2019 Spring CS141 Final Report
Deep Learning & Optimizing Matrix Multiplication

Zishen Wan

May 2019

1 Introduction

Neural networks have become ubiquitous in applications including computer vision, speech recognition and natural language processing. Matrix-Matrix multiplication (M×M) is a basic building block in these wide range of neural networks and deep learning applications.

In convolutional neural networks (CNN), fully connected layers are implemented with matrix multiplication, and more than 96% of the connections are in the FC layers [1]. Figure 1 shows how a matrix multiplication is used for the FC layer. The height of the filter matrix is the number of 3-D filters(M) and the width is the number of weights per 3-D filter (input channels(C) × width (W) × height (H)); the height of the input feature maps matrix is the number of activations per 3-D input feature map ($C \times W \times H$), and the width is the number of 3-D input feature maps (N); finally, the height of the output feature map matrix is the number of channels in the output feature maps (M), and the width is the number of 3-D output feature maps/batch size (N). Similarly, the CONV layer in a DNN can also be mapped to a matrix multiplication using a relaxed form of the Toeplitz matrix.

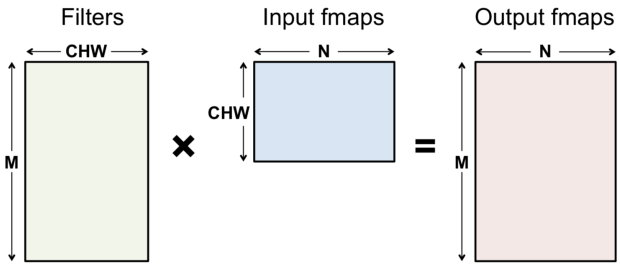

Figure 1: Mapping to matrix multiplication for fully connected layers.

In recurrent neural network (RNN), M×M are performed on the new input and the hidden state at each time step, producing a new hidden state and the output. LSTM is a widely used structure of RNN cell that provides more complex hidden unit computation. Each LSTM cell can be decomposed into eight Matrix Multiplication operations, two for each: input gate, forget gate, output gate, and one temporary memory cell.

During M×M, the operation time and memory access is usually the bottleneck[2], especially when the matrix is larger than the cache capacity. Considering the case that $\mathbf{B} \times \mathbf{C} = \mathbf{A}$, if each matrix is 4096×4096 and each element is 4-byte, then each matrix is 64MB. With a naive M×M implementation, it will take hours to get the result with 16MB cache. Such speed is unacceptable for deep learning applications, especially for real-time applications that are latency-sensitive, such as pedestrian detection in an autonomous vehicle. Therefore, optimizing and speedup matrix-multiplications is very important and something we should consider when building AI-enabled systems.

2 Baseline: Naive Matrix Multiplication

source code: void naivemm

Our baseline is Naive Matrix Multiplication. Figure 2 [3] shows how rows of **B** are combined with columns of **C**. This combination involves the element-wise multiplication of the values from **B**'s row and the **C**'s column and final summing of all the elements of the resulting vector, i.e., doing an inner product. Each such inner product produces the value for one element of **A**. The pseudo-code is listed below.

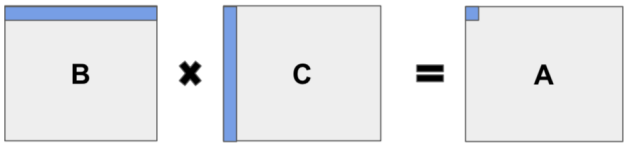

Figure 2: Naive Matrix Multiplication

Algorithm 1 Naive Matrix Multiplication

Input: Matrix **B**, **C**
Output: Matrix **A**

```
for (i = 0; i < out_rows; i++) do
    for (j = 0; j < out_cols; j++) do
        for (k = 0; k < in_cols; k++) do
            A[i][j] = B[i][k] × C[k][j]
```

We test the execution time of naive M×M on different input matrix sizes, the results are summarized in Table 1.

