss[i]);
    }
    inner += loss[label];

    #pragma omp parallel for
    for (int i = 0; i < count; ++i)
    {
        loss[i] *= (i == label) - loss[i] - inner;
    }
}
```

2.1.5 Optimization on back propagation

a) Optimization Goal

- Improve the speed of back propagation, the process to calculate loss can be operated at the same time.

b) Optimization Process

- Utilizing memory coalescing in the process of accumulating sum which eliminates memory accessing separately.

```
static void backward(LeNet5 *lenet, LeNet5 *Δs, Feature *errors, Feature ...
    *features, double(*actiongrad)(double))
{
    DOT_PRODUCT_BACKWARD(features->layer5, errors->layer5, ...
        errors->output, lenet->weight5_6, Δs->weight5_6, Δs->bias5_6, ...
        actiongrad);

    #pragma omp parallel for
    for(int x = 0; x < GETLENGTH(lenet->weight4_5); ++x)
        for (int y = 0; y < GETLENGTH(*(lenet->weight4_5)); ++y)
            CONVOLUTE_FULL(errors->layer5[y], errors->layer4[x], ...
                lenet->weight4_5[x][y]);

```

```
10      #pragma omp parallel for
11      FOREACH(i, GETCOUNT(errors->layer4))
12          ((double *)errors->layer4)[i] *= actiongrad(((double ...
               *)features->layer4)[i]);
13
14      #pragma omp parallel for
15      FOREACH(j, GETLENGTH(errors->layer5))
16          FOREACH(i, GETCOUNT(errors->layer5[j]))
17              Δs->bias4_5[j] += ((double *)errors->layer5[j])[i];
18
19      #pragma omp parallel for
20      for (int x = 0; x < GETLENGTH(lenet->weight4_5); ++x)
21          for (int y = 0; y < GETLENGTH(*(lenet->weight4_5)); ++y)
22              CONVOLUTE_VALID(features->layer4[x], Δs->weight4_5[x][y], ...
                   errors->layer5[y]);
23
24      SUBSAMP_MAX_BACKWARD(features->layer3, errors->layer3, errors->layer4);
25
26      #pragma omp parallel for
27      for(int x = 0; x < GETLENGTH(lenet->weight2_3); ++x)
28          for (int y = 0; y < GETLENGTH(*(lenet->weight2_3)); ++y)
29              CONVOLUTE_FULL(errors->layer3[y], errors->layer2[x], ...
                   lenet->weight2_3[x][y]);
30
31      #pragma omp parallel for
32      FOREACH(i, GETCOUNT(errors->layer2))
33          ((double *)errors->layer2)[i] *= actiongrad(((double ...
               *)features->layer2)[i]);
34
35      #pragma omp parallel for
36      FOREACH(j, GETLENGTH(errors->layer3))
37          FOREACH(i, GETCOUNT(errors->layer3[j]))
38              Δs->bias2_3[j] += ((double *)errors->layer3[j])[i];
39
40      #pragma omp parallel for
41      for (int x = 0; x < GETLENGTH(lenet->weight2_3); ++x)
42          for (int y = 0; y < GETLENGTH(*(lenet->weight2_3)); ++y)
43              CONVOLUTE_VALID(features->layer2[x], Δs->weight2_3[x][y], ...
                   errors->layer3[y]);
44
45      SUBSAMP_MAX_BACKWARD(features->layer1, errors->layer1, errors->layer2);
46
47      #pragma omp parallel for
48      for(int x = 0; x < GETLENGTH(lenet->weight0_1); ++x)
49          for (int y = 0; y < GETLENGTH(*(lenet->weight0_1)); ++y)
50              CONVOLUTE_FULL(errors->layer1[y], errors->input[x], ...
                   lenet->weight0_1[x][y]);
51
52      #pragma omp parallel for
53      FOREACH(i, GETCOUNT(errors->input))
54          ((double *)errors->input)[i] *= actiongrad(((double ...
               *)features->input)[i]);
55
56      #pragma omp parallel for
57      FOREACH(j, GETLENGTH(errors->layer1))
58          FOREACH(i, GETCOUNT(errors->layer1[j]))
59              Δs->bias0_1[j] += ((double *)errors->layer1[j])[i];
60
61      #pragma omp parallel for
62      for (int x = 0; x < GETLENGTH(lenet->weight0_1); ++x)
```

```
63          for (int y = 0; y < GETLENGTH(*(lenet->weight0_1)); ++y)
64              CONVOLUTE_VALID(features->input[x], ∆s->weight0_1[x][y], ...
                    errors->layer1[y]);
65
66      SUBSAMP_MAX_BACKWARD(features->layer1, errors->layer1, errors->layer2);
67  }
```

2.2 Optimization using CUDA Platform

In this subsection, the optimization process of LeNet-5 by CUDA will be introduced, different optimization methods are implemented.

2.2.1 Optimization on testing

a) Optimization Goal

- Improve the speed of the testing process, as the function predict operates on every test data independently.

b) Optimization Process

- Rewrite the testing function to a CUDA kernel, shared memory is used to store the prediction results for every test data.
- Related code

