CNN Optimizations

Zhengqi Dong, Yunlu Deng, Pingcheng Dong
{dong760, yldeng, pingchen }@bu.edu

1. Problem Statement

The code we are going to optimize is multilayer-perception-in-C[4]. The writer is the user "manoharmukku" on github. Here is the link and we also add it in the reference part: https://github.com/manoharmukku/multilayer-perceptron-in-c

1.1. What is MLP and our code structure

Multiple Perception (MLP) is a kind of mode of feedforward artificial neural network(ANN) in machine learning. MLP can be used in a lot of problems.

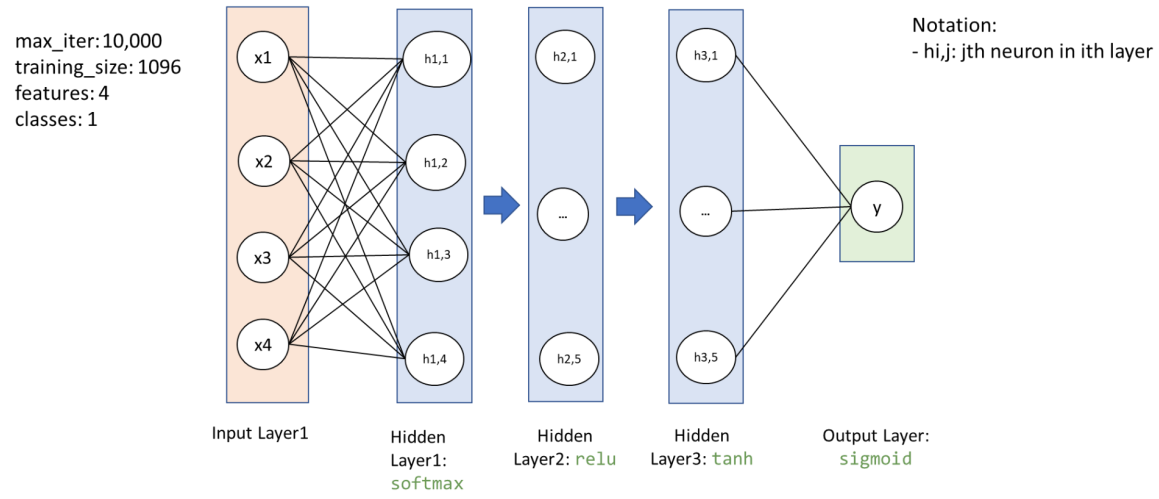

Figure 1: MLP and code instruction

As we can see in the figure 1, MLP is a fully connected class which consists of three kinds of layers of the nodes, input layer, hidden layerN(N=1,2,3, etc), output layer. Each layer has several nodes and these nodes are fully connected. MLP uses a supervised learning technique called backpropagation for training the model.[3] Two nodes will be connected by a nonlinear activation function and the code will these layers to calculate the weight. The weight is the final result of our training.
The code we are going to optimize is based on a single MLP model. By changing the input arguments, we can choose the specific way to train the model, such as the number of hidden

layers, each hidden layer size, the kind of nonlinear activation functions and so on. There is an example in figure 1 and the next is the table of the each argument meaning:

Program parameters explanation:

> Argument 0: Executable file name *Ex:* **./MLP**
>
> Argument 1: Number of hidden layers *Ex:* **3**
>
> Argument 2: Number of units in each hidden layer from left to right separated by comma (no spaces in-between) *Ex:* **4,5,5**
>
> Argument 3: Activation function of each hidden layer from left to right separated by comma (no spaces in-between) *Ex:* **softmax,relu,tanh**
>
> Argument 4: Number of units in output layer (Specify 1 for binary classification and k for k-class multi-class classification) *Ex:* **1**
>
> Argument 5: Output activation function *Ex:* **sigmoid**
>
> Argument 6: Learning rate parameter *Ex:* **0.01**
>
> Argument 7: Maximum number of iterations to run during training *Ex:* **10000**
>
> Argument 8: Path of the csv file containing the train dataset *Ex:* **data/data_train.csv**
>
> Argument 9: Number of rows in the train dataset (Number of samples) *Ex:* **1096**
>
> Argument 10: Number of columns in the train dataset (Number of input features + 1 (output variable)). The output variable should always be in the last column *Ex:* **5**
>
> Argument 11: Path of the csv file containing the test dataset *Ex:* **data/data_test.csv**
>
> Argument 12: Number of rows in the test dataset (Number of samples) *Ex:* **275**
>
> Argument 13: Number of columns in the test dataset (Number of input features + 1 (output variable)). The output variable should always be in the last column *Ex:* **5**

Figure 2: Input arguments[4]

For this project, we are going to optimze the project. The original code is consisted of 100% C language without any optimize method. So, it is possible to use several optimization method to shorten the training time, such as serial, parallel, GPU and so on. In this project, our group will try to use all these ways to achieve this goal. Here is the plan:

1.Single-core CPU optimization(serial): loop unrolling, blocking;

2.Multi-core CPU optimization(parallel): OpenMP, pthread, SIMD vectorization;

3.GPU optimization: Naive single and multiple CUDA block implementation, cache and constant memory, two strategy of tiled algorithms

1.2 Functions running time

Determine which function to be optimized first!
The first step of this project is to determine the optimization part. There is a good tool that can help us to do that — program profiling.
“Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.”[5]
“The flat profile shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which functions burn most of the cycles, it is stated concisely here. See The Flat Profile.”[6]
To understand where our program spent its time and on which pieces of code while it was executing, we need to profile the program. That is, we want to measure the time that each function has spent, and it further helps us to determine where the critical functions are, the pieces of code we should optimize with.
Here is the result of the Flat profiler:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
52.62	12.20	12.20	10960000	0.00	0.00	back_propagation
20.51	16.95	4.75	43840000	0.00	0.00	calculate_local_gradient
15.50	20.54	3.59	43840000	0.00	0.00	mat_mul
5.01	21.70	1.16	10960000	0.00	0.00	forward_propagation
1.36	22.02	0.32	10960000	0.00	0.00	d_relu
1.21	22.30	0.28	10960000	0.00	0.00	relu
0.86	22.50	0.20	10960000	0.00	0.00	softmax
0.82	22.69	0.19	10960000	0.00	0.00	tan_h
0.58	22.83	0.14	10960000	0.00	0.00	d_softmax
0.52	22.95	0.12	10960000	0.00	0.00	d_tanh
0.37	23.03	0.09	10960000	0.00	0.00	sigmoid
0.26	23.09	0.06	10000	0.00	0.00	randomly_shuffle
0.26	23.15	0.06	1	0.06	23.17	mlp_trainer
0.09	23.17	0.02	10960000	0.00	0.00	d_sigmoid
0.09	23.19	0.02	1	0.02	0.02	write_csv
0.02	23.20	0.01				identity
0.00	23.20	0.00	1100	0.00	0.00	mat_mul_classify
0.00	23.20	0.00	275	0.00	0.00	relu_classify
0.00	23.20	0.00	275	0.00	0.00	sigmoid_classify
0.00	23.20	0.00	275	0.00	0.00	softmax_classify
0.00	23.20	0.00	275	0.00	0.00	tan_h_classify
0.00	23.20	0.00	2	0.00	0.00	read_csv
0.00	23.20	0.00	1	0.00	0.00	initialize_weights
0.00	23.20	0.00	1	0.00	0.02	mlp_classifier

Figure 3: Profile result

From the above result, we see that our program spent 52.62% (12.20 sec) of time in back_propagation, and 20.51% (4.75 sec) on calculate_local_gradient, and 15.50% (3.59 sec) on mat_mul (We only consider the top3 for our project).

