On Improving the Memory Access Patterns During The Execution of Strassen's Matrix Multiplication Algorithm

Hossam ElGindy and George Ferizis
School of Computer Science & Engineering
The University of New South Wales
Sydney, NSW, Australia
{hossam,gferizis}@cse.unsw.edu.au
FAX: +61 2 9385 5995

Abstract

Matrix multiplication is a basic computing operation. Whereas it is basic, it is also very expensive with a straight forward technique of $O(N^3)$ runtime complexity. More complex solutions such as Strassen's algorithm exist that reduce this complexity to $O(N^{\log_2 7})$; the recursive nature of such algorithms place a large burden on memory systems due to temporary storage and the lack of locality in their access patterns

In this paper we propose a scheme for reordering the matrix entries stored in memory. This reordering provides two major benefits: a simple method to transform the recursive algorithm into an iterative one, and also a simple method for maintaining memory locality over the entire operation. These two features both provide an improvement in performance that grows as the problem size increases.

The proposed reordering scheme has been implemented in C. Testing of our C implementation, which eliminates the need for unnecessary storage of matrix elements from previous iterations, with matrices of size up-to 2048×2048 exhibits improvement of 27.05% and 8.9% over the original algorithm and another reordering scheme respectively.

1 Introduction

Matrix multiplication is a common problem in comptuer science, that is expensive to solve with a complexity of $O(N^3)$. Many algorithms have been proposed to reduce this complexity, with algorithms such as Strassen's matrix multiplication algorithm [Str69] or variants of it such as Winograd's [FP74] being able to reduce it to $O(N^{\log_2 7})$ or less. They achieve this reduction by recursively dividing the larger problem into smaller sub-problems then combining the partial results from solving these sub-problems.

This recursive approach presents serveral problems: the recursive overhead in maintaining a stack, finding the optimal point of truncation for the recursion and the need to deal with odd sized matrices.

We propose a reordering method that simplifies the unrolling of the recursion into an iterative call and also causes an increase in memory locality as the algorithm executes. It is shown that this has positive effects on performance that increases as the size of the matrices that are being multiplied increase.

Previous research efforts into reordering of matrix elements [CLPT99, FW97] have used hierarchical data structures such as quad trees to reorder the elements of the matrices being multiplied. These orderings (such as Z-Morton) are used to allow the final matrices left for multiplication to be contiguously laid out in memory and hopefully, fit into cache.

These orderings however do not deal with any locality issues as the matrix is being broken down and then merged back together in the recursion. They also do not take try to extract the benefits of reordering a step further, by removing the recursive overhead and any associated temporary storage this incurs. The reordering system that we propose is designed to deal with locality in the actual recursive stages, as well as giving the ability to remove the recursion from the algorithm.

The resulting iterative approach also allows for the depth of recursion to be chosen at run-time, and in a similar method used by Chatterjee etal [CLPT99], statically pad an irregular matrix in a near optimum manner.

In this paper we describe Strassen's algorithm and then describe the reordering method that we have used, as well as how it leads us to be able to iteratively execute the algorithm. The results from experiments done comparing this method with other methods are listed as well as implementation details.

2 Strassen's algorithm

The multiplication of two matrices can be expressed as $C = AB$, where A, B and C are $N \times N$ matrices.

The use of straight forward techniques to calculate the result of this product requires N^3 multiplications. This value grows rapidly as the size of the matrices being multiplied increases, two matrices of dimensions 1024×1024 would require $107,374,1824$ calculations. Strassen [Str69] proposes a recursive method to multiply two matrices that has approx. $N^{\log_2 7}$ operations, which reduces the number from the previous by a factor of 8. This method relies on recursively breaking down each of the matrices that are to be multiplied into 4 sub-matrices, corresponding to the north-west, north-east, south-west and south-east quadrants and performing arithmetic operations on them.

If N is an multiple of 2, we can break each matrix into 4 $\frac{N}{2} \times \frac{N}{2}$ sub-matrices, as shown in equation(1). Otherwise we are presented with the problem of odd sized matrices. This problem, along with solutions to it are discussed later.

