DEGREE PROJECT IN COMPUTER ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2016

Optimizing Strassen's multiplication algorithm for modern processors

A study in optimizing matrix multiplications for large matrices on modern CPUs

ROBERT WELIN-BERGER

ANTON BÄCKSTRÖM

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Optimizing Strassen's multiplication algorithm for modern processors
Optimering av Strassens multiplikationsalgoritm för moderna processorer

A study in optimizing matrix multiplications for large matrices on modern CPUs

ROBERT WELIN-BERGER
ANTON BÄCKSTRÖM

Examensrapport vid CSC
Handledare: Michael Schliephake
Examinator: Örjan Ekeberg

Abstract

This paper examines how to write code to gain high performance on modern computers as well as the importance of well planned data structures. The experiments were run on a computer with an Intel i5-5200U CPU, 8GB of RAM running Linux Mint 17.

For the measurements Winograd's variant of Strassen's matrix multiplication algorithm was implemented and eventually compared to Intel's math kernel library (MKL). A quadtree data structure was implemented to ensure good cache locality. Loop unrolling and tiling was combined to improve cache performance on both L1 and L2 cache taking into regard the out of order behavior of modern CPUs. Compiler hints were partially used but a large part of the time critical code was written in pure assembler. Measurements of the speed performance of both floats and doubles were performed and substantial differences in running times were found.

While there was a substantial difference between the best implementation and MKL for both doubles and floats at smaller sizes, a difference of only 1% in execution time was achieved for floats at the size of 2^{14}. This was achieved without any specific tuning and could be expected to be improved if more time was spent on the implementation.

Referat

Den här rapporten undersöker hur man skriver kod som medför en hög prestanda på moderna datorer så väl som vikten av väl genomtänkta datastrukturer. Experimenten utfördes på en dator med en Intel i5-5200U CPU, 8GB med RAM som kör Linux Mint 17.

För utvärderingarna implementerades Winograds variant av Strassens matrismultiplikationsalgoritm och jämfördes i slutänden mot Intels math kernel library (MKL). En quadtree struktur implementerades för att säkerställa god cachelokalitet. Looputrullning och blockning kombinerades för att förbättra cacheutnyttjandet av både L1- och L2-cacharna med avseende på out-of-order beteendet hos moderna CPUer. Kompilatorledtrådar användes till viss del men en stor del av den prestanda kritiska koden skrevs helt i assembler. Mätningar av prestanda för både floats och doubles gjordes och substantiella skillnader i körtid upptäcktes.

Medan det var en substantiell skillnad mellan den bästa implementationen och MKL för både doubles och floats vid mindre storlekar så uppnåddes en prestandaskillnad av endast 1% för floats vid en storlek av 2^{14}. Det här uppnåddes utan några specifika finkalibreringar och kan antas förbättras om mer tid spenderades på implementationen.

Contents

Chapter 1

Introduction

Processors have seen many improvements in the past few years. The main ones being improvements in register sizes enabling instructions to be executed symmetrically on multiple values. Processors have also received more layers of cache accessible from many cores on the same dies. Other improvements are improvements regarding better prefetching and branch prediction. The problem with these new advances being that they do not present any performance gain unless the code is specifically written with this in mind. To best test the advantage of these gains, matrix multiplication will be implemented to investigate how it compares to other implementations, naive ones as well as state of the art versions.

Matrix multiplication is often a requisite in mathematics. The most obvious case being in linear algebra and linear transformations. However, many other problems may be represented as matrices, such as group theory and many of its related fields. Some of these matrices may be huge and hence might take a long time to compute.

More often than not, execution speed of matrix multiplication is of great importance since it's often the limiting factor in many other algorithm. The calculations are typically run on large clusters but there are limitations to what can be done by increasing the size of the cluster, not to mention the fact that execution time is far from linear with the size of the cluster in all practical implementations.

The fastest known algorithms for matrix multiplication are those that are based on the results of the Coppersmith–Winograd algorithm. These have a time complexity on the order of $\mathrm{O}(n^{2.37})$ and vary depending on implementation. Unfortunately these algorithms are only of academic interest as their high constant factors make them slower than other algorithms on any data that can be computed on modern hardware. With this in mind, the algorithms subject to implementation are the naive one and Strassen's algorithm.

Strassen's matrix multiplication algorithm is of $\mathrm{O}(n^{2.81})$, which makes it the fastest algorithm in use and therefore the one examined in this paper. More specifically the report covers the Winograd variant of Strassen as it has a slightly lower constant than the original Strassen. The naive matrix multiplication algorithm and some variations of it are implemented for comparison. In order to analyze the performance of the implementation and theory several factors will be examined such as the FLOPS, latency and cache.

Normally, this algorithm is executed on large clusters due to being easy to multithread, however, in this case focus lies on the per thread performance and no threading will be implemented.

