EC527 CNN Optimization

Boston University
Zhengqi Dong(M.S. RAS)
Pingcheng Dong
Yunlu Deng

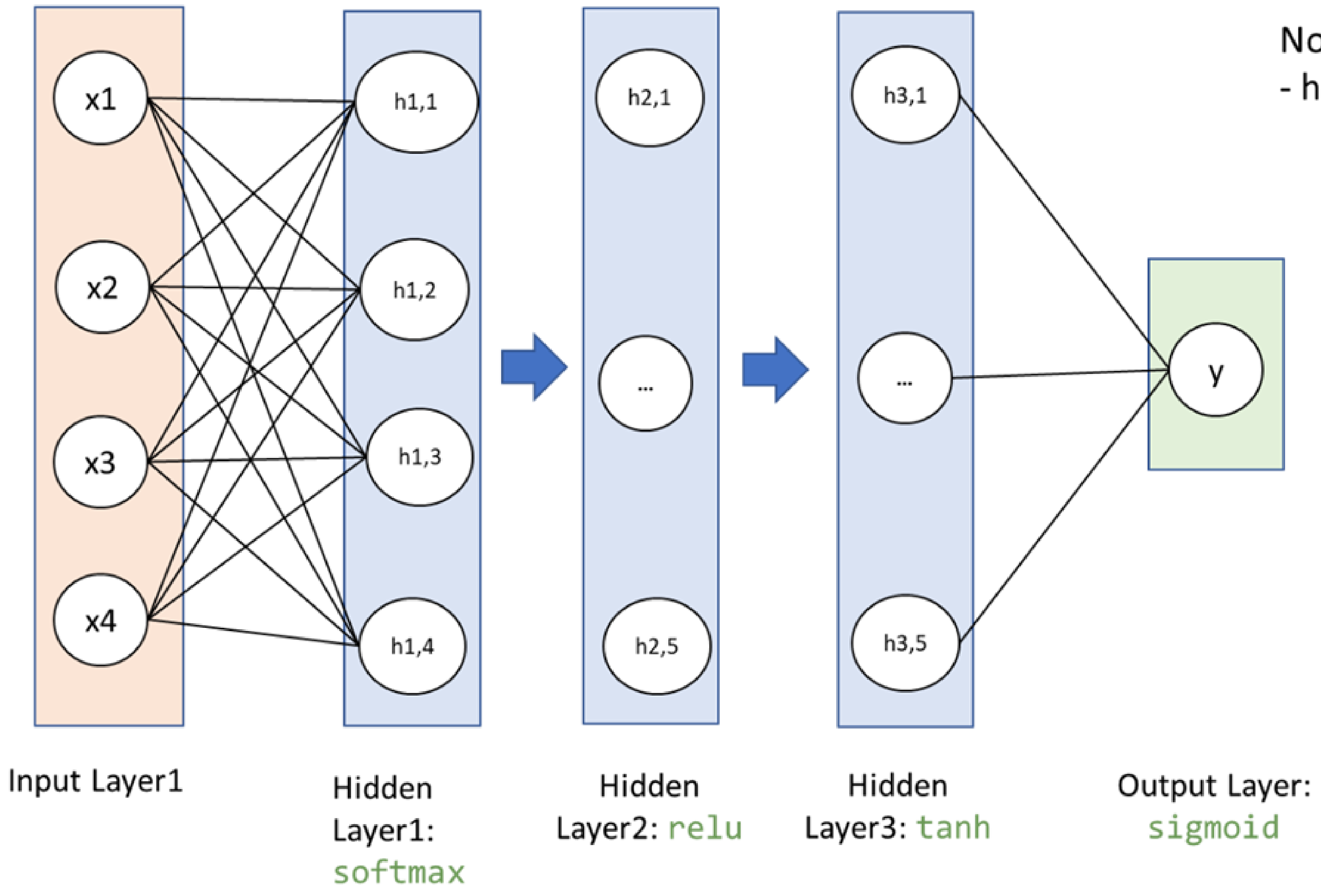
05/01/2022

Background

Problem overview: MLP(multi-layer perceptron)

BOSTON UNIVERSITY

max_iter: 10,000
training_size: 1096
features: 4
classes: 1



Notation:
- hi,j: jth neuron in ith layer

```
$ ./MLP 3 4,5,5 softmax,relu,tanh 1 sigmoid 0.01 10000 data/data_train.csv 1096 5
data/data_test.csv 275 5
```

Courtesy: Implementation of Multi Layer Perceptron in C, https://github.com/manoharmukku/multilayer-perceptron-in-c

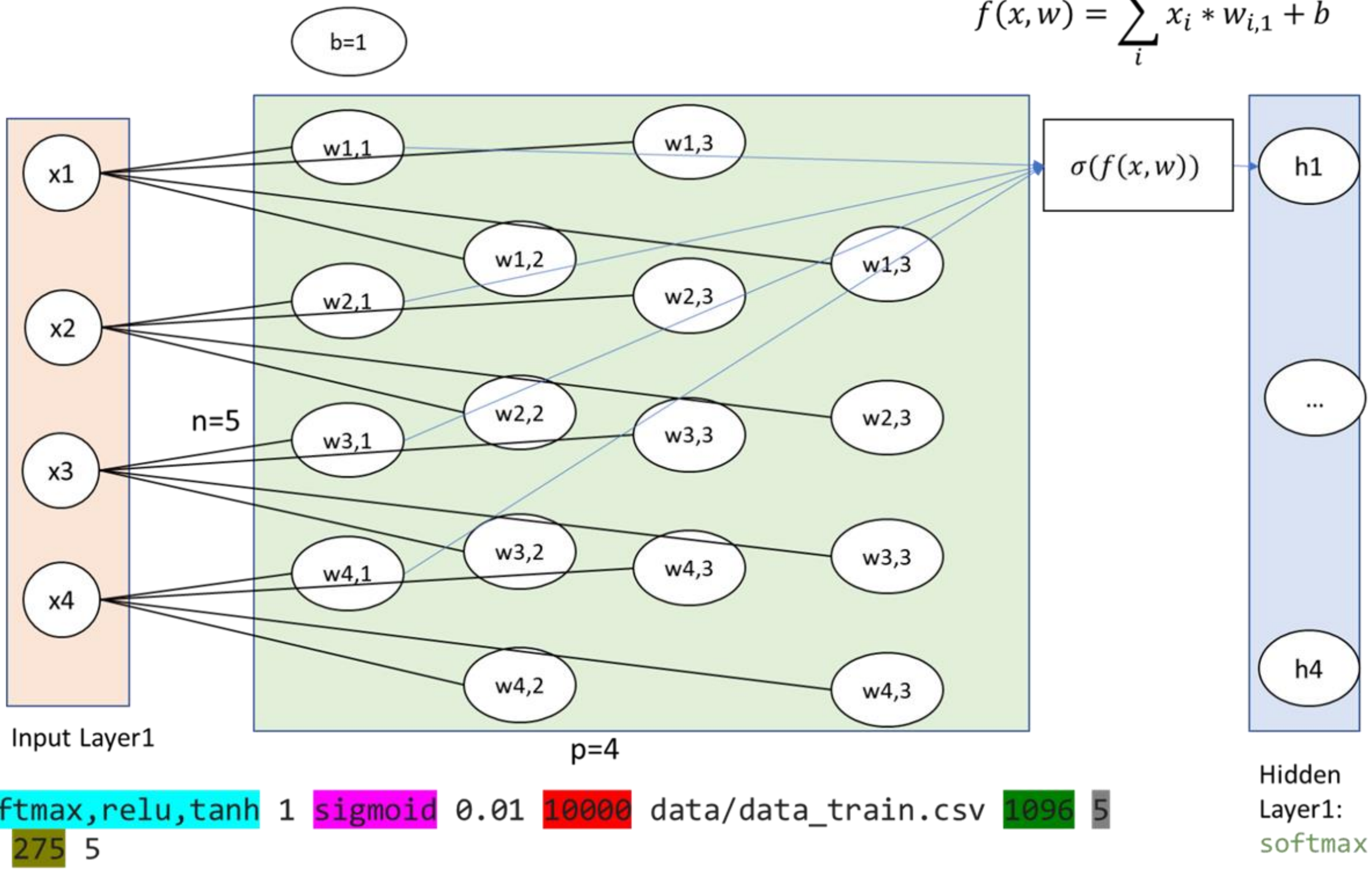
Problem overview: MLP(multi-layer perceptron)