```
1       LeNet5 *gpulenet;
2       cudaMalloc((void**)&gpulenet, sizeof(LeNet5));
3       cudaMemcpy(gpulenet, lenet, sizeof(LeNet5), cudaMemcpyHostToDevice);
4       image *gputest_data;
5       cudaMalloc((void**)&gputest_data, COUNT_TEST * sizeof(image));
6       cudaMemcpy(gputest_data, test_data, COUNT_TEST * sizeof(image), ...
            cudaMemcpyHostToDevice);
7       uint8 *gputest_label;
8       cudaMalloc((void**)&gputest_label, COUNT_TEST * sizeof(uint8));
9       cudaMemcpy(gputest_label, test_label, COUNT_TEST * sizeof(uint8), ...
            cudaMemcpyHostToDevice);
10
11      int *gpuresult;
12      cudaMalloc((void**)&gpuresult, sizeof(int)*BLOCK_NUM);
13
14      testing <<< BLOCK_NUM, THREAD_NUM, THREAD_NUM * sizeof(int) >>> ...
            (gpulenet, gputest_data, gputest_label, COUNT_TEST, gpuresult);
15
16      int sum[BLOCK_NUM];
17      cudaMemcpy(&sum, gpuresult, sizeof(int) * BLOCK_NUM, ...
            cudaMemcpyDeviceToHost);
18      cudaFree(gpulenet);
19      cudaFree(gputest_data);
20      cudaFree(gputest_label);
21      cudaFree(gpuresult);
22
23  __global__ static void testing(LeNet5 *lenet, image *test_data, uint8 ...
        *test_label, int total_size, int *gpuresult)
24  {
```

```
25      //  int right = 0;
26      const int tid = threadIdx.x;
27      const int bid = blockIdx.x;
28      extern __shared__ int shared[];
29      shared[tid] = 0;
30      for (int i = bid * THREAD_NUM + tid; i < total_size; i += BLOCK_NUM * ...
            THREAD_NUM) {
31          uint8 l = test_label[i];
32          int p = Predict(lenet, test_data[i], 10);
33          shared[tid] += l == p;
34      }
35      __syncthreads();
36      if (tid == 0)
37      {
38          for (int i = 1; i < THREAD_NUM; i++)
39          {
40              shared[0] += shared[i];
41          }
42          gpuresult[bid] = shared[0];
43
44      }
45  }
```

2.2.2 Optimization on TrainBatch

a) Optimization Goal

- Speedup the TrainBatch process for it trains a batch of training data in parallel, which is suitable for CUDA application.

b) Optimization Process

- Because the stuct of Feature and LeNet5 are very large(72KB and 415KB separately), the sum of deltas is taken out of CUDA kernel. We get batch of Lenet5 data to be handled by CPU.
- Related code(Optimized)

```
1       LeNet5 *tgpulenet;
2       cudaMalloc((void**)&tgpulenet, sizeof(LeNet5));
3       image *gputrain_data;
4       cudaMalloc((void**)&gputrain_data, COUNT_TRAIN * sizeof(image));
5       uint8 *gputrain_label;
6       cudaMalloc((void**)&gputrain_label, COUNT_TRAIN * sizeof(uint8));
7       LeNet5 *gputrain;
8       cudaMalloc((void**)&gputrain, sizeof(LeNet5)*batch_size);
9       // Feature *gpufeatures;
10      // cudaMalloc((void**)&gpufeatures, sizeof(Feature)*batch_size);
11      // Feature *gpuerrors;
12      // cudaMalloc((void**)&gpuerrors, sizeof(Feature)*batch_size);
13      LeNet5 *ΔS= (LeNet5 *) malloc(sizeof(LeNet5)*batch_size);
14
15      LeNet5  ttt = { 0 };
16      cudaMemcpy(gputrain_data, train_data, COUNT_TRAIN * sizeof(image), ...
            cudaMemcpyHostToDevice);
17
```

