CNN Optimizations

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

1. Problem Statement

1.1. Dataset

What are the primary data structures? What is the memory reference pattern?

1.2 What is MLP

1.3 Functions running time

2. Single-Core CPU Optimization

3. Multi-Core CPU Optimization

2.2.1 Choose Optimization part
2.2.2 Dependence

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] have shown that this type of network can be very memory (weight) and computational 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 size 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 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 XXX 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.

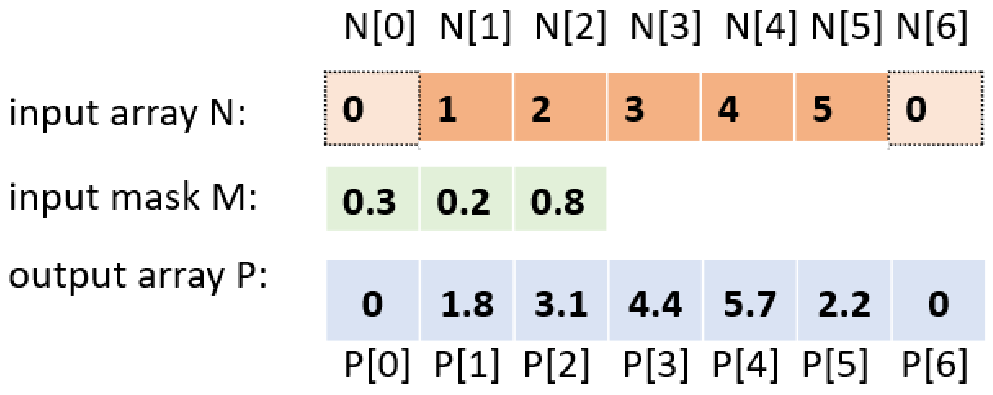

Figure 1: 1D conv boundary case

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 1 and 2:

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 1: 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 2: 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 XXX, and the corresponding code is shown in Code 4.

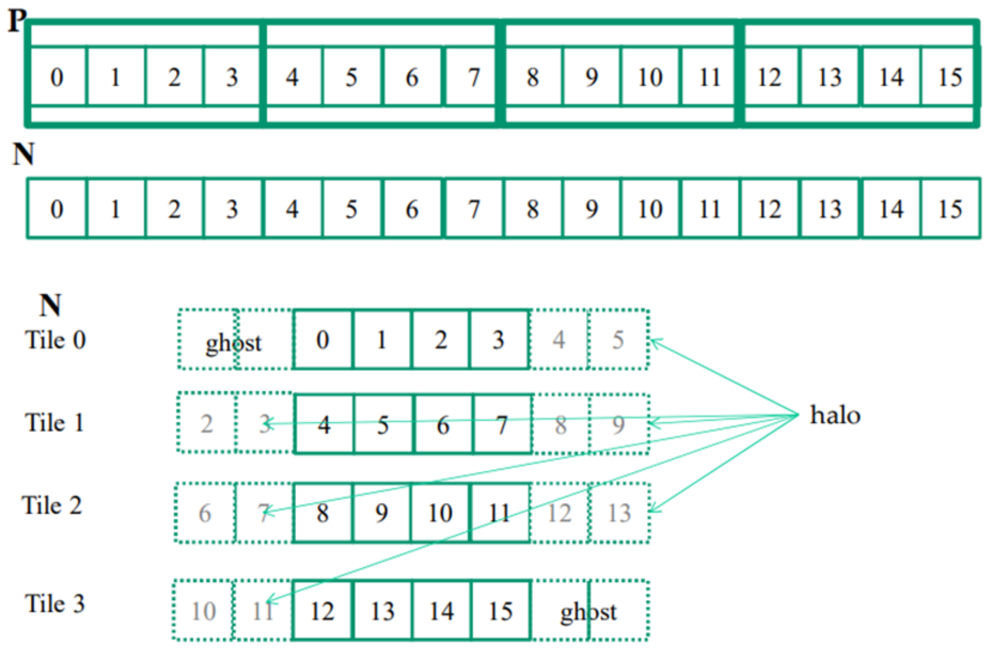

Figure XXX: Stragegy 1 of 1D CNN tiled algorithm

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 XXX, and the code that I implemented for strategy 2 is shown in Code 5:

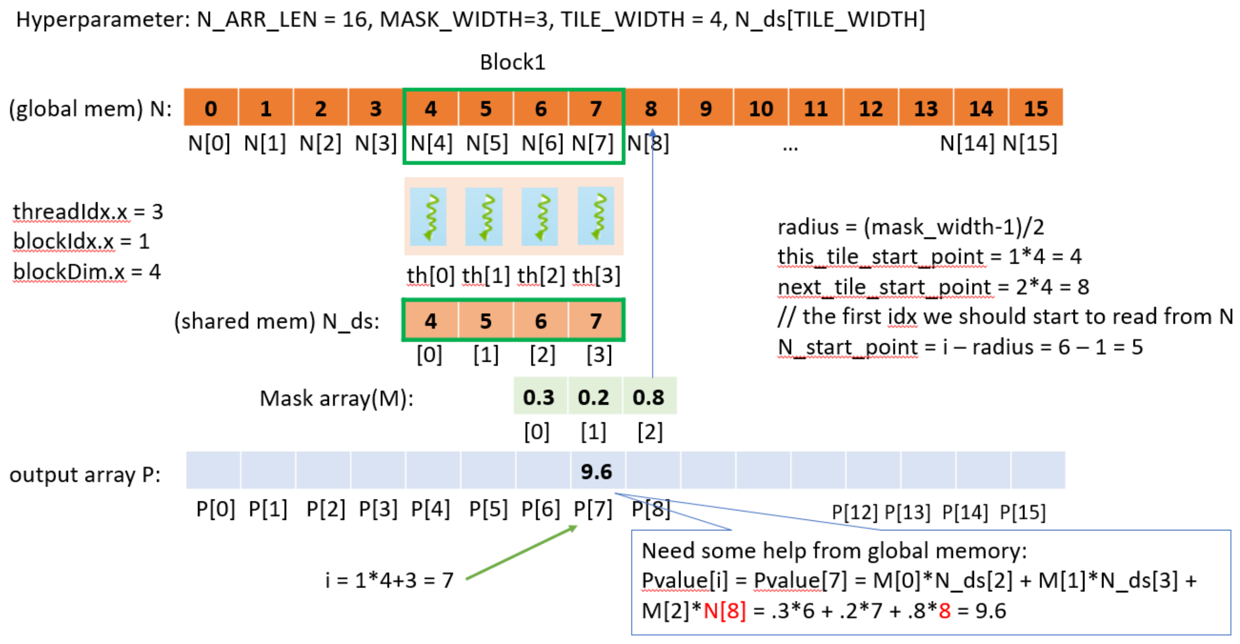
Hyperparameter: N_ARR_LEN = 16, MASK_WIDTH=3, TILE_WIDTH = 4, N_ds[TILE_WIDTH]
Block1
(global mem) N:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
N[0] N[1] N[2] N[3] N[4] N[5] N[6] N[7] N[8] ... N[14] N[15]
threadIdx.x = 3
blockIdx.x = 1
blockDim.x = 4
th[0] th[1] th[2] th[3]
radius = (mask_width-1)/2
this_tile_start_point = 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
(shared mem) 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:
9.6
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+3 = 7
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

Figure XXX: 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 3: 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 4: Benchmark Performance for Code 5(constant memory)

Note: The result in Table 1-3 and Table 4 were ran twice in 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 between shared memory and global 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.

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).

- For the parallel (and GPU) parts, how were the data and computations partitioned?
- Where does the time go? What is the arithmetic intensity?
- Overview of your optimized codes. What are the optimizations? What problems did you have?
- Experiments and results. What worked?

- Quality of work: What worked? What didn’t? Why? What limit did you hit? How many ideas from the semester did you try – how well did you understand them?