Flat Profiler only shows which functions get called and by how many times. With call graphs, we can go deeper, a more granular way to profile the program. We can understand the function dependency of a function, such as which functions are called within, and by how many times, and this will help us to eliminate the unnecessary functions that take a lot of time. Here is a snippet of call graph result:

```
69   index % time    self  children    called     name
70                                                 <spontaneous>
71   [1]    100.0    0.00   23.19                 main [1]
72                   0.06   23.11       1/1           mlp_trainer [2]
73                   0.00    0.02       1/1           mlp_classifier [16]
74                   0.00    0.00       2/2           read_csv [24]
75   -----------------------------------------------
76                   0.06   23.11       1/1           main [1]
77   [2]     99.9    0.06   23.11       1         mlp_trainer [2]
78                  12.20    5.34 10960000/10960000     back_propagation [3]
79                   1.16    4.35 10960000/10960000     forward_propagation [4]
80                   0.06    0.00   10000/10000       randomly_shuffle [14]
81                   0.00    0.00       1/1           initialize_weights [25]
82   -----------------------------------------------
83                  12.20    5.34 10960000/10960000     mlp_trainer [2]
84   [3]     75.6   12.20    5.34 10960000         back_propagation [3]
85                   4.75    0.59 43840000/43840000     calculate_local_gradient [5]
86   -----------------------------------------------
87                   1.16    4.35 10960000/10960000     mlp_trainer [2]
88   [4]     23.7    1.16    4.35 10960000         forward_propagation [4]
89                   3.59    0.00 43840000/43840000     mat_mul [6]
90                   0.28    0.00 10960000/10960000     relu [8]
91                   0.20    0.00 10960000/10960000     softmax [9]
92                   0.19    0.00 10960000/10960000     tan_h [10]
93                   0.09    0.00 10960000/10960000     sigmoid [13]
94   -----------------------------------------------
95                   4.75    0.59 43840000/43840000     back_propagation [3]
96   [5]     23.0    4.75    0.59 43840000         calculate_local_gradient [5]
97                   0.32    0.00 10960000/10960000     d_relu [7]
98                   0.14    0.00 10960000/10960000     d_softmax [11]
99                   0.12    0.00 10960000/10960000     d_tanh [12]
100                  0.02    0.00 10960000/10960000     d_sigmoid [15]
```

Figure 4: call graph result

With this call graph, we can see that back_propagration is the one that takes more of time: itself takes 12.20 sec, and 5.34 sec by its child (or callee). The second function that takes most of the time is forward_propagation, which takes 1.16 sec by itself, and 4.35 sec by its child.

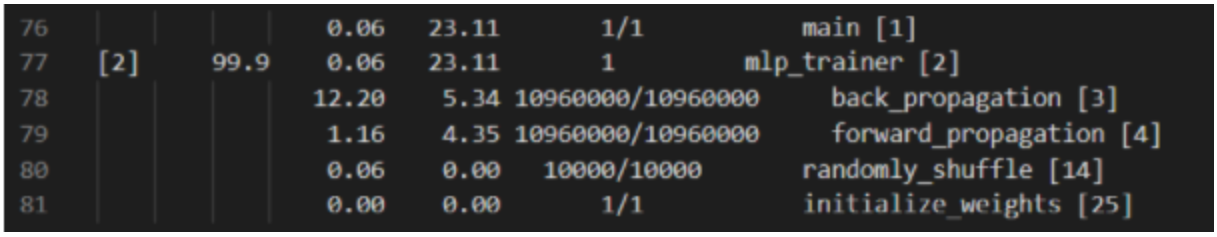

Figure 5: call graph part1

By looking at the child function of back_propagration, we see that calculate_local_gradient is the one that takes most of the time, 4.75 sec by itself, and 0.59 sec by its child.

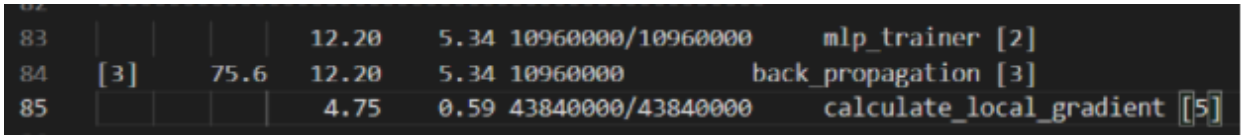

Figure 6: call graph part2

By looking at the child function of forward_propagation, we see that mat_mul is the one that takes most of the time, 3.59 sec by itself, and 0.00 sec by its child.

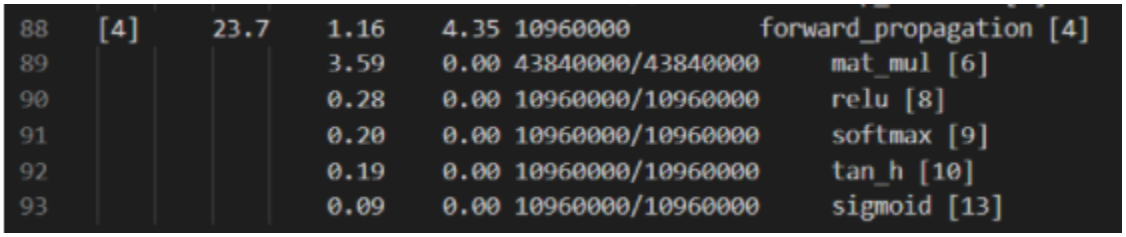

Figure 7: call graph part3

Therefore, by ranking the time cost of each function in descending order, we will mainly focus on three functions: back_propagation(52.62%), calculate_local_gradient(20.51%), mat_mul (15.50%).

