HPC on convolutional computation, CSC 746, Fall 2021

Zhuozhuo Liu*
San Francisco State University

ABSTRACT

We implemented convolution, a popular operation in modern deep neural networks and implement two methods, im2col and multi-threading, to improve its speed performance. Since computing convolution can be converted to a general matrix multiplication(GEMM) problem, methods that improve performance of GEMM could be applied to convolution as well. We found that for im2col it's faster when we optimize for read than write. We also found that OMP accelerated the speed more than 60 times of the speed of the naive solution. The next steps that is to optimize im2col and GEMM further.

1 INTRODUCTION

Convolution is an important convolution in Convolutional Neural Networks(CNN). In the project, we studied the performance optimization of convolution and implemented im2col and GEMM to accelerate the speed. We implemented 5 programs cumulatively and optimized im2col and GEMM separately. The basic convolution implementation used 4 naive for loops. The im2col(optimize for read) implementation expanded the input tensor to be a matrix that is ready to do GEMM and then applied naive GEMM on it. The im2col(optimize for write) program tried a different way to iterate for loops so that it saved memory access on write. Then we applied multi-threading on GEMM to accelerate the speed. We calculate the speedup by comparing to the basic version and analyze the result using heat maps.

2 BACKGROUND AND PREVIOUS WORK

Convolution is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other. More specifically, in deep learning it involves an input matrix (M*N matrix), a kernel matrix (K*K matrix) and an output matrix. To calculate convolution, the kernel shape will be applied on input matrix, then performs dot multiplication with the part of the input matrix that it's currently on so that we will get a dot product. The dot product will be one pixel of the output matrix. We iterate the kernel matrix to different location of the input matrix to get all output pixels.

The idea of converting convolution to GEMM was firstly used by Yangqing Jia, who is the author of Caffe [**?**], a deep-learning library, to improve the performance of convolution on CPU GPU. Consider that we have a H*W image with depth C at each input location, and a kernel with shape of (N, C, K, K), where N is number of filters and K is the size of the kernel matrix. For each location of the input, we get a K*K patch and apply N filters to it. The methods reduced a H*W input image with depth C to a matrix of size (H*W, K*K*C), which is usually known as im2col operation. The originally four dimensional kernel (N, C, K, K) can also be reshaped to (N, K*K*C) as well. With this, a convolution operation can be converted to a general matrix multiplication (GEMM).

For the performance of GEMM, tiling/block algorithm can largely improve the performance. Tiling create small tiles of matrix and

*email:zliu15@mail.sfsu.edu

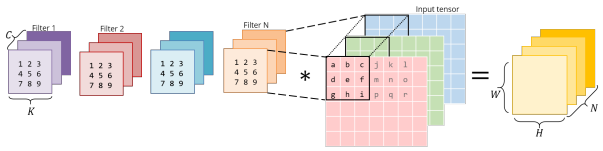

Figure 1: kernel with shape of (N, C, K, K) applies on H*W image with depth C. Credit to Sahni. [**?**]

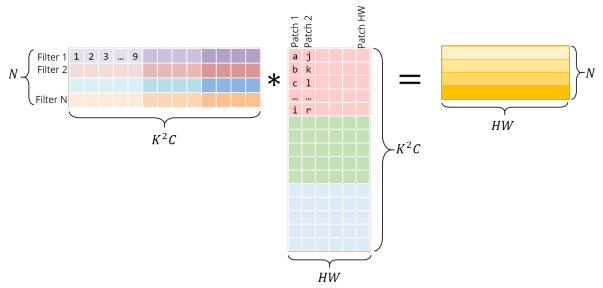

Figure 2: Convert a convolution to GEMM. Credit to Sahni. [**?**]

combine together with method of copying to local memory to exploit spatial locality. [**?**] Copying to local memory is to copy tiles to a buffer and modify code to use data directly from the buffer instead of reading or writing to the memory every time. Gunnels et al. computed C := AB + C (A, B, C are matrices) by storing part of the matrix contiguously in some packed format that fits in cache memory. Goto and Geijn improved the algorithm and made it fits multilevel memories. [**?**]

Loop unrolling is a technique which improves the performance by reducing the number of loop iterations. We will manually increase the statements in each loop, which may save some time of checking the conditions. In addition, loop unrolling, in conjunction with other code optimizations, can increase instruction-level parallelism and improve memory hierarchy locality. [**?**]

Parallel computing was largely used in GEMM optimization in previous studies and thus can be used to optimize our program as well.