What this paper brings to the field is an analysis of optimizations that are implemented on the Haswell architecture, the results are equally relevant for all new and coming processors since instruction sets are mostly only extended. It should also be noted that these techniques are general and can be applied to many other algorithms apart from matrix multiplication.

1.1 Problem statement

This thesis investigates what coding optimizations can be applied to Strassen's matrix multiplication algorithm in order to implement it as efficiently as possible for modern processors using the C language. The areas subject to evaluation are in prioritized order:

- Vectorization and SIMD
- Data localization and caching
- Quadtrees using Morton ordering
- Loop unrolling and tiling

1.2 Scope of Thesis

The implementation and measurements are restricted to large quadratic matrices with dimension represented on the form 2^k where $8 < k < 15$. The limitations are put in place since matrices with different properties requires individual tuning to be comparable. The reason of this specific restriction is that these types of matrices make up the core of most Strassen implementations.

1.3 Thesis Overview

Section 2 introduces the reader to the Strassen algorithm and goes through the data structures that are needed for this implementation. Section 2 also goes through vectorization and what new instructions are available on modern hardware. In section 3 a closer look is taken at how and what has been implemented and more details are given regarding evaluation. In section 4 the results are presented and explained under what circumstances they have been achieved. In section 5 the results are interpreted and discussed and potential future work is discussed. In section 6 the final conclusions are presented.

Chapter 2

Background

This section aims to introduce techniques and findings that are known to optimize the Strassen algorithm as well as code in general. Some findings are specific for Strassen although in most cases Strassen was mainly used to show a more general technique as it is a relatively easy example.

2.1 Matrix multiplication

Matrix multiplication works by taking the dot product of rows from the A matrix and columns of the B matrix to create the elements of the C matrix. More specifically, given that A is a $n \times m$ matrix and B is a $m \times p$ matrix that can be written on the form:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}, \; B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mp} \end{bmatrix}$$

The matrix C is the product of AB and will be an $n \times p$ matrix that can be represented as:

$$C = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{np} \end{bmatrix}$$

The matrix C can be calculated by the formula:
$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$

When calculated on a computer it is of great importance to use the correct data types. Most systems have access to floats and doubles when accuracy is of the essence. Floats are typically 32 bit (4 byte) while doubles are 64 bit (8 byte). Floats therefore have around 7 decimals digits of precisions compared to the 15 to

16 digits of doubles. Floats can not contain as large numbers as doubles either with their maximum being close to $3e38$ while doubles can hold $1.7e308$.

2.2 Strassen

Strassen is a recursive algorithm used to calculate the multiplication of two matrices faster than the naive algorithm. Strassen is often chosen as it is the fastest algorithm that can practically be implemented on modern hardware. Strassen starts by splitting the given matrices into two 2x2 submatrices as shown in table 2.1. Strassen then starts adding and subtracting them in accordance to table 2.2 then it does 7 multiplications recursively and then recombining them to get the correct result as shown in table 2.3. If this was to be done with naive matrix multiplication, 8 matrix multiplication would be required because each of the 4 submatrices in C is the result of the sum of two matrix multiplications e.g. $C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$.

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

Table 2.1. Splitting the matrices into 2x2 matrices

$S_1 := A_{21} + A_{22}$	$P_1 := A_{11} * B_{11}$	$U_1 := P_1 + P_2$
$S_2 := S_1 - A_{11}$	$P_2 := A_{12} * B_{21}$	$U_2 := P_1 + P_6$
$S_3 := A_{11} - A_{21}$	$P_3 := S_4 * B_{22}$	$U_3 := U_2 + P_7$
$S_4 := A_{12} - S_2$	$P_4 := A_{22} * T_8$	$U_4 := U_2 + P_5$
$T_5 := B_{12} - B_{11}$	$P_5 := S_1 * T_5$	$U_5 := U_5 + P_3$
$T_6 := B_{22} - T_5$	$P_6 := S_2 * T_6$	$U_6 := U_3 - P_4$
$T_7 := B_{22} - B_{12}$	$P_7 := S_3 * T_7$	$U_7 := U_3 + P_5$
$T_8 := T_6 - B_{21}$		

Table 2.2. Additions and multiplications

$$C = \begin{bmatrix} R_1 & R_5 \\ R_6 & R_7 \end{bmatrix}$$

Table 2.3. Result matrix

Strassen can be computed in $O(n^{log_2 7+O(1)}) \approx O(n^{2.8074})$. Since additions and subtractions are n^2, any constant number of them does not affect the time complexity as the multiplication is closer to n^3. As the constants of Strassen is larger than in the naive algorithm, it is sensible to calculate a cutoff point where the recursive Strassen algorithm is switched over to the naive algorithm. This cutoff point is referred to as the recursion truncation point. When using the Strassen-Winograd variant the exact formula for the recursion truncation point is [1]:

$$\left(naive < strassen \right) \Rightarrow \left(\frac{n^3}{7 * (n/2)^3 + (n/2)^2 * 15} < 1 \right)$$

This means that for:
$n = 16 \implies 0.901$
$n = 32 \implies 1.007$
This would indicate that the recursion truncation point should be around 32 for the Strassen algorithm without vectorization, and a higher value for the versions with vectorization[2]. In practice tests has to be run to find the truncation point since it can be different depending on matrix size, hardware and implementation.