Further analyzation:

Analyzation in back_propagation.c

Based on the analyzation above, in order to get more detailed information, we insert several time record functions into this file, like:

```
clock_gettime(CLOCK_REALTIME, &time_start);
for (i = 0; i < n_layers-1; i++)
    weight_correction[i] = (double**)calloc(layer_sizes[i]+1, sizeof(double*));
clock_gettime(CLOCK_REALTIME, &time_stop);
time_stamp[0] = time_stamp[0] + interval(time_start, time_stop);
```

Figure 8: time record example

After compiling and running, here is the result:

```
PS C:\Users\111\multilayer-perceptron-in-c> ./MLP 3 4,5,5 softmax,relu,tanh 1 sigmoid 0.01 1000 data/data_train.csv 1096 5 data/data_test.csv 275 5
Training:
---------
func0       func1       func2       func3       func4       func5       func6       func7
0.252509    2.130107    0.255550    0.511152    0.059468    0.396791    0.529015    0.812962

Done.
```

Figure 9: result

So that our main objective is to optimize the function:

func1:

```
clock_gettime(CLOCK_REALTIME, &time_start);
for (i = 0; i < n_layers-1; i++)
    for (j = 0; j < layer_sizes[i]+1; j++)
        weight_correction[i][j] = (double*)calloc(layer_sizes[i+1], sizeof(double));
clock_gettime(CLOCK_REALTIME, &time_stop);
time_stamp[1] = time_stamp[1] + interval(time_start, time_stop);
```

Figure 10: function1

func3:

```
clock_gettime(CLOCK_REALTIME, &time_start);
calculate_local_gradient(param, n_layers-1, n_layers, layer_sizes, layer_inputs, layer_outputs, expected_output, local_gradient);
clock_gettime(CLOCK_REALTIME, &time_stop);
time_stamp[3] = time_stamp[3] + interval(time_start, time_stop);
```

Figure 11: function2

func5 and func6:

```
int k;
clock_gettime(CLOCK_REALTIME, &time_start);
for (i = n_layers-2; i >= 1; i--) {
    //calculate_local_gradient(param, i, n_layers, layer_sizes, layer_inputs, layer_outputs, expected_output, local_gradient);

    for (j = 0; j < layer_sizes[i]; j++)
        for (k = 0; k < layer_sizes[i-1]+1; k++)
            weight_correction[i-1][k][j] = (param->learning_rate) * local_gradient[i][j] * layer_outputs[i-1][k];
}
clock_gettime(CLOCK_REALTIME, &time_stop);
time_stamp[5] = time_stamp[5] + interval(time_start, time_stop);

/*----------------- Update the weights -----------------------------------*/
clock_gettime(CLOCK_REALTIME, &time_start);
for (i = 0; i < n_layers-1; i++) {
    for (j = 0; j < layer_sizes[i]+1; j++) {
        for (k = 0; k < layer_sizes[i+1]; k++) {
            param->weight[i][j][k] -= weight_correction[i][j][k];
        }
    }
}
clock_gettime(CLOCK_REALTIME, &time_stop);
time_stamp[6] = time_stamp[6] + interval(time_start, time_stop);
```

Figure 12: function3

2. Single-Core CPU Optimization

Code directory:
https://github.com/drago1234/EC527_CNNs_Optimization/tree/main/multilayer-perceptron-in-c-single-core

2.1 code can be optimized

From the previous time analysis part, we have found that the most time consuming part of the program is in the trainer and back_propagation part. After running the time test for each part of the back_progation, we found that there are one double for loop and two triple for loops, which cost most of the time. Hence, we decided to optimize these three for loops.

```
/*---------- Calculate weight corrections for all layers' weights ------------------*/
// Weight correction for the output layer
calculate_local_gradient(param, n_layers-1, n_layers, layer_sizes, layer_inputs, layer_outputs, expected_output, local_gradient);
for (i = 0; i < param->output_layer_size; i++)
    for (j = 0; j < layer_sizes[n_layers-2]+1; j++)
        weight_correction[n_layers-2][j][i] = (param->learning_rate) * local_gradient[n_layers-1][i] * layer_outputs[n_layers-2][j];

// Weight correction for the hidden layers
int k;
for (i = n_layers-2; i >= 1; i--) {
    calculate_local_gradient(param, i, n_layers, layer_sizes, layer_inputs, layer_outputs, expected_output, local_gradient);

    for (j = 0; j < layer_sizes[i]; j++)
        for (k = 0; k < layer_sizes[i-1]+1; k++)
            weight_correction[i-1][k][j] = (param->learning_rate) * local_gradient[i][j] * layer_outputs[i-1][k];
}

/*----------------- Update the weights -----------------------------------*/
for (i = 0; i < n_layers-1; i++) {
    for (j = 0; j < layer_sizes[i]+1; j++) {
        for (k = 0; k < layer_sizes[i+1]; k++) {
            param->weight[i][j][k] -= weight_correction[i][j][k];
        }
    }
}
```

Figure 13: Single-code optimization functions

2.2 ways to optimize

We decided to try to unroll the for loop and try to use the openmp.

2.3 unroll the for loops

We have tried to unroll the loop by 2, 4 and 8. Then we run the speed test at the size of 10000 and 100000. We got some positive results. We found that after unrolling by 2, the cost of time will decrease significantly, and after unrolling by 4, there will be some slight decrease in time consuming, but after unrolling by 8, the cost of time is more than unrolling by 4.

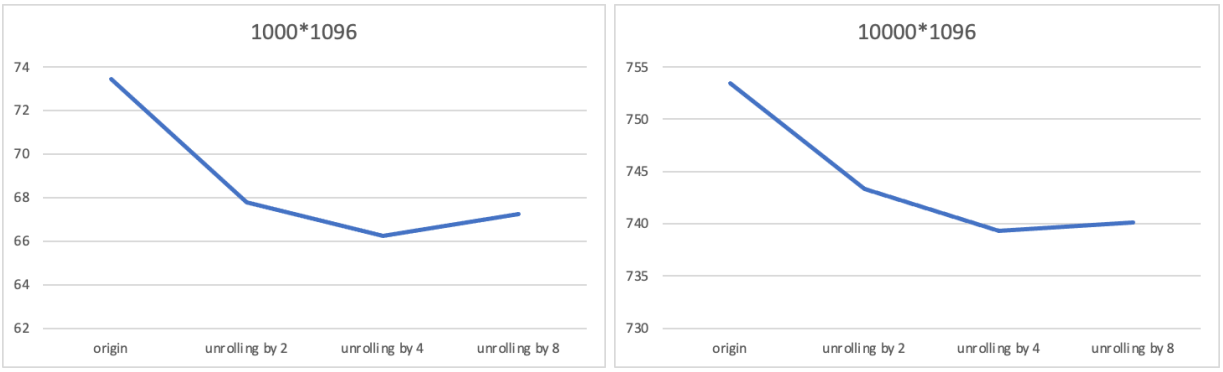

Figure 14: Single-core unroll result

2.4 try to use openmp

As we can see, there are many for loops in the back_propagartion, so we decided to try openmp for all of the for loops. The platform we tried is i5-8279U which has 4 cores and 8 threads with 2.4Ghz. However, we found that it is not satisfying for the results of openmp.