BOSTON UNIVERSITY

max_iter: 10,000
training_size: 1096
features: 4
classes: 1

sample MLP: details between input and layer1

$$f(x,w) = \sum_i x_i * w_{i,1} + b$$




```
$ ./MLP 3 4,5,5 softmax,relu,tanh 1 sigmoid 0.01 10000 data/data_train.csv 1096 5
data/data_test.csv 275 5
```

Courtesy: Implementation of Multi Layer Perceptron in C, https://github.com/manoharmukku/multilayer-perceptron-in-c

Single Core CPU Optimization

Pingcheng Dong

Time analyzation

1. Flat profiler

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00   23.19                 main [1]
                0.06   23.11       1/1           mlp_trainer [2]
                0.00    0.02       1/1           mlp_classifier [16]
                0.00    0.00       2/2           read_csv [24]
-----------------------------------------------
                0.06   23.11       1/1           main [1]
[2]     99.9    0.06   23.11       1         mlp_trainer [2]
               12.20    5.34 10960000/10960000     back_propagation [3]
                1.16    4.35 10960000/10960000     forward_propagation [4]
                0.06    0.00   10000/10000       randomly_shuffle [14]
                0.00    0.00       1/1           initialize_weights [25]
-----------------------------------------------
               12.20    5.34 10960000/10960000     mlp_trainer [2]
[3]     75.6   12.20    5.34 10960000          back_propagation [3]
                4.75    0.59 43840000/43840000     calculate_local_gradient [5]
-----------------------------------------------
                1.16    4.35 10960000/10960000     mlp_trainer [2]
[4]     23.7    1.16    4.35 10960000          forward_propagation [4]
                3.59    0.00 43840000/43840000     mat_mul [6]
                0.28    0.00 10960000/10960000     relu [8]
                0.20    0.00 10960000/10960000     softmax [9]
                0.19    0.00 10960000/10960000     tan_h [10]
                0.09    0.00 10960000/10960000     sigmoid [13]
-----------------------------------------------
                4.75    0.59 43840000/43840000     back_propagation [3]
[5]     23.0    4.75    0.59 43840000          calculate_local_gradient [5]
                0.32    0.00 10960000/10960000     d_relu [7]
                0.14    0.00 10960000/10960000     d_softmax [11]
                0.12    0.00 10960000/10960000     d_tanh [12]
                0.02    0.00 10960000/10960000     d_sigmoid [15]
```

2. Further analyzation: 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:

```c
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);
```

```
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.
```

for loops inside the back_propagation

```c
/*----------- 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];
        }
    }
}
```

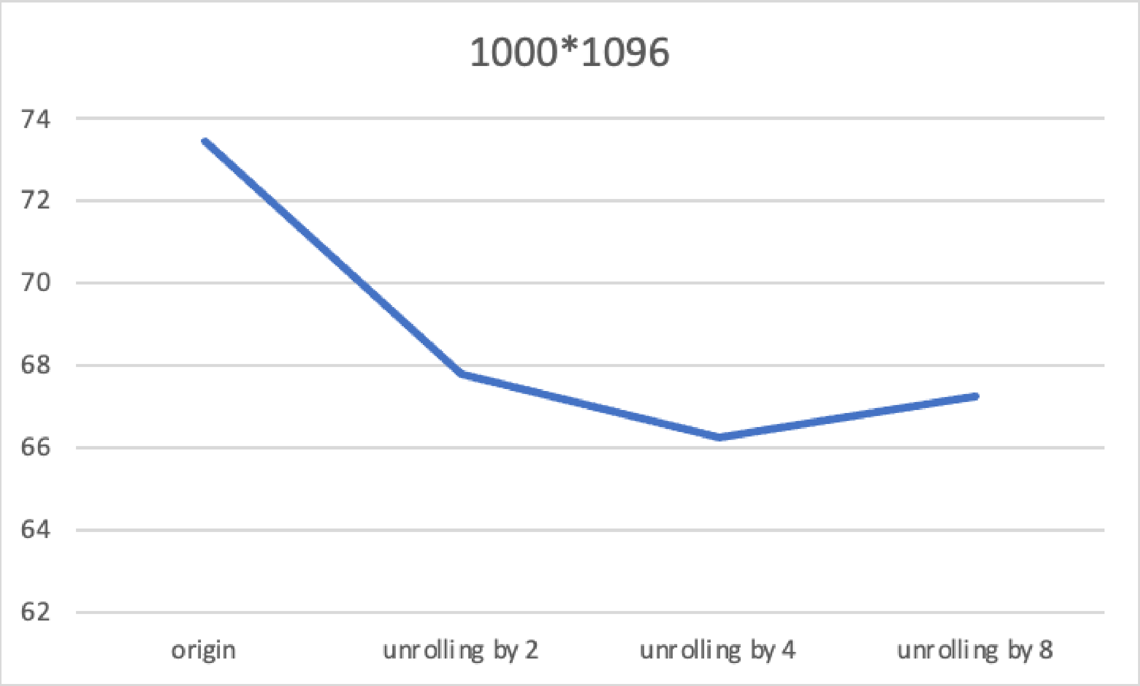
ways to optimize

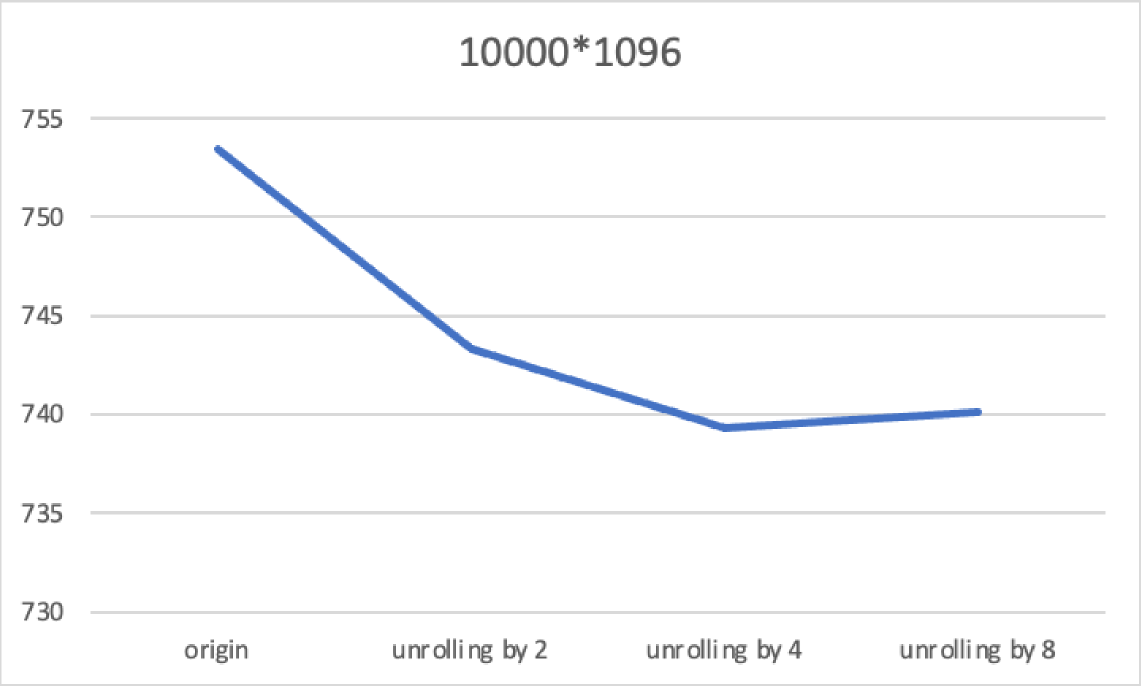
1. unrolling the for loops
2. use openmp

unrolling for loops

time difference before openmp
the platform is i5-5250U 2.3GHz

size	origin	unrolling by 2	unrolling by 4	unrolling by 8
1000*1096	73.43245	69.768331	66.23826	67.23843
10000*1096	753.47324	745.29493	739.28194	740.12887

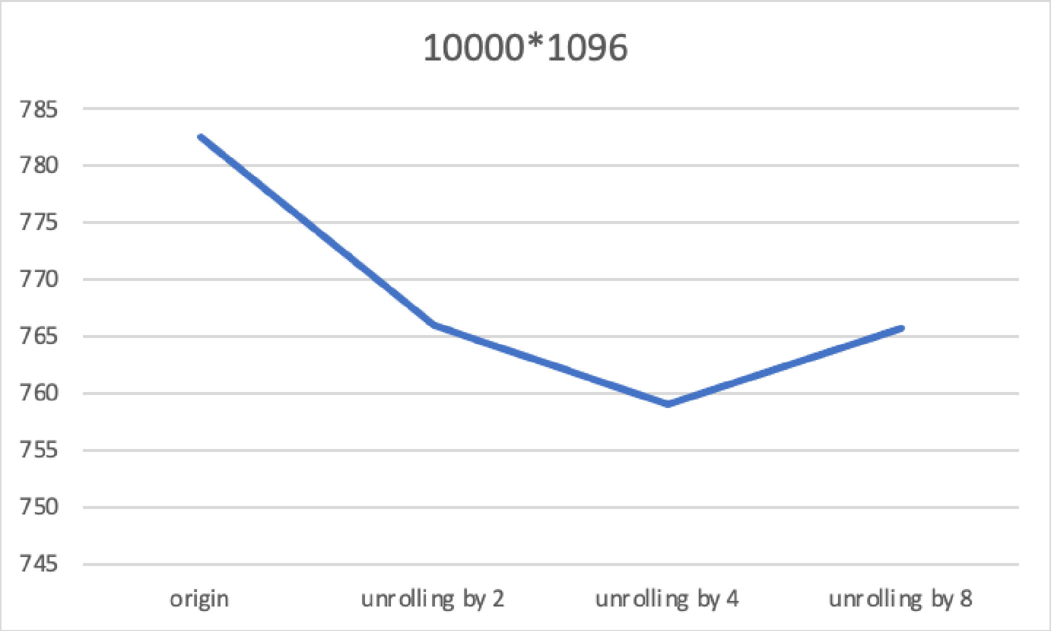


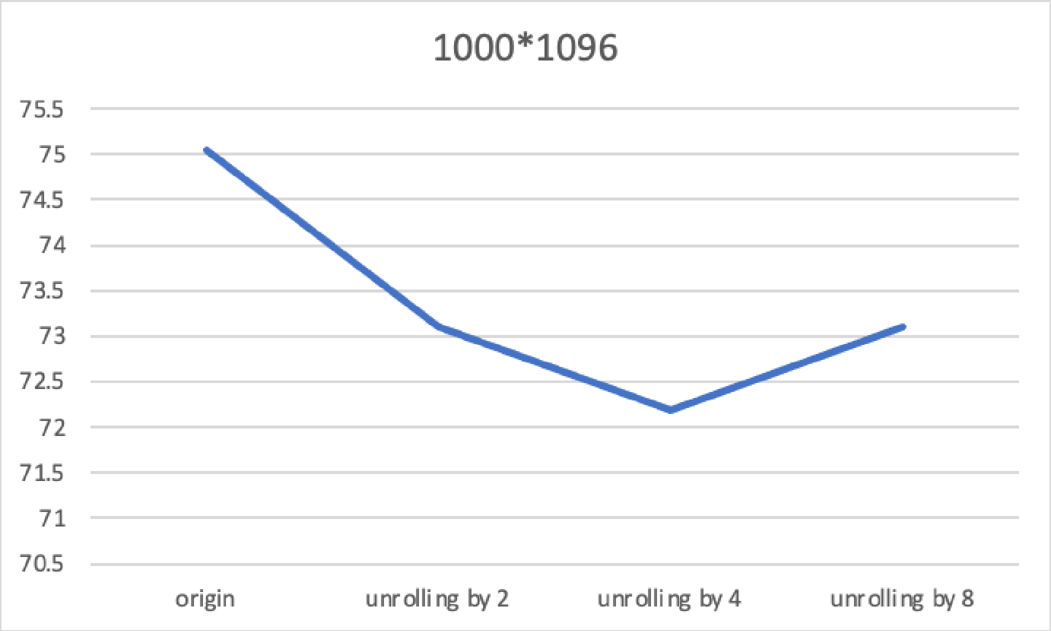


openmp

use openmp with unrolling
the platform is i5 - 8279U 4 cores 8 threads

size	origin	unrolling by 2	unrolling by 4	unrolling by 8
1000*1096	75.04	73.0986	72.1875	73.1005
10000*1096	782.56	766	758.9855	765.6478





the openmp did not perform well inside the back_propagation function

Multi-Core CPU Optimization

Yunlu Deng

Introduction

BOSTON UNIVERSITY

Steps:

1. Analyse the code and choose the optimize part;

