Optimizing Strassen's Algorithms

Chihang Wang

6 Jan 2016

1 Introduction

The last report discussed the underlying maths of the Strassen algorithm. A baseline Strassen algorithm which only works for power matrix multiplication was implemented. In this report, I discuss how the algorithm is generalized to work on any size of matrix multiplication.

2 Terminology

size:
A matrix which has m rows and n columns is said to have size (m, n)
power matrix:
A power matrix is a matrix of the size $(2^m, 2^m)$
matrix multiplication size:
we use (m, n, k) to refer to a matrix multiplication between a (m, k) sized matrix and a (k, n) sized matrix, the result matrix will be of the size (m, n)

3 Strassen's Algorithm

Strassen's algorithm computes 7 sub-matrices and then use them to compute the output matrix. The outline is as the follow:

$$M_1 = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$$

$$M_2 = (A_{2,1} + A_{2,2}) \times B_{1,1}$$
$$M_3 = A_{1,1} \times (B_{1,2} - B_{2,2})$$
$$M_4 = A_{2,2} \times (B_{2,1} - B_{1,1})$$
$$M_5 = (A_{1,1} + A_{1,2}) \times B_{2,2}$$
$$M_6 = (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2})$$
$$M_7 = (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})$$

Afterwards, the four sub-submatrices is computed as:

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$
$$C_{1,2} = M_3 + M_5$$
$$C_{2,1} = M_2 + M_4$$
$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

3.1 Problem with the size

The baseline version of the algorithm works for power matrices only. For other sizes of matrices, the submatrices will be of different sizes, and the matrix arithmetic/multiplication becomes a little miserable. Consider the following $(3, 3, 3)$ matrix multiplication:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

submatrix splition will give the following:

$$A_{1,1} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} A_{1,2} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$A_{2,1} = \begin{bmatrix} 1 & 1 \end{bmatrix} A_{2,2} = \begin{bmatrix} 1 \end{bmatrix}$$

Consider the computation of M_5, which is:

$$M_5 = (A_{1,1} + A_{1,2}) \times B_{2,2}$$

There are two problems:

1. The addition on the left hand side is adding a (2, 2) matrix with a (2, 1) matrix, which is not clearly defined.
2. $B_{2,2}$ is a (1, 1) matrix, the size of two matrices mismatch and hence the multiplication is undefined

3.2 Use of padding

3.2.1 Overview

The natural choice is zero-padding. For the previous question. We can pad the two input matrices to the following:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Then we can use the baseline Strassen algorithm to correctly give the output:

$$\begin{bmatrix} 3 & 3 & 3 & 0 \\ 3 & 3 & 3 & 0 \\ 3 & 3 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

After which we then copy the desired section back to the result matrix.

3.2.2 Implementation

The most brute implementation is to pad the input matrices to power matrices. Since we already have the baseline version. This merely means we need to add a pre-processing stage of zero padding, and a post-processing stage to copy the result to the desired output matrix. The pseudo code is given below:

The main problem is the wasted computation. For example, (1025, 1025, 1025) matrix computation will first transform the problem to (2048, 2048, 2048) matrix multiplication. The wasted computation is 7x, far exceeding the gain from using the Strassen algorithm. Also, the extra memory required to hold the new, zero-padded matrices can be a problem if memory is a concern.
One modification which significantly improves the situation is to realize that the base case, i.e. when normal matrix multiplication routine is called instead

Algorithm 1 StrassenPadToPowerMatrix

```
procedure StrassenPadToPowerMatrix(A, B, C)
    $s \leftarrow getDimensionOfNewDimension(A, B)$
    $newA \leftarrow newMatrix(s, s, 0)$
    $copyTo(A, newA)$
    $newB \leftarrow newMatrix(s, s, 0)$
    $copyTo(B, newB)$
    $newC \leftarrow newMatrix(s, s, 0)$
    $StrassenBaseLine(newA, newB, newC)$
    $copyTo(newC, C)$
```

of recursive call to strassen, may not be (2, 2, 2). For example, if the base case is (32, 32, 32), any matrix multiplication of which the size is smaller than (32, 32, 32) will not recurse down. (960, 960, 960) will recurse 5 levels to reach (30, 30, 30), and then use regular algorithm to compute the product. This suggest that padding to a power matrix is not necessary. We only need to pad it to a size which can be written as $(a * 2^n)$ where $a < 32$

For the (1025, 1025, 1025) example above, it is sufficient to pad it to (1088, 1088, 1088), as $1088 = 17 \times 2^6$, after 6 level of recursive call, the base case is reached.