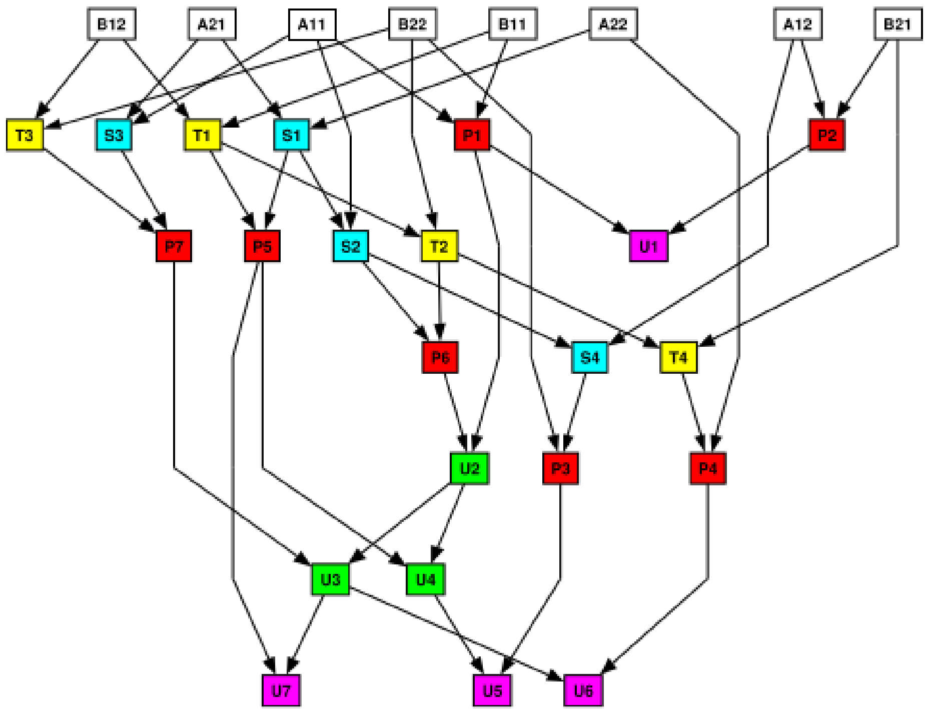

Figure 2.1. Winograd's task dependency graph by P. Bjørstadt et al [1]

The Strassen algorithm is not able to perform all of its calculations in place but need additionally memory. Strassen needs a total of 6 memory areas with the size of a submatrix for every recursion level, however, as it can make use of the 4 result matrices it only needs to allocate two additional matrices. This can be deduced by doing an exhaustive search for the minimum amount of submatrices needed at peak for every combination based on the restrictions in figure 2.1

2.3 Vectorization and SIMD

Normal instructions or so called single instruction single data (SISD) perform one instruction on one piece of data. Vectorization works by using single instruction multiple data (SIMD) instructions. The multiple data refers the to act of performing the current operation on an array of values at once as shown in figure 2.2 as opposed to the common one value. When calculating the speed of a program, it is common practice to look at the floating point operations per second (FLOPS) throughput and compare it to the theoretical maximum. A computer's maximum FLOPS is calculated as follows:

$$FLOPS = \#cores \times clock \times \#operations/cycle \times vectorsize/32 \ \times \ 2 \ with \ FMA$$

In the formula, #cores refers to the amount of cores on the CPU and "clock" is the clock rate. Furthermore, "#operations/cycle" is equal to the number of operations one core can execute every cycle and for most modern processors this value is four. Vectorsize divided by 32 is the amount of floats the processor can operate on at any one time using SIMD instructions. Lastly, the result of the entire formula is multiplied by two if FMA or equivalent instructions are performed as they are considered to execute two instructions, i.e. one multiplication and one addition. The most important aspect of the formula is the fact that the amount of FLOPS is linear to the size of the vector, leading to 100% speed increase for every element in the array excluding the original one. Based on previous reports, an approximate 90%+ of the maximal FLOPS [3] can be reached using vectorization for matrix multiplication.

