Strassen's Matrix Multiplication in Parallel Environments

Yulei Liao 2015201953

Renmin University of China
2015201953@ruc.edu.cn

***Abstract*—I focus on how to calculate $A \times B$ quickly and accomplish efficient single-precision matrix multiplication in parallel environments of Strassen's algorithm. Supposing A, B are $n \times n (n = 2^k)$ matrices, I give the best parameters of this algorithm in different parallel environments.**

I. Introduction

Matrix multiplication is an important problem in parallel computing. In single-core algorithm, the complexity of classical three-loop algorithm is $O(n^3)$. The complexity of Strassen's original algorithm [1] and Winograd's variant [2] of this algorithm are $O(n^{2.81})$.

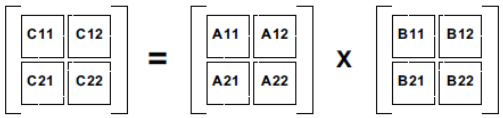

Fig. 1. Block Decomposition of A, B, C

Stassen's algorithm computes the product C of two matrices A and B by first decomposing each matrix into 4 roughly equal size blocks as in Fig. 1. Strassen's algorithm [1] computes C by the following equations:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \qquad C_{11} = M_1 + M_4 - M_5 + M_7 \\
M_3 &= A_{11}(B_{12} - B_{22}) \qquad C_{12} = M_3 + M_5 \\
M_4 &= A_{22}(B_{21} - B_{11}) \qquad C_{21} = M_2 + M_4 \\
M_5 &= (A_{11} + A_{12})B_{22} \qquad C_{22} = M_1 - M_2 + M_3 + M_6 \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

We note that this recursive algorithm needs lots of temporary matrices such as M_i. To solve this problem, Douglas el al. [3] provide an implementation of Strassen's algorithm which uses two temporary matrices at each level of the recursion.

II. Methodology

Though the complexity of Strassen's algorithm is $O(n^{2.81})$, the application and release of storage space for temporary matrices and recursion cost too much time. Junjie Li el al. [4] describe an efficient algorithm and the best parameters on GPUs. I realize it in both CPUs and GPUs and find out the best parameters in the experiment. Kernels are the parallel calls.

A. Zero-Level Adaptation

Zero-level adaptation is the classical $O(n^3)$ matrix multiplication. TABLE I gives the sequence of kernel calls.

TABLE I
Zero-Level Adaptation

Step	Computation	Kernel
1	$C = A \times B$	$mul(A, B, C)$

B. One-Level Adaptation

One-level adaptation uses Strassen's method non-recursively. TABLE II gives the sequence of kernel calls.

TABLE II
One-Level Adaptation

Step	Computation	Kernel
1	$(A_{11}, A_{12}, A_{21}, A_{22}) = A$	$split(A_{11}, A_{12}, A_{21}, A_{22}, A)$
2	$(B_{11}, B_{12}, B_{21}, B_{22}) = B$	$split(B_{11}, B_{12}, B_{21}, B_{22}, B)$
3	$T_1 = A_{11} - A_{21}$	$sub(A_{11}, A_{21}, T_1)$
4	$T_2 = B_{22} - B_{12}$	$sub(B_{22}, B_{12}, T_2)$
5	$C_{21} = T_1 \times T_2$	$mul(T_1, T_2, C_{21})$
6	$T_1 = A_{21} + A_{22}$	$add(A_{21}, A_{22}, T_1)$
7	$T_2 = B_{12} - B_{11}$	$sub(B_{12}, B_{11}, T_2)$
8	$C_{22} = T_1 \times T_2$	$mul(T_1, T_2, C_{22})$
9	$T_1 = T_1 - A_{11}$	$sub(T_1, A_{11}, T_1)$
10	$T_2 = B_{22} - T_2$	$sub(B_{22}, T_2, C_{22})$
11	$C_{11} = T_1 \times T_2$	$mul(T_1, T_2, C_{11})$
12	$T_1 = A_{12} - T_1$	$sub(A_{12}, T_1, T_1)$
13	$C_{12} = T_1 \times B_{22}$	
14	$C_{12} = C_{22} + C_{12}$	$mul\_add(T_1, B_{22}, C_{22}, C_{12})$
15	$T_1 = A_{11} \times B_{11}$	
16	$C_{11} = C_{11} + T_1$	
17	$C_{12} = C_{11} + C_{12}$	$mul\_inc\_inc\_inc$
18	$C_{11} = C_{11} + C_{21}$	$(A_{11}, B_{11}, T_1, C_{21}, C_{11}, C_{12})$
19	$T_2 = T_2 - B_{21}$	$sub(T_2, B_21, T_2)$
20	$C_{21} = A_{22} \times T_2$	
21	$C_{21} = C_{11} - C_{21}$	mul_sub_inc
22	$C_{22} = C_{11} + C_{22}$	$(A_{22}, T_2, C_{11}, C_{21}, C_{22})$
23	$C_{11} = A_{12} \times B_{21}$	
24	$C_{11} = T_1 + C_{11}$	$mul\_add(A_{12}, B_{21}, T_1, C_{11})$
25	$C = (C_{11}, C_{12}, C_{21}, C_{22})$	$merge(C_{11}, C_{12}, C_{21}, C_{22}, C)$

C. Multi-Level Adaptation

Multi-level adaptation is the strassen's algorithm. Compared with one-level adaptation, it replaces all multiplication calls with recursive calls.

