Programming Assignment 1 Report

Mengtong Zhang, Yunjia Zeng

March 30, 2019

In this report, we proposed a modified Strassen's algorithm to improve the efficiency of traditional Strassen's algorithm for matrix multiplication. Instead of recurring to the base case of 1×1 matrix, we stop recursing and switch to conventional matrix multiplication when the dimension of input is less than some cross-over point. The goal of this report is to find the optimal cross-over point for different matrix dimensions. First, theoretical analysis will be performed to minimize the total number of operations for our modified Strassen's algorithm. Next, we will implement the modified algorithm and find the experimental cross-over point which minimize the runtime of matrix multiplication.

Mathematical Analysis

In this section, we will provide some theoretical analysis for determining the cutoff point for optimizing the runtime of Strassen's Algorithm. Consider the variant of Strassen's algorithm: to multiply two n by n matrices, start using Strassen's algorithm, but stop the recursion at some size n_0, and use the conventional algorithm below that point.

To find n_0, first let's see the runtime of two matrix multiplication algorithms. Here we assume that the cost of any single arithmetic operation (adding, subtracting, multiplying, or dividing two real numbers) is 1, and that all other operations are free.

Suppose now we are trying to compute,

$$C_{n\times n} = A_{n\times n}B_{n\times n}$$

Conventional matrix multiplication

For conventional matrix multiplication, we need to compute each elements of C, which is n^2 elements. And for each element $C_{i,j}$, $C_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$, it will cost n multiplications and $n-1$ additions. So the total number of operations for conventional matrix multiplication is:

$$C(n) = (2n-1)n^2$$

Strassen's Algorithm

The basic idea of Strassen's algorithm is to use devide-and-conquer that requires only 7 multiplications of size $n/2$ matrix. Let

$$A = \begin{bmatrix} X & Y \\ Z & W \end{bmatrix}, \; B = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$
$$A \cdot B = \begin{bmatrix} XE + YG & XF + YH \\ ZE + WG & ZF + WH \end{bmatrix}.$$

Let

$$\begin{aligned} &P_1 = X \cdot (F - H),\, P_2 = (X + Y) \cdot H,\, P_3 = (Z + W) \cdot E \\ &P_4 = W \cdot (G_E),\, P_5 = (X + W) \cdot (E + H) \\ &P_6 = (Y - W) \cdot (G + H),\, P_7 = (X - Z) \cdot E + F, \end{aligned}$$

and

$$\begin{aligned} XE + YG &= P_5 + P_4 - P_2 + P_6 \\ XF + YH &= P_1 + P_2 \\ ZE + WG &= P_3 + P_4 \\ ZF + WH &= P_5 + P_1 - P_3 - P_7. \end{aligned}$$

Then it is easy to see the recurrence of Strassen's algorithm as

$$T(n) = 7T(\frac{n}{2}) + 18((\frac{n}{2})^2)$$

For Strassen's Algorithm, we may need to use padding strategy to deal with odd dimensions. The padding strategy we used is: for each recursion of Strassen's Algorithm, check if the size of the matrix is odd, if yes, pad 1 row and 1 column to it so that the dimension becomes even, the proceed the recursion. There are some other padding strategies. We will discuss why we choose this one later.

In this way, Strassen's Algorithm will cost 7 multiplications on size $\lceil \frac{n}{2} \rceil$ matrices and 18 additions on size $\lceil \frac{n}{2} \rceil$ matrices. Now we can conclude that the total number of operations for Strassen's Algorithm is:

$$T(n) = 7T(\lceil \frac{n}{2} \rceil) + 18(\lceil \frac{n}{2} \rceil^2)$$

Cross-over point

Since Strassen's algorithm is a recursive algorithm, at some point in the recursion, once the matrices are small enough, we may want to switch from recursively calling Strassen's algorithm and just do a conventional matrix multiplication. That point is the cross-over point. In this part, we will give the formula for calculating total number of operations for different input size and cross-over point.