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \tag{1}$$

It can be observed from this that the following equations are true:

$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned} \tag{2}$$

From these equations the following has been derived.

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned} \tag{3}$$

where

$$\begin{aligned} P_1 &= A_1 \cdot B_1 \\ P_2 &= A_2 \cdot B_2 \\ P_3 &= A_3 \cdot B_3 \\ P_4 &= A_4 \cdot B_4 \\ P_5 &= A_5 \cdot B_5 \\ P_6 &= A_6 \cdot B_6 \\ P_7 &= A_7 \cdot B_7 \end{aligned} \tag{4}$$

where

$$\begin{aligned} A_1 &= a \\ A_2 &= a + b \\ A_3 &= c + d \\ A_4 &= d \\ A_5 &= a + d \\ A_6 &= b - d \\ A_7 &= a - c \end{aligned} \tag{5}$$

and

$$\begin{aligned} B_1 &= g - h \\ B_2 &= h \\ B_3 &= e \\ B_4 &= f - e \\ B_5 &= e + h \\ B_6 &= f + h \\ B_7 &= e + g \end{aligned} \tag{6}$$

One larger matrix multiplication has been reduced to 7 smaller matrix multiplications. It is possible to break each multiplication down recursively, until a matrix size is reached that can be operated on easily. So it can be seen the algorithm follows the following steps:

1. Recursively divide the matrices A and B using the operations above until the truncation point of the recursion is reached.

   This is the point at which the matrices are deemed of a size that is sufficiently small that they can be multiplied using standard techniques that run in $O(N^3)$ time.

2. Multiply these matrices.

3. Using the equations above combine the results of the multiplications to produce a resultant matrix C.

It should be noted that variations to Strassen's algorithm exist such as Winograd's [FP74]. They use a different set of arithmetic equations that take advantage of partial sums calculated, for additional savings in the number of additions calculated. For the purposes of this experiment they weren't considered for comparison, as the paritioning of them matrices in both algorithms are identical, the memory access patterns are the same.

3 Algorithm Refinement

Our implementation of Strassen's Matrix Multiplication algorithm involved some refinements. These refinements will be examined in this section.

They refinements are:

- The execution of the algorithm on odd sized matrices.
- The reordering method that was used and the benefits it presents.
- The unrolling of the recursion into an iterative loop and the associated changes in the amount of memory used during execution

3.1 Odd sized matrices

As can be observed Strassen's algorithm operates by dividing a matrix into four equal submatrices. If a matrix has odd dimensions, this cannot be done by simply finding the pivot points in the matrix and splitting along the axises they would provide.

Strassen [Str69] originally proposed a technique named **static padding** that involves the padding of the initial matrices with extra columns and rows so that all of the matrices produced during the entire procedure would be even in dimension.

Another technique is **dynamic padding**. This technique adds extra columns and rows to the matrices as needed during the recursive procedure.

Another technique is **dynamic peeling**. **Dynamic peeling** relies on the removal of columns and rows to make the matrices even in dimension. The remaining portions of the matrix have Strassen's algorithm executed on them and extra calculations are done later to place the stripped columns into the result of the execution of Strassen's algorithm.

Similarly to Thottethodi et al [TCL98] we use static padding. Assuming two $N \times N$ matrices are given for multiplication, with the size of the final matrix desired for multiplication being $M \times M$, the resultant value of $\log_2\left(\frac{N}{M}\right)$ must be an integer for the entire process to work correctly. There are two variables that may be changed N or M. Changing N or M would correspond with either padding the initial matrices that are given for multiplication, or changing the size of the final matrix to be multiplied.

