

Matrix-Matrix Multiplication

Craig Douglas
March 20, 2001

$$C = A \cdot B, \text{ where}$$

$$A : M \times K, B : K \times N, \text{ and } C : M \times N.$$

Outline:

Standard

Strassen

Parallel

Standard Form

$$C_{ij} = \sum_{\ell=1, K} A_{i\ell} B_{\ell,j}$$

- There are **6** ways to compute this sum depending on how A and B are stored (row or column wise).
- Solving memory bank conflicts and using cache blocking makes a huge difference on run times.
- Fast implementations are hardware dependent. Fastest ones are frequently written in assembly language (or close to it) with hand optimization only.
- Complex data (2 reals) gives implementors problems.

Memory bank effects: Assume 4 banks and try to access $v = \{v_1, \dots, v_{12}, \dots\}$.

Scheme A

Bank 1	v_1, v_5, v_9
Bank 2	v_2, v_6, v_{10}
Bank 3	v_3, v_7, v_{11}
Bank 4	v_4, v_8, v_{12}

Stride 1: no conflicts

Stride 2,4,6, \dots : conflicts

Stride 3,5,7, \dots : no conflicts

Scheme B

Bank 1	v_1, v_2, v_3
Bank 2	v_4, v_5, v_6
Bank 3	v_7, v_8, v_9
Bank 4	v_{10}, v_{11}, v_{12}

Stride 1,2,3: conflicts

Stride 4,5,6: no conflicts

Cache effects:

- On most superscalar computers, 95% of peak can be achieved as long as all three matrices fit into cache. Efficiency drops by a factor of 4 to 10 when the data does not fit entirely into cache.
- Prefetching data into cache, using the data twice (which is a very hard algorithm to implement well, and having an expensive cache memory system leads to codes that do not have a huge fall off at the cache memory limit (e.g., IBM's ESSL *dgemm* routine).
- Some vendors aim at small matrices, others at large. Not many aim at both.

Strassen Form

Matrices can be decomposed into 2×2 submatrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

We can factor the multiplication algorithm using the blocks. This can be recursively done again until some threshold (*mindim*) is reached where it is no longer effective computationally. In general,

$$96 \leq \textit{mindim} \leq 192.$$

Asymptotically, this technique leads to a $O(N^{\log_2 7})$ algorithm instead of a $O(N^3)$ for square matrices.

Strassen-Winograd variant:

$$\begin{array}{lll} S_1 = A_{21} + A_{22} & M_1 = S_2 S_6 & T_1 = M_1 + M_2 \\ S_2 = S_1 - A_{11} & M_2 = A_{11} B_{11} & T_2 = T_1 + M_4 \\ S_3 = A_{11} - A_{21} & M_3 = A_{12} B_{21} & \\ S_4 = A_{12} - S_2 & M_4 = S_3 S_7 & \\ S_5 = B_{12} - B_{11} & M_5 = S_1 S_5 & C_{11} = M_2 + M_3 \\ S_6 = B_{22} - S_5 & M_6 = S_4 B_{22} & C_{12} = T_1 + M_5 + M_6 \\ S_7 = B_{22} - B_{12} & M_7 = A_{22} S_8 & C_{21} = T_2 - M_7 \\ S_8 = S_6 - B_{21} & & C_{22} = T_2 + M_5 \end{array}$$

There are only 7 matrix-matrix multiplications and 15 matrix-matrix additions and subtractions.

Extra storage is required to hold sums of quadrants of A and B . The amount is roughly the same as the quadrants. The extra storage is quite expensive (big bucks).

What is normally computed is not $C = AB$,
but

$$C \leftarrow \alpha \cdot op(A)op(B) + \beta \cdot C, \text{ where}$$

$$op(X) = \begin{cases} X, \\ X \text{ transpose}, \\ X \text{ conjugate transpose}, \\ X \text{ conjugate}, \end{cases}$$

and

$$op(A) : M \times K, \quad op(B) : K \times N, \quad \text{and} \quad C : M \times N.$$

This complicates memory usage and can add MN extra storage if $\beta \neq 0$.

