Fast Matrix Multiplication

Chathuri Peli Kankanamalage

Introduction

This project is to multiply large matrices whose entries are integers with values between -16 and +16. In this project we use two algorithms to implement matrix multiplication. First algorithm that we implement is the basic matrix multiplication algorithm. Then we implement Strassen's Algorithm, which is an optimized matrix multiplication algorithm. The goal of the project is to speed the algorithm to utmost speed and discuss the steps that we follow in order to achieve the speed.

1.0 Algorithm 1: Basic Matrix Multiplication

Inputs: Matrix A of n x p
Matrix B of p x m
Output: Matrix C of n x m such that C= AB

In basic matrix multiplication, we traverse through three "for" loops. Below is the implementation of `Product_basic` function.

```
void Product_basic(int *A, int Am, int An, int *B, int Bn,  int *C){
        int i, j, k;
        for (i=0; i < Am; i++){
                for (j=0 ; j < Bn ; j++){
                        *(C + i * Bn +j) = 0;
                        for (k=0; k < An; k++){
                                *(C + i * Bn + j) += (*(A + i * An +k))*(*(B + k * Bn +j));
                        }
                }
        }
}
```

This includes Am x Bn x An multiplications and scalar additions.

2.0 Algorithm 2: Strassen's Algorithm

Strassen's algorithm is an optimized version of matrix multiplication. Suppose we are multiplying two square matrices A and B of size mxm. If we use basic multiplication algorithm to multiply A, and B, we have to deal with m^3 multiplication with $(m^3 - m^2)$ additions. Strassen's algorithm tries to reduce the number of multiplications by dividing the matrices in to sub matrices recursively. His algorithm consists of 7 multiplications with 8 additions when multiplying two 2x2 matrices. Initial algorithm only works for square matrices in which the size is a power of 2. To make it work for arbitrary size matrices (square / rectangular), we need to do some additional padding / peeling. There are several ways that you can do the padding and peeling. Most common methods are below.

1. Static padding
2. Dynamic padding
3. Dynamic peeling

Basic idea of static padding is to make sure both input matrices are of the size of power 2. When rectangular or square matrices which are not of the size of power of 2 provided, we add columns and rows with 0s until they become square matrices of power of 2. Once the algorithm execution completes, we remove all the extra columns and rows added (which are all 0s) and return the desired matrix. There are improved versions of static padding where you do not have to make the input matrices as the size of power 2. You just need to make them even according to the cutoff value you choose. This is discussed in detail at the later part of the report.

In dynamic padding, padding happens at each time Strassen's algorithm calls. If the matrix contains odd number of rows, extra row can be added and if the matrix contains odd number of columns, extra column can be added. This padding needs to be done throughout the execution of the algorithm.

Another approach is dynamic peeling. When the input matrices are odd number dimensions, we consider it as a square matrix with extra row or extra column. Input matrix is divided into 4 matrices with a square matrix (m-1 x n-1), 2 thin matrices (1xn and mx1) and a single element. Strassen's algorithm is applied to multiply 2 square matrices and basic multiplication is applied to multiply other parts of the matrices.

According to literature, scientists discovered dynamic approaches work better compare to static padding. If we are using static padding as it is, there can be inefficiencies in situations such as when size of the matrix is closely larger than a power of 2. For example, suppose the size of the matrix is 37. Now if we find the closest power of 2 to 37, 64 which is almost double the original matrix. To avoid such inefficiencies, there are approaches to use cut-off value when finding the closest even number.

In our project, static padding with improvement, dynamic padding and dynamic peeling approaches are implemented. We will compare the performance numbers of these three approaches with the basic matrix multiplication.

Another important factor to investigate is the size of the cut off value in where we need to distinguish between calling the basic algorithm or the Strassen's algorithm. Strassen's algorithm works efficiently for large matrices. During each recursion, matrices will be divided in to sub matrices. So there comes a time, size of the sub matrix is smaller than the cut-off value. At that point, we need to use basic matrix multiplication algorithm. This cut-off value has direct impact on the performance of the Strassen's algorithm. As mentioned earlier, this cut-off value is used in improved static padding mechanism. Literature suggests, optimal even size (q) of the matrix should be $\frac{c}{2} < q \leq c$ where c is the cut-off value.