We take the example of two matrices with dimensions 258×258 that are to be multiplied, with the matrix size at the truncation point of the recursion to be 16×16. If the final size at the truncation point is fixed at 16×16, we must pad the matrices to be of dimension 512×512. This results in the number of items being stored increasing by 294%. This is clearly not a desirable option. If the dimensions of the matrices at truncation point are changed to 17×17, it can be seen that the initial matrices need only be padded

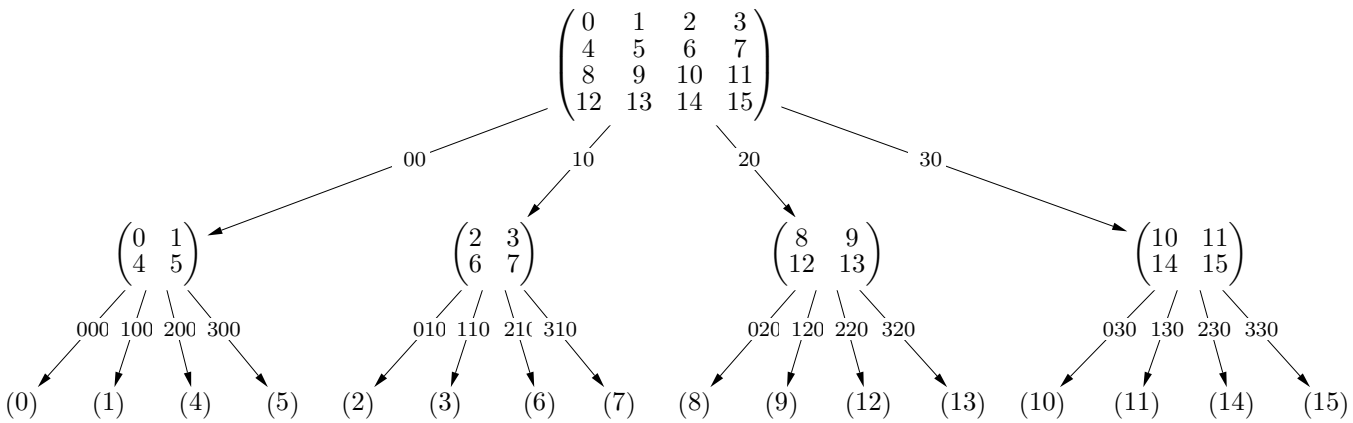


Figure 1: Quad-tree indexing with the numbering reversed. The keys for a node are shown in the directed edge pointing to it.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

Figure 2: Example 4x4 matrix

(0 2 8 10 1 3 9 11 4 6 12 14 5 7 13 15)

Figure 3: Array of values reordered

to a dimension of 272×272. This is a dramatic improvement on the previous scenario with the number of items being stored only increasing by 11%.

The technique we use, adopts this strategy. It attempts to find an optimum level of padding by changing both the size of the final matrix for multiplication and padding the initial matrix.

3.2 Reordering method

Past work into reordering has attempted to give a performance increase with respect to locality during the splitting of the matrices [CLPT99]. This work involves reordering the elements of the matrices into an arrays that it can be split into 4 contiguous blocks until the truncation point for the recursion is reached.

This however does not take into account any locality in how the matrices will be placed back together, with the subsequent additions and subtractions that occur after the sub-matrices have been multiplied.

As can be seen Strassen's algorithm breaks a matrix into 4 sub-matrices, which then are operated on to produce 7 matrices. The process is repeated with these sub-matrices until the truncation point of the recursion is reached.

Given 4 submatrices A, B, C, D, all the operations between the matrices occur on elements in the same position in each matrix, ie., any operation will involve only $(a_{i,j}, b_{i,j}, c_{i,j}, d_{i,j}) \forall (i, j)$. Our reordering scheme attempts to place these 4 values as close to each other as possible, while still maintaining the matrices that will be finally multiplied in contiguous memory blocks.

This reordering is achieved by taking a standard level-ordering such as simple quad tree indexing and reversing the order in which the numbers per level are placed together to achieve an order. An example of this is seen in figure 1.

As can be seen, the higher order portion of the final index produced is based on the lowest depth of the tree, not the highest depth as is usually the case. An example of this reordering for a 4×4 matrix in figure 2 can be seen in figure 3.