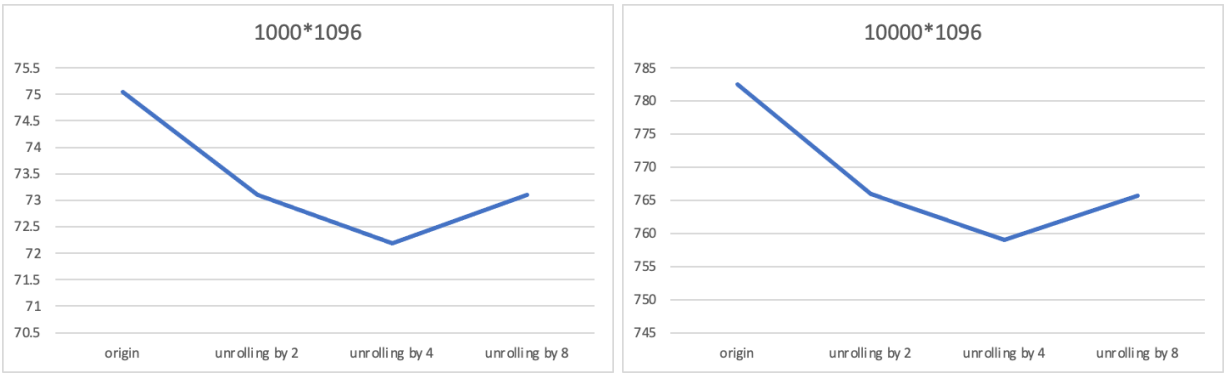

Figure 15: Single-core unroll-OpenMP result

2.5 conclusion

After we have tried the way above, we found that unrolling the for loops is an effective way to optimize the back_propation, but the openmp is not suitable for this. We have figured that the iteration in the back_propation is decided by the layer size which is not as large enough to overcome the extra time of openmp to create threads and control threads. Therefore, we decided to use multiple threads in the trainer which have large iterations.

3. Multi-Core CPU Optimization

3.1 Optimization code

Pthread:
https://github.com/drago1234/EC527_CNNs_Optimization/tree/main/multilayer-perceptron-in-c-openmp

OpenMP:
https://github.com/drago1234/EC527_CNNs_Optimization/tree/main/multilayer-perceptron-in-c-pthread

3.2 Choose Optimization part

In part 1.2, we have discussed the parts of code this project would like to optimize. However, not all parts of code optimization will work for a specific optimization method. For example, the multi-core CPU optimization part includes OpenMP and Pthread. The key precondition for these two methods is the assignment of each method should be large enough. Otherwise, the cost of threads can not be compensated by the time saving of optimization methods. For example, a failure version of pthread optimization chooses to insert the pthread into the function mat_mul which has been mentioned in figure 7. The result shows that the execution time will be 1000 times slower.
The organization of original code likes running a small step with a large number of times. So the main loop would be the best part for me to insert the threads. In this way, each thread can work on a large number of assignments. The main loop:

```
for (j = 0; j < param_in->n_iterations_max; j++) {
    //printf("Iteration %d of %d(max)\r", j+1, param_in->n_iterations_max);
    // Randomly shuffle the data
    randomly_shuffle(indices, param_in->train_sample_size/Num_threads);

    for (k = 0; k < param_in->train_sample_size/Num_threads; k++) {
        training_example = indices[k];
        // Perform forward propagation on the jth training example
        //printf("Iteration %d of %d(max), %d, %d\r", k, param_in->train_sample_size, T_ID, j);
        forward_propagation(param_in, training_example, n_layer_in, n_layer_size_in, layer_inputs, layer_outputs, weight_in[T_ID]);
        //printf("Iteration %d of %d(max), %d\n", k, param_in->train_sample_size, T_ID);
        // Calculate the error

        // Perform back propagation and update weights
        back_propagation(param_in, training_example, n_layer_in, n_layer_size_in, layer_inputs, TestTimes, weight_in[T_ID]);
    }
}
```

Figure 16: Multi-core optimization function

For this double loop, under the condition of figure 1, the number of innermost code will be execute 10000*1096 times. It is large enough to use OpenMP and Pthread here.

3.3 Dependence

Dependence is the key factor which depends if the code can be paralized. In order to use OpenMP and Pthread to optimize the code, it is necessary to make sure that the paralyzation will not influence the final result. That means, the variable in one thread which is going to be read will not be written by another thread. For example, thread 1 writes the value 1 into the variable A and is going to read it 10ms later. However, during this 10ms, thread 2 may write the value 2 into the variable Also that when thread 1 reads A, it will get a wrong value.
In the original code, there is a triple array called “para->weight” that breaks the dependence rule. As it shows in these two figures:

```
for (i = 0; i < n_layers-1; i++) {
    for (j = 0; j < layer_sizes[i]+1; j++) {
        for (k = 0; k < layer_sizes[i+1]; k++) {
            param->weight[i][j][k] -= weight_correction[i][j][k];
        }
    }
}
```

Figure 17: Para->weight write in function back_propagation

```
switch (param->hidden_activation_functions[layer_no-1]) {
    case 1: // identity
        d_identity(layer_sizes[layer_no], layer_inputs[layer_no], layer_outputs[layer_no], layer_derivatives[layer_no]);

        for (i = 0; i < layer_sizes[layer_no]; i++) {
            double error = 0.0;
            for (j = 0; j < layer_sizes[layer_no+1]; j++)
                error += local_gradient[layer_no+1][j] * param->weight[layer_no][i][j];

            local_gradient[layer_no][i] = error * layer_derivatives[layer_no][i];
        }

        break;
```

Figure 18: Para->weight read in function back_propagation

As we can see in figure 17 and 18, the variable “para->weight” breaks the dependence rule and it saves the result of weight correction after the loop. So I use another variable, a four dimensional array called “weight-in” which can be seen in figure 16 in order to save the weight correction result of each thread and merge them together at the end.