3.0 Correctness

When measure the correctness, we need to consider several factors.

- Whether the results generated by multiplication is correct
- Whether the algorithm works for larger size matrices
- Whether the program release all the memory allocated during the execution of the algorithm

To verify whether the correct results are being generated, I ran the same matrices with basic algorithm and the Strassen's algorithm and check whether both results are the same. Also verified whether the results are correct using online matrix multiplication tools.
To verify whether the program can handle very large matrices, I ran matrices of size 1000*1000 up to 5000*5000 using the test code and check whether I encounter unexpected quitting and segmentation faults. According to the test code, if the answer is incorrect, it suggests that the output is incorrect.
To verify the third point, I ran "valgrind", which is a tool to check memory usage of the program to make sure that we release all the memory used by the execution.

4.0 Performance

To measure the performance of the Strassen's algorithm, I ran tests in HULK machine with increasing matrix size. I increase the matrix size by 100 at each iteration. Same data files are used for all the strategies when taking performance numbers. Performance numbers are taken for all the 4 strategies:

- Basic matrix multiplication
- Strassen's algorithm with static padding
- Strassen's algorithm with dynamic padding
- Strassen's algorithm with dynamic peeling

We compare the performance numbers that we get for all the cases.

Below is the graph we get when running three cases for matrix of size 100*100*100 up to 3000 * 3000 * 3000. In this scenario, I keep the cut off as 12.

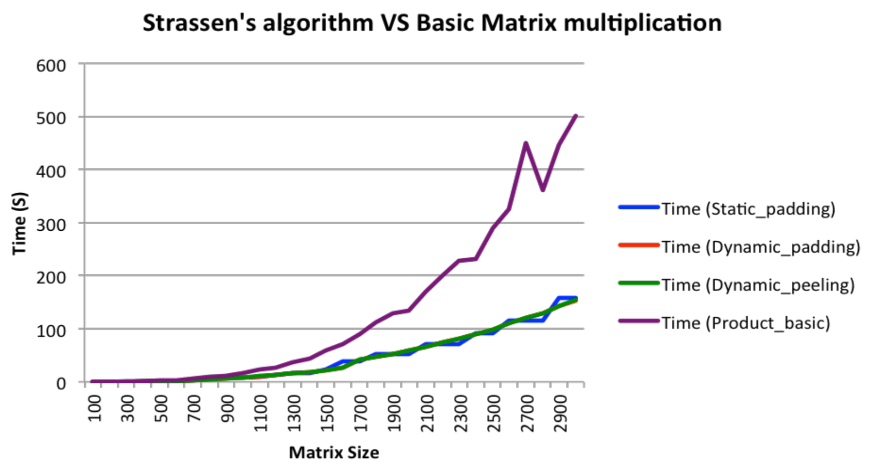

Figure 1: Performance of Strassen vs basic matrix multiplication

Also we calculate the speed up achieved by each strategy of Strassen's algorithm compare to basic multiplication.