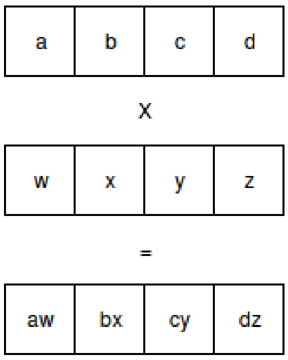

Figure 2.2. Example of a SIMD multiplication

SIMD instructions were first introduced during the 1970s although the modern SIMD instructions were not introduced until 1999 by Intel in the form of their SSE instruction set. The SSE instructions sets has since received several extensions with the latest version being SSE4.2 which was released in 2008 and later replaced by AVX and AVX2.

2.4 AVX2

The latest set of vector instructions from Intel known as AVX2 operates on 16 256-bit registers compared to the previous SSE4 instruction set which operates on 8 128-bit registers. AVX2 is a superset of the SSE4 instructions and indirectly replaces the SSE4 instructions by mirroring the names while appending a 'v' to the beginning. The replaced instructions are limited to operate on the lower 128 bits of the lower half of the 16 registers available to AVX2. The 256-bit registers are referred to as ymm registers and may only be used by the AVX2 instructions not derived from the SSE4 instructions. The lower 128-bits of the same registers are referred to as xmm registers as can be seen in figure 2.3. Although AVX comes with a few important instructions not available in the old SSE instructions and four times as much register memory, the key difference is the double vector-size which allows for twice as many operations at a time, effectively doubling the FLOPS.

One more thing that has been added together with the AVX2 instruction set is the FMA instruction set. FMA stands for Fused Multiply Add and means that both a multiplication and an addition can be performed at the same time, thus speeding up the code.

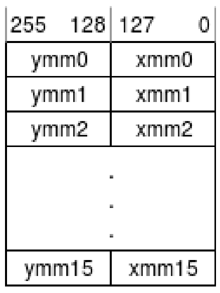

Figure 2.3. Registers used for AVX2 instructions

2.5 Memory management

Every Intel CPU with architecture subsequent to and including Haswell has 64kB of L1 cache storage and 256kB of L2 cache storage. This is of importance when the algorithm transitions into the naive algorithm at the recursion truncation point. As the dimension of the matrices are 32×32 there is enough memory to store almost 8 matrices in the L1 cache, greatly exceeding the two input matrices and the result matrix needed. This excess memory allows for a lot of prefetching of data which should grant increased performance. For the L1 cache there is 32kB of data cache that is 64B/line and is 8-WAY. A miss in the L1 cache results in latency of between 8-11 cycles, assuming the data can be found in the L2 cache.

2.6 Morton ordering

Instead of storing a matrix by the rows, Morton ordering allows the conversion of multidimensional arrays into one-dimensional arrays while preserving spacial locality [2][4][5]. It works by ordering the elements in a Z-like pattern repeated recursively in each quadrant of the matrix according to figure 2.4. By ordering the elements in a fashion similar to the one they are accessed may grant performance boosts by prefetching the data prior to usage.

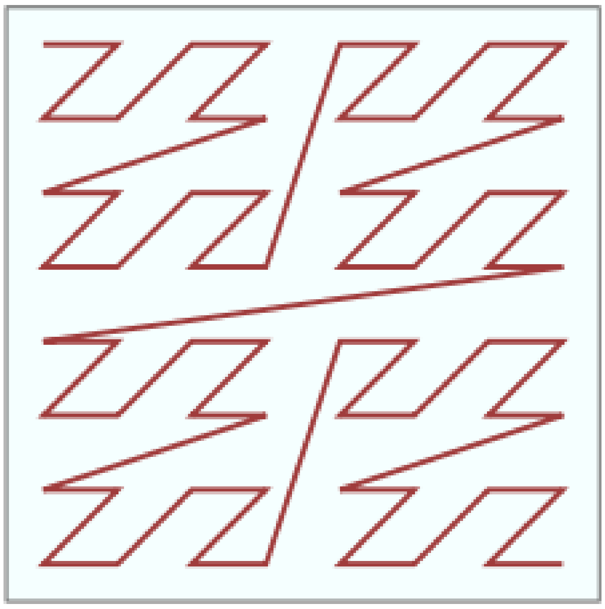

Figure 2.4. An 8x8 morton ordered matrix [6]

Strassen works by recursively splitting the matrix in four blocks. Arranging the elements of the matrix in a Z-like order therefore improves the spacial locality of the algorithm as well as negating the need of knowing the original matrix's dimension in each sub-part of the algorithm. An effective version of Strassen using Morton ordering has been demonstrated previously [5].

2.7 Quadtree

Quadtree is a tree based structure referring to quadratic sections of a matrix or similar structure. Quadtrees work well with morton ordering as both are implemented recursively in each quadrant of the matrix. Combining these two techniques grants an increase in memory locality of the matrix and therefore increased cache performance and faster execution times.

