Strassen's Performance on Convolutional Neural Network

Chihang Wang

11th January 2016

1 Introduction

This document reports the performance of implemented Strassen's algorithm on the actual Convolutional Neural Network. In the previous report, it is found that for matrices of size larger than (2048, 2048, 2048), optimized Strassen's algorithm starts to beat the performance of cache-blocking BLAS algorithm. However, It is found that previously optimized Strassen's algorithm is slower than the normal cache-blocking algorithm in the actual convolutional neural networks. This is due to the size of the matrices in the convolutional neural network. The following sections describe the details.

2 Performance on LeNet5 & Cifar10

The performance are the same for both algorithm. This is because for these two neural networks, the parameters are so small, that the base case is called immediately without invoking the recursion.

3 Performance on VGG16

The algorithms are then compared on the VGG model. It is found that Strassen algorithm returns slightly slower results. The size of matrix multiplication involved in VGG are as the follow:

Layer	M	N	K
conv1	64	50176	576
conv2	128	12544	576
conv3	256	3136	1152
conv4	256	3136	2304
conv5	512	784	2304
conv6	512	784	4608
conv7	512	196	4608
conv8	512	784	4608

The base-case decision is such that stop recursion and call cache-blocking algorithm soon as one of (M, N, K) is below 256. This means the first 2 and last 2 layers will call cache-blocking algorithm right away. For the other 4 layers, the running time is collected and presented below:

Layer	cache-blocking	Strassen
conv3	90	105
conv4	180	209
conv5	90	118
conv6	180	224

Intuitively, the difference is due the overhead of blocking setup, addition, etc. In order to have a better understanding, valgrind is used to determine the instruction count in each of the functions involved. The outcome of conv6 is presented below:

	cache-blocking	Strassen
Total Instruction	825M	780M
Actual Multiplication	792M	615M
Total Memory read	106M	140M
Total Memory write	14M	41M

This result is very suggestive. Although the number of instructions is less, especially the number of instructions spent in the micro-kernel of multiplication routine. However, the number of reads and writes is far more in the Strassen's algorithm. Although both program reports a L1 cache-miss of 13%, but the actual number of stall is greater in Strassen, hence it becomes slower.

P.S. The "Actual Multiplication" is not the number of assembly instruction doing multiplication. But the number of instructions within the cache-blocking's micro-kernel which does a very tight loop of multiplication. All of the actual multiplication is done within the micro-kernel.

4 Reason of the Overhead

The reason why cache-blocking algorithm is so fast is because there is a packing stage, which moves a row stride of A, and a column stride of B into contiguous memory space. Therefore, in the inner loop, all the memory accesses are regular and hence prefetching can be enabled.

(.... detailed ommited, can be added)

For my laptop, the configuration of BLAS decided to pack a $(128, 384)$ entries from A, and $(384, 4096)$ entries from B. Then, in which iteration of inner loop, $(8, K)$ from A is multiplied with $(K, 8)$ from B, and updating $(8, 8)$ of C.

Now, when K is a multiple of 384, the inner loop will working at its optimal speed, as the setup instructions will contribute to a smaller factor. However, for other sizes of K, the proportional of overhead will be slightly larger.

However, the real difference comes from the fact that Strassen needs to make 7 new utility matrices, all these needs packing of the different matrices constructed from addition/subtraction of submatrices. Since they are different, there is no reuse of packed stride as well.

Furthermore, as said previously, the optimal performance of cache-blocking is reached for size (128, 384, 4096), this means the original matrix needs to be of size around (256, 768, 8192) for the Strassen algorithm to be 7/8x faster than cache-blocking alone. For slightly smaller matrices, the performance is also slightly better, but we are aiming at something around (2048, 2048, 2048) from emperical study, which is not generally the case in convolutional neural network.

5 Possible optimization

Since most of the time is spent on packing of small matrices, we could try to pre-pack the matrices into row sride for A, and column stride for B. This means, for (8, 8) matrix of A:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

The previous memory layout will be:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 1 & 2 & 3 & 4 & 5 & 6 & 1 & 2 & 3 & 4 & 5 & 6 & 1 & 2 & 3 & 4 & 5 & 6..... \end{bmatrix}$$

Now it will be (where zero is added where necessary):

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 5 & 6 & 0 & 0 & 5 & 6 & 0 & 0 & 5 & 6..... \end{bmatrix}$$