3.4 Result

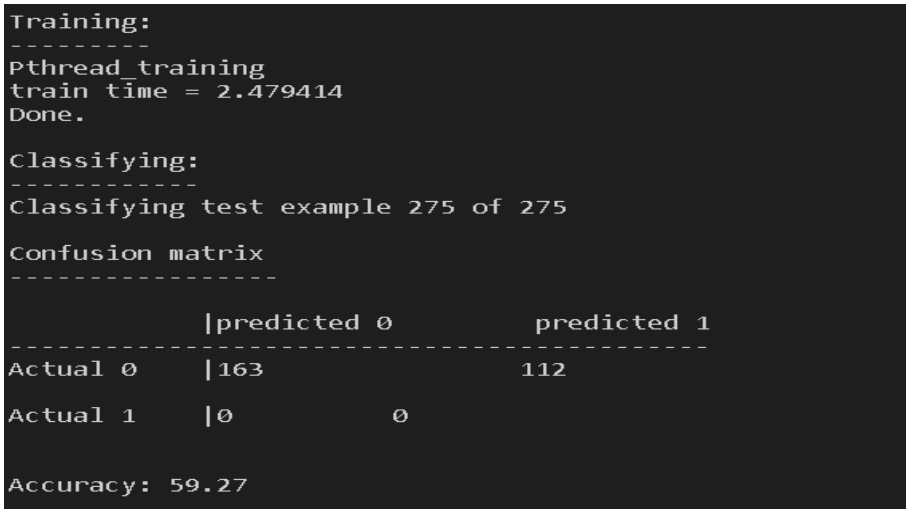

Figure 18: Output example

	Origin_src	Pthread_4	Pthread_8	Pthread_16	OpenMP_3	OpenMP_4	OpenMP_5
1000*1096_Time(s)	7.53	2.48	1.64	1.65	2.88	2.4	1.98
10000*1096_Time(s)	75.04	24.96	20.34	17.77	26.59	23.46	21.08
100000*1096_Time(s)	758.56	249.49	183.97	188.49	295.87	283.51	287.03
1000*1096Optimize_rate(%)	100	303.62	459.14	456.36	261.45	313.75	380.30
10000*1096Optimize_rate(%)	100	300.64	368.92	422.28	282.21	319.86	355.97
100000*1096Optimize_rate(%)	100	304.04	412.32	402.44	256.38	267.56	264.27

Table 1: Final result table

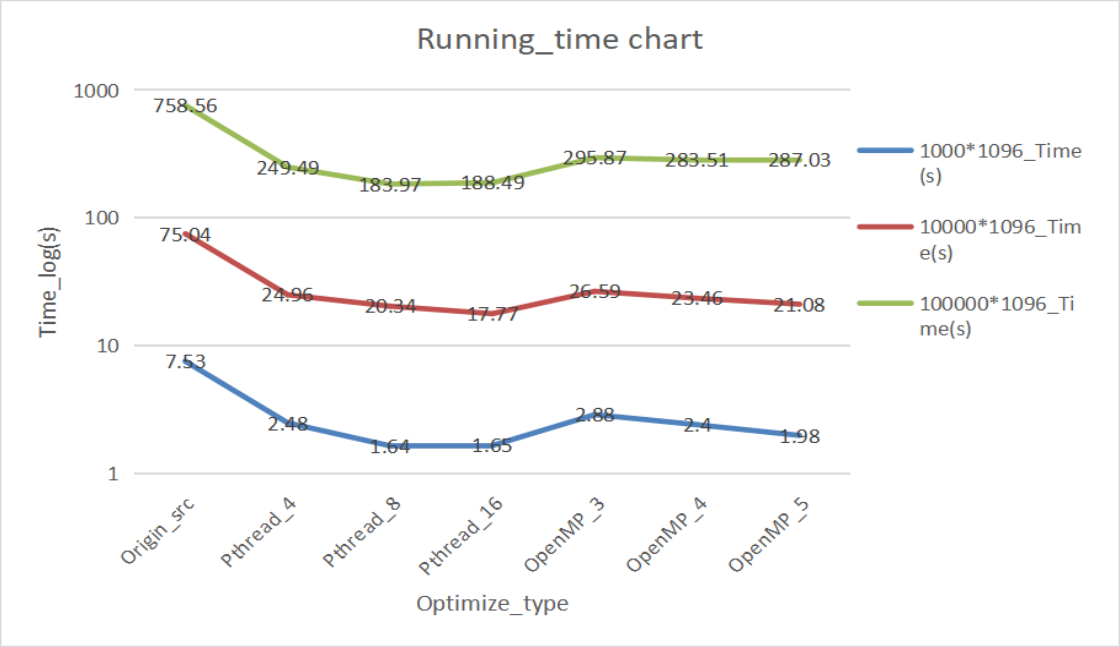

Figure 19: Running time optimization plot

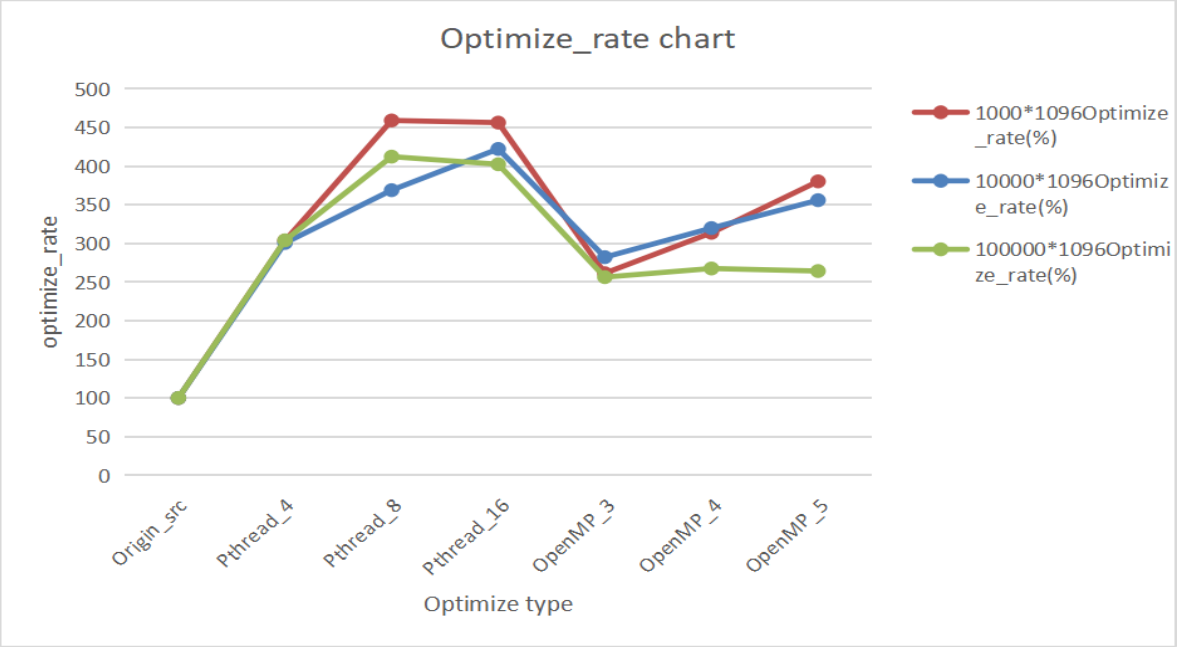

Figure 20: Optimization rate plot

The figure 18 shows an example of the output of the code which includes the optimization method, training time, result and accuracy. The project aim is to try our best to shorten the time as much as possible. The table 1 shows the result of the test we have done. As we can see in table 1, the pthread can optimize the code up to 456.36% better than the original code and the OpenMP can do so up to 380.30%. The optimization has made a huge improvement.