Table 1: Execution time of naive M×M of different sizes

	8x8	32x32	128x128	256x256	512x512	1024x1024
naive	0.000004	0.000222	0.009652	0.086646	0.852689	16.331204

We can find that the execution time is short when the size of matrix is small, but will increase dramatically when the size of matrix becomes larger. If the naive computation is ordered such that a single row of **B** is combined successively with each of the columns of **C**, then there is apparently good reuse of the elements of the row of **B**. However, considering the memory hierarchy in Figure 3[3], the size of a row in **B** is often larger than the memory, e.g., cache, closest to the compute units. This results in poor reuse, because values from **B** in the small memory cannot be held long enough to be available to be reused for computation on the next column of **C**. Therefore they must be reloaded from the next level of the memory hierarchy resulting in significant inefficiency.

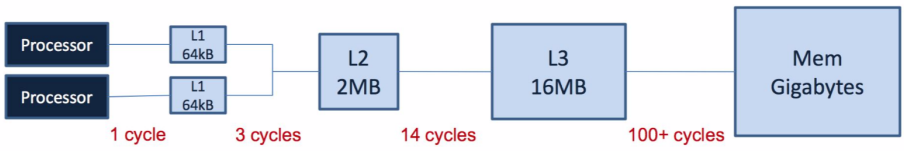

Figure 3: Memory Hierarchy of Computer

Through examining the algorithm, we can find: in the first loop, we read row i of **B** into memory; in the second loop, we read $A(i, j)$ into memory and read column j of **C** into memory; in the third loop, we do multiply-and-add (MAC) operation and write $A(i, j)$ back to slow memory. Based on previous analysis and assuming we fetch and write through DRAM, the total slow memory fetches we actually have are $N^3(\mathbf{C})+N^2(\mathbf{B})+2N^2(\text{Write \& Read } \mathbf{A}) = N^3+3N^2$, where N is the size of input square matrix. If $N = 1024$, we totally fetch 1076887552 (˜1077 Million) times from DRAM. Assuming each DRAM fetch cost 100 cycles and the frequency is 2GHz, then 1077 Million DRAM fetches will take $1077 \times 10^6 \times 100 \times (1/(2 \times 10^9))$ = 53.85s. If $N = 4096$, we will have about 68 Billion slow memory fetches, which will take 1 hour to run task under the same computer condition.

3 Tiled Matrix Multiplication

source code: void tiling

To ameliorate the memory inefficiencies that result from calculating the inner products (naive method) on full rows and columns, we can partition or tile the computation to fit in the various levels of the memory hierarchy. The principle behind tiling is illustrated in Figure 4[3], where the inner products are done on a 2-D partition, or tile, of the full matrices. For each pair of tiles in the matrices, the same naive approach can be employed on the partial rows of matrix **B** and partial columns in the matrix **C** to create a tile of partial results in the matrix **A**. As computations for all the pairs of tiles are done the subsequent partial results are added to the partial results in **A** from previous partial result computations. If

a single tile of **B** is used repeatedly to create a series of partial results, and if the tile is small enough to be held in the memory closest to the compute units, then reuse in that memory will be higher.

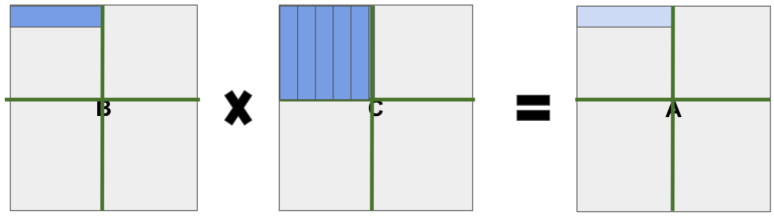

Figure 4: Tiled Matrix Multiplication

Algorithm 2 Naive Matrix Multiplication

Input: Matrix **B**, **C**
Output: Matrix **A**

1: **for** $(it = 0; it < out\_rows; it+ = T)$ **do**
2: **for** $(jt = 0; j < out\_cols; jt+ = T)$ **do**
3: **for** $(kt = 0; k < in\_cols; kt+ = T)$ **do**
4: **for** $(i = it; i < it + T; i + +)$ **do**
5: **for** $(j = jt; j < jt + T; j + +)$ **do**
6: **for** $(k = kt; k < kt + T; k + +)$ **do**
7: $\mathbf{A}[i][j] = \mathbf{B}[i][k] \times \mathbf{C}[k][j]$

We test the execution time of tiling matrix multiplication on different input matrix sizes and tile sizes. The results are summarized on Table 2 and Figure 5.