Suppose the input dimension is n, the cross-over point is n_0, we assume n and n_0 are both power of 2 for convenience of computation:

$$
\begin{aligned}
T(n,n_0) &= 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 = 7^2T(\frac{n}{2^2}) + 7\times 18(\frac{n}{2^2})^2 + 18(\frac{n}{2})^2 \\
&= \cdots = 7^{log_2(\frac{n}{n_0})}T(n_0) + \sum_{i=1}^{log_2(\frac{n}{n_0})}(7^{i-1}\cdot 18(\frac{n}{2^i})^2) \\
&= 7^{log_2(\frac{n}{n_0})}(2n_0-1)n_0^2 + \sum_{i=1}^{log_2(\frac{n}{n_0})}(7^{i-1}\cdot 18(\frac{n}{2^i})^2)
\end{aligned}
$$

Obviously, the total operation number $T(n)$ is related with both n and n_0. To find the optimal n_0 such that $T(n)$ is minimized, next we will do some theoretical estimation of $T(n)$ under different n_0 and n.

Comparison of Two Padding Strategies

In the previous sections, we have mentioned there are different padding methods to deal with odd number dimensions. Suppose the input dimension is n, they can be described as:

- One-time Padding: If $log_2 n$ is not an integer, then we pad 0s to the original input matrix such that their dimension becomes $2^{\lceil log_2 n\rceil}$, which is power of 2. Then we can easily apply Strassen's algorithm. Let $n^{'} = 2^{\lceil log_2 n\rceil}$, Then the total number of operations will be:

$$T(n,n_0) = T(n^{'},n_0) = 7^{log_2(\frac{n^{'}}{n_0})}(2n_0-1)n_0^2 + \sum_{i=1}^{log_2(\frac{n^{'}}{n_0})}(7^{i-1}\cdot 18(\frac{n^{'}}{2^i})^2)$$

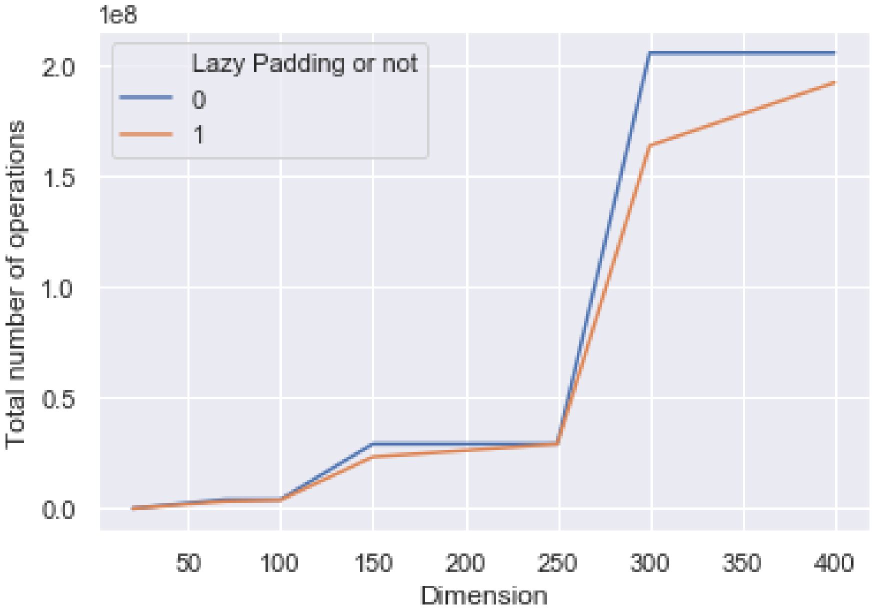
- Lazy Padding: If n is odd, then we pad 1 row and 1 column to the input so that the dimension becomes even, then recursively call Strassen's algorithm. Every time before recursively calling Strassen's, we check whether the dimension is odd or not, if yes, then pad 1 row and 1 column to the input. In this way, the total number of operations can be calculated by the recursive relation:

$$T(n,n_0) = 7T(\lceil\frac{n}{2}\rceil) + 18(\lceil\frac{n}{2}\rceil^2)(n > n_0)$$

  when $n \leq n_0$, we then use the formula for conventional multiplication to calculate the number of operations.

Next, we will illustrate the theoretical efficiency of this two padding strategy by plotting the total number of operations for n=20,70,100,150,250,300,400 under different padding strategies. The total number of operations are calculated based on the formula we have given above. Since the difference between these two padding strategies only happens in Strassen's algorithm, we will set cross-over point = 2, which is nearly pure Strassen's.