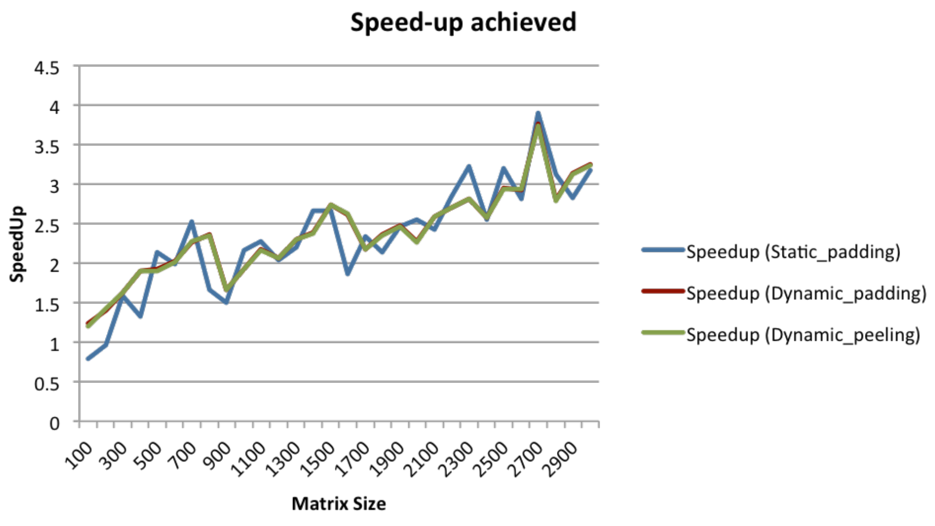

Figure 2: Speed up gain compare to basic matrix multiplication

As we can see in the graphs, Strassen algorithm is much faster compare to basic matrix multiplication. Speed up is almost reaching 3 for most of the cases, which means Strassen algorithm is 3 times faster than basic matrix multiplication for larger matrices. From above results, we figured that dynamic peeling is much better approach compare to other approached used. Due to that, when finding the optimal cut-off I used dynamic peeling.

4.1 Determining the optimal cut-off value

As mentioned earlier, cut-off value plays an important role for the performance of the Strassen's algorithm. When the matrix size is less than the cut-off size, matrix multiplication is done by the basic matrix multiplication. To figure out the optimal cut-off value, I measure the performance with changing cut-off values such as 12, 24, 36, 48, 52, 56, 60, 64, 72 and 84 and see how the performance numbers vary. To get the performance numbers, I used dynamic peeling approach. Initially I increase the cut off by 12. But when I update the cut off from 48 to 60, I saw a considerable amount of improvement. To get a better understanding, I change the cut off by 4 around that range. Speed up is also calculated compare to basic matrix multiplication in order to figure out the cut-off clearly.