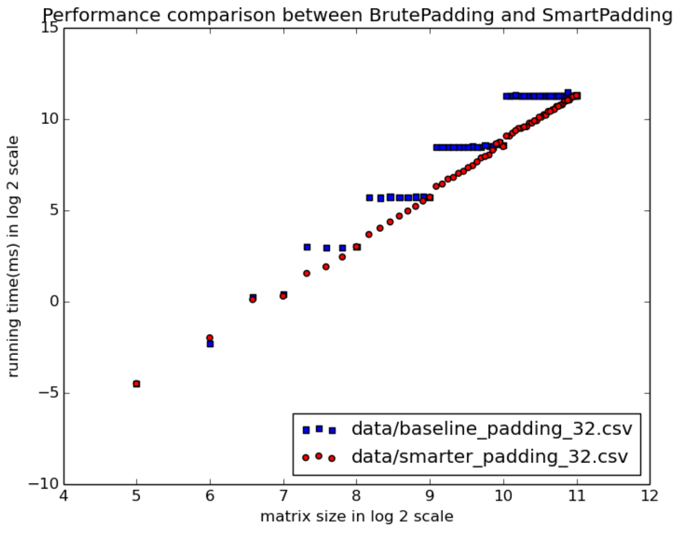

Figure 1: Running Time without compiler optimization

Figure 1 compares two padding strategies. The disadvantage of naive padding is evident.

3.3 An optimized pad-free version

3.3.1 Overview

It is possible to have an algorithm which does not involve padding. When memory is a concern, especially in mobile ssystem, this can be desirable. The key reason why padding can be avoided is because the computed values are thrown away anyway. The padding is added to make the algorithm regular. We hack the algorithm to enable it to deal with irregular input.

3.3.2 Irregular matrix addition/subtraction

The computation of M_i requires matrix subtraction and addition. When the size mismatches, the zero-padding version first zero-pad the smaller matrix to a equal-sized one and then perform the standard matrix arithmetic routine. However, we realize that the zero-padded values does not contribute to the output anyway in the addition/subtraction. Hence, we can devise a "partial" addition/subtraction routine which works as the following:

Algorithm 2 PartialAddition

```
for $position$ in $C$ do
    if $position$ defined in $Smaller$ then
        $C[position] \leftarrow Larger[position] + Smaller[position]$
    else
        $C[position] \leftarrow Larger[position]$
```

3.3.3 Irregular matrix multiplication

If we say A has the size (m, k), B has the size (k, n), C (m, n). After splitting the matrices, we expect the submatrices to have the size:

$$\begin{bmatrix} C_{1,1}(m_1, n_1) & C_{1,2}(m_1, n_2) \\ C_{2,1}(m_2, n_1) & C_{2,2}(m_2, n_2) \end{bmatrix} = \begin{bmatrix} A_{1,1}(m_1, k_1) & A_{1,2}(m_1, k_2) \\ A_{2,1}(m_2, k_1) & A_{2,2}(m_2, k_2) \end{bmatrix} \times \begin{bmatrix} B_{1,1}(k_1, n_1) & B_{1,2}(k_1, n_2) \\ B_{2,1}(k_2, n_1) & B_{2,2}(k_2, n_2) \end{bmatrix}$$

where

$$m = m_1 + m_2$$

$$m_2 + 1 \geq m_1 \geq m_2$$

$$n = n_1 + n_2$$

$$n_2 + 1 \geq n_1 \geq n_2$$
$$k = k_1 + k_2$$
$$k_2 + 1 \geq k_1 \geq k_2$$

For M_1, $A_{1,1} + A_{2,2}$ will have the size (m_1, k_1) after partial addition, $B_{1,1} + B_{2,2}$ (k_1, n_1). Hence, the multiplication will be of the size (m_1, k_1, n_1), the result is of the size (m_1, n_1)

For M_2, the multiplication has size (m_2, k_1, n_1), the result is of the size (m_2, n_1)

For M_3, the multiplication has size (m_1, k_1, n_2), the result is of the size (m_1, n_2)

For M_4, there is a problem, LHS is $A_{2,2}$, which has the size (m_2, k_2), RHS is $(B_{1,1} - B_{2,1})$, which has the size (k_1, n_1). When k is an odd number, $k1 \neq k2$, the dimension mismatches, hence problematic.