We can use parts of C if $\beta = 0$. We assume columnwise storage. Only two work areas are needed: W_{MK} and W_{KN} . When M is odd, an additional short vector of length $N/2$ is required.

The size of each large work area is

$$W_{MK} : \left\lfloor \frac{M \max(K, N) + M + \max(K, N) + 4}{4} \right\rfloor$$

and

$$W_{KN} : \left\lfloor \frac{KN + K + N + 4}{4} \right\rfloor,$$

The bound for the work areas is

$$\frac{1}{3}[M \max(K, N) + KN] + \frac{1}{2}[M + \max(K, N) + K + 3N] + 32.$$

Step	W_{MK}	C_{11}	C_{12}	C_{21}	C_{22}	W_{KN}	Operation
1.						S_7	$B_{22} - B_{12}$
2.	S_3						$A_{11} - A_{21}$
3.				M_4			$S_3 S_7$
4.	S_1						$A_{21} + A_{22}$
5.						S_5	$B_{12} - B_{11}$
6.					M_5		$S_1 S_5$
7.						S_6	$B_{22} - S_5$
8.	S_2						$S_1 - A_{11}$
9.		M_1					$S_2 S_6$
10.	S_4						$A_{12} - S_2$
11.			M_6				$S_4 B_{22}$
12.			T_3				$M_5 + M_6$
13.	M_2						$A_{11} B_{11}$
14.		T_1					$M_1 + M_2$
15.			C_{12}				$T_1 + T_3$
16.		T_2					$T_1 + M_4$
17.						S_8	$S_6 - B_{21}$
18.				M_7			$A_{22} S_8$
19.				C_{21}			$T_2 - M_7$
20.					C_{22}		$T_2 + M_5$
21.		M_3					$A_{12} B_{21}$
22.		C_{11}					$M_2 + M_3$

Odd M , N , or K cause corruption problems in memory unless special care is taken. Rows or columns may be duplicated to solve the problem conceptually.

Specifically, we do certain steps differently:

Step	Array	Step	Function
4.	S_1	(a)	If K is odd, then copy first column of A_{21} into W_{MK} .
		(b)	Complete S_1 .
10.	S_4	(a)	If K is odd, then pretend first column of $A_{21} = 0$ in W_{MK} .
		(b)	Complete S_4 .
11.	M_6	(a)	If M is odd, then save first row of M_5 .
		(b)	Calculate most of M_6 .
		(c)	Complete M_6 using (a) based on M odd or not.
21.	M_3	(a)	Calculate M_3 using an index shift.

Numerical Experiments

On a 500 MHz Pentium III running Linux (gcc, g77, $mindim = 64$).

M,N,K	<i>dgemm</i>	<i>dgemmw</i>
100	0.033	0.033
120	0.050	0.050
144	0.067	0.067
172	0.133	0.117
206	0.567	0.200
247	0.917	0.317
355	2.717	0.900
426	4.683	1.500
511	8.050	2.600
613	13.883	4.250
735	23.899	7.083
882	41.748	11.950
1000	60.648	17.116

dgemmw runs away and hides from *dgemm*!

A Parallel Form

Form	Matrix		Total Loads
	Mults	Adds	
Standard	8	4	12
Strassen	7	15	22

When loads and stores are expensive, the standard form is better. This is the case in distributed memory parallel computers. It is not necessarily true on shared memory computers.

Aha! There is something for the class to investigate:

- Standard across processors, but Strassen within a processor.
- Standard only
- Strassen only
- MPI versus OpenMP

This is a simple parallel method based on communication being equally expensive. This is not entirely realistic on today's computers, which are distributed shared memory nodes.

More complicated ones can be found by searching on *Parallel Matrix-Matrix Multiplication* on Google. The one at the University of Texas is a good start.