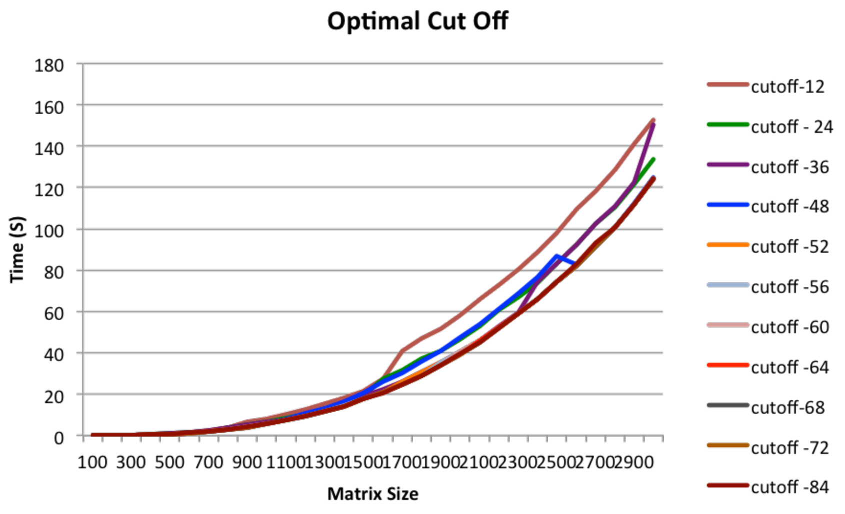

Figure 3: Performance of dynamic peeling with different cut off values

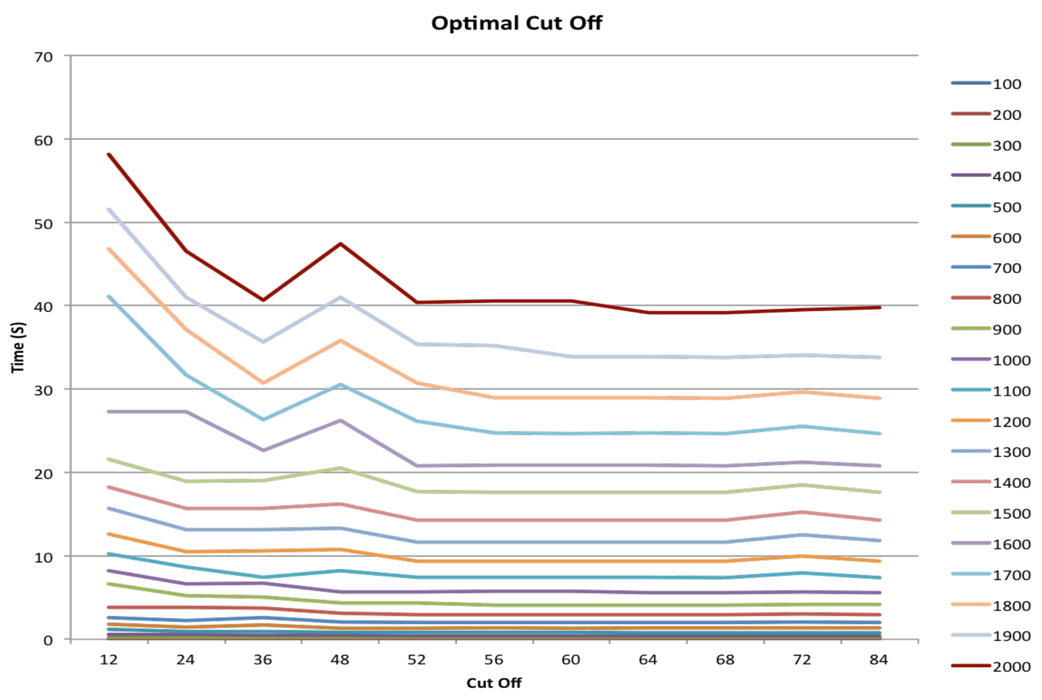

Figure 4: Dynamic peeling with different cut off values

Figure 4 shows a better indicator of the optimal cut off value compare to Figure 3. From figure 4, we can see that optimal cut off value is a range around 52 – 72.

I calculated the speed up gain with changing cut off values compare basic matrix multiplication in order get a better understanding of the optimal cut off value.