Examination of the reordered array shows that $a_{(0,0)}$, $b_{(0,0)}$, $c_{(0,0)}$ and $d_{(0,0)}$, are the first 4 values in the array. They are then followed by $a_{(0,1)}$, $b_{(0,1)}$, $c_{(0,1)}$ and $d_{(0,1)}$.

The first 4 values will produce the values for $a_{1:(0,0)}$, $a_{2:(0,0)}$ and so on until $a_{7:(0,0)}$ with the arithmetic operations shown in equation (5). The next 4 will create $a_{1:(0,1)}$, $a_{2:(0,1)}$ and so on until $a_{7:(0,1)}$.

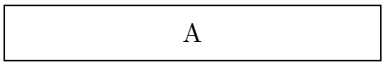
When the next stage of iteration attempts to compute the values for the submatrices produced by $A1$, it will find they are not in contiguous blocks of memory as was the case in the first iteration, but are separated by a distance of 7 values. This impacts negatively on the memory locality we wish to achieve throughout the execution of the algorithm, as related values move further apart. This is overcome by passing over the array several times, and is described in the next section.

This distance increases with the number of iterations and conforms to the geometric sequence of 7^i where i is the iteration, starting at 0.

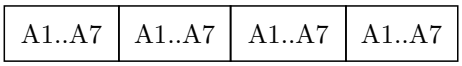
This distance relationship is shown in figure 4.

3.3 Unrolling of recursion

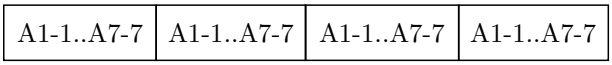
The regularity shown in figure 4 allows the recursion in Strassen's algorithm to be unrolled into an iterative process.



Initial array holding A



Array after iteration 1, with values spaced apart by 7



Array after iteration 2, with values spaced apart by 49

Figure 4: The layout of the resulting array with only a single pass through the initial array.

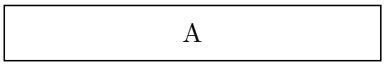
This iterative process scans over the array for each iteration and calculates all the matrices that would be made for that iteration by examining the relevant values. The matrices in both arrays are then multiplied. For each iteration the resulting array is again passed over and merged using the equations shown in equations (3).

As can be seen the values being read move apart from each other as the number of iterations increases. This clearly does not maintain locality.

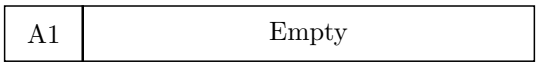
This was solved by passing over the array multiple times, 7 during the splitting phase and 4 during the merging phased, and only writing out the results of one set of the relevant equations per pass.

The example in figure 5 shows how the splitting phase is done for matrix A. The first pass goes through the matrix and computes all the values corresponding to $A1$ and places them into the array contiguously. The second pass will then do the same for $A2$ and so on and so forth.

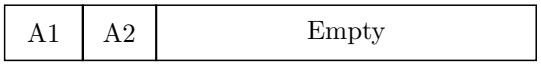
This process allows for all values to be placed in a contiguous section for each iteration, at the cost of multiple passes over the array.



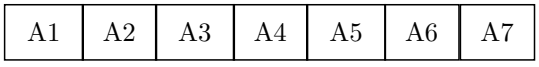
Initial array holding A



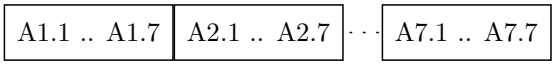
Array after the first pass of iteration 1



Array after the second pass of iteration 1



Array after the final pass of iteration 1



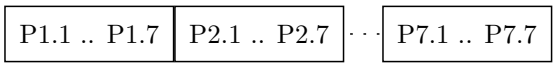
Array after the final pass of iteration 2

Figure 5: The layout of the resulting array with multiple passes through the initial array.