Table 2: Execution time of tiled M×M of different input sizes and tile sizes

	8x8	32x32	128x128	256x256	512x512	1024x1024
naive	0.000004	0.000222	0.009652	0.086646	0.852689	16.331204
tiling (tile size=N/2)	0.000003	0.000211	0.009269	0.078224	0.851741	7.519277
tiling (tile size = N/4)	0.000004	0.000212	0.009382	0.076737	0.740546	7.444965
tiling (tile size = N/8)	0.000006	0.000214	0.009281	0.075443	0.628263	7.345624

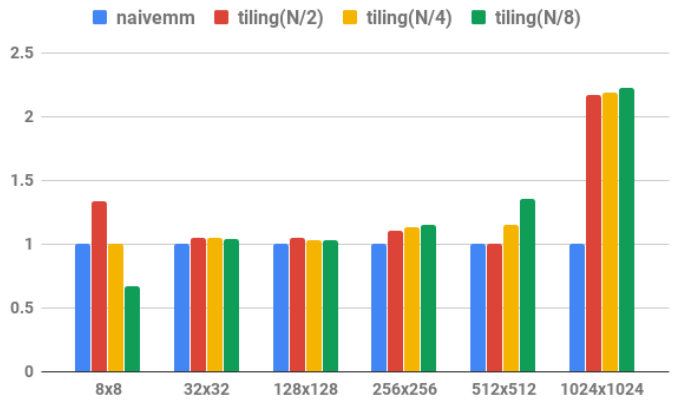

Figure 5: Speed up for tiled matrix multiplication under different matrix sizes

We can find that when the size of input matrix are small, the execution time for naive and tiled method are almost same, but when size = 1024, tiled method has significant advantages and can achieve 2.2x speed up. As analyzed before, this is because after titled the partition data can be stored in cache and we don't have to fetch them from DRAM, which can reduce execution time. For different tile sizes, we find that N/8 is most efficient, because in a certain range, the smaller partitioned data, the higher probability they can be all stored in cache, resulting in less execution time.

4 Extra Credit: Permutation Matrix Multiplication

source code: (1)void permutation; (2)void tiling_permutation

In naive and tiled M×M method, we find every time we fetch different elements of **B** and **C** and do MAC operation, which leave space for us to explore more efficient data reuse method. To further optimize M×M efficiency, we can fetch the same element from **B** every time and perform multiplication with all the elements in **C**. After that, we fetch the second elements from **B** and do the same operation as before.

Looking at the code, it's obvious that the index of matrix has nothing with the order of *for* loop. So we can exchange the last two *for* loops, in this case, index of **B** will keep the same inside the innermost loop and we don't need to fetch it from memory, which increase the efficiency. The results are summarized in Table 3 and Figure 6.

Algorithm 3 Permutation Matrix Multiplication

Input: Matrix **B**, **C**
Output: Matrix **A**

```
1: for (i = 0; i < out_rows; i++) do
2:     for (k = 0; k < in_cols; k++) do
3:         for (j = 0; j < out_cols; j++) do
4:             A[i][j] = B[i][k] × C[k][j]
```

Table 3: Execution time of permutation M×M of different input sizes

	8x8	32x32	128x128	256x256	512x512	1024x1024
naive	0.000004	0.000222	0.009652	0.086646	0.852689	16.331204
permutation	0.000003	0.000222	0.009127	0.070145	0.552844	4.412685
permutation + tiling*	0.000007	0.000217	0.009215	0.069065	0.540312	4.391978