3 IMPLEMENTATION

We optimized im2col and omp seperately. There are 5 programs in total that did the job of convolutional computation. Program 2 and 3 are 2 ways to optimize im2col. Program 2 was optimized the locality for read. Program 4 built upon program 2 and optimized GEMM. Program 3 was optimized the locality for write. Program 5 built upon program 3 and optimized GEMM.

Since convolution is very complicated (e.g., 6-7 for loops). We simplified the program by hard-coded a few parameters. We preset INPUT_CHANNEL to 3, since the input tensors are normally of 3 channels: red, green and blue. We preset OUTPUT_CHANNEL to 1. We preset FILTER_DIMENSION to 3.

We will let users to pass 2 critical parameters that defined the size of the input: the dimension of the input tensor, and how many filters/kernels it has to apply to each tensor. for example, ./convolution 10

100ẅill calculate a matrix of size 10*10, with 3 channels, applying 100 filter of size 3*3.

For all programs, inputs are the same: 3 input tensors, n filters, the dimension of each tensor and the number of filters. Input tensors and filters will be filled with random numbers to mock up an image input, so we don't need to pass a image.

3.1 Program 1: Naive convolution

This program (see Listing 1) implemented a basic convolution.

```
void basic_convolution(
    float *in_data,
    float *out_data,
    float *filter,
    int channel_dimension,
    int total_filters) {
        for (int filter_count = 0; filter_count < total_filters; filter_count++) {
            for (int channel_count = 0; channel_count < INPUT_CHANNEL; channel_count++) {
                for (int i = 0; i < channel_dimension; i++) {
                    for (int j = 0; j < channel_dimension; j++) {
                        out_data[filter_count * channel_dimension * channel_dimension
                            + (channel_dimension * i + j)] += convolve_pixel(
                                in_data, i, j,
                                channel_dimension,
                                filter,
                                channel_count,
                                filter_count
                            );
                    }
                }
            }
        }
    }
```

Listing 1: the implementation of naive convolution

3.2 Program 2: im2col(optimize for read) + GEMM

We expanded the input by calling im2col() (see Listing 2). Then we did a basic GEMM between the expanded input data and filters.

3.3 Program 3: im2col(optimize for write) + GEMM

The previous program iterated i and j based on the input instead of output so we thought it might waste some of the locality when writing on the output. We then tried a different way by changing the iteration order to iterate i and j based on the output to benefit writing to the input (see Listing 3). We hope this will accelerate the program.

3.4 Program 4: im2col(optimize for read) + GEMM(optimized)

After optimized im2col, we next optimized GEMM (see Listing 4). There are many ways to do it, what we did is to implement multi-threading by using OpenMP. This program was built on program 2.

3.5 Program 5: im2col(optimize for write) + GEMM(optimized)

Similar to program 4, we optimized GEMM (see Listing 4) by OMP. This program was built on program 3.

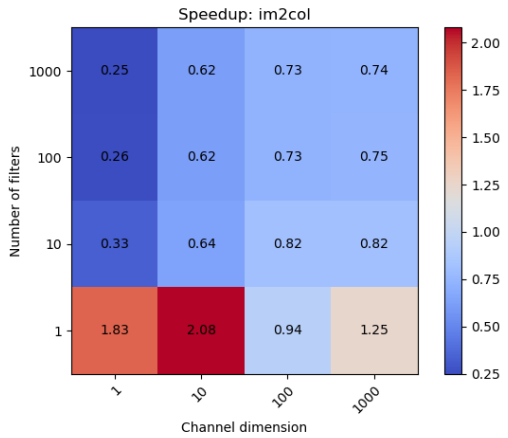

Figure 3: The speedup of program 2 on 16 configuration. When the number of filters are larger than 10, im2col actually impeded the speed. This figure was produced by the program `plot.py` file.

4 Results

4.1 Computational platform and Software Environment

All tests were running on Cori KNL node. The processor was Intel Xeon Phi Processor 7250. The clock rate was 1.4 GHz. The size of L1 cache was 64 KB. The size of L2 cache was 1MB. The memory size was 96 GB for DDR4 and 16 GB for MCDRAM. Cmake version was 3.14.4. GCC version was 8.3.0. Optimization level was O3 by default. OpenMPI version was 4.0.2. [1]

4.2 Methodology

We first validated the program by comparing the output matrices of 4 programs. We then measured the elapsed run time of the program in second. We used the run time of the basic version as a baseline and calculated the speedup for program 2, 3 and 4 by using elapsed_time(basic) / elapsed_time(current program).