4. GPU Optimization (Zhengqi Dong)

A short intro to 1D CNNs optimization with CUDA

As we have seen previously, the MLP is a simple example of a fully connected neural network, where every neuron in one layer has a connection to every neuron in the next layer. This type of network is “structure agnostics”, a general-purpose connection that makes no assumption about the feature in the input data. However, as we see in many research papers [1-2, 7] have shown that this type of network can be very memory (weight) and computationally expensive to train, and a convolutional neural network(CNN) is often used in feature extraction. Therefore, instead of optimizing the original vector-matrix multiplication, we will generalize to a convolution operation that can take the various sizes of mask_width as input to perform the computation. For simplicity we will start to optimize the performance of 1D convolutions, and the CPU version implementation of baseline CNN has shown below in Code1:

Code1: 1D CNN CPU baseline code

```
void conv_1D(float *N, float *M, float *P, int mask_width, int
N_rowlen){
      int i;
      float Pvalue;
      int halo_width = (mask_width - 1) / 2;
      int left_end = halo_width; int right_end = N_rowlen-halo_width;
      for (i = halo_width; i < N_rowlen-halo_width; i++){
            Pvalue = 0;
            for (int j = 0; j < mask_width; j++){
                  Pvalue += N[i - halo_width + j] * M[j];
            }
            P[i] = Pvalue;
      }
}
```

The idea of the convolution is pretty simple, where each output data element is a weighted sum of a collection of neighboring input elements, and the weights that are used in weighted sum calculation are defined by the input mask array, which commonly referred to as the convolution kernel or convolution masks. In this function `conv_1D`, we have an input array float *N, mask array float *M, and an output array float *M that has the same length `N_rowlen` as the input array N. For avoiding the boundary condition where the calculation involves missing input elements, 0 will be used to for those filled-in elements. This technique is often known as padding, and those missing elements are typically referred to as “ghost cells” or “halo cells” in literature. For example in the Figure 21 below, the calculation of P[1] will be:

`P[1] = N[0]*M[0]+N[1]*M[1]+N[2]*M[2]=1.8`

As we can see the calculation will involve one missing input element that filled with zero, and those missing element that circled with dash-line will be referred to as “halo cells”. The width of halo_cell can be formulated as `(mask_width - 1) / 2` or `ceil(mask_width / 2)`, where the length of mask_width is assumed to be odds number.

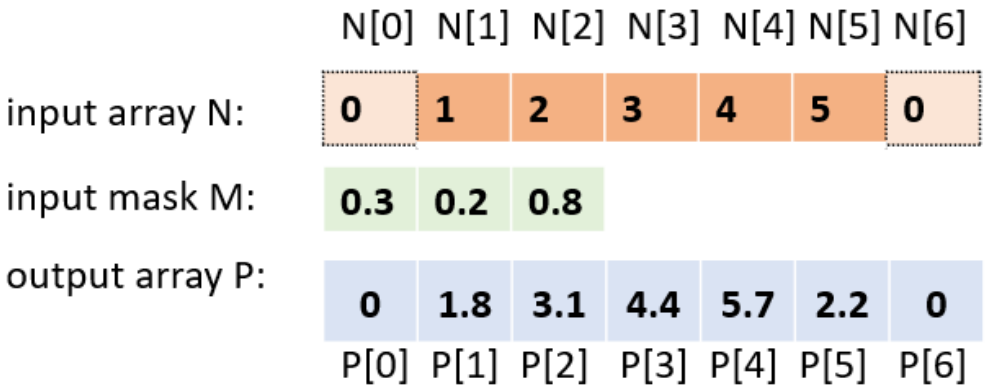

Figure 21: 1D conv boundary case

After we have defined the boundary position, the right end and left end, of the output array, we then can simply just use for loop to compute the weighted sum for each output element P[i]. As

we can see that each innermost loop involves one floating multiplication and addition, and two unique reads to the global memory, so the Arithmetic Intensity(AI), or the ratio of floating-point arithmetic operation per byte of global memory access, is about 1 in the kernel function, which is a very small portion of the peak performance.

Based on previous analysis, we observed that the calculation of output array P can be decomposed to multiple individual isolated computations, and this makes the convolution operation to be an ideal problem for parallel computing with CUDA. In general speaking, we can simply just map the computation involved for each output element to each CUDA thread, and each CUDA thread will perform the convolution operation accordingly. The baseline GPU implementation of convolution operation is shown below:

Code2: 1D CNN GPU Baseline code

```
__global__ void cuda_conv_1D_multi_block(float *N, float *M, float
*P, int mask_width, int N_rowlen){
     int i = blockIdx.x * blockDim.x + threadIdx.x; // i is [0,
P_ARR_LEN-1]
     float Pvalue = 0;
     int halo_width = (mask_width - 1) / 2; // assume mask_width is
odd number
     if (i < halo_width || i > (N_rowlen - 1 - halo_width)) return;
// 1 off for idx
     for (int j = 0; j < mask_width; j++){
          Pvalue += N[i - halo_width + j] * M[j];
     }
     P[i] = Pvalue;
}
```

1D CNN with cache and constant memory

The AI for the kernel function is still one. However, if we look at the code closer, we can observe several properties for the mask array: 1) The mask array M that is used in convolution operation is fairly small compared to the input array length; 2) The contents of the mask array M does not change throughout the whole execution; 3) The execution order that we need to access mask array is the same for each computation of output array P[i], or even better, that actual computation order for each our array P[i] can be different. With all those properties mentioned, we can see that the mask array is an excellent candidate for constant memory. Therefore, we can optimize the performance by using the constant memory, as shown in Code 3 below:

Code 3: 1D CNN with constant memory

```
#define MASK_WIDTH 3  // array size for mask (M)
__constant__ float d_mask_constant[MASK_WIDTH];
```

```
__global__ void
cuda_conv_1D_multi_block_with_mask_in_constant_memory(float *N, float
*P, int mask_width, int N_rowlen){
      int i = blockIdx.x * blockDim.x + threadIdx.x; // i is [0,
P_ARR_LEN-1]
      float Pvalue = 0;
      //Skip halo cells
      if (i < halo_width || i > (N_rowlen - 1 - halo_width)) return;
      for (int j = 0; j < mask_width; j++){
            Pvalue += N[i - halo_width + j] * d_mask_constant[j];
      }
      P[i] = Pvalue;
}
```

Now, by putting the mask array into constant memory, we reduce the number of global memory access for each iteration to one, so the ratio of floating-point arithmetic to memory access has doubled to two. The result that we compute for Code 1 and 2 has been shown in Table 2 and 3:

rowlen	CPU(msec)	GPU(msec)	Speedup
1000	0.0600	0.5790	0.1036
100,000	2.5730	1.3468	1.9105
10,000,000	119.0000	33.7104	3.5301
1,000,000,000	10150.0000	3327.1500	3.0507

Table 2: Benchmark Performance for Code 2(GPU baseline)

rowlen	CPU(msec)	GPU(msec)	Speedup
1000	0.059	0.5508	0.1071169
100,000	2.513	1.372	1.8316327
10,000,000	124.8	34.298	3.6386961
1,000,000,000	9804.3121	3110.98877	3.15151

Table 3: Benchmark Performance for Code 3(constant memory)

Tailed Algorithm

The above results were ran in SCC with NVIDIA Tesla V100-SMX2. We can see that the performance has increased slightly, and I think this is because we hit the memory bandwidth. Therefore, in order to handle this issue, we need to find a way to reduce the parallel algorithm. A good approach is to use the tiled algorithm, where we have all threads within a block

collaboratively to load the input element into a on-chip memory(e.g., shared memory in GPU), and then those corresponding output element within a block can simply just reading the content of input array from shared memory rather than reading from global memory each time.