When comparing to a row based memory structure later referred to as linear memory, quadtrees have been observed to improve performance by more then 20% while also providing significant improvements over other forms of memory localization techniques [4].

2.8 Loop unrolling

Loop unrolling is a technique to minimize the amount of instructions needed per operation by doing many operations per loop. Normally 3 instructions would have to be executed for each intended operation, a compare, a branch and the intended operation. This is often unnecessary, if prior knowledge about the data is possessed, blocks of data can be operated over at a time by executing multiple hard-coded operations for every loop. By doing this the amount of instructions per operation can be lowered until reaching a ratio closer to 1 instruction per operation. Often, this can be done by the compiler by supplying hints and compiler flags, but sometimes this has to be done by hand.

Chapter 3

Method

To evaluate the different implementations a set of restrictions have been put in place. The matrices to be examined are quadratic with dimension 2^k. The reason being that it makes it far easier to implement the algorithms and focus can be spent on a broader comparison. The values are also restricted so that the result can be stored in a float without overflowing.

Different versions of matrix multiplication will be implemented to test the effectiveness of the various speedup methods and the level of speedup each of them provide.

3.1 Strassen

The version of Strassen that is going to be implemented is the Winograd variant of the Strassen algorithm. It is first going to be implemented using the default linear memory layout and then later going to be implemented using quadtrees. To simplify the rest of the implementations they are both going to use a linear memory layout at the truncation point. This ensures a symmetric behavior after the call and allows for a comparison with fewer factors. Unfortunately this has the effect that the matrices have to be copied multiple times in the linear version to ensure their linearity of memory.

3.2 Linear Layout

In the linear memory layout version the memory is represented with an array of pointers into the array. This enables the linear variants to access positions with precomputed offsets which enhances caching. The problem with this implementation is that while the offsets can be easily calculated the cache is very quickly going to be flooded by all the values that have a very high repeating distance pattern.

3.3 Quadtrees

The quadtree implementation that was used for this implementation is a group of pointers into a Morton ordered linear representation of the matrix. The truncation point needs to be the same as the size of the matrix that the assembler variant uses. A lot of the code for testing, adding and subtracting can be exactly the same for both versions of the memory layout since they both implemented using a linear memory space at the core. The base Node structure can be seen in figure 3.1.

```c
struct Quad {
  float *matrix;
  uint32_t elements;
  struct Quad *children[4];
};
```

quad.c

Figure 3.1. Structure of a quadtree node in C

3.4 Naive multiplication

This is only implemented as a point of reference to determine the relative speedup of each individual optimization technique and algorithm. The problem with this code is mainly that as soon as the sizes are getting large there are going to be a lot of cache misses and the CPU will be spending most of the time waiting for data to be fetched from higher levels of cache or even RAM. A lot can be learned from examining the behavior of this implementation since other versions are going to have a similar behavior when their block sizes become too large. A simplified version of this can be seen in figure 3.2

```c
for (size_t i = 0; i < SIZE; i++)
  for (size_t j = 0; j < SIZE; j++)
    for (size_t k = 0; k < SIZE; k++)
      C[i][j] += A[i][k] * B[k][j];
```

naive.c

Figure 3.2. Naive matrix multiplication algorithm implemented in simplified C

3.4.1 Tiling

The awful cache locality of the naive matrix multiplication algorithm can be mitigated by performing block-wise computations that are adapted to the size and characteristics of the L1 cache. Thanks to the better cache locality inherent to operating on data that is close in a linear representation, this version runs much faster than the version using the linear memory layout. The tiling version still uses the naive algorithm at its core and therefore has the same time complexity. This has the effect that it will run much slower than Strassen for larger matrices. A simplified version of this can be seen in figure 3.3. The tile size would be set during compile time for performance reasons.

```c
for (size_t i = 0; i < size; i += TILESIZE)
  for (size_t j = 0; j < size; j += TILESIZE)
    for (size_t k = 0; k < size; k++)
      for (size_t x = i; x < i + TILESIZE; x++)
        for (size_t y = j; y < j + TILESIZE; y++)
          C[x][y] += A[x][k] * B[k][y];
```

tiled.c

Figure 3.3. Tiled algorithm implemented in simplified C

3.5 Broadcast assembler

This assembly implementation of the naive matrix multiplication algorithm is based on the vbroadcastsd and vbroadcastss instructions for floats and doubles respectively. It takes the value in the lowest of an xmm vector register or a memory address and sets every value in the target ymm register to that value. This is used to broadcast a value from the left hand matrix into a vector register that can then be multiplied with a row from the right hand side to compute a partial result of that 256 bit segment as visualized in figure 3.4. This enables the full computation of two 4x4 double matrices with only the following instructions.

- 4 loads of the right hand matrix
- 16 broadcast of the individual values in the right hand matrix
- 16 FMA instructions to perform the calculations.