```
18      cudaMemcpy(gputrain_label, train_label, COUNT_TRAIN * sizeof(uint8), ...
            cudaMemcpyHostToDevice);
19
20      for (int i = 0, percent = 0; i <= total_size - batch_size; i += ...
            batch_size)
21      {
22          cudaMemset(gputrain, 0, sizeof(LeNet5)*batch_size);
23          cudaMemcpy(tgpulenet, lenet, sizeof(LeNet5), cudaMemcpyHostToDevice);
24          TrainBatch << < trainBLOCK_NUM, trainTHREAD_NUM >> > (tgpulenet, ...
                gputrain_data + i, gputrain_label + i, batch_size, gputrain);
25          cudaMemcpy(&s, gputrain, sizeof(LeNet5) * batch_size, ...
                cudaMemcpyDeviceToHost);
26
27          double buffer[GETCOUNT(LeNet5)] = { 0 };
28          for (int mm = 0; mm < batch_size; ++mm)
29          {
30              ttt = &s[mm];
31              FOREACH(j, GETCOUNT(LeNet5))
32                  buffer[j] += ((double *)&ttt)[j];
33          }
34          FOREACH(nn, GETCOUNT(LeNet5))
35              ((double *)lenet)[nn] +=  buffer[nn] * ALPHA / batch_size;
36
37          if (i * 100 / total_size > percent)
38              printf("batchsize:%d\ttrain:%2d%%\n", batch_size, percent = i ...
                    * 100 / total_size);
39      }
40      cudaFree(tgpulenet);
41      cudaFree(gputrain_data);
42      cudaFree(gputrain_label);
43      cudaFree(gputrain);
44      free(&s);
45
46     __global__ void TrainBatch(LeNet5 *lenet, image *inputs, uint8 ...
          *labels, int batchSize, LeNet5 *gputrain )
47  {
48      const int tid = threadIdx.x;
49      const int bid = blockIdx.x;
50      const int num = bid*blockDim.x + tid;
51      Feature gpufeatures = { 0 };
52      Feature gpuerrors = { 0 };
53      gputrain[num] = { 0 };
54
55      load_input(&gpufeatures, inputs[num]);
56      forward(lenet, &gpufeatures, relu);
57      load_target(&gpufeatures, &gpuerrors, labels[num]);
58      backward(lenet, gputrain+num, &gpuerrors, &gpufeatures, relugrad);
59  }
```

3 Optimization Result

In this section, we introduce the optimization result of our project in OpenMp and CUDA. We optimize both the process of training and testing the LeNet-5 CNN. We run our program on GHC46 cluster machine. The speed up in training can save our time in training, so we can train more times of CNN to get better result. The speed up in testing can make the application more interactive and efficient. The details of result will show below.

3.1 OpenMP Optimization Result

From Table. 1, we can see the speed up by OpenMP for training LeNet-5 process with 60000 samples is 11.48x, and the speed up for testing LeNet-5 process with 10000 samples is 11.70x. So the average time for training one sample is 0.1768ms, and the average time for testing is 0.0584ms.

process	number of sample	sequential	OpenMP	Speed Up
training	60000	121.8046	10.6057	11.48x
testing	10000	6.8313	0.5840	11.70x

Table 1: LeNet Speed Up by OpenMP

3.2 CUDA Optimization Result

As shown in Table 2, the result of training and testing LeNet-5 with CUDA speedup. For a better precision, batch size is in the order of 300 around. It limits the thread number in TrainBatch and we can see the speedup of training is not very satisfying. The intermediate structure and return structure are very large, which only can be stored in the local memory and global memory. And the data copy between host and device is time consuming meanwhile. On the contrary, the speed up for testing process is very obvious and much better than the result gotten from OpenMP optimization.

process	number of sample	sequential	OpenMP	Speed Up
training	60000	121.8046	31.7278	3.838x
testing	10000	6.8313	0.1197	57.07x

Table 2: LeNet-5 Speed Up by CUDA

4 Conclusion

For this project, we understand the architecture of LeNet-5 CNN and applied the techniques we learned in the course to optimize the code. Our system gets more than 96% accuracy in digit recognition with 60000 samples for training and 10000 samples for testing.

Specifically, through optimization using OpenMP, we better understand the multi-thread acceleration and acquired useful experience from tackling problems, which gets above 11x speed up.

By rewriting LeNet-5 in CUDA, we learned how to better utilize GPU parallel computing resources, and have a better understanding of shared memory, with which gets 57x speed up in testing and 3.8x speed up in training.

The speed up of CNN is very important, as the CNN is applied in everywhere in our life. The speed up of CNN can make our lives more efficient and helpful. For example, in disease detection domain, if we can speed up 60x, before we need 60 hours, now we just need 1 hour to get the result. Time is life. The speed up can help us detect diseases quicker and cure the patients.

Thank you for your reading.

References

[1] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.

[2] Izotov, P. Yu, et al. "CUDA-enabled implementation of a neural network algorithm for handwritten digit recognition." Optical Memory & Neural Networks 20.2 (2011): 98-106.

[3] Jaderberg, Max, Andrea Vedaldi, and Andrew Zisserman. "Speeding up convolutional neural networks with low rank expansions." arXiv preprint arXiv:1405.3866 (2014).

[4] Tanomoto, Masakazu, et al. "A cgra-based approach for accelerating convolutional neural networks." Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on. IEEE, 2015.

[5] Alhamali, Abdulrahman, et al. "FPGA-Accelerated Hadoop Cluster for Deep Learning Computations." Data Mining Workshop (ICDMW), 2015 IEEE International Conference on. IEEE, 2015.

[6] fan-wenjie. LeNet-5 https://github.com/fan-wenjie/LeNet-5