Universiteit Leiden
The Netherlands

Opleiding Informatica

Automatically finding the best blocking size

for matrix multiplication

Yorick Bolster

Supervisors:
Aske Plaat & Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

14/07/2017

Abstract

An algorithm can only be computed as fast as the hardware will allow, but how do we make sure the hardware is used to the fullest possible extent? Matrix multiplication is a frequently used algorithm that can be improved by blocking. This thesis will try to identify the best blocking size for the matrix multiplication blocking algorithm on two different architectures and on various matrices. A number of blocking sizes will be tested, some derived from the cache size, which is determined at run-time by the program, others will be static values set from the start. The data collected from these tests will be used to determine the best blocking size for the local caches, and the best blocking size for the shared caches of both architectures. This outcome will be tested to see if it holds across different sizes and shapes of matrices. Lastly, it will be determined if a pattern can be found throughout the different architectures and the different matrices, so that a rule can be determined for calculating the best blocking size from the cache size.

Contents

Chapter 1

Introduction

In this chapter we give an introduction to the problem addressed in this thesis, and a concise description of the background to this problem. Linear algebra is used in various scientific disciplines, as it is a very useful way of storing and presenting data. In Linear Algebra and its applications 5th edition [1], each chapter of the book gives an example of a use of linear algebra. The examples range from Wassily Leontief describing the input and output of all subsections of the United States economy in a system of linear equations as described in chapter 1, to R. Lamberson using linear algebra to show the effect of deforestation to the population of the northern spotted owl as described in chapter 5. Other examples include using linear algebra to calculate the airflow on a plane or to organizing control systems in spaceflight, described in chapter 2 and 4 respectively. This sample of applications of linear algebra shows that its uses are diverse, and could possible never be said to have been fully explored.

A good example of how matrix multiplication has a place in almost anybody's life is its use in computer graphics, rotations in 3D space can be done with the use of multiplying a matrix representing an object by a special rotation matrix [2]. Matrix multiplications are calculated by computers to save time, thereby saving costs or increasing profits. If these computations could be done even faster, the cost saving or increase in profits would be even better. The way computers process these matrices creates a problem in modern systems that lends itself to a specific kind of optimization.

1.1 Problem definition

In modern systems memory speed is the restricting factor in getting computations done. Caches are used to try to solve this problem by providing small and fast storage close to the processor. However, replacement policies on these caches cannot always correctly predict what to keep and what to throw out, so they are almost never used optimally. The cache replacement policy is decided based on the most common actions a processor normally does, this results in there being instances of use where there is ineffective use of the cache.

One such instance can occur when a program uses large loops inside another loop that iterates over an array, large nested loops iterating over an array if you will. If the outer loop is very large, and all the information inside the inner loop is reused many times over, it results in ineffective use of the cache when the cache is not big enough. A cache that is big enough and can hold all the information for the inner loop throughout all outer loops would not be used ineffectively.

A replacement policy can have a large effect on how well the cache is being used when the situation with ineffective cache-use occurs. The worst case replacement policy for this is First-In-First-Out (FIFO), as the first index of the inner loop that was used, will also be the first index of the inner loop that is used again, and should actually be discarded last, see Section 2.2. Matrix Multiplication relies heavily on nested loops, and subsequently makes inefficient use of the cache(s) when large matrices have to be computed.

Loop blocking, also referred to as loop tiling, is a technique that utilizes a blocking size to make sure small chunks of the data that are reused in the loop are kept in the cache. Many experiments have been done with loop blocking under many different circumstances, such as loop blocking with a maximum blocking size [3], and loop tiling with the blocking size as a parameter [4]. In this thesis both parameters and fixed sizes will be used. The parameters will be set to a size that is ideal for the cache size assuming a FIFO replacement policy and fully associative cache. The model of a FIFO replacement policy is used because it is simple and also the worst case scenario.