The final algorithm contains zero-level, one-level and multi-level adaptation. The value of τ_1, τ_2 can be changed to make it faster.

```
strassen(A, B, C, n)
    if n <= t1
        run zero-level adaptation
    else if n <= t2
        run one-level adaptation
    else
        run multi-level adaptation
        (contains recursive calls)
```

III. Experiment

First, I set $\tau_1 = \tau_2 = +\infty$, so I can get $GFLOPS/s$ by using zero-level adaptation. Then, I set $\tau_1 = 1, \tau_2 = +\infty$, so I can get $GFLOP/s$ by using one level adaptation. Compare them and I can get the value of τ_1.

Now τ_1 is fixed. First, I set $\tau_2 = +\infty$, so I can get $GFLOPS/s$ by using zero-level and one-level adaptation. Then, I set $\tau_2 = \tau_1$, so I can get $GFLOP/s$ by using zero-level and multi-level adaptation. Compare them and I can get the value of τ_2.

A. Pthread

Fig. 2 shows zero-level apaptation has the highest $GFLOPS/s$. Also core dumped in the experiment when I try to create large number of threads. It is best to use the classical measure.

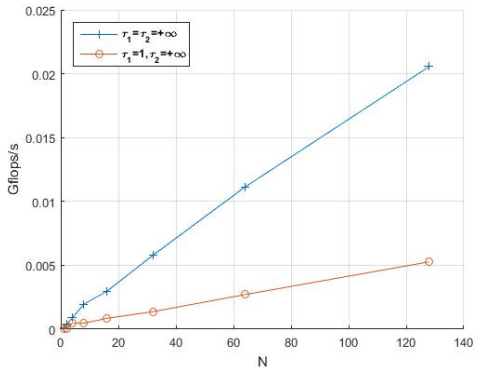

Fig. 2. Strassen's Algorithm using pthread

B. OpenMP

Fig. 3 shows that zero-level adaptation performs better than one-level adaptation when $N \leq 64$. So I get $\tau_1 = 64$.

Then I set $\tau_1 = 64$. Fig. 3 shows that zero-level and one-level adaptation performs better than zero-level and multi-level adaptation when $N \leq 128$. So I get $\tau_2 = 128$.

While N is large enough, multi-level adaptation performs best.

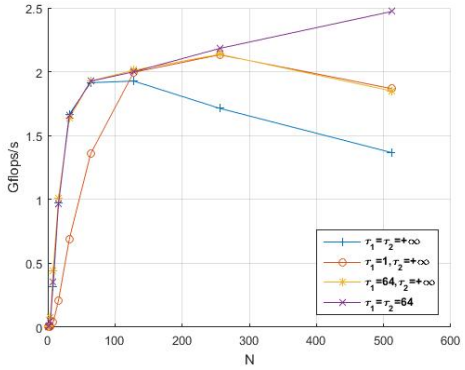

Fig. 3. Strassen's Algorithm using OpenMP

C. CUDA

Fig. 4 shows that zero-level adaptation performs better than one-level adaptation when $N \leq 1024$. So I get $\tau_1 = 1024$.

Then I set $\tau_1 = 1024$. Fig. 4 shows that zero-level and one-level adaptation performs better than zero-level and multi-level adaptation when $N \leq 4096$. So I get $\tau_2 = 4096$.

We note that one-level adaptation performs better than multi-level adaptation, because GPUs are slower than CPUs at recursive calls, and N is not very large in my experiment.

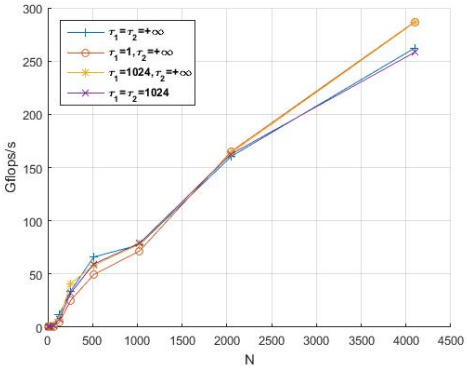

Fig. 4. Strassen's Algorithm using CUDA

IV. Conclusion

I have accomplished efficient Strassen's matrix multipication parallel algorithms in CPUs and GPUs. I use diffrent adaptation to avoid too much recursion. My experiments show how to choose the best parameters and have got rough values.

References

[1] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[2] S.Winograd, "On multiplication of 2 2 matrices," *Linear Algebra and its Applications*, vol. 4, no. 4, pp. 381–388, Oct. 1971.

[3] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith, "Gemmw: a portable level 3 blas winograd variant of strassen's matrix-matrix multiply algorithm," *Journal of Computational Physics*, vol. 110, no. 1, pp. 1–10, 1994.

[4] J. Li, S. Ranka, and S. Sahni. (2011) Strassen's matrix multiplication on gpus.