For program 2, 3, and 4, we also recorded the run time of im2col stage and GEMM state respectively.

we tested the combination of 2 parameters: Dimension of source matrices (D): 1, 10, 100, 1000, and Number of filters (NF): 1, 10, 100, 1000. So there were 16 combinations tested.

4.3 Program 2 im2col(optimize for read) + GEMM

Program 2 optimized the locality for read. Figure 3 shows the heat map of the speedup on 16 configurations. It shows that when the number of filters are larger than 10, im2col actually impeded the speed. More filters brought more operations. It indicated that the naive im2col and GEMM didn't accelerate the performance. We will need to optimize im2col and GEMM to reach better performance.

4.4 Program 3 im2col(optimize for write) + GEMM

Program 3 optimized the locality for write. Figure 4 shows the heat map of the speedup on 16 configurations. As we can see, it shows the similar pattern from the optimized version and it didn't improve the speed significantly. Moreover, when we check the run time for the im2col operation separately, we found that it's much slower than the im2col operation in program 2. Since im2col is a independent operation that will not be influenced by the rest of the operations, we can include that the program didn't improved the performance. It might because that the locality for read is more important than write since we have more read operations.

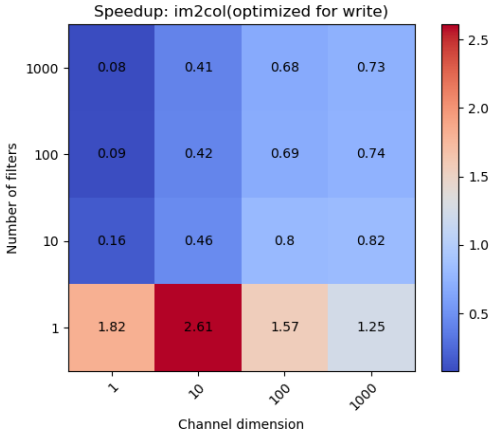

Figure 4: The heat map of the speedup of program 3 on 16 configurations. It shows the similar pattern from the optimized version and it didn't improve the speed significantly. This figure was produced by the program `plot.py` file.

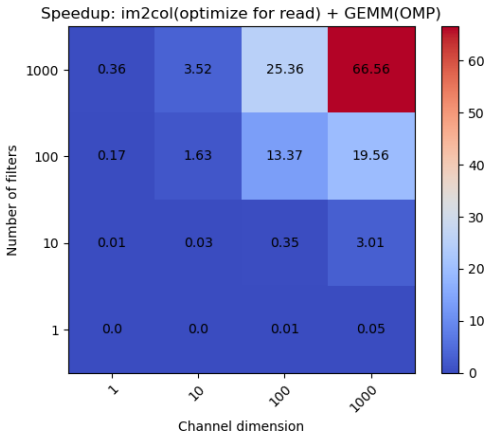

Figure 5: The performance of Program 5. It was paralleled well on larger input on the maximum speed of 66. This figure was produced by the program `plot.py` file.

4.5 Program 4 im2col(optimize for read) + GEMM(optimized)

Program 4 was built on program 2 and optimized GEMM by using OMP to parallel it. Figure 5 shows that after converting to GEMM, it was paralleled well on larger input (i.e., large NF and D) on the maximum speed of 66. The effort to do im2col can only be paid off when doing larger size of GEMM. It didn't reach the ideal speedup because of the serial part (im2col) and when different threads write to the cache.

4.6 Program 5 im2col(optimize for write) + GEMM(optimized)

Program 5 was built on program 4 and optimized GEMM by using OMP to parallel it. Figure 6 shows that it performed worse than program 4 since when GEMM is optimized, the performance difference between 2 versions of im2col become obvious.