* tile size = N/8

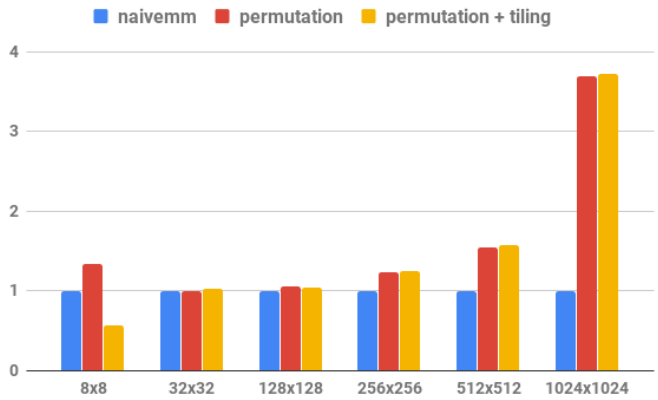

Figure 6: Speed up for permutation matrix multiplication under different matrix sizes

From 3 and Figure 6, we can find permutation M×M method have significant advantages over naive and tiling, which can achieve 3.70x speed up than naive method when N=1024. Combining permutation and tiling, we can achieve highest 3.71x speed up for N=1024 now. Therefore, optimizing data reuse will significantly reduce execution time.

5 Extra Credit: Loop Invariant Matrix Multiplication

source code: (1) void naivemm_invariants_v1; (2) void permutation_invariants_v1; (3) void tiling_invariant_v1; (4) void naivemm_invariants_v2; (5) void permutation_invariants_v2; (6) void tiling_invariant_v2;

Apart from multiplication of elements in matrix, the calculation of matrix's index will also take up execution time. Looking into the code of naive method, we find that *a_index* keeps the same inside innermost loop. However, we still calculate it every time, which is useless and time consuming. Therefore, we can move *a_index* into the second loop so we don't need to calculate it in the innermost loop. We call it *loop invariant v1* algorithm.

Algorithm 4 Loop Invariant v1 Matrix Multiplication

Input: Matrix **B**, **C**
Output: Matrix **A**

```
1: for (i = 0; i < out_rows; i++) do
2:     for (j = 0; j < out_cols; j++) do
3:         a_index = i * out_cols + j
4:         for (k = 0; k < in_cols; k++) do
5:             b_index = i * in_cols + k
6:             c_index = k * out_cols + j
7:             A[a_index] += B[b_index] × C[c_index]
```

Inspired by *loop invariant v1* algorithm, we can minimize unnecessary index calculations to the greatest extent. So we come up with *loop invariant v2* algorithm, making sure we only calculate the index parameters when they changes.

Algorithm 5 Loop Invariant v2 Matrix Multiplication

Input: Matrix **B**, **C**
Output: Matrix **A**

```
1: for (i = 0; i < out_rows; i++) do
2:     a_i_outcols = i * out_cols
3:     b_i_incols = i * in_cols
4:     for (j = 0; j < out_cols; j++) do
5:         int a_index = a_i_outcols + j
6:         for (k = 0; k < in_cols; k++) do
7:             b_index = b_i_incols + k
8:             c_index = k * out_cols + j
9:             A[a_index] += B[b_index] × C[c_index]
```

We test the execution time of loop invariant algorithm on naive, titled and permutation method, the results are summarized in Table 4 and Figure 7. We can find loop invariant algorithm can further reduce run time and improve efficiency. We can achieve highest 4.54x speed up for N=1024 now.