```
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]);
    }
}
```

function mlp_trainer

function back_propagation

2. Make sure about dependence;

```
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];
        }
    }
}
```

```
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;
```

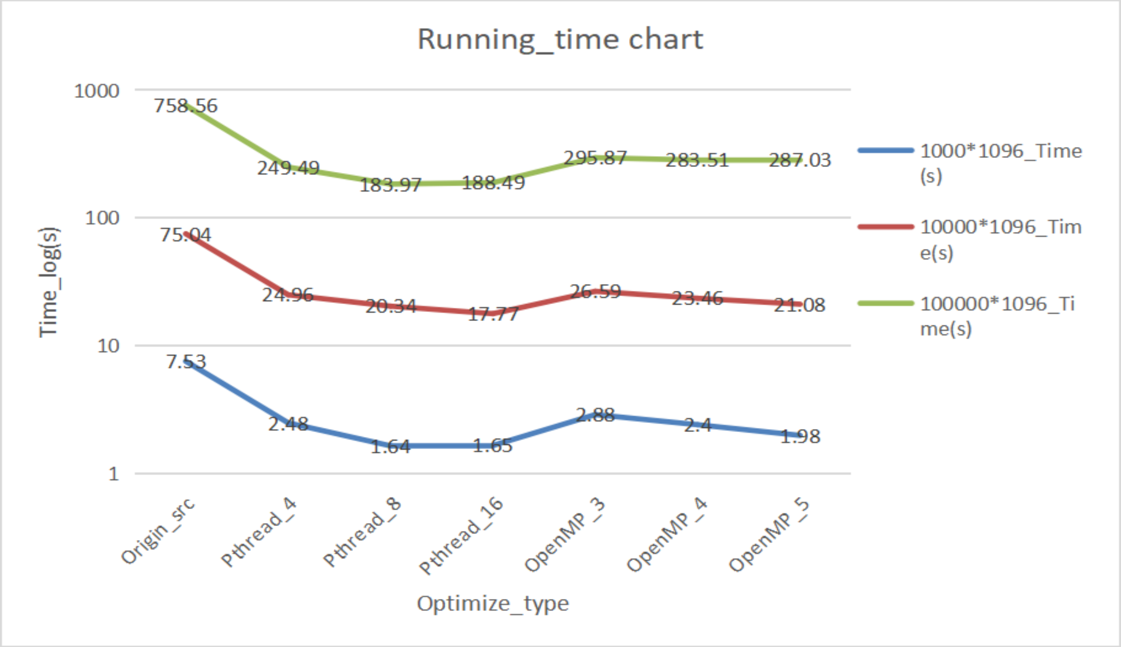
Result

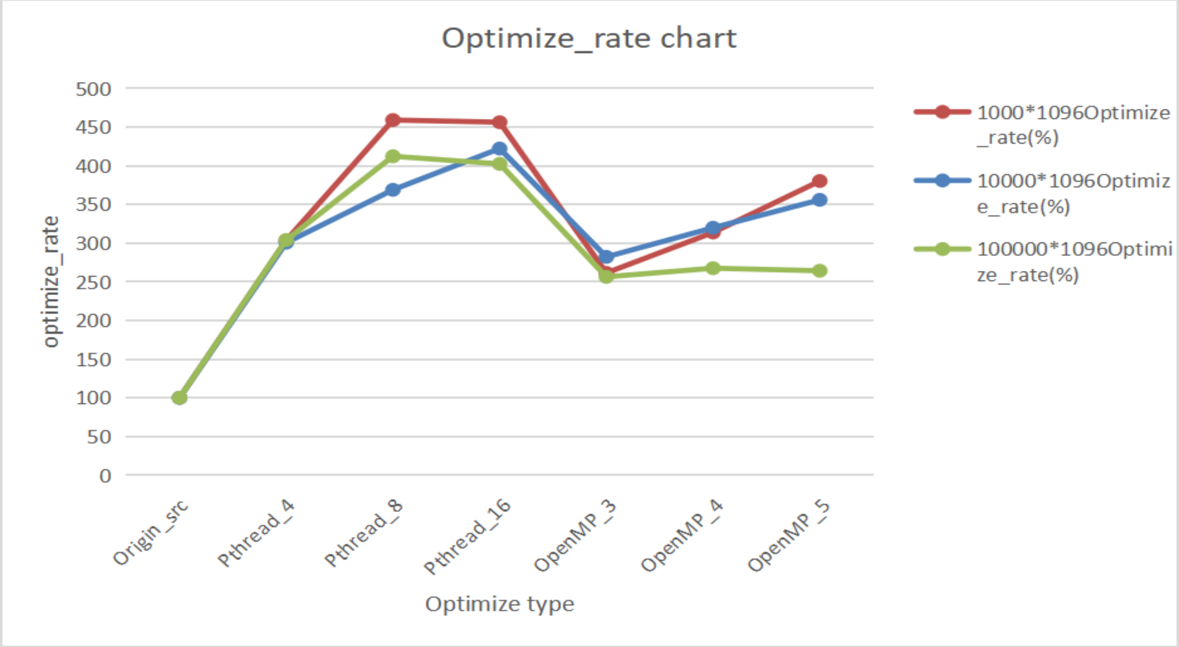
BOSTON UNIVERSITY

	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 Result of Optimization

```
Training:
---------
Pthread_training
train time = 2.479414
Done.