We can see that for large dimensions, obviously lazy padding shows more efficiency with a less theoretical total number of operations.



Theoretical Estimation

In this section, we calculate the total number of operations using the formula for different n and n_0:

$$T(n, n_0) = 7^{log_2(\frac{n}{n_0})}(2n_0 - 1)n_0^2 + \sum_{i=1}^{log_2(\frac{n}{n_0})} (7^{i-1} \cdot 18(\frac{n}{2^i})^2)$$

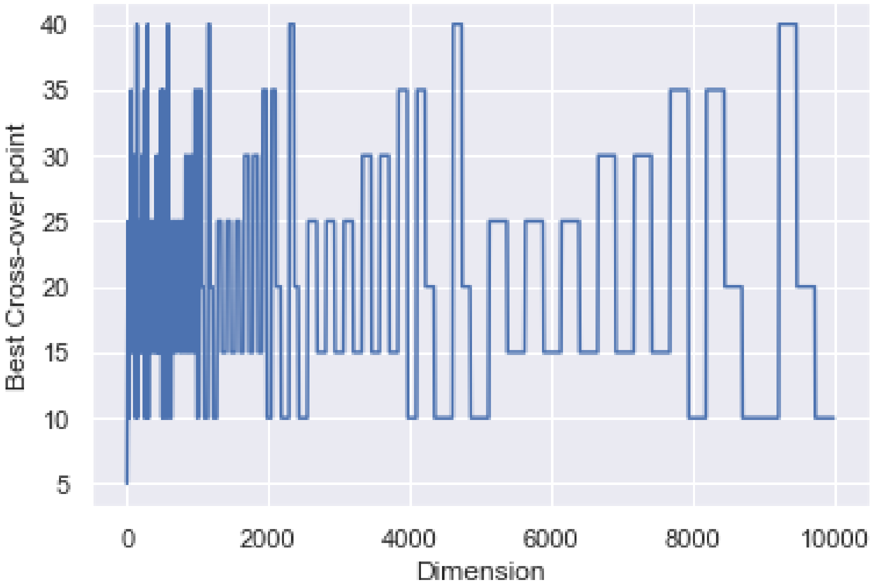
Here is a table shows the total number of operations, where the * sign marked the best cross-over point for each input size.

	n=4	n=8	n=16	n=32	n=64	n=128	n=256	n=512
$n_0 = 2$	156	1380	10812	80292	580476	4137060	29254332	205959972
$n_0 = 4$	112*	1072	8656	65200	474832	3397552	24077776	169724080
$n_0 = 8$		960*	7872*	59712*	436416*	3128640*	22195392*	156547392*
$n_0 = 16$			7936	60160	439552	3150592	22349056	157623040
$n_0 = 32$				64512	470016	3363840	23841792	168072192
$n_0 = 64$					520192	3715072	26300416	185282560
$n_0 = 128$						4177920	29540352	207962112
$n_0 = 256$							33488896	235601920
$n_0 = 512$								268173312

After estimating the number of operations for different input dimensions and cross-over points, we find:

- For $n = 4$, it is better to use conventional multiplication.
- For $n \geq 8$, the cross-over point is around 8.

Also, we may want to figure out the cross-over point for dimensions which are not power of 2. Next, we will illustrate the number of operations for some ordinary input dimensions:



	n=20	n=70	n=100	n=150	n=250	n=300	n=400
$n_0 = 2$	66396	3299520	3888714	23375424	29224668	164032968	192526986
$n_0 = 4$	28323	1433943	3149206	10316385	24048112	72619695	156291094
$n_0 = 8$	15975	828891	2104771	6081021	22165728*	42972147	105113779
$n_0 = 16$	15100*	606627*	1721983	5780896	22319392	40871272	86357167
$n_0 = 32$		618534	1624375*	5183047*	23812128	36686329*	81574375*
$n_0 = 64$		613725	1777500	5589894	25091469	39534258	89077500
$n_0 = 128$				5968125	27515625	42181875	99490000
$n_0 = 256$						47497500	112440000

For dimensions other than power of 2, we can find that different dimensions will have different best cross-over points. And most of the cross-over points are 16,32 or 64. It is probably because these matrix require more operations for padding using Strassen's, so it is better for them to stop Strassen's and switch to conventional multiplication earlier.