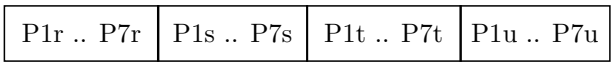
This method is used to split the array for every iteration except for the last one. As there is no need to maintain locality on this level for splitting as the method mentioned previously and shown in figure 4 is used.

To maintain locality during the merging phase a similar method is used. As shown in figure 6, the output from the multiplication will have the 7 values needed to produce values for each sub-matrix grouped together. Placing the results from all of these simply in a row in a similar method to figure 4 will cause the same problems as the splitting phase.

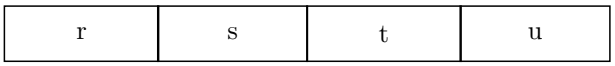
It can be seen that the values related in these matrices come from the sets of r, s, t, u. Thus we do a similar operation to above and group all values of r together, all values of s together and so on.



Array after the multiplication



Array after iteration 1 of merging.



Array after iteration 2 of merging.

Figure 6: The layout of the resulting array with multiple passes through the initial array.

3.4 Memory Use

With a change to an iterative algorithm from a recursive algorithm we find that not only memory access patterns are changed but that the amount of memory that is being used also changes. The differences between the recursive and iterative processes in respect to the amount of memory used will now be examined.

For this comparison, the size of the initial matrices for multiplication are $N \times N$ with the size of the matrices at the truncation point of the recursion being $M \times M$.

3.4.1 Recursive Approach

The first parameter to be introduced is the depth of recursion. This value is:

$$\log_2 \frac{N}{M} \tag{7}$$

.

Non-Parallel Solution

For a non-parallel approach to solving this problem, for each level of recursion 6 matrices need to be stored at maximum while the 7th matrix is being computed. For any level of recursion i the size of each matrix is:

$$\frac{N^2}{4^i} \tag{8}$$

.

Thus we are presented with the figure below for the maximum memory use by the elements of a single matrix and it's sub-matrices during the entire execution of the recursive algorithm:

$$N^2 + \sum_{1 \le i \le \log_2\left(\frac{N}{M}\right)} 6\left(\frac{N}{2^i}\right)^2 \tag{9}$$

With the geometric sequence in equation (10), equation (9) approaches the value shown in equation (11). Thus the number of elements being stored during the multiplication of both matrices can be seen in equation (12).

$$\lim_{i\to\infty}\left(\sum_i \frac{1}{4^i}\right) = \frac{4}{3} \tag{10}$$

$$\lim_{i\to\infty}\left(N^2 + \sum_i 6(\frac{N}{2^i})^2\right) = 9N^2 \tag{11}$$

$$18N^2 \tag{12}$$

Parallel Solution

A parallel solution needs to store at maximum every value in the recursive call as the tree grows (assuming a thread is started for every recursive call). Each level would require the following number of matrices to be associated with it:

$$7^i \tag{13}$$

where i starts at 0.

Thus for the entire recursive call, we observe the following is the maximum number of elements being stored at any time:

$$\frac{8}{3} \times \left(\frac{7}{4}\right)^{log_2 \frac{N}{M}} N^2 \tag{14}$$

3.4.2 Iterative Approach

During the iterative approach there is no dependency between any iteration i and any previous iterations. Therefore with no temporary storage from previous iterations, we need only examine the number of items being stored during the final iteration.

During an iteration i every 4 values will be substituted for 7 values. Thus we see the following:

$$size(i+1) = \frac{7}{4} size(i) \tag{15}$$

If this series is expanded, we can see that for an arbitrary iteration i, a matrix that began at size $N \times N$ will be of size:

$$\left(\frac{7}{4}\right)^i N^2 \tag{16}$$

.

With the number of iterations being the same as the depth of recursion ($\log_2 \frac{N}{M}$) as shown in equation (7), the following equation gives the memory use for both matrices:

$$2 \times \left(\frac{7}{4}\right)^{\log_2 \frac{N}{M} - 1} N^2 \tag{17}$$

As there are no dependencies between iterations, this value is the same for a parallel implementation.