Thus the inner loop requires 36 instructions to perform 64 multiplications and additions. The float version operates in a similar fashion but operates on a 8x8 block as 8 floats can fit in a single ymm register.

- 8 loads of the right hand matrix
- 64 broadcast of the individual values in the right hand matrix
- 64 FMA instructions to do the actual calculations.

Thus the inner loop requires 136 instructions to perform 512 multiplications and additions.

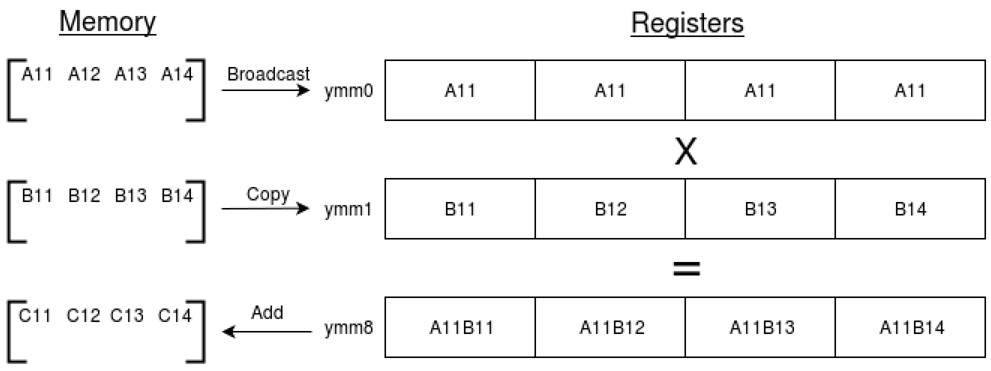

Figure 3.4. Data transfer during broadcast assembler

3.6 Comparison

When comparing the implementations it is important to isolate different properties. Therefore not all implementations are compared to every other implementation. The first subject of comparison are the different memory layouts. These are only compared to make sure there are no errors or corner cases in the Quadtree version as most previous work indicates it should be faster or equivalent in speed to using the linear memory approach. The next evaluation ensures an optimal usage of the cache is present by measuring varying block sizes for the different data sizes and algorithms. These comparisons are performed for every data type individually and the fastest combinations will be selected for further evaluation. The final process is to optimize the fastest version and compare it to the state of the art. The choice for state of the art implementation is the Intel math kernel library (MKL), a library of optimized math routines for science, engineering, and financial applications which has been hand optimized by Intel engineers for every version of their processors.

3.7 Benchmarking details

The algorithms are benchmarked on an Intel i5-5200U processor with 8GB of RAM and 8GB of SSD swap space. The machine is running the operating system Linux Mint 17 and the compiler being used is gcc. The time including converting it to a quadtree will be benchmarked to ensure that any possible precomputation will be taken into consideration. All measurements will be run for at least 1 second and at least 10 rounds to remove any anomalies in runtime. The state of the art program used for comparison run on the same machine but will instead be compiled using Intel's compiler ICC as it will not compile properly using another compiler.

Chapter 4

Results

This chapter will present the running times of the different implementations. Most of them are straight forward and many of them are only of interest when compared to each other. Graphs will be used to show general trends and patterns while the data is represented in tables when the data mainly overlap.

4.1 Reference implementation

This is the state of the art implementation used as a benchmarking reference to compare against. Here multithreading is also tested for feasibility. As can be seen from figure 4.1 the speedup from using 2 threads is almost two times as is expected. At the largest tested size the machine was reaching its RAM limit and since the multithreaded version uses more memory it started swapping to disk.

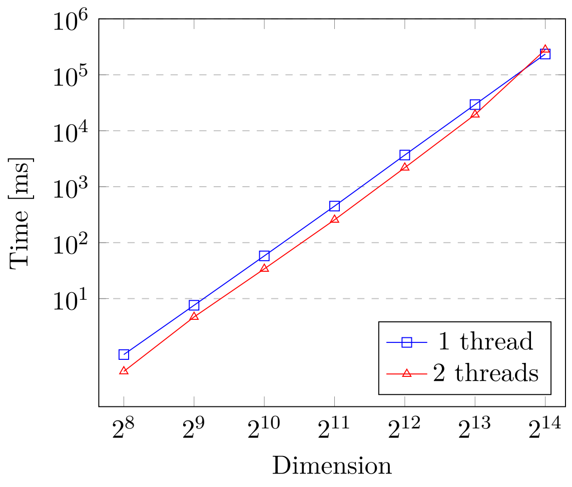

Figure 4.1. Threading performance of MKL for matrix multiplication using doubles

4.2 Loop expansion and caching optimization

This section examines the first experiment to determine what gains are possible to achieve using basic cache optimization techniques and compiler aided loop expansion. When looking at the smaller sizes the effect of loop unrolling can be seen as there is no indication that cache has any effect since there is no difference between float and double. Looking at table 4.1 it can be concluded that there is no difference in execution times between floats and doubles without vectorization until cache starts becoming an issue. As the sizes get larger the difference between the two versions become increasingly significant as the linear version starts missing cache progressively more.