Take the previous example

$$B_{1,1} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \; B_{2,1} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

The computation of $B_{1,1} - B_{2,1}$ using partial subtraction gives:

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

This means we are computing

$$\begin{bmatrix} 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

Which is undefined.

The old approach is to zero-pad $A_{2,2}$ to:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

However, notice that extra zero-padded rows are unnecessary. Increasing zero-padded rows only increases zeroed rows in the product matrix, hence this can be avoided, which means we have reduced the problem to:

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

Next, extra zero-padded columns are also unnecessary. They merely imply that the corresponding rows in the RHS matrix does not contribute to the final output. Hence, instead of zero-padding LHS, we can reduce the RHS dimension, which gives:

$$\begin{bmatrix} 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \end{bmatrix}$$

This approach means that we are multiplying (m_2, k_2) with (k_2, n_1), the result is of the size (m_2, n_1)

For M_5, similar problem exist, we apply the same strategy to have the multiplication of size (m_1, k_2, n_2), the result is of the size (m_1, n_2)

For M_6, the multiplication has size (m_1, k_1, n_1), the result is of the size (m_1, n_1)

For M_7, the multiplication has size (m_1, k_2, n_1), the result is of the size (m_1, n_1)

3.3.4 Further refinement

There is one more minor detail subject to modification. Notice that the size of $M_1, M_2, M_3, M_4, M_5, M_7$ are all smaller or equal to the submatrices of $C_{i,j}$ that they are added or subtracted to. However, M_6, which is added to $C_{2,2}$, has the size of (m_1, n_1), but $C_{2,2}$ has the size of (m_2, n_2).

This means that there are wasted computations when $m_1 != m_2$. To avoid this, we can adopt the following:

$$M_6 = (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2})$$

Notice the last row and last column of the M_6 is unused. So, in the computation of $(A_{2,1} - A_{1,1})$, we can ignore the last row of $A_{1,1}$ by treating $A_{1,1}$ as a (m_2, k_1) matrix, instead of a (m_1, k_1) one. In the computation of $(B_{1,1} + B_{1,2})$, we can ignore the last column of $B_{1,1}$ by treating $B_{1,1}$ as a (k_1, n_2) matrix instead of a (k_1, n_1) one.

3.4 Reduce the Space requirement

A simple version of the algorithm may first compute all of the 7 utility matrices M_i, and then assign them to C_j. This means that the extra memory space required will be up to $7x$ of the input matrices.

A better approach is to compute M_i one by one. Immediately after computation, assign it to the designated submatrices of C_j, after which, the space can be used to compute the next utility matrix.

Also, notice that during the computation of M_i, temporary space is needed to hold the intermediate value from addition/subtraction. For example, in the computation of M_2, it is likely we need the following:

$$T = A_{2,1} + A_{2,2}$$

$$M_2 = T \times B_{1,1}$$

In order to avoid this temporary space in memory critical system, one can do the following:

$$A_{2,1} += A_{2,2}$$

$$M_2 = A_{2,1} \times B_{1,1}$$

$$A_{2,1} -= A_{2,2}$$

This will not affect the overall asyptotic complexity of Strassen's algorithm, since addition/subtraction is $O(n^2)$. However, the overhead will be larger.

3.5 Internal Implementation of a Matrix

It worth briefly mention how a matrix is implemented internally. While it is natural to represent a matrix by a 2 dimensional array, or pointer to pointer, it is much more convenient to represent it as a single pointer in a row-major fashion.

For a newly constructed matrix of the size (m, n), a new space of $m \times n$ will be allocated in the heap, and the pointer Ptr to the first entry will be returned. However, it is important to notice that (Ptr, m, n) is not sufficient to represent the matrix efficiently.

For instance, consider in the matrix multiplication routine when we want to pass submatrix $A_{1,1}$ to the subroutine. While A is well-defined with the triplet $(PtrA, ma, na)$, $A_{1,1}$ is not defined by $(PtrA, ma/2, na/2)$! This is because that while A is a contiguous memory space, $A_{1,1}$ is not. The difference between the (i, j) position and $(i + 1, j)$ position is not $(ma/2)$, but (ma).

To summarize, to define a matrix, we need a quadriple $(Ptr, m, n, incRow)$ where:
Ptr:
the pointer to the first entry
m:
the number of rows
n:

the number of columns
incRow:
the number of element between (i, j) element and $(i+1, j)$ element

This convention makes techniques discussed in section 3.3 handy. In order to treat a (k_1, n_1) matrix as a (k_2, n_1) matrix, simply change the quadriple from $(Ptr, k_1, n_1, incRow)$ to $(Ptr, k_2, n_1, incRow)$

Splitting is also handy, we move the Ptr to the new starting entry, and modify the height and width of the matrices to the desired values, and yet keep the $incRow$ constant.