3.4.3 Comparison

It is immediately obvious that the parallel implementation of the iterative approach has better memory use than the parallel implementation of the recursive approach. However this is not true of the non-parallel approach. To find when the iterative approach gives better memory use than the recursive, when the process is executed in a non-parallel manner, we compare equations (12) and (17).

$$\begin{aligned} 2 \times \left(\frac{7}{4}\right)^{\log_2 \frac{N}{M} - 1} N^2 &\leqslant 18N^2 \\ \left(\frac{7}{4}\right)^{\log_2 \frac{N}{M} - 1} &\leqslant 9 \\ \log_2 \frac{N}{M} - 1 &\leqslant \log_{\frac{7}{4}} 9 \approx 3.9 \\ \log_2 \frac{N}{M} - 1 &< 4 \\ \log_2 \frac{N}{M} &< 5 \end{aligned} \tag{18}$$

From equation (18) it can be seen that the number of iterations must be less than or equal to 4 for the iterative approach to provide better memory use.

4 Implementation

This algorithm was implemented using the C language to gain experimental results.

The implementations were run on an AMD Athlon 2000+ system, with 512 MB of RAM. Runtime was measured using standard C functions for measuring the process runtime.

4.1 Reordering

Whereas the reordering we suggest is most easily implemented recursively, it has been implemented iteratively to keep in the spirit of removing the recursion from the execution of the algorithm. The reordering of values is done twice for any implementation that has reordering, once for the initial inputs, and once to reorder the final output into a row major ordering.

The inputs are assumed to be in row major order.

4.2 Splitting

The splitting phase operates in the manner described below:

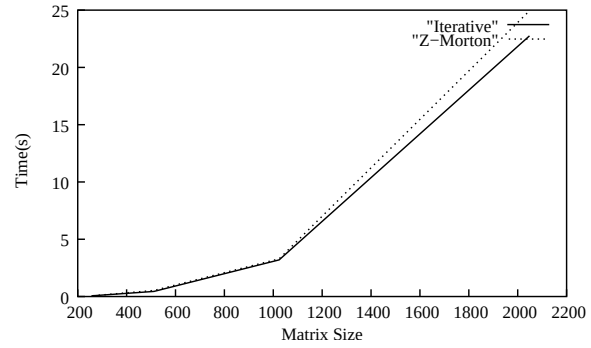
```
int array: arrayin
int array: arraytemp
for i = 1 to 7
   Scan the array arrayin and process
   every 4 consecutive values to compute
   all the elements Ai where Ai is
   defined in equation 5.
arrayin = arraytemp
```

This occurs at every iteration except the last one, where the number of iterations is the depth of the recursion that would occur. This is $\log_2 \frac{N}{M}$ where N is the size of the matrices to be multiplied and M is the size of the final matrices that will be multiplied using normal techniques.

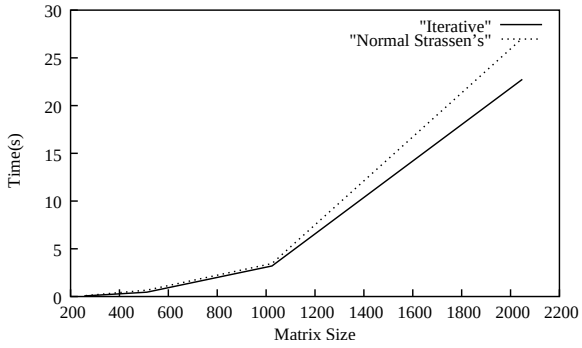
It should be noted that by the word value, a single integer is not meant. What is meant is a series of integers, where the size of the series is equal to the number of items in the final matrices at the truncation point of recursion.

This is the set of operations for the matrix A. We would use the other set of equations (6) for matrix B, in a similar fashion.