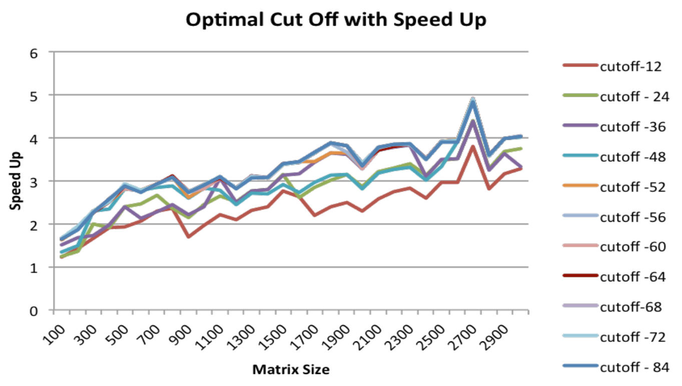

Figure 5: Speed up gain with different cut off values

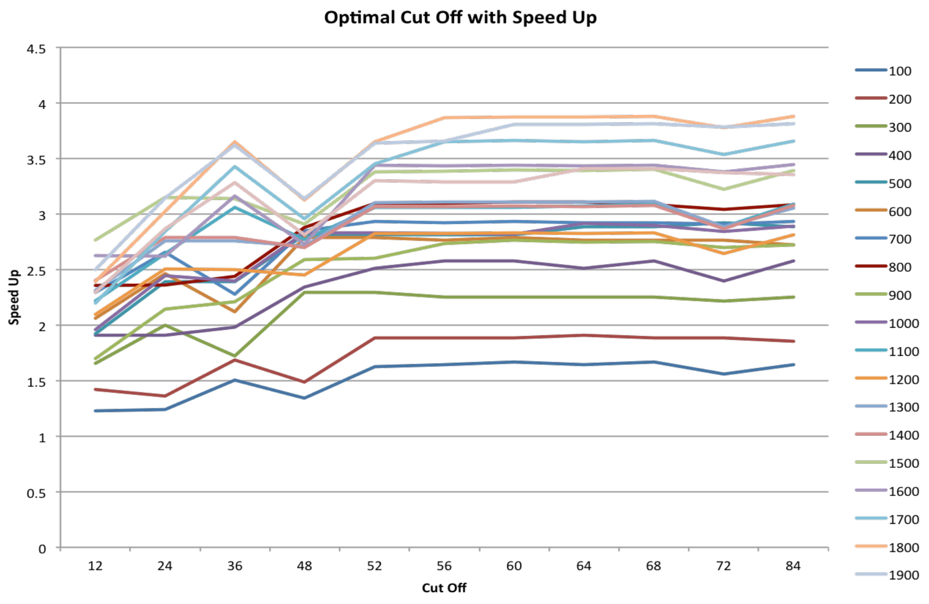

Figure 5: Speed up gain with different cut off values

In the initial test case, I choose the cut off as 12. According to the results, it seems 12 is not the optimal cut off for Strassen's algorithm. Below graph shows how fast dynamic peeling executes when we use an optimal cut off value.

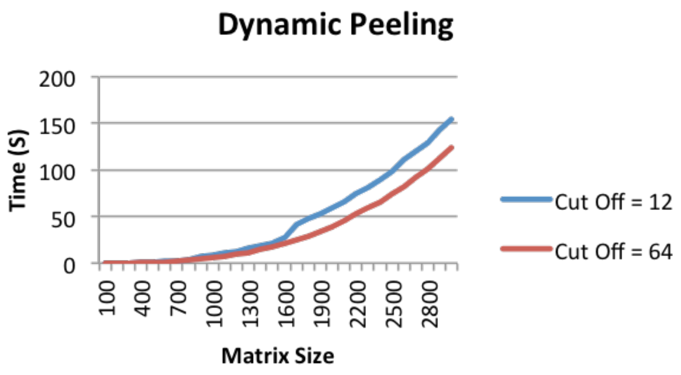

Figure 6: Time difference with different cut offs

According to above graphs, we can see that we can't say an exact value as the cut-off. We can define a region for cut-off where the Strassen's algorithm runs fast. According to the results I get, I can say that cut off should be around 52 – 72 region. According to the speed up graphs, we can reach speed up to 5 which means Strassen's algorithm is 5 times faster than the basic matrix multiplication. Speed up gain depends on the machine I used. I ran the same test in my desktop and I was able to get much better speed up numbers.

5.0 Discussion

When comparing the performance numbers, Strassen's algorithm performs really well compare to basic matrix multiplication. Speed up gain is more than 5 when using a better cut off value for large matrices compare to basic matrix multiplication. When implementing the Strassen's algorithm, we need to allocate memory for intermediate matrices several times. For the case of dynamic padding and dynamic peeling, we need to allocate memory for all the sub matrices that is being created since we do the padding and peeling dynamically. In static padding, we do not allocate memory for sub matrices during recursions but need to allocate memory for a bigger matrix than input matrices at the very beginning. Even with the extra memory allocation, still the performance numbers outnumber basic matrix multiplication for a great extent. According to the results, we saw that Strassen's algorithm can be perform 5, 6 times faster than the basic algorithm when we choose an optimal cut off value.

Performance numbers can be varying due to several factors. One major factor that affects the performance is the load of the machine. I ran all the tests in HULK machine, which is a shared resource. I also ran the same tests in a desktop machine of my lab (with IU network) and able to achieve lot faster results compare to HULK.

Also to get more accurate performance numbers, we should do the same test several times and get the average.