3.6 Size of Base Case

Strassen's algorithm uses 18 matrix addition/subtraction to reduce the number of matrix multiplication from 8 to 7. We know the matrix multiplication has complexity $O(n^3)$, while the addition/subtraction is $O(n^2)$

Hence, mathematically, the place to stop to call recursively should satisfy the following:

$$n^3 \geq 18 \times n^2$$

which is when $n = 18$. Hence, we should stop recursive call when the matrix multiplication is of the size smaller than (18, 18, 18).

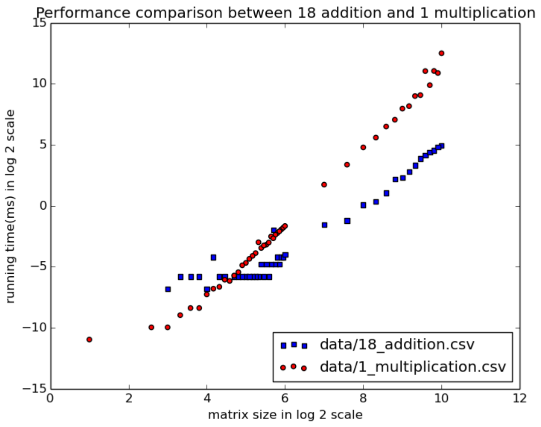

Figure 2: Compare 18 addition and 1 multiplication

The figure suggest that setting base case to 32 is a good option. In practice, call-stack setup in recursive call, etc, contribute to the overhead, hence it is desirable to increase the base case to even larger.

Figure below compares the performance with that of ATLAS GEMM:

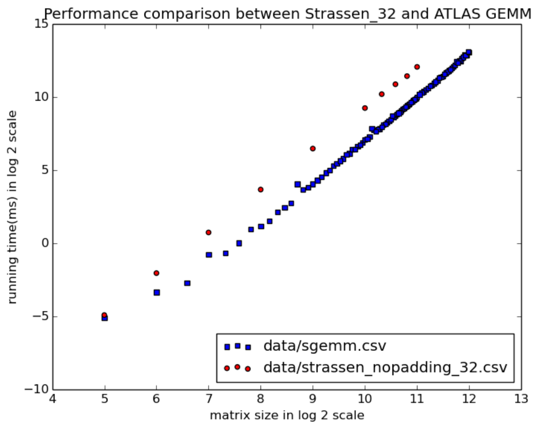

Figure 3: Compare to ATLAS Algorithm

Evidently, ATLAS GEMM is still much faster. It is known that ATLAS GEMM uses an efficient cache-blocking algorithm. If using ATLAS GEMM as the baseline algorithm, we have the following performance:

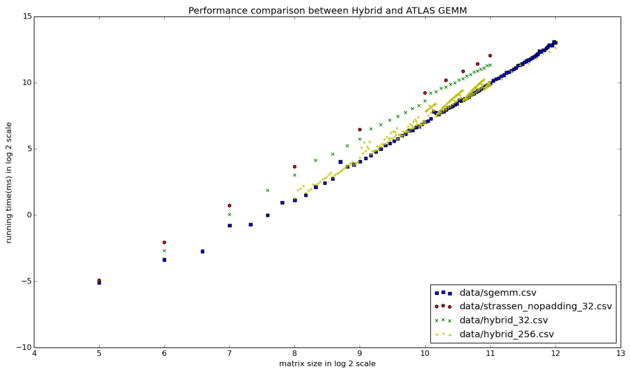

Figure 4: compare hybrid ATLASGEMM

When we increase the size of base case to $(256, 256, 256)$, the performance matches the performance of ATLAS GEMM for small matrices size, and beat it when the size becomes large.

The table below illustrate the speed up of hybrid version more clearly:

Dimension	GEMM	Hybrid
256	2.32	2.37
512	16.6	18
1024	145	127
2048	1042	927
4096	8148	6482

Table 1: Running time (ms) comparison between GEMM and Hybrid

......

1. Discussion on why ATLAS is faster 2. Choice of Base case. When smaller, cache oblivious. When larger, more optimization such as pre-fetching, assembly level can take place.