Cross-over points may also not power of 2. To further narrower the range of cross-over points, we choose input size $= 0, 10, 20, \ldots, 10000$, and choose cross-over points $= 0, 5, 10, 15, \ldots, 100$, for each input size, pick the smallest number of operation and the corresponding best cross-over point. The results are shown in the figure above.
To sum up, the theoretical cross-over points are within the range of $(5, 40)$ for dimension under 10000.

Experimental Analysis

In this section, we will first explain the modified Strassen's algorithm we designed to solve this problem. And then the algorithm will be applied to different inputs to find the experimental cross-over point.

Algorithm Design and Implementation

To perform the analysis, we first implemented the ordinary algorithm for matrix multiplication. We also implemented the ordinary matrix addition algorithm *matrix_add*() and subtraction algorithm *matrix_sub*() to be used in the Strassen's algorithm *strassen*(). All *matrix_add*(), *matrix_sub*(), and *strassen*() take two matrices A and B as inputs. There is also a function *matrix_generator*() that takes a dimension *dim* as input to generator two $dim \times dim$ random binary square matrices for testing use. The algorithm works well and the output of matrices of Strassen's algorithm and the ordinary matrix multiplication algorithm are the same.

The Padding Function

To deal with the odd input of n, we pad the input matrix with 0s first implemented a wrapper function *wrapper_strassen*() to find the next power of 2 for padding appropriate number of 0s so that we can apply the Strassen's algorithm recursively without worrying about the odd matrix sizes. But later in implementation we find it is more efficient to follow another methods of padding. To transform a $n \times n$ matrix into a $n \times n$ matrix inside the recursion, m need to be a power of 2 such that it must be only divisible by 2. Therefore, as long as m is divisible by 2, it will hit the cutting point eventually. Thus it is more efficient by just adding 0s when we need them for multiplications along the recursion. The improved method now checks if n is divisible by 2, before each time Strassen's algorithm goes down the recursion tree. If n is not divisible by 2, the padding algorithm transforms the matrices into $m \times m$ matrices where $m = n + 1$ by adding one row and column of 0s to each matrix. Now rather than what we do in the wrapper before, the padding algorithm add at most one row and column of 0s at each level of recursion. The runtime was cut down significantly this way.

The Strassen's Function

In the first try, we implemented Strassen's algorithm that follows the guideline of the matrices where n is a power of 2. The key of the algorithm is to break both matrices into four $\frac{n}{2} \times \frac{n}{2}$ matrices and multiply them. Therefore we can perform this process until we have 1×1 matrices witch are just simple numbers. Instead of performing eight recursive multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices (the ordinary divide-and-conquer method), Strassen's perform only seven:

1. Divide the input matrices A and B and output matrix C into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices, therefore in the function we initialized $A11, B11, C11$, $A12, B12, C12$, $A21, B21, C21$, and $A22, B22, C22$ respectively.

2. Rather than the vanilla version of Strassen's algorithm in the textbook, we replace the containers of $S_1, S_2, S_3, \ldots, S_10$ with two middle-ware *temp*1 and *temp*2 that are recyclable (we overwrite each time) in order to save some memory. Each of which is also $\frac{n}{2} \times \frac{n}{2}$ and is the sum or difference of two matrices created in the previous step.

3. Use the sub-matrices created in the previous two steps to recursively compute seven matrix products $P_1, P_2, P_3, \ldots, P_7$. Each matrix P_i is $\frac{n}{2} \times \frac{n}{2}$.

4. Compute the desired sub-matrices $C11, C12, C21, C22$ of the result matrix C by adding and subtracting various combinations of P_i matrices.