	256	512	1024	2048	4096
Linear float	82	654	8508	77002	788621
Linear double	81	861	9637	89678	785962
Tiled float	13	108	937	7999	64142
Tiled double	13	128	1051	8524	68955

Table 4.1. Runtime [ms] at different matrix sizes for naive matrix multiplication with different memory layouts

4.3 Memory layout

Here the memory layout is examined to determine its effect on the runtime of the program. Both versions are using the same recursion truncation point which goes poorly for the linear version with its much larger overhead. The amount of extra copying and poor cache locality means that the copying itself has to start to read from RAM at larger sizes. Here it can also be seen that the float version of the Quad implementation is more than twice as fast as the double version, indicating that it is better optimized than the double version.

	512	1024	2048	4096	8192
Linear float	88	629	4063	30361	210675
Linear double	92	647	4476	30812	216759
Quad float	7	49	341	2276	15984
Quad double	19	133	910	6251	43657

Table 4.2. Runtime [ms] at different matrix sizes for Strassen's matrix multiplication algorithm with different memory layouts

4.4 Cache optimization

All of these test are performed using Quadtrees as it was the best layout in both theory as well as practice. When looking at graph 4.2 an interesting behavior can be seen. For smaller sizes the implementation runs increasingly slower relatively MKL and then gets progressively faster. This behavior is not seen in graph 4.3 but a similar progressive speedup can be seen. It should be pointed out that at 2^{14} in graph 4.2, the 128 block version is only 1% slower than MKL.

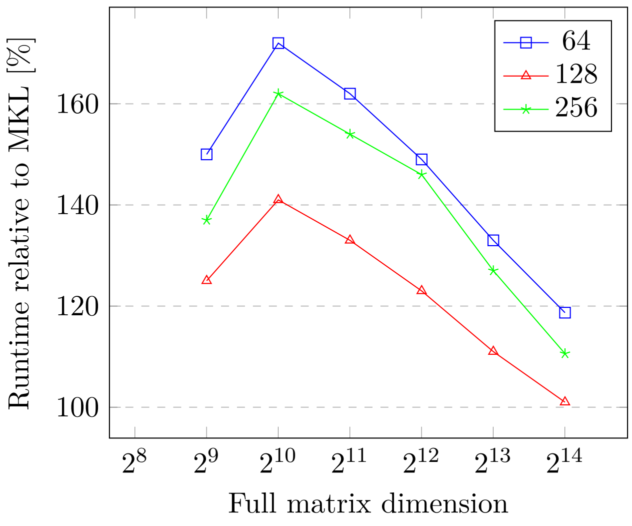

Figure 4.2. Strassen using floats at different truncation points

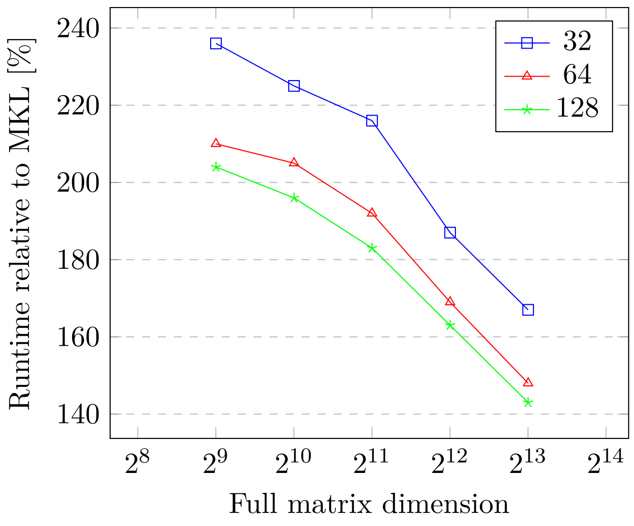

Figure 4.3. Strassen using doubles at different truncation points

Chapter 5

Discussion

This chapter aims to discuss the observations and results presented in the previous chapter. The first observation being discussed is the correlation between the runtime when using floats as opposed to doubles. The second and third section is about how the usage of quadtrees and tiled memory affects the speed of Strassen's multiplication algorithm at different recursion truncation points. The last section explains how our best implementations compare to MKL and what to expect if other tests would have been performed.