6.0 Techniques used to speed up the algorithm

To speed up the Strassen's algorithm, we followed several approaches.

1. Implementing several variations of the Strassen's algorithm

In this project we implemented several variations of Strassen's algorithm such as static padding, dynamic padding and dynamic peeling. We compare the results that we get for these three variations to decide which approach is the best.

2. Minimize the memory allocation for the Strassen's algorithm

When implementing static padding and dynamic peeling, I reused input matrices as much as possible in order to reduce the extra matrices that we need to create. Dynamic padding is implemented without giving much consideration to memory usage since I can use it as an indicator of the effect of memory usage.
According to Strassen's algorithm, input matrices are divided into four sub matrices of size/2. Instead of allocating extra memory for these 4 sub matrices, we can use the same input matrix if we use the stride of the input matrix when doing the other calculations. Also at the end of the Strassen's algorithm, we can use the same output matrix when combining results. Since these improvements are done at the Strassen's algorithm level and it is called recursively, such improvements have a considerable impact on the performance.

Here are some code snippets, which uses strides of input matrices.

```
void add_matrix(int m, int n, int *A, int as, int *B, int bs, int *C, int cs) {
    int i = 0, j = 0;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            *(C + i * cs + j) = *(A + i * as + j) + *(B + i * bs + j);
        }
    }
}
```

```
void subtract_matrix(int m, int n, int *A, int as, int *B, int bs, int *C,
int cs) {
    int i = 0, j = 0;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            *(C + i * cs + j) = *(A + i * as + j) - *(B + i * bs + j);
        }
    }
}
```

3. Figure out an optimum cut-off value

As discussed earlier, cut-off value plays an important role for the performance of the Strassen's algorithm. In static padding, it is used when finding the closest even number that should be used, which helps to reduce the size of the padded matrix to a great extent. This improvement helps static padding a lot. According to our results, we can see that static padding also performs equally well with dynamic peeling.

In dynamic peeling, results taken with cut off 12 and results taken with cut off 64 vary in a considerable amount.

Matrix Size (m x m x m)	Time (S), c=12	Time (S), c=64
1000	8.36	5.54
1100	10.52	7.43
1200	12.77	9.34
1300	15.75	11.65
1400	18.42	14.28
1500	21.8	17.62
1600	27.26	20.84
1700	41.38	24.71
1800	47.65	28.93
1900	52.33	33.85
2000	58.78	39.16
2100	66.11	45.93
2200	74.09	52.56
2300	80.81	59.07
2400	89.82	65.9
2500	98.88	74.23
2600	110.66	82.12
2700	120.27	91.93
2800	129.35	100.66
2900	142.81	111.9
3000	154.64	124.07

Table 1: Dynamic peeling with cut off 12 and 64

According to the above results, we can see that time changes by 30 seconds when the matrix size is 3000 which is a considerable gain.

7.0 Conclusion

In this project we compare the performance of the basic matrix multiplication and improved Strassen's algorithm. Strassen's algorithm is implemented using three strategies: static padding, dynamic padding and dynamic peeling. According to the performance numbers we get, we can conclude that the Strassen's algorithm is much faster compare to basic matrix multiplication. Also we can conclude that dynamic peeling approach is the better option among the three strategies.
Also we conclude that cut off value we choose plays an important role for the performance of the Strassen's algorithm. We ran considerable amount of tests in order to find the optimal cut off value. According to our results, cut off value can be in a range instead of an exact value. For our scenario, we get the range as 52 – 72 and we choose 64 as the optimal cut off value. With that cut off value, we are able to achieve speed-up to 5,6 times for large matrices.

8.0 References

1. Huss-Lederman, Steven, et al. "Implementation of Strassen's algorithm for matrix multiplication." *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*. IEEE, 1996.
2. Jacobson, Elaine M., et al. *Strassen's Algorithm for Matrix Multiplication: Modeling, Analysis, and Implementation*. Center for Computing Sciences, Institute for Defense Analyses, 1996.