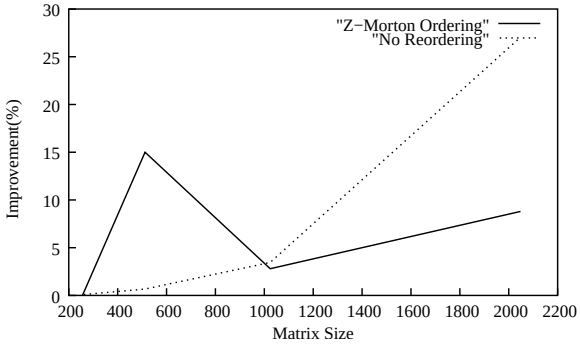
As can be seen for each iteration *arrayin* is passed over seven times. Each pass takes in 4 values at a time from the array and evaluates one of the seven arithmetic equations for that matrix before outputting a single value into array *arraytemp*. This is how the locality is maintained, with the use of iteration.



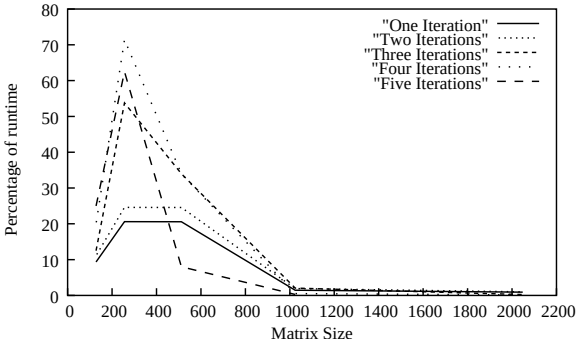
(a) Z-Morton ordering [CLPT99] and our iterative approach



(b) Regular Strassen's algorithm and our iterative approach



(c) Relative Performance Figures



(d) Reordering time as a percentage of total run time

Figure 7: Results

4.3 Multiplication

Following this the values in both matrices are multiplied. From each array, subsets of integers that contain enough integers to create matrices of the size that had been set for multiplication are taken and added. These are multiplied and placed into an array.

4.4 Merging

A similar method is used for the merging portion of the implementation, however it only requires 4 passes as the number of terms is only 4.

```
int array: arrayin
int array: arraytemp
for i in r, s, t and u
   Scan the array arrayin and process
   every 7 consecutive values to compute
   all the elements i where i is
   defined in equation 3.
arrayin = arraytemp
```

It works by taking 7 values in from the array and working one of 4 arithmetic equations on them in order. This again occurs for every iteration.

5 Results

The following results came from the implementation being written in C. The comparative results in figure 7 were run with a final matrix size of 128×128 which was found to be optimum for matrices larger than 256×256 in dimension.

The relative results show that with the exception of a spike for the 256×256 matrix multiplication, the performance improvement increases as the size of the matrix increases. The reason for this spike is not clearly understood. For the maximum sized matrices multiplied(2048×2048) the performance increase was found to be 8.9% and 27.05% for Z-Morton reordering and no reordering respectively.

In figure 7(d) we can see that the reordering of the matrices takes a substantial time for smaller matrices, however as the matrix size grows the relative cost becomes negligible. As expected with more iterations to be run, the reordering cost as a percentage is higher. This is due to the reordering taking longer to calculate due to the number of indexes being larger.

6 Conclusion

We have introduced a method of reordering that attempts to achieve better memory locality for all stages of Strassen's matrix multiplication algorithm. This reordering method has also been used to transform Strassen's algorithm from a recursive process to an iterative process. This could lead into future work in attempting to place the algorithm directly into hardware using a purely iterative process, without the aid of a co-processor as in other implementations of Strassen's algorithm.

References

[CLPT99] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, 1999.

[FP74] P. C. Fischer and R. L. Probert. Efficient procedures using matrix algorithms. *Automata, Languages and Programming*, pages 413–427, 1974.

[FW97] Jeremy D. Frens and David S. Wise. Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.*, 32(7):206–216, July 1997.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[TCL98] Mithuna S. Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of Super Computing 98*, 1998.

[WF99] David S. Wise and Jeremy D. Frens. Morton-order Matrices Deserve Compilers' Support. *Technical Report 533, Computer Science Dept., Indiana University*, 1999.