The problem is that in practice using the simple model of assuming a FIFO replacement policy and fully associative cache cannot guarantee the best result. Different replacement policies, n-way associative caches, and smart caches create factors that this simple model does not accommodate. The questions that this thesis will try to answer about blocking algorithms for matrix multiplication are: With the cache size known, will the model of a fully associative cache with a FIFO replacement policy determine the best blocking size for loop-blocking for the local caches, and for the shared smart cache? If not, could another rule of finding the best blocking size through the cache size be created? Will the best overall performing blocking sizes perform the best over different shapes of matrices being multiplied? Lastly, will the overall best performing blocking size perform the best over all different matrix sizes?

1.2 Thesis Overview

This bachelor thesis was done at LIACS under the supervision of Kristian Rietveld and Aske Plaat. The thesis is divided into chapters, this chapter contains the introduction where the problem is introduced. Chapter 2 includes definitions and explanations on important concepts concerning the research question, explaining the background to the problem. Chapter 3 explains how the program that acquires data on the time it takes to do matrix multiplication with different matrix- and blocking-sizes is implemented, and also explains how this data will be used. Chapter 4 discusses the data that was acquired through the experiments. Chapter 5 recaps the conclusions drawn from the data.

Chapter 2

Background

In this chapter the background to the problem is laid out, expanding on how the problem in Section 1.1 has come to be. The gap between processor and memory speed, caches, the Berkeley Dwarfs, autotuning, loop blocking, dense linear algebra, and BLAS and ATLAS will be discussed.

2.1 Processor and memory speed gap

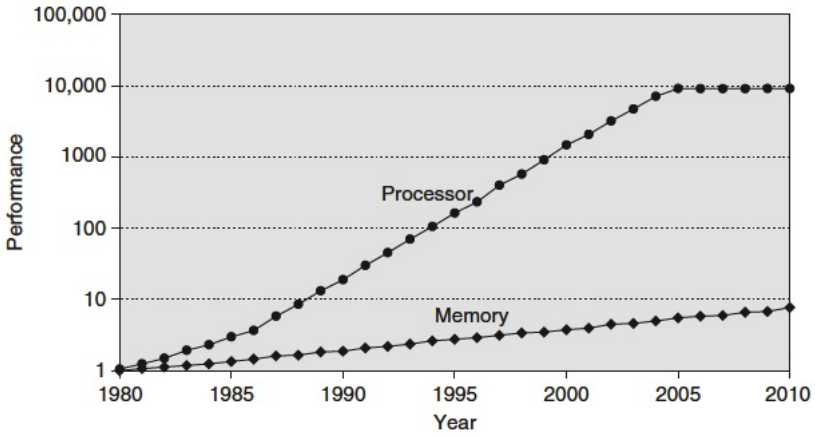
A regular processor is able to do significantly more computations than a main memory can provide data for during almost any process. The difference between a single core's performance and memory performance has been increasing exponentially from 1980 to 2005, see Figure 2.1 , and has not changed significantly since then. This disparity between processor improvement and main memory speed improvement is called the memory wall [5]. Single core processors stopped becoming exponentially faster from around 2005 mostly due to the power wall problem, the amount of electrical power used was too high for the small amount of space and could not be dissipated fast enough. The invention of multicore processors however, have further increased the magnitude of the memory wall.

Caches, see Section 2.2, are used to mitigate this problem. The more effectively the caches are being used, for example through blocking algorithms, the more the effects of the memory wall are mitigated.

2.2 Caches and replacement policies

Caches are used to mitigate the speed gap problem by providing small and fast storage close to the processor. When caches are used in multiple levels, becoming bigger the further away from the processor they are, it is called multi-level caching. All modern processors make use of multi-level caching, using various sizes and

Figure 2.1: The gap in performance between processor and memory, with 1980 as a baseline. The processor speed is measured as the difference in time between memory requests for a single core, and the memory speed is measured as the latency. Note that this is on a logarithmic scale. Figure taken from Computer Architecture: A Quantitative Approach, Fifth Edition, page 73.



either two or three levels. The last cache before the main memory (last level cache, or LLC) is often a cache that is shared among cores, while the others belong to a single core.