Classifying:
------------
Classifying test example 275 of 275

Confusion matrix
-----------------

            |predicted 0          predicted 1
---------------------------------------------
Actual 0    |163                   112

Actual 1    |0          0


Accuracy: 59.27
```





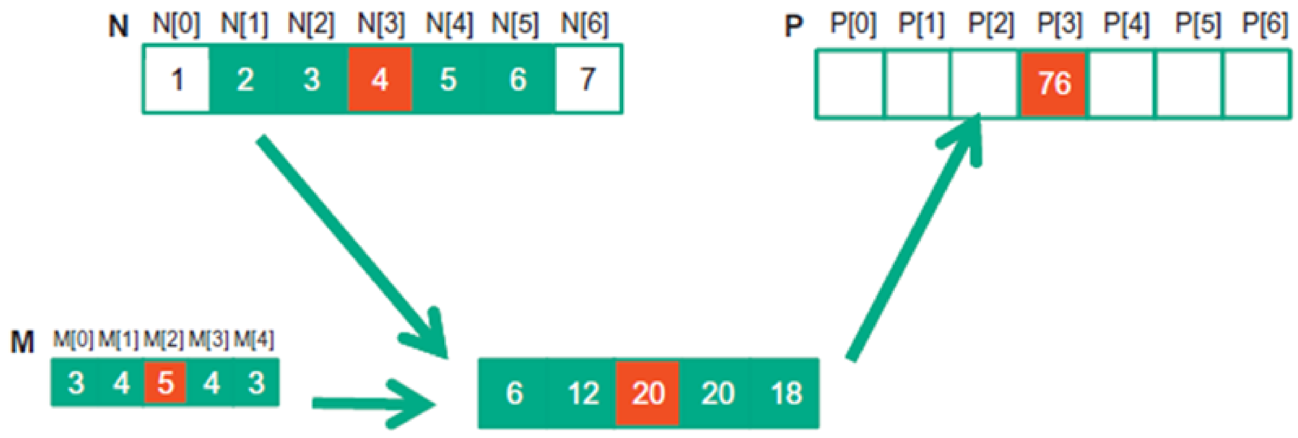
GPU Optimization

Zhengqi Dong

1D Conv General Idea:

BOSTON UNIVERSITY

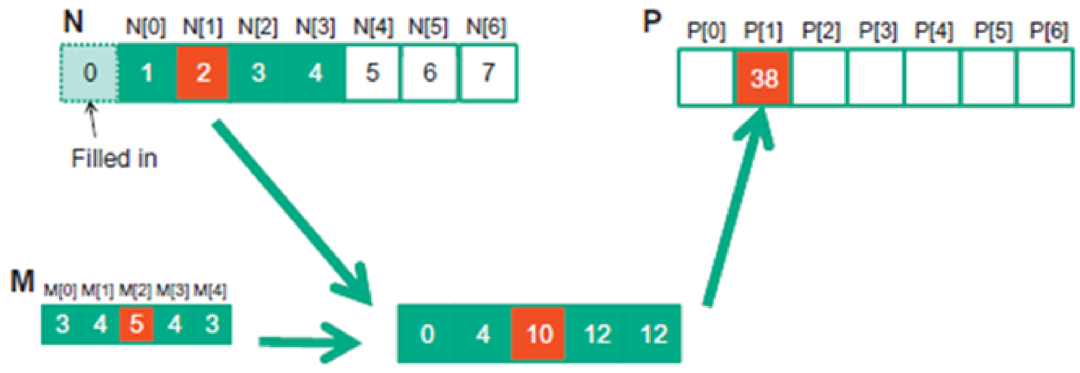
1D example



calculation for P[3]

```
P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]
     = 2*3 + 3*4 + 4*5 + 5*4 + 6*3
     = 76
```

1D boundary case: "ghost cell"



Courtesy: Kirk, David B., and W. Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

1D conv: CPU version(with same padding)