Table 4: Execution time of loop invariant M×M of different input sizes

	8x8	32x32	128x128	256x256	512x512	1024x1024
naive	0.000004	0.000222	0.009652	0.086646	0.852689	16.331204
naive + loop invariant v1	0.000003	0.000187	0.008808	0.065088	0.768943	14.361044
naive + loop invariant v2	0.000004	0.000183	0.008617	0.063601	0.715668	12.711219
tiling*	0.000006	0.000214	0.009281	0.075443	0.628263	7.345624
tiling*+ loop invariant v1	0.000007	0.000188	0.008681	0.067209	0.547763	6.512802
tiling*+ loop invariant v2	0.000004	0.000182	0.008632	0.064516	0.527266	6.297586
permutation	0.000003	0.000222	0.009127	0.070145	0.552844	4.412685
permutation + loop invariant v1	0.000003	0.000186	0.008034	0.057647	0.447837	3.715339
permutation + loop invariant v2	0.000003	0.000183	0.007951	0.057388	0.450879	3.596119

* tile size = N/8

6 Extra Credit: Loop Unrolling Matrix Multiplication

source code: (1) void naivemm_unrolled; (2) void tiling_unrolled; void permutation_unrolled; (3) void tiling_invariant_v2_unrolled; (4) void permutation_invariants_v2_unrolled

Considering the assembly code of M×M implementation, MACs operation/cycle ≈ 1/6, which means we have loop iteration overhead. The loop overhead can be amortized over more computation, so we can perform two MACs computation in the innermost loop every time. Now MACs operation/cycle ≈ 2/8 = 1/4. We call it *loop unrolling* algorithm.

We measure execution time of loop unrolling algorithm on naive, tiling, permutation and loop invariant method. We test it for unrolling step size=2 and 4 separately, the results are summarized in Table 5 and Figure 8. We can find loop unrolling algorithm can significantly reduce run time, and s=4 is better than s=2 because it can further reduce loop

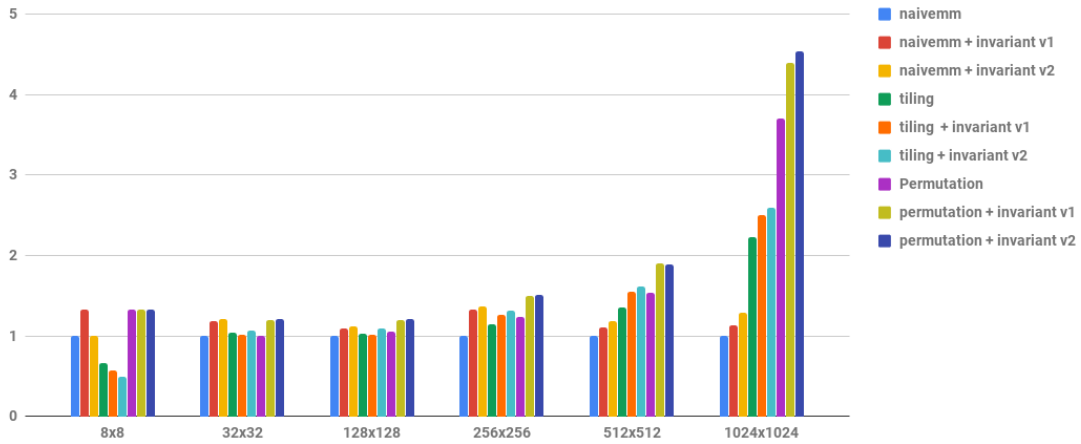

Figure 7: Speed up for loop invariant matrix multiplication under different matrix sizes

Algorithm 6 Loop Unrolling Matrix Multiplication

Input: Matrix **B**, **C**
Output: Matrix **A**

```
for (int i = 0; i < out_rows; i++) do
    for (int j = 0; j < out_cols; j++) do
        for (int k = 0; k < in_cols; k = k + 2) do
            int a_index = i * out_cols + j
            int b_index = i * in_cols + k
            int c_index = k * out_cols + j
            A[a_index] = A[a_index] + B[b_index] × C[c_index] + B[b_index + 1] × C[c_index + out_cols]
```

iteration overhead. Combining all permutation, invariant v2 and unrolling method, we finally achieve **1.846584s** for N=1024 matrix multiplication, which is **8.844x** speed up compared with naive method!