There are two strategies to implemented tailed algorithm:

Strategy 1:

After we determined the size of the input array length and the number of threads per block, we can then partition the data and computation based on the block or tile, and the number of block for the computation is determined with the following formula:

`int dimGrid = (P_ARR_LEN + THREADS - 1) / THREADS;`

This equivalent to "`int dimGrid = ceil(P_ARR_LEN / NUM_THREADS_PER_BLOCK)`", which will make sure that we only have enough space for all the elements in the output array. Then all the threads within a block/tile will work collaboratively to load the data from the input array, and some threads might do extra work for those ghost cells. A schematic diagram is shown in Figure 22, and the corresponding code is shown in Code 4.

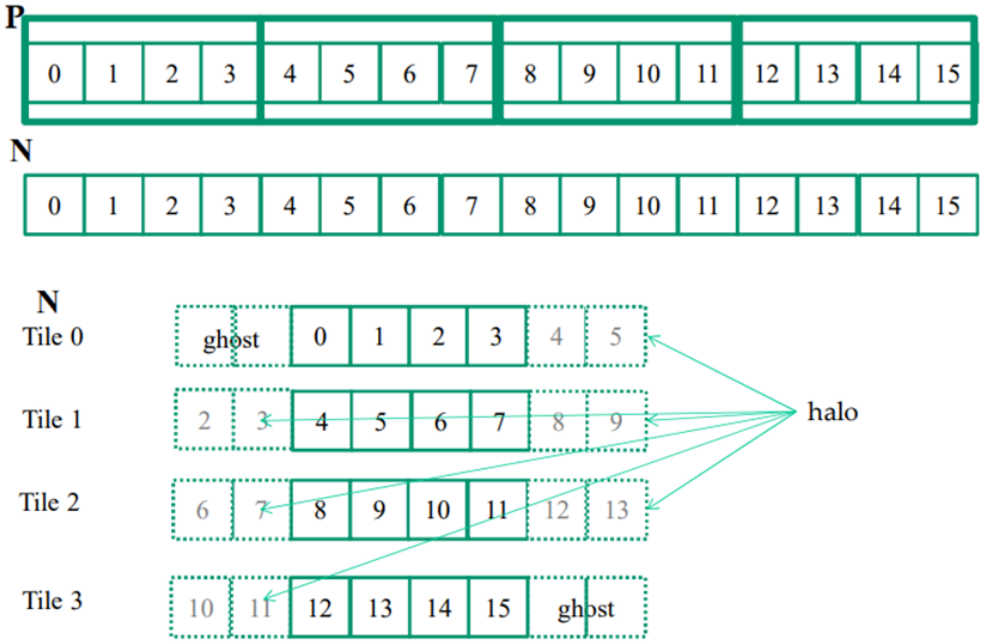

Figure 22: Stragegy 1 of 1D CNN tiled algorithm [8]

Code 4: 1D CNNs with tailed Algorithm Strategy 1

```
...[Host code]
define NUM_THREADS_PER_BLOCK 16
int GRID = (P_ARR_LEN + NUM_THREADS_PER_BLOCK - 1) / THREADS;
size_t SHMEM = (NUM_THREADS_PER_BLOCK + HALO_CELL*2) * sizeof(int);
cuda_conv_1D_tiled_and_shared_memory_kernel<<<GRID,
NUM_THREADS_PER_BLOCK, SHMEM>>>(d_input, d_output_data, MASK_WIDTH,
N_ARR_LEN);
...
// Strategy 1:
```

```
__global__ void cuda_conv_1D_tiled_strategy1(float *N, float *P, int
mask_width, int N_rowlen){
      int tid = blockIdx.x * blockDim.x + threadIdx.x;
      extern __shared__ int s_array[];

      int halo_width = (mask_width - 1) / 2;     // assume mask_width
is odd number
      // Loading first set of data into shared memory, starting with
halo cell
      s_array[threadIdx.x] = N[tid];
      // Maximum Size of the shared memory array
      int n_padded = blockDim.x + 2 * halo_width;
      // Loading second set of data into shared memory, starting from
offset
      int s_offset = threadIdx.x + blockDim.x;
      // Global offset for the input in DRAM
      int g_offset = tid + blockDim.x;
      if (s_offset < n_padded) {
        s_array[s_offset] = N[g_offset];
      }

      __syncthreads();
      float Pvalue = 0;
      for (int j = 0; j < mask_width; j++){
            Pvalue += s_array[threadIdx.x + j] * d_mask_constant[j];
      }
      if (tid >= halo_width || tid < (N_rowlen -  halo_width)){
            P[tid+halo_width] = Pvalue;
      }
}
```

Strategy 2:

The idea is similar to strategy 1. In stead of manually handling those halo cells for two set of loading, we will just directly loading those halo cell from global memory. A simple example is shown in Figure 23, and the code that I implemented for strategy 2 is shown in Code 5:

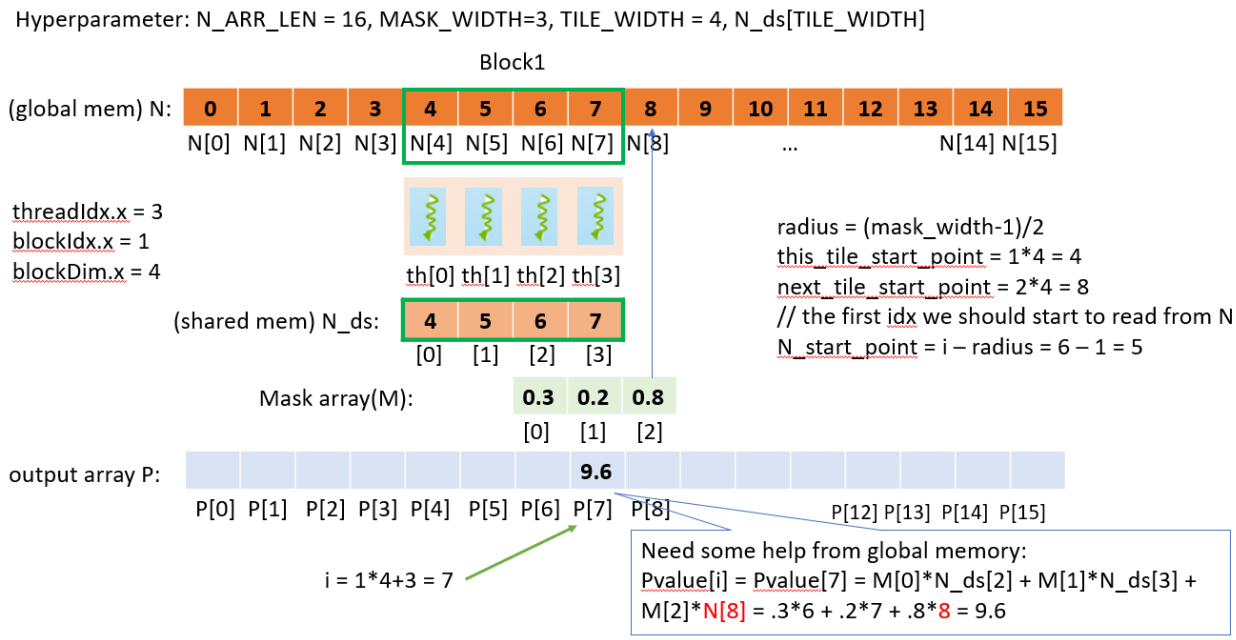

Figure 23: Stragegy 2 of 1D CNN tiled algorithm

Code 5: 1D CNNs with tailed Algorithm Strategy 2

```
// Strategy 2:
#define TILE_WIDTH 4
__global__ void cuda_conv_1D_tiled_and_shared_memory_kernel2(float
*N, float *P, int mask_width, int N_rowlen){
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      // Instantiate shared array s_array
      __shared__ float s_array[TILE_WIDTH];

      // Load data with corresponding idx from N
      s_array[threadIdx.x] = N[i];

      // Making sure all threads have finished loading data into
shared memory.
      __syncthreads();

      int halo_width = (mask_width - 1) / 2;     // assume mask_width
is odd number
      if (i < halo_width || i > (N_rowlen - 1 - halo_width)) return;
// 1 off for idx

      int this_tile_start_point = blockIdx.x * blockDim.x;
      int next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
      int N_start_point = i - halo_width;              // the first
idx we should start to read from N
```

```
      float Pvalue = 0;
      /* Go through each item in mask. If the corresponding item
needed from input array is in ghost cell, then we will read it from
global memory, otherwise, we will read it from shared memory! */
      for (int j = 0; j < mask_width; j++){
            int N_index = N_start_point + j;      //    the
corresponding idx of M[j] for N[N_index]
            // Check the boundary
            if (N_index >= 0 && N_index < N_rowlen){
                  // Decide whether should read from global memory or
shared memory?
                  int reading_from_s_array = ((N_index >=
this_tile_start_point) && (N_index < next_tile_start_point));
                  if (reading_from_s_array){
                        Pvalue += s_array[threadIdx.x - halo_width + j]
* d_mask_constant[j];
                  }else {
                        Pvalue += N[N_index] * d_mask_constant[j];
                  }
            }
      }
      P[i] = Pvalue;
}
```

Result of Strategy 1 vs Strategy 2:

rowlen	CPU(msec)	GPU(msec)	Speedup
1000	0.00205	0.4489	0.0045667
100,000	1.604	0.8683	1.8472878
10,000,000	148.706	51.3228	2.8974647
1,000,000,000	20518.374	6361.1801	3.2255609

Table 4: Benchmark Performance for Code 4(constant memory)

rowlen	CPU(msec)	GPU(msec)	Speedup
1000	0.01804	0.463392	0.0389303
100,000	2.4873	1.288352	1.9306059
10,000,000	116.99418	36.04224	3.2460297

1,000,000,000	9834.4849	3451.381348	2.8494344

Table 5: Benchmark Performance for Code 5(constant memory)

Note: The result in Table 2-4 and Table 5 were ran twice at different time in SCC, but same GPU (Tesla V100-SXM2) and CPU(Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz) architecture were used.

Conclusion:

By splitting the works into numerous CUDA threads, we did make great improvements (3.5x on an array with length 10,000,000 and 3.05x on 1,000,000,000) compared with the performance of the CPU baseline. In addition, we tried putting mask array into constant memory to reduce the memory access latency, and this help us to improve the performance by 2-3% but not dramatically. In the end, we tried to apply the tiled algorithm by managing all threads within a block to load the corresponding input elements collaboratively. There are two strategies were conducted to implement the tiled algorithm for 1D CNNs. However, the result is not as great as we expected. I think the main reason for that might be related to the overhead for transferring data from input array in global memory to shared memory and the waiting time for synchronization of all threads before performing the computation for the output array.

Note: All the performances evaluated above have been verified with the correct results that were produced from the CPU, and they are all being computed correctly. All the codes are stored in the file `test_1d_conv.cu`, and can be compiled easily by calling `make`.

Compilation Instruction:

By running the following command to compile the CUDA file:

```
nvcc -g -G -O1 -pg test_1d_conv.cu -o test_1d_conv
```

Note: -g is for debugging, and -O1 is for optimization, and -pg allows us to use Linux command gprof, a program profiling utility, with the command, "`gprof ./test_1d_conv gmon.out > analysis.txt`"

And then type the following command to run the program (Arguments are optional):

```
./test_1d_conv_no_padding [N_ARR_LEN] [NUM_THREADS_PER_BLOCK]
[task_id]
```

- argv[0]: `task_id`, which task to run (cuda_conv_1D_single_block - 1, cuda_conv_1D_multi_block - 2, multi_block_with_mask_in_constant_memory - 3, tiled algorithm with Strategy 1 - 4, tiled algorithm with strategy 2 - 5)
- argv[1]: `N_ARR_LEN`, length of input array N, with 1024 as default
- argv[2]: `NUM_THREADS_PER_BLOCK`, number of threads per block, with 16 as default. (It's better to use a number that is multiple of NUM_THREADS_PER_BLOCK, so we won't see many unmatched errors for the last block)

5. Reference

[1] Lawrence, Steve, et al. "Face recognition: A convolutional neural-network approach." *IEEE transactions on neural networks* 8.1 (1997): 98-113.
[2] Li, Zhiyuan, Yi Zhang, and Sanjeev Arora. "Why are convolutional nets more sample-efficient than fully-connected nets?." *arXiv preprint arXiv:2010.08515* (2020).
[3] Wikipedia. “Multilayer perceptron”. https://en.wikipedia.org/wiki/Multilayer_perceptron
[4] manoharmukku. “multilayer-perception-in-C”. https://github.com/manoharmukku/multilayer-perceptron-in-c
[5] Wikipedia. https://en.wikipedia.org/wiki/Profiling_(computer_programming)
[6] “Introduction to Profiling”. https://sourceware.org/binutils/docs/gprof/Introduction.html#Introduction
[7] Zuhair, A. and Hassani, H., 2021. Comparing the Accuracy of Deep Neural Networks (DNN) and Convolutional Neural Network (CNN) in Music Genre Recognition (MGR): Experiments on Kurdish Music. arXiv preprint arXiv:2111.11063., https://arxiv.org/pdf/2111.11063.pdf
[8] Courtesy: Kirk, David B., and W. Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan kaufmann, 2016.