5.1 Float vs Double

As can bee seen in table 4.1, when using the naive matrix multiplication algorithm there is little to no speed difference when using floats compared to doubles. This can likely be attributed to many reasons but the dominant one would be the fact that the relative performance difference is overshadowed by the cache misses. However, when looking at the Strassen implementation in table 4.2 using floats yield a consistent 2.7 times speedup compared to doubles. This is something that in combination with the relative poor results in graph 4.3 compared to graph 4.2 clearly indicates that the double implementations are lacking something that the float versions have. This is probably because the float version has 4 times as many instructions for each load of the right hand matrix. This means that the effect of a cache miss is larger by up to a magnitude of four rather then the otherwise expected two times. The expected 2 times comes from the FLOPS formula explained in section 2.3. Since floats require four bytes and doubles require 8, a SIMD instruction may operate on 8 floats simultaneously but only 4 doubles using AVX2 instructions.

5.2 Linear memory vs Quadtree for Strassen

In table 4.2 it can be seen that the Linear memory version is around 12 to 13 times slower than the Quadtree version using the same truncation point. Should it be tried to be used for an actual implementation a much higher truncation point would be chosen to take the much larger overhead into consideration. There are several things that make the Quadtree faster but the most dominant reason is the fact that the Linear memory version requires a lot of copying of the data for each recursion. This is mostly because of the naivety of the implementation and something that could be lessened at cost of cache locality. Unfortunately that would make it more difficult to write a fast assembler version and was therefore never prioritized since it was expected for the Quadtree version to be faster. The small difference in execution times that can be seen between floats and doubles here can be entirely attributed to cache misses because of the doubled size of the matrices memory footprint.

5.3 Truncation points

When looking at the truncation points there are two things that affect the runtime, time complexity and cache misses. Because doubles are twice as large as a float it is possible that it will start having cache misses earlier than the float version which can bee seen in both of the graphs 4.2 and 4.3. In both versions the 128 version ends up being the fastest, but while it dominates in the float version it is much closer to the 64 version for doubles. Since the float versions are fastest at size 128 there is no need to test 256 for the doubles because it has strictly worse performance for larger sizes relative to floats. In regards to time complexity it gets much harder to calculate and while it can affect the runtime by about 10% per level shift, we can see from both graph 4.2 and graph 4.3 that the difference can be much larger than 10%, indicating that cache is the dominant factor.

5.4 Comparison to MKL

In graph 4.2 and 4.3 it can be seen that the 3 lines are decreasing after 2^{10}. The fact that they are all getting closer to the speed of MKL for larger matrix sizes indicates that MKL might be using an algorithm optimized for smaller matrices, meaning it has less overhead but worse time complexity than Strassen. MKL might also be using different algorithms for different sizes and is unfortunately something that we have no control over or even knowledge about. We only tested to the size of 2^{14} As most modern computers have 8GB of RAM, the next size being 32768×32768 is of no interest to calculate as it would require a substantial amount of transfers between primary and secondary memory increasing the runtime exponentially, measuring disk optimization rather than anything else.

Chapter 6

Conclusion

The results show that what was best in theory also came to be the best in practice, but it also shows that there are many details that require accounting for. The cache optimizations quadtrees and tiling proved vital to the performance of the many different implementations and even the smallest of difference could give up to 20% performance increase as shown in graphs 4.2 and 4.3. Quadtrees as a staple amongst data structures showed itself once again to be effective and delivered a substantial performance gain as shown in 4.2.

When looking at the relative performance measurement of graphs 4.2 and 4.3 it becomes clear that it is not sufficient to use the same implementation for different data types when maximum performance is desired. The need to test theory vs reality has also been shown with the recursion truncation point being very far off from where the theory calculated it to be.

It has been shown that at larger sizes MKL either uses the same combination of algorithms that was used here or they use something with equivalent performance. While it has been shown how to effectively calculate matrix multiplication for larger matrices there is still a lot of future work in smaller matrix sizes where quirks, tricks and alternative implementations are more important as we have pushed the theory to its currently known limits.

Bibliography

[1] Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. Memory efficient scheduling of strassen-winograd's matrix multiplication algorithm. *CoRR*, abs/0707.2347, 2007.

[2] Mithuna Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck. Tuning strassen's matrix multiplication for memory efficiency. In *IN PROCEEDINGS OF SC98 (CD-ROM*, 1998.

[3] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra. Optimizing matrix multiplication for a short-vector {SIMD} architecture – {CELL} processor. *Parallel Computing*, 35(3):138 – 150, 2009. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[4] Hossam ElGindy and George Ferizis. On improving the memory access patterns during the execution of strassen's matrix multiplication algorithm. In Vladimir Estivill-Castro, editor, *Twenty-Seventh Australasian Computer Science Conference (ACSC2004)*, volume 26 of *CRPIT*, pages 109–115, Dunedin, New Zealand, 2004. ACS.

[5] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. 2002.

[6] David Eppstein. Four iterations of the z-order curve., 2008.