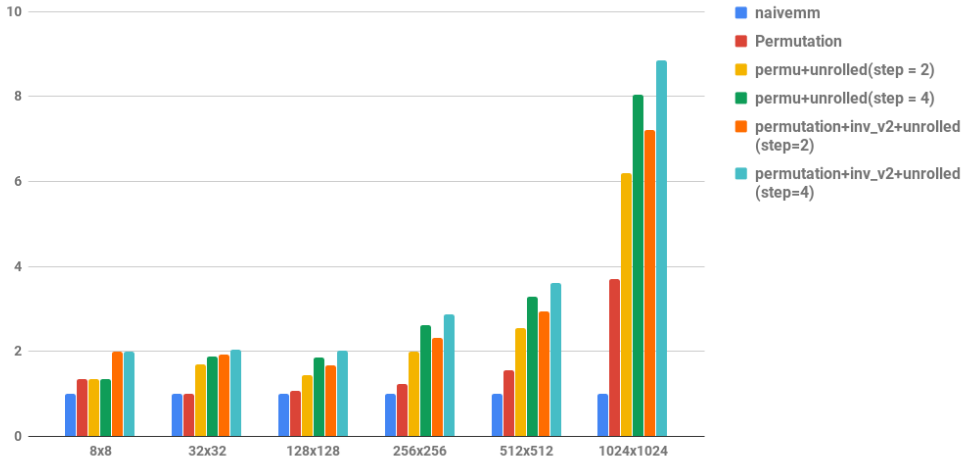

Figure 8: Speed up for loop unrolled matrix multiplication under different matrix sizes

7 Summary and Further work

Matrix multiplication is ubiquitous in deep learning and optimizing it is very important for AI-enabled systems. In this project, we employ tiling, permutation, loop invariant and loop unrolling method to maximize data reuse and minimize unnecessary multiplication operation. Compared with N=1024 naive implementation (16.331204s), we finally achieve 1.846584s execution time and 8.844x speed up.

Table 5: Execution time of loop unrolled M×M of different input sizes and unrolled sizes

	8x8	32x32	128x128	256x256	512x512	1024x1024
naive	0.000004	0.000222	0.009652	0.086646	0.852689	16.331204
naive + unrolled (s[*]=2)	0.000003	0.000187	0.008808	0.065088	0.768943	14.361044
naive + unrolled (s=4)	0.000002	0.000098	0.005702	0.039117	0.463527	9.501525
tiling[**]	0.000003	0.000211	0.009169	0.078224	0.881741	7.519277
tiling + unrolled (s=2)	0.000008	0.000156	0.007094	0.048602	0.417193	4.813046
tiling + unrolled (s=4)	0.000009	0.000119	0.005507	0.037015	0.312987	3.876697
tiling + invariant v2 + unrolled (s=4)	0.000009	0.000121	0.005262	0.034388	0.296961	3.589833
permutation	0.000003	0.000222	0.009127	0.070145	0.552844	4.412685
permutation + unrolled (s=2)	0.000003	0.000131	0.006746	0.432880	0.336451	2.637761
permutation + unrolled (s=4)	0.000003	0.000119	0.005208	0.033122	0.258809	2.029511
permutation + invariant v2 + unrolled (s=2)	0.000002	0.000115	0.005817	0.037329	0.290623	2.263806
permutation + invariant v2 + unrolled (s=4)	0.000002	0.000109	0.004771	0.030176	0.235661	**1.846584**

[*] s: unrolling step size
[**] tile size = N/8

There are lots of other tricks we can use to optimize matrix multiplication, such as pruning, quantization, precision and computation transform optimizations (e.g Gauss' transform, Strassens transform and Winograd transform).

References

[1] I. Sutskever A. Krizhevsky and G. E. Hinton. Imagenet classifica- tion with deep convolutional neural networks. In *NIPS*, 2012.

[2] S.Yao K.Guo B.Li E.Zhou J.Yu T.Tang N.Xu S. Song Y. Wang J.Qiu, J.Wang and H. Yang. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*, 2016.

[3] U.Gupta L.Pentecost J.Zuckerman Z.Yedidia D.Brooks, V.Reddi. Cs141 lecture notes. 2019.