We call Strassen's function recursively at each step of calculating $P_1, P_2, P_3, \ldots, P_7$. Before calling,

Algorithm 1 The modified Strassen's Algorithm

```
procedure STRASSEN(matrix A, matrix B, crossover)
    $n \leftarrow$ length of $A$
top:
    if $n <= crossover$ then return Multiply(A,B)
    else
loop:
        while $n$ divided by 2 is not 0 do
            pad A.
            pad B.
            $n \leftarrow n + 1$
        $half \leftarrow$ n // 2
        $A11, A12, A21, A22 \leftarrow$ empty half × half matrix
        $B11, B12, B21, B22 \leftarrow$ empty half × half matrix
        $C \leftarrow$ empty n × n matrix
        $temp1, temp2 \leftarrow$ empty half × half matrix
        for $i \leftarrow 1:half$ do
            for $j \leftarrow 1:half$ do
                A11[i][j] ← A[i][j]
                B11[i][j] ← B[i][j]
                A12[i][j] ← A[i][j + half]
                B12[i][j] ← B[i][j + half]
                A21[i][j] ← A[i + half][j]
                B21[i][j] ← B[i + half][j]
                A22[i][j] ← A[i + half][j + half]
                B22[i][j] ← B[i + half][j + half]
        $P1 \leftarrow (A11 + A22) \times (B11 + B22)$
        $P2 \leftarrow (A21 + A22) \times B11$
        $P3 \leftarrow A11 \times (B12 - B22)$
        $P4 \leftarrow A22 \times (B21 - B11)$
        $P5 \leftarrow (A11 + A12) \times B22$
        $P6 \leftarrow (A21 - A11) \times (B11 + B12)$
        $P7 \leftarrow (A12 - A22) \times (B21 + B22)$
        Adjust and overwrite temp1, temp2accordingly when necessary
        $C11 \leftarrow (P1 + P4) - P5 + P7$
        $C12 \leftarrow (P3 + P5)$
        $C21 \leftarrow (P2 + P4)$
        $C11 \leftarrow (P1 + P3) - P2 + P6$
        for $i \leftarrow 1:half$ do
            for $j \leftarrow 1:half$ do
                C[i][j] ← C11[i][j]
                C[i][j + half] ← C12[i][j]
                C[i + half][j] ← C21[i][j]
                C[i + half][j + half] ← C22[i][j]
output $C$
```

we adjust and overwrite $temp1$ and $temp2$ based on needs. At beginning, $temp1$ was set to the result of $matrix\_add(A11, A22)$, and $temp2$ was set to the result of $matrix\_add(B11, B22)$, since $P1 = strassen(temp1, temp2)$. Since $P2 = (A21 + A22) \times B11$, we then overwire $temp1$ as the result of $matrix\_add(A21, A22)$, then call $strassen(temp1, B11)$. We do similar process later on for all P_1 to P_7.

Test and Results

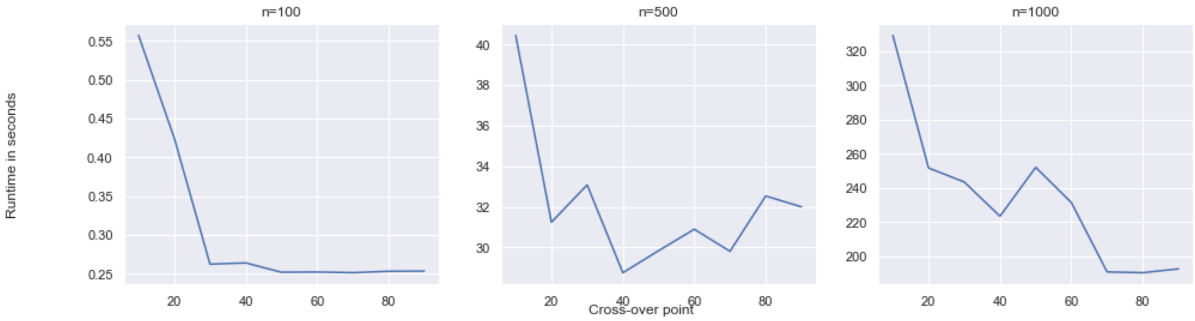
In this section, we will run the algorithm above for different input sizes and cross-over points to find the experimental best cross-over points. We are using Macbook Air 2013 with "1.3GHz dual-core Intel Core i5" and "4GB of 1600MHz LPDDR3" memory. To make the implementation faster, we generate matrices only containing entries 0 or 1. And for each input size, we keep the matrix same to find the best cross-over point.