Initially, a heuristic iteratively partitions the processors and matrices A , B , and C until each processor has a submatrix multiplication to perform independently and in parallel with the others.

Assume there are p processors, which can be factored into the product of primes:

$$p = \prod_{i=1}^n p_i.$$

Without loss of generality, assume $p_{i-1} \leq p_i$, $2 \leq i \leq n$.

Start with one set of p processors. In step i , the heuristic partitions each set of processors into p_i subsets and divides the maximum of $\{M_i, K_i, N_i\}$ by p_i . Also, the number of processor partitions grows by a factor of p_i , as does the number of submatrix multiplications. Each submatrix multiplication decreases in complexity by an identical factor of p_i . After n steps, each processor has an independent submatrix multiplication.

Example with $p = 12 = 2 \times 2 \times 3$

The prime factorization is part of the heuristic used here. There are better algorithms that can be found by search through Google on the phrase *parallel matrix-matrix multiplication*. (Other algorithms can be found be not including *parallel*).

0. Processors 1 – 12 collaborate on one matrix multiplication. The matrices are partitioned initially as whole matrices:

$$\begin{bmatrix} aaaa \\ aaaa \\ aaaa \end{bmatrix} \times \begin{bmatrix} bbbb \\ bbbb \\ bbbb \\ bbbb \end{bmatrix} = \begin{bmatrix} cccc \\ cccc \\ cccc \end{bmatrix}.$$

1. Use the first 2 in the prime factorization of 12. Two processor groups are formed: 1 – 6 and 7 – 12, which collaborate on two submatrix multiplications:

$$\begin{bmatrix} aaaa \\ aaaa \\ aaaa \end{bmatrix} \times \begin{bmatrix} bbb & | & bb \\ bbb & | & bb \\ bbb & | & bb \\ bbb & | & bb \end{bmatrix} = \begin{bmatrix} ccc & | & cc \\ ccc & | & cc \\ ccc & | & cc \end{bmatrix}.$$

2. Use the second 2 in the prime factorization of 12. Split both processor groups in half: 1 – 3, 4 – 6, 7 – 9, and 10 – 12, which collaborate on four submatrix multiplications:

$$\begin{bmatrix} aa & | & aa \\ aa & | & aa \\ aa & | & aa \end{bmatrix} \times \begin{bmatrix} bbb & | & bb \\ bbb & | & bb \\ \hline bbb & | & bb \\ bbb & | & bb \end{bmatrix} = \begin{bmatrix} ccc & | & cc \\ ccc & | & cc \\ ccc & | & cc \end{bmatrix}.$$

3. Use the last prime, 3, in the prime factorization of 12. We get 12 processor groups with one processor in each group, which collaborate on 12 submatrix multiplications:

$$\left[\begin{array}{c|c} aa & aa \\ \hline aa & aa \\ \hline aa & aa \end{array} \right] \times \left[\begin{array}{c|c} bbb & bb \\ \hline bbb & bb \\ \hline bbb & bb \\ \hline bbb & bb \end{array} \right] = \left[\begin{array}{c|c} ccc & cc \\ \hline ccc & cc \\ \hline ccc & cc \end{array} \right].$$

4. Each processor performs its matrix multiplication independent of the others.

Numerical Experiments

On a Sequent Symmetry, $M = N = K = 500$.

p	Time	Speedup	Scaling %
	Using <i>matmulp/DGEMMW</i>		
1	543.28		
2	277.71	1.9562	97.81
4	159.49	3.4063	85.15
8	86.84	6.2561	78.20
12	55.93	9.7135	80.94
16	49.55	10.9642	68.52
	Using classical DGEMM only		
1	1532.14		
2	774.02	1.9794	98.97
4	388.51	3.9436	98.59
8	195.78	7.8258	97.82
12	131.94	11.6123	96.76
16	102.02	15.0180	93.86

The speedup is better with the slower algorithm. (Only wall clock times should count in supercomputing.)