4.7 Next steps

Our findings shows that the combination of im2col and GEMM accelerated the speed of convolution. The next steps that we shall

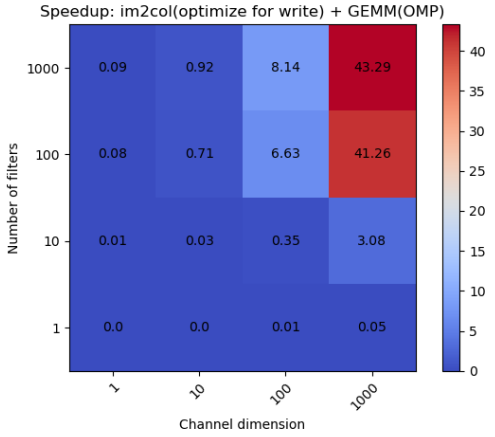

Figure 6: The performance of Program 5. It performed worse than program 4 since when GEMM is optimized. When GEMM is optimized, the performance difference between 2 versions of im2col become obvious. This figure was produced by the program `plot.py` file.

focus on is to optimize im2col and GEMM further. We may optimize the im2col operation by reducing memory access such as continuous memory address read. We may optimize GEMM by tiling, vectorization, unrolling, and many existing methods that were used to optimize GEMM.

References

[1] https://docs.nersc.gov/systems/cori/. Cori system specification.

```
void im2col(float *in_data, float *im2col_data,
    float *filter, int channel_dimension) {
        int n_patch = channel_dimension *
    channel_dimension;
        // im2col: convert input data to col data
        for (int channel_count = 0; channel_count
    < INPUT_CHANNEL; channel_count++) {
            int im2col_start_i = channel_count *
    FILTER_DIMENSION * FILTER_DIMENSION;
            int in_start_i = channel_count *
    channel_dimension;
            for (int i = 0; i < channel_dimension;
     i++) {
                for (int j = 0; j <
    channel_dimension; j++) {
                    int kernel_index = 0;
                    for (int x_offset = -1;
    x_offset < FILTER_DIMENSION - 1; x_offset++) {
                        for (int y_offset = -1;
    y_offset < FILTER_DIMENSION - 1; y_offset++) {
                            int new_i = i +
    x_offset;
                            int new_j = j +
    y_offset;
                            int im2col_i =
    im2col_start_i + kernel_index;
                            int im2col_j = i*
    channel_dimension + j;
                            if (new_i < 0 || new_i
     >= channel_dimension || new_j < 0 || new_j >=
     channel_dimension) {
                                im2col_data[
    im2col_i * n_patch + im2col_j] = 0.0;
                            } else {
                                im2col_data[
    im2col_i * n_patch + im2col_j] = in_data[(
    in_start_i + new_i)*channel_dimension+new_j];
                            }
                            kernel_index++;
                        }
                    }
                }
            }
        }
    }

void im2col_convolution(...) {
        float *im2col_data = (float *)malloc(
    sizeof(float) * n_rows * n_patch);
        // Convert image to column
        im2col(in_data, im2col_data, filter,
    channel_dimension);
        // Vector matrix multiplication
        dgemm(...);
}
```

Listing 2: the implementation of convolution using im2col

```
void im2col_optimized_locality(float *in_data,
    float *im2col_data, float *filter, int
    channel_dimension) {
        int cols = channel_dimension *
    channel_dimension;
        int rows = FILTER_DIMENSION *
    FILTER_DIMENSION;
        // im2col: convert input data to col data
        for (int channel_count = 0; channel_count
    < INPUT_CHANNEL; channel_count++) {
            int im2col_start_i = channel_count *
    FILTER_DIMENSION * FILTER_DIMENSION;
            int in_start_i = channel_count *
    channel_dimension;
            for (int i = 0; i < rows; i++) {
                int start_i = (int)i /
    FILTER_DIMENSION;
                int start_j = i % FILTER_DIMENSION
    ;
                int j = 0;
                while (j < cols) {
                    int new_i = start_i + (int)j /
     channel_dimension - 1;
                    int new_j = start_j + j %
    channel_dimension - 1;
                    if (new_i < 0 || new_i >=
    channel_dimension || new_j < 0 || new_j >=
    channel_dimension) {
                        im2col_data[(
    im2col_start_i+i)*cols+j] = 0.0;
                    } else {
                        im2col_data[(
    im2col_start_i+i)*cols+j] = in_data[(
    in_start_i+new_i)*channel_dimension+new_j];
                    }
                    j++;
                }
            }
        }
    }
```

Listing 3: the implementation of im2col which optimized writing

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < total_filters; i++) {
   for (int j = 0; j < cols; j++) {
       float temp_out = 0.0;
       for (int k = 0; k < rows; k++) {
           temp_out +=
           filter[i*FILTER_DIMENSION*
    FILTER_DIMENSION*INPUT_CHANNEL+k]
            * M[k*cols+j];
       }
       out[i*cols+j] = temp_out;
   }
}
```

Listing 4: the implementation of GEMM which utilized multi-threading