First, we try our algorithm under the condition that both n and n_0 are power of 2. Here is the table of the time records for different input sizes n and different choices for cross-over points n_0. The * sign marks the shortest running time (in seconds).

	n=4	n=8	n=16	n=32	n=64	n=128	n=256	n=512
$n_0 = 2$	0.00027	0.0022	0.0169	0.1090	0.5078	3.5803	24.3758	186.2126
$n_0 = 4$	0.00008*	0.0008	0.0071	0.0672	0.1906	1.4479	10.2282	74.6720
$n_0 = 8$		0.0003*	0.0033	0.0268	0.1512	0.8599	5.8818	41.9590
$n_0 = 16$			0.0013*	0.0101	0.1213	0.6394	4.7826	33.5140
$n_0 = 32$				0.0089*	0.1028*	0.6604	4.1219*	28.8278*
$n_0 = 64$					0.1042	0.5564*	4.4461	29.7232
$n_0 = 128$						0.58258	4.6861	38.3818
$n_0 = 256$							5.56094	38.3127
$n_0 = 512$								43.0341

From the table above, we can see the best cross-over point varies in 8,16, 32 and 64. In the theoretical analysis, the cross-over point is around 8 for power of 2 dimensions. However, the experimental analysis give us a result which is close to the theoretical best cross-over point for non-power of 2 dimensions.

Similarly as the procedure of theoretical analysis, next we will perform our modified Strassen's algorithm on ordinary dimensions. Specially for $n = 250$, we tried different type of input matrix. Our original input matrix only contains 0 or 1, now we tried input matrix with entries randomly selected between 1 and 1000 to see whether it will influence the runtime. The runtime for the new type of input matrix is shown in ().



	n=20	n=70	n=100	n=150	n=250	n=300	n=400
$n_0 = 2$	0.0719	3.2010	3.0604	24.1308	25.5198(25.0557)	175.8413	194.1363
$n_0 = 4$	0.0150	0.7419	1.2453	6.1787	10.3743(10.0695)	48.4783	75.0930
$n_0 = 8$	0.0048	0.5088	0.5872	2.1478	6.2524(6.1693)	17.4353	33.2611
$n_0 = 16$	0.0028*	0.1352	0.3572	1.5398	5.0564(4.7002)	11.2496	23.3878
$n_0 = 32$		0.0909*	0.2992*	0.8983*	4.3442(4.4317)	8.2663	14.9620*
$n_0 = 64$		0.1100	0.3060	0.9283	3.9202*(4.4818)	8.1270*	16.4691
$n_0 = 128$				1.1193	4.3486(4.4725)	8.9753	17.7248
$n_0 = 256$						8.2802	17.3701

From the table above, we can see except for $n = 250$ and $n = 300$, we have found the experimental cross-over points is exactly the same with theoretical ones. We also find that the experimental cross-over point is larger than theoretical ones, which is probably because we have ignored some operations in Strassen's (such as padding 1 row and 1 column operations) in the theoretical analysis. Thus, in the implementation Strassen's is more time-consuming than we expected. In this case, the algorithm tends to choose to switch to conventional multiplication earlier, which will lead to a larger cross-over point.

Another finding is that changing type of matrixes will not influence runtime very much. After changing the entries to be a random number between 1 and 1000 for $n = 250$, the runtime is only changed slightly.

Finally, we are going to try cross-over points that are not power of 2. We will choose input size $n = 100, 500, 1000$ and record the time for running modified Strassen's with cross-over points $n_0 = 10, 20, 30, 40, 50, 60, 70, 80, 90$. The results are shown in the above figure.

We can see that for $n = 100, 500, 1000$, the experimental cross-over points lie in $(40, 80)$, which is also larger than our theoretical estimation.

Conclusion

- The theoretical cross-over points for matrix size < 10000 is between 5 and 40. And if we assume both cross-over point and input dimension are power of 2, the theoretical cross-over point is around 8.

- The experimental cross-over points are close to theoretical ones but they are typically larger. It is probably because we have introduced more operations into the Strassen's algorithm during implementation, so actually Strassen's is more time-consuming than we expected. Therefore, the algorithm tends to stop it earlier with larger cross-over point. Another possible reason may due to the use of Python as programming language for this analysis. The list creation maybe slow and too memory consuming. If we use Java or C++, especially C++, the strassen algorithm will have better performance and the result maybe closer to the theoretical result.

- The experimental cross-over points for dimensions that are power of 2 is between 16 and 64. And for other dimensions under 1000, the experimental cross-over points is between 40 and 80.