```
void conv_1D(float *N, float  *M, float *P, int mask_width, int
N_rowlen) {
    int i; float Pvalue;
    // Return directly, if threadIdx exceed the size of P
    int halo_width = (mask_width - 1) / 2;
    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;
    }
}
```

i =1: N[0]*M[0]+N[1]*M[1]+N[2]*M[2]=1.8
i=2: N[1]M[0]+N[2]M[1]+N[3]M[2]=3.1
...
i=5: N[4]M[0]+N[5]M[1]+N[6]M[2]=2.2

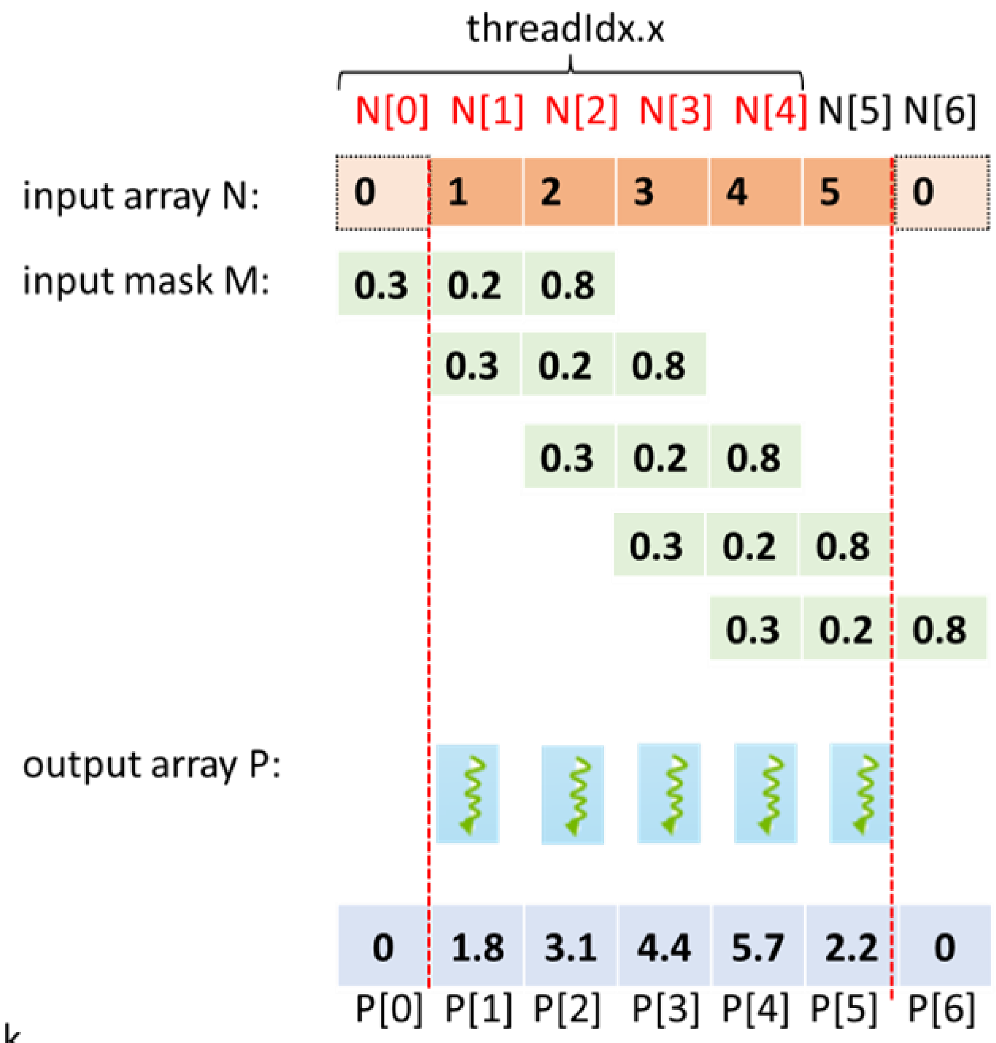
	N[0]	N[1]	N[2]	N[3]	N[4]	N[5]	N[6]
input array N:	**0**	**1**	**2**	**3**	**4**	**5**	**0**
input mask M:	**0.3**	**0.2**	**0.8**				
output array P:	**0**	**1.8**	**3.1**	**4.4**	**5.7**	**2.2**	**0**
	P[0]	P[1]	P[2]	P[3]	P[4]	P[5]	P[6]

Assumption:
- Padding: same padding is used
- mask_width: odd number, e.g., mask_width = 3
- halo_cell = ceil(mask_width/2) = 1
- i in range [halo_width, N_rowlen - 1 - halo_width]

1D conv: kernel function for single block (with same padding)

BOSTON UNIVERSITY

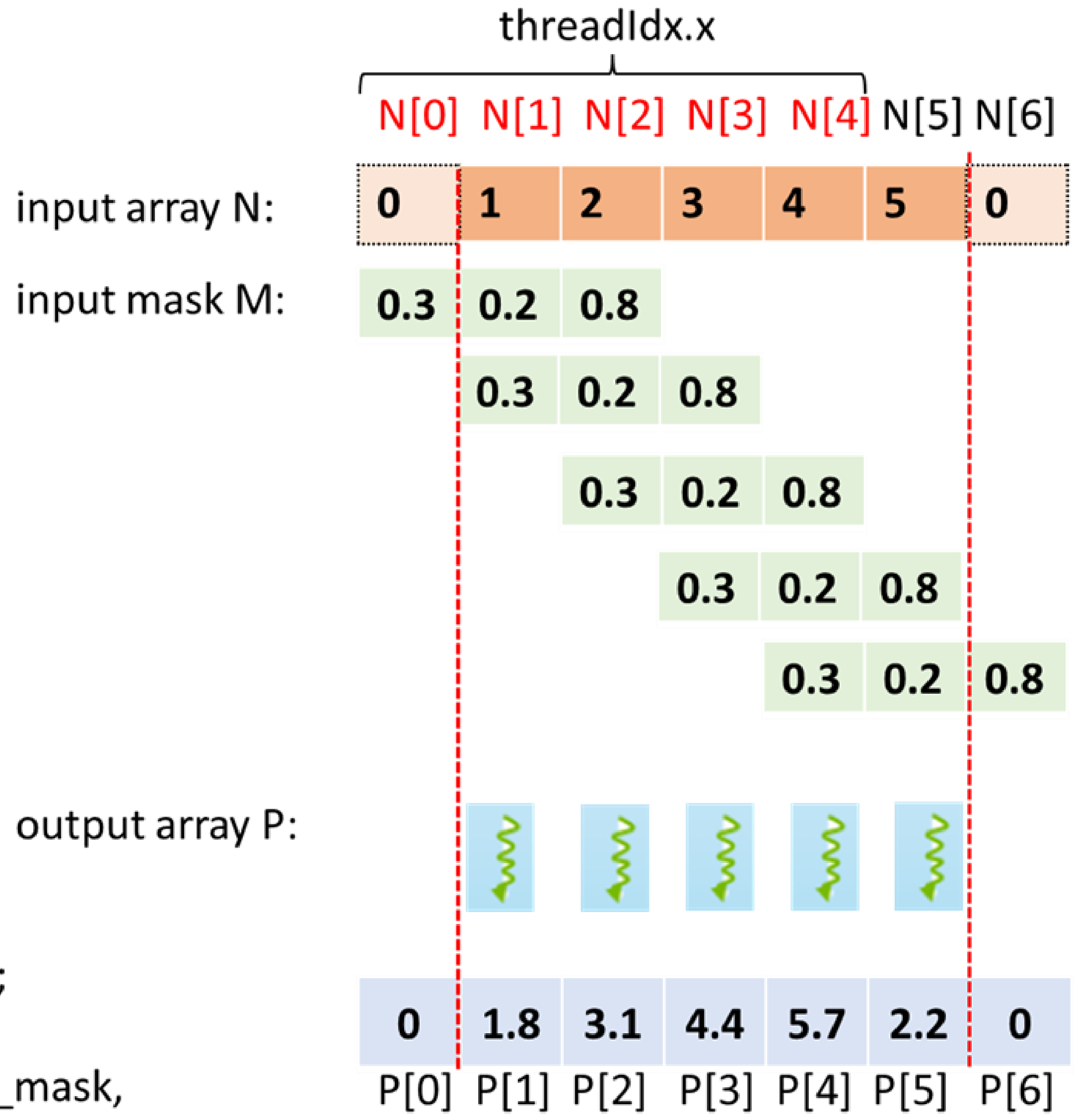
```
__global__ void conv_1D _single_block(float *N, float  *M,
float *P, int mask_width, int N_rowlen) {
    int i = threadIdx.x; // For output array P
    int j; float Pvalue = 0;

    // Return directly, if threadIdx exceed the size of P
    int halo_width = (mask_width - 1) / 2;
    if (i < halo_width || i > (N_rowlen - 1 - halo_width)) return;
    // 1 off for idx
    for (j = 0; j < mask_width; j++){
        Pvalue += N[i - halo_width + j] * M[j];
    }
    P[i] = Pvalue;
}
```




```
cuda_single_block_conv_1D<<<1, N_ARR_LEN>>>(d_input, d_mask,
d_output_data, MASK_WIDTH, N_ARR_LEN);
```

1D conv: kernel function for multi block (with same padding)

```
__global__ void convolution_1D_multi_block(float *N, float
*M, float *P, int mask_width, int N_rowlen) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    float Pvalue = 0;
    // Return directly, if threadIdx exceed the size of P
    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] * d_mask_constant[j];
    }
    P[i] = Pvalue;
}

dim3 dimGrid(ceil(P_ARR_LEN / NUM_THREADS_PER_BLOCK), 1);
dim3 dimBlock(NUM_THREADS_PER_BLOCK, 1);
cuda_conv_1D_multi_block<<<dimGrid, dimBlock>>>(d_input, d_mask,
d_output_data, MASK_WIDTH, N_ARR_LEN);
```



1D conv multi-block Result (tested on SCC Tesla V100):

BOSTON UNIVERSITY

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

Takeaway:

1. GPU will be faster if we're running on a larger array.
2. Floating-point arithmetic calculation to global memory is 1.0 in the kernel, which is pretty bad. --> Might need to consider leverage the shared_memory to close to peak performance.
3. Length of input array(N) must be multiple of NUM_THREADS_PER_BLOCK. Otherwise, we might see many unmatched result, e.g., NUM_THREADS_PER_BLOCK=256, P_ARR_LEN=1000.

1D conv with putting mask_array into constant memory:

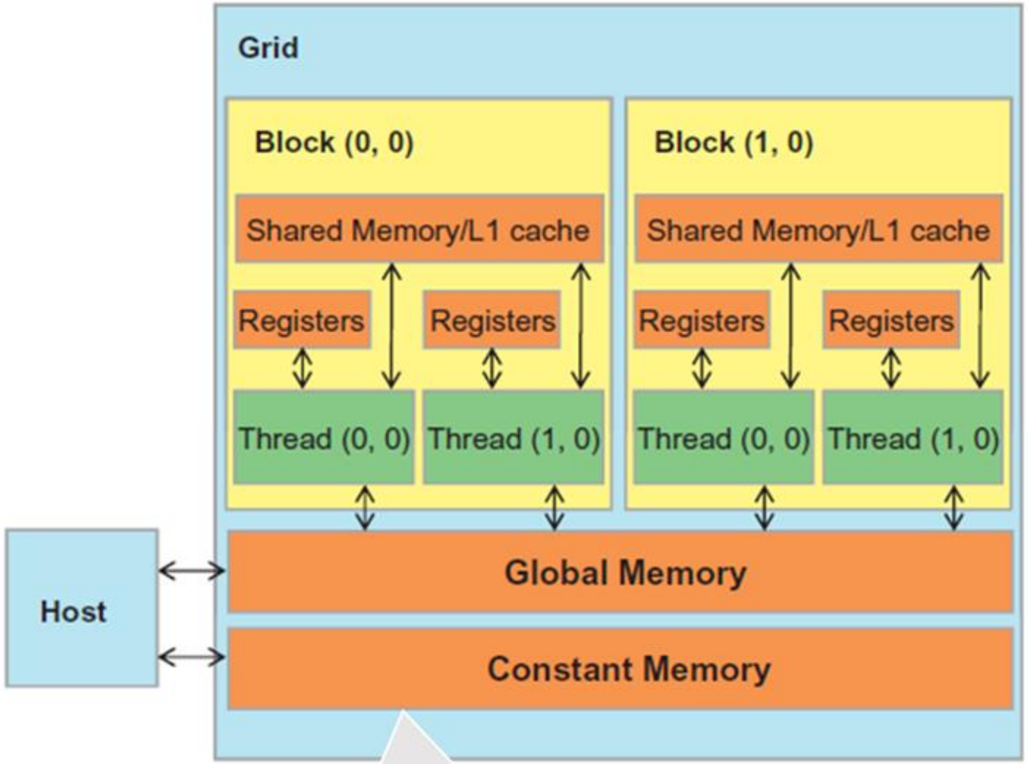
Observation:

1. First, the size of M array is typically small (often less than 100). That's MASK_WIDTH << N_ARR_LEN.
2. Second, the contents of M are not changed throughout the execution of the kernel.
3. Third, all threads need to access the mask elements and in the same order, from M[0] to M[MASK_WIDTH – 1].

Update:

- Place mask array(M) in constant memory; (FYI: you have 64 KB constant memory in total)
- Allocated and initialized mask in a mask `h_mask` array in the host memory, and transferred to device constant memory before launched kernel function:

```
cudaMemcpyToSymbol(d_mask h_mask, Mask_Width*sizeof(float));
```



Constant memory is a good place for mask array(M).

```
// Declare in host code:
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

1D conv with putting mask_array into constant memory:

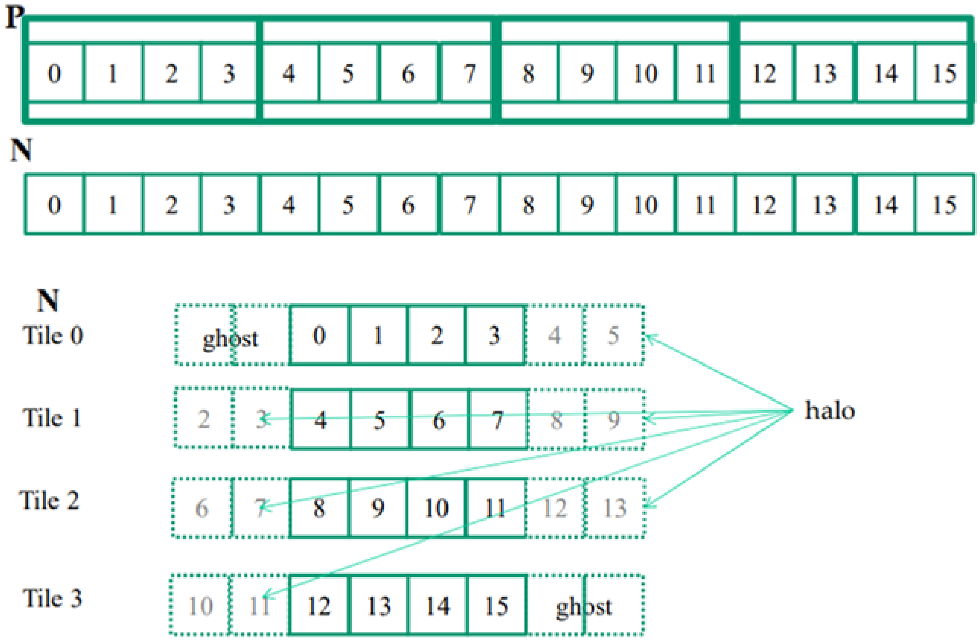
BOSTON UNIVERSITY

rowlen	CPU(msec)	GPU(msec)	Speedup
1000	0.059	0.5508	0.107117
100,000	2.513	1.372	1.831633
10,000,000	124.8	34.298	3.638696
1,000,000,000	9804.3121	3110.98877	3.15151

Takeaway:

- With the use of constant memory and caching, we have effectively doubled the ratio of floating-point arithmetic to memory access to 2.

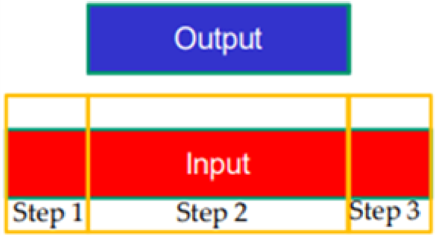
Tiled 1D Convolution Basic Idea:

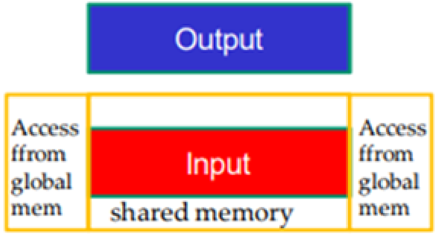


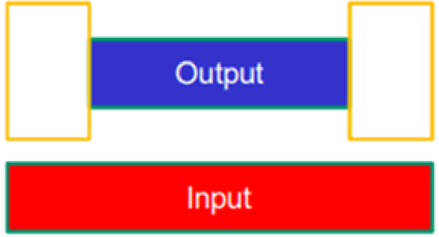
Strategy 1:
- Block size covers output tile
- Use multiple steps to load input tile

Strategy 2:
- Block size covers output tile
- Load only "core" of input tile
- Access halo cells from global memory

Strategy 3:
- Block size covers input tile
- Load input tile in one step
- Turn off some threads when calculating output







Courtesy: Kirk, David B., and W. Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

Tiled 1D Conv on Strategy 2: General Case

Hyperparameter: N_ARR_LEN = 16, MASK_WIDTH=3, TILE_WIDTH = 4, N_ds[TILE_WIDTH]

Block1

input array N:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
N[0]	N[1]	N[2]	N[3]	N[4]	N[5]	N[6]	N[7]	N[8]		...				N[14]	N[15]

th[0] th[1] th[2] th[3]

threadIdx.x = 2
blockIdx.x = 1
blockDim.x = 4

shared array N_ds:

4	5	6	7
[0]	[1]	[2]	[3]

Mask array(M):

0.3	0.2	0.8
[0]	[1]	[2]

output array P:

						8.3									
P[0]	P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]	P[8]				P[12]	P[13]	P[14]	P[15]

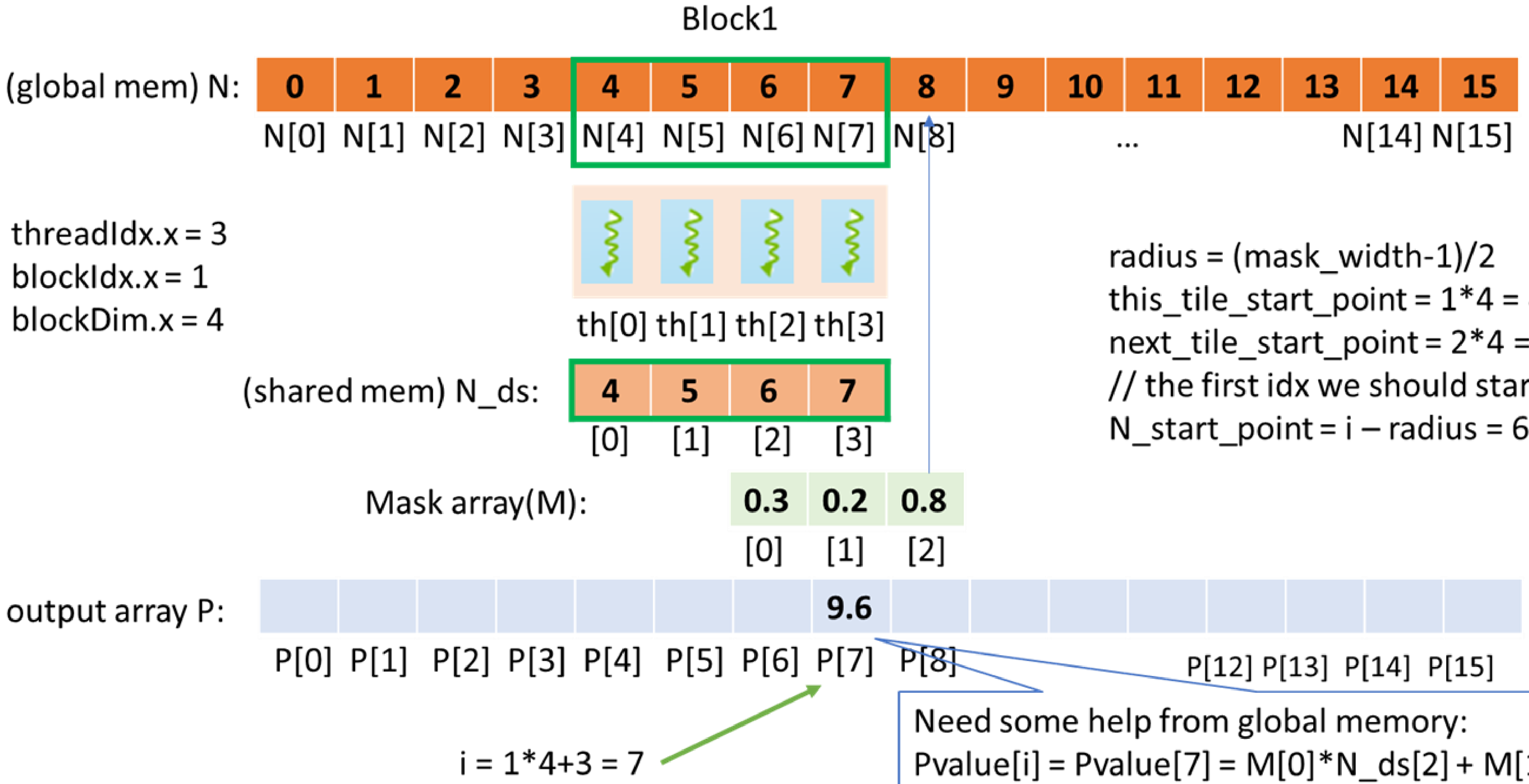
i = 1*4+2 = 6

radius = (mask_width-1)/2
this_tile_start_point = 1*4 = 4
next_tile_start_point = 2*4 = 8
// the first idx we should start to read from N
N_start_point = i – radius = 6 – 1 = 5

Directly access shared memory:
Pvalue[i] = Pvalue[6] = M[0]*N_ds[1] + M[1]*N_ds[2] + M[2]*N_ds[3] = .3*5 + .2*6 + .8*7 = 8.3

Tiled 1D Conv on Strategy 2: Boundary Case 1

Hyperparameter: N_ARR_LEN = 16, MASK_WIDTH=3, TILE_WIDTH = 4, N_ds[TILE_WIDTH]



radius = (mask_width-1)/2
this_tile_start_point = 1*4 = 4
next_tile_start_point = 2*4 = 8
// the first idx we should start to read from N
N_start_point = i – radius = 6 – 1 = 5

Need some help from global memory:
Pvalue[i] = Pvalue[7] = M[0]*N_ds[2] + M[1]*N_ds[3] + M[2]*N[8] = .3*6 + .2*7 + .8*8 = 9.6

Tiled 1D Conv on Strategy 2: code

```
#define TILE_WIDTH 4
__global__ void convolution_1D_tiled_cache_kernel(float *N, float *P, int mask_width, int
N_rowlen){
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   __shared__ float N_ds[TILE_WIDTH];
   N_ds[threadIdx.x] = N[i];
   __syncthreads();

   int halo_width = (mask_width - 1) / 2;
   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;
   for (int j = 0; j < mask_width; j++){
      int N_index = N_start_point + j;
      if (N_index >= 0 && N_index < N_rowlen){
         int reading_from_N_ds = ((N_index >= this_tile_start_point) && (N_index <
next_tile_start_point));
         if (reading_from_N_ds){
            Pvalue += N_ds[threadIdx.x - halo_width + j] * d_mask_constant[j];
         }else {
            Pvalue += N[N_index] * d_mask_constant[j];
         }
      }
   }
   P[i] = Pvalue;
}
```

Tiled 1D Conv on Strategy 2: Result

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

Takeaway:

- The result is not as good as we expected, but we are stilling working on it...

Note:

- All result computed previous set NUM_THREADS_PER_BLOCK=16

Conclusion

What we achieved so far:

- Multi Core:
  - Pthread optimization: 4.5x best improvement, the more thread, the better.
  - OpenMP optimization: 3.8x best improvement, the more thread, the better.
- GPU Optimization:
  - It's only useful when we have a large input array (>100,000 float).
  - By putting mask_array into constant memory does help to improve the performance
  - The performance with tiled algorithm did not perform very well as we expected, and we will spend more time to investigate it...

Future works:

- Multi Core
  - SIMD vectorization.
  - Combine with single core optimization to get better performance.
- GPU Optimization:
  - If time allowed, we will apply tiled algorithm on 2D conv operations as well

Thanks for listening!
Any Question?

Reference:

BOSTON UNIVERSITY

[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

Backup Slides

Tiled 1D conv on with Strategy 2 result (tested on my PC RTX2070, tasked=5):

BOSTON UNIVERSITY

N_rowlen = P_rowlen = 1024

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
   1024,                3,             1024,        0.01711
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 5 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 1.947328 (msec)
ZERO RESULT in 0:    0.0000  0.0000  -nan %
ZERO RESULT in 1023:    0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/1024 results (from GPU) are zero
```

N_rowlen = P_rowlen = 100,000

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
 100000,              3,            100000,          3.595
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 5 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 4.122752 (msec)
ZERO RESULT in 0:    0.0000  0.0000  -nan %
ZERO RESULT in 99999:    0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/100000 results (from GPU) are zero
```

N_rowlen = P_rowlen = 10,000,000

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
10000000,           3,        10000000,          163.6
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 5 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 147.237503 (msec)
ZERO RESULT in 0:    0.0000  0.0000  -nan %
ZERO RESULT in 9999999: 0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/10000000 results (from GPU) are zero
```

Takeaway:

- Make sense when we have a very large array

1D conv multi-block Result (tested on my PC RTX2070, tasked=2):

BOSTON UNIVERSITY

Hyperparameter: {NUM_THREADS_PER_BLOCK=16; MASK_WIDTH 3; TOL 0.05; }

N_rowlen = P_rowlen = 1024

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
   1024,           3,          1024,         0.0174
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 2 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 1.543776 (msec)
ZERO RESULT in 0:   0.0000  0.0000  -nan %
ZERO RESULT in 1023:    0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/1024 results (from GPU) are zero
```

N_rowlen = P_rowlen = 100,000

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
 100000,           3,        100000,         1.221
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 2 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 3.005472 (msec)
ZERO RESULT in 0:   0.0000  0.0000  -nan %
ZERO RESULT in 99999:   0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/100000 results (from GPU) are zero
```

N_rowlen = P_rowlen = 10,000,000

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
10000000,          3,       10000000,         148.8
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 2 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 111.096252 (msec)
ZERO RESULT in 0:   0.0000  0.0000  -nan %
ZERO RESULT in 9999999: 0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/10000000 results (from GPU) are zero
```

Takeaway:

1. GPU will be faster if we're running on a larger array.
2. Floating-point arithmetic calculation to global memory is 1.0 in the kernel, which is pretty bad. --> Might need to consider leverage the shared_memory to close to peak performance.
3. Length of input array(N) must be multiple of NUM_THREADS_PER_BLOCK. Otherwise, we might see many unmatched result, e.g., NUM_THREADS_PER_BLOCK=256, P_ARR_LEN=1000.

1D conv multi-block and mask in constant memory result (tested on my PC RTX2070, taskid=3) :

BOSTON UNIVERSITY

Hyperparameter: {NUM_THREADS_PER_BLOCK=16; MASK_WIDTH 3; TOL 0.05; }

N_rowlen = P_rowlen = 1024

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
   1024,              3,          1024,        0.01769
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 3 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 3.052384 (msec)
ZERO RESULT in 0:   0.0000  0.0000  -nan %
ZERO RESULT in 1023:    0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/1024 results (from GPU) are zero
```

N_rowlen = P_rowlen = 100,000

```
N_lenth, Mask_length, output_length, 1D conv time(msec)
 100000,              3,        100000,          1.923
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 3 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 2.552192 (msec)
ZERO RESULT in 0:   0.0000  0.0000  -nan %
ZERO RESULT in 99999:   0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/100000 results (from GPU) are zero
```

N_rowlen = P_rowlen = 10,000,000

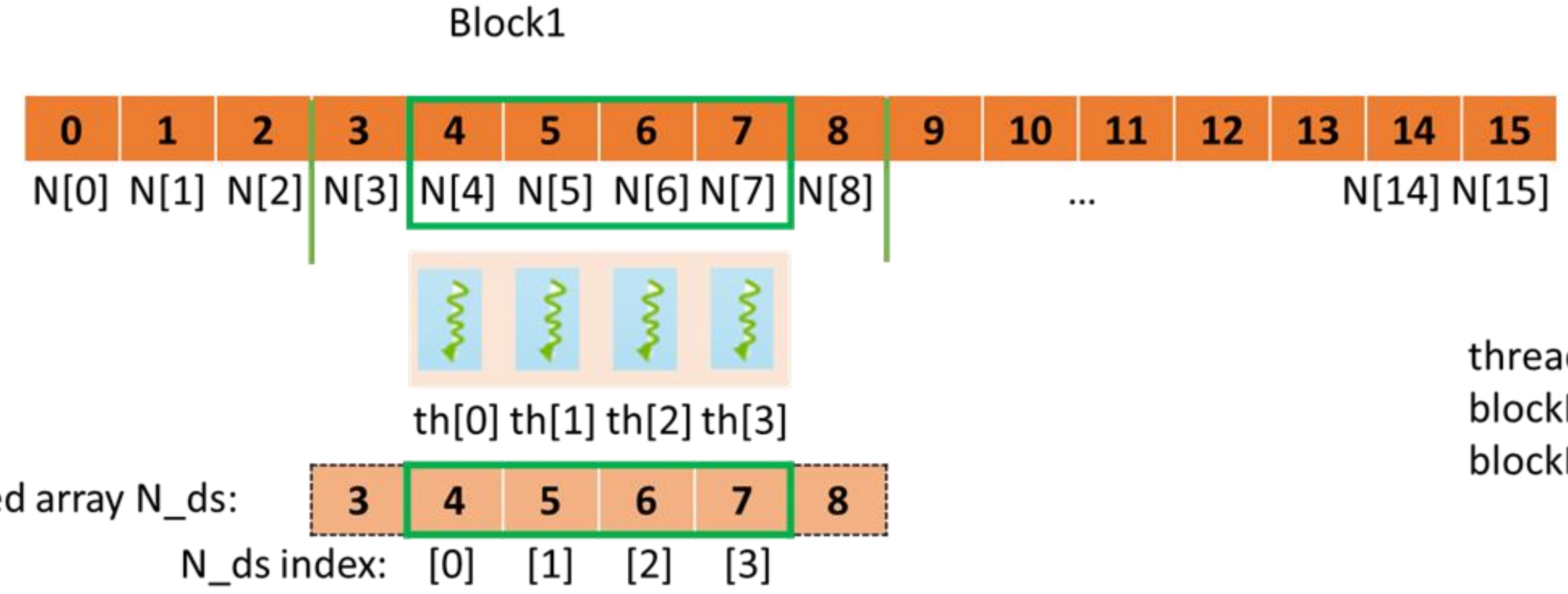
```
N_lenth, Mask_length, output_length, 1D conv time(msec)
10000000,             3,        10000000,          139.5
==========> All CPU tests are done! Now, running GPU code!
==============>Running taskid #: 3 on GPU!
1(1D single block) --> 2(1D multi-block) --> 3(1D mulit-block with mask in constant
memory) --> 4(tiled algo with Strategy 1) --> 5(tiled algo with Strategy 2(shared +
global memory))

GPU time: 104.104767 (msec)
ZERO RESULT in 0:    0.0000  0.0000  -nan %
ZERO RESULT in 9999999: 0.0000  0.0000  -nan %

@ERROR: TEST FAILED: 2/10000000 results (from GPU) are zero
```

Takeaway:

- With the use of constant memory and caching, we have effectively doubled the ratio of floating-point arithmetic to memory access to 2.

(My example) Strategy 1: Loading the left halo (with no padding)

Hyperparameter: N_ARR_LEN = 16, MASK_WIDTH=3, TILE_SIZE = 4, N_ds[6]



threadIdx.x = 2
blockIdx.x = 1
blockDim.x = 4

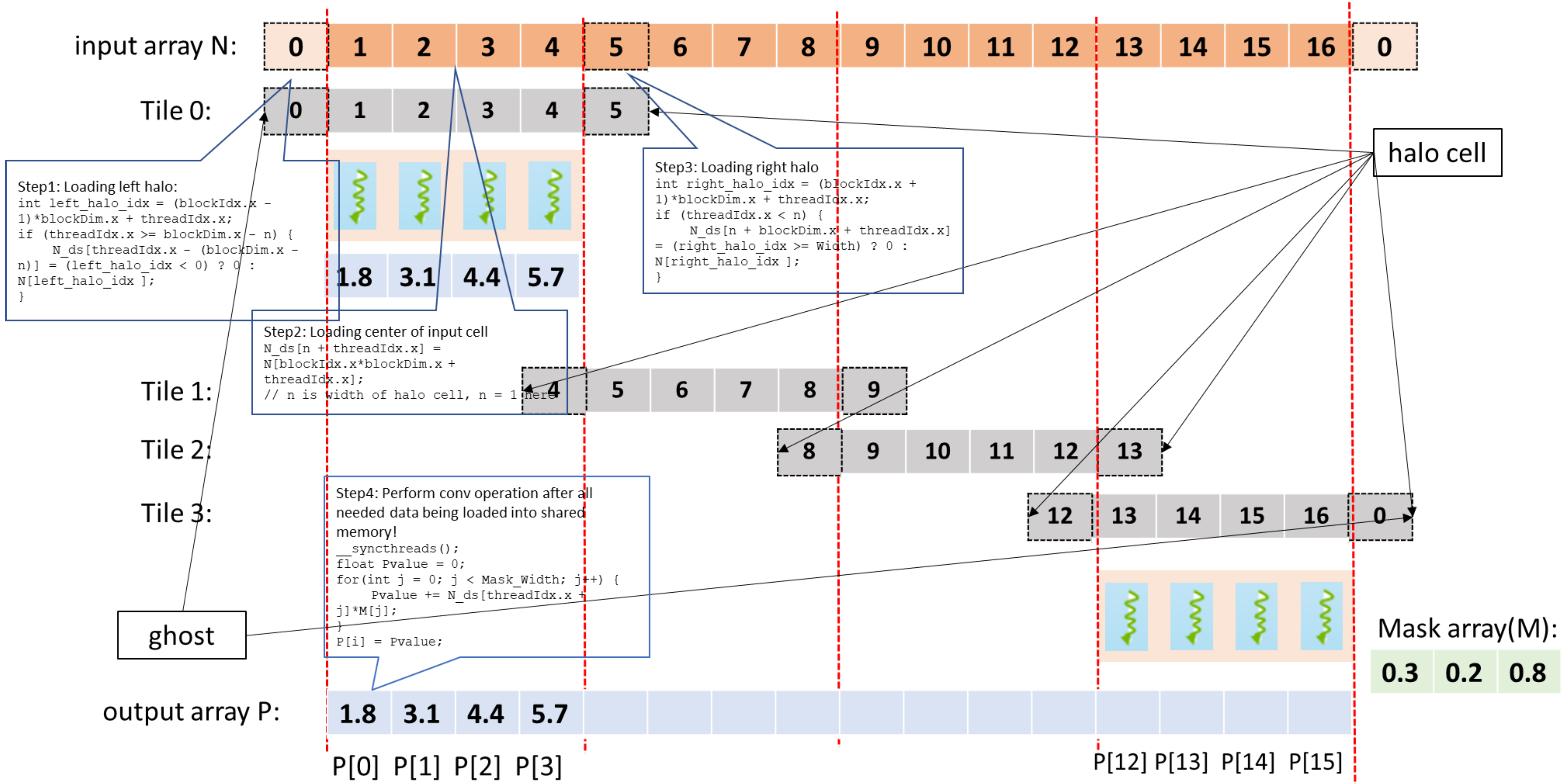
i = 1*4+2 = 6

left_halo_idx = (1-1)*4+2 = 2
//assume mask_width is odd number
n = (mask_width-1)/2 = 1
blockDim.x – n = 4-1 = 3

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
__shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
int n = Mask_Width/2;
int left_halo_idx = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] = (left_halo_idx < 0) ? 0 : N[left_halo_idx ];
}
```

Courtesy: Kirk, David B., and W. Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

(My example) Strategy 1: All in one (with no padding)



Note: N_length must be multiple of TILE_SIZE, e.g., N_length = 16, TILE_SIZE=4