Cache hierarchies can be inclusive or exclusive. When an inclusive cache has data in it, it means that all lower level caches also have this data. For an exclusive cache this is explicitly not true. If an exclusive cache has a miss in L1, and a hit in L2, it will swap a line in L1 with the needed line in L2. If an inclusive cache has a miss in L1, and a hit in L2, it will copy the line from L1 to L2. Swapping costs more time than simply copying, but an exclusive cache hierarchy can store more data.

Caches need replacement policies in order to determine what to do when it is full and a miss is encountered. There are various replacement policies that all try to predict which cache entry is going to be the least useful in the future. These policies make use of information such as the amount of time that passed since use, the order of being loaded in the cache, or how many times an entry has been used. Some use more than one of these variables.

The replacement policies of caches do have some limitations in what information they put in which part of the cache. Some caches can put any information anywhere in the cache, and others can only use map certain parts of main memory to certain parts of the cache, this is referred to as the associativity of a cache. A cache that is free to put any information from the main memory anywhere in the cache is called a fully associative cache. The opposite of this is a direct mapped cache, which has each part of main memory mapped to a place in the cache. A direct mapped cache takes very little time to look up if an entry is in the cache, but has a larger amount of misses, and a fully associative cache has the opposite trade-off. In order to balance the trade-off of benefits and downsides between these two caches, the n-way associative cache was designed. In an n-way set associative cache, each part of main memory can be mapped to n possible places.

The FIFO replacement algorithm in combination with a fully associative cache is used to model the blocking sizes in this thesis, for this model see Section 3.1.3. The reason for this is that is constitutes the worst possible scenario for when the problem described in Section 1.1 occurs. When doing matrix multiplications, the

information within the inner loop is used many times over. When the inner loop does not fit inside the cache, data has to be discarded. The FIFO replacement policy discards exactly the information that is the first to be used again. Consider the scenario that an inner loop that requires x amount of entries that will be reused at each iteration of the outer loop, with the outer loop requiring no entries for data, and the cache having space for x-1 entries. If this cache has a FIFO replacement policy and a fully associative cache, it would have a cache miss at every data request, because during the second iteration of the loop, every data request will replace the data needed for the next request.

Smart caches are shared caches that were designed by Intel in order to make better use of the last level cache (LLC). Traditionally each core was given a part of the LLC to store information in, which resulted in a number of problems. If one core were to run a very data-hungry program, and the other core would run a program needing almost no data, a large part of the LLC would not be used while another part would constantly have too little. Another problem occurs when a process migrates to another core, in this case the program would encounter a large number of compulsory misses. The smart cache solves these problems by having all cores have access to the entire LLC. All cores can put new data anywhere in the LLC, while still following the replacement policy. This naturally results in a core that asks for a lot of data, to get a larger share of the core assigned to it.

2.3 The Berkeley Dwarfs

In 2004 Phil Collela gave a presentation on software requirements for scientific computing [6] in which he identified seven so called motifs of scientific computing. Each of these motifs had a distinctive combination of computation and data access. These seven motifs would in the future be referred to as the seven dwarfs. The introduction of this concept inspired people at Berkeley university to identify more dwarfs to add to the list. The distinctive combination of computation and data access by which a dwarf can be identified can be found in all programs that are within the same class. Even if their implementation is done differently at a higher level, the lower level operations that actually make up the resulting program will still be the similar [7]. This means that the definition of a dwarf is not found in its exact implementation, but in its underlying pattern. The dwarf mine (or Berkeley dwarfs) now consists of thirteen dwarfs whose implementation has been useful for years throughout many changes in industry and science, and are expected to stay relevant for many more. The dwarf that will be used in this thesis is dense linear algebra, of which specifically the matrix multiplication computation.

2.4 Dense linear algebra

A computation is said to use *dense* linear algebra when it uses a data structure that is in size proportional to the amount of entries in the matrix. When a computation uses a data structure that is a size proportional to the amount of non-zeros, it is said to use sparse linear algebra. There are three levels of dense linear algebra.

Listing 2.1: standard matrix multiplication

```
Matrix multiplication algorithm:

let a be an n x m matrix;
let b be an m x p matrix;
let c be an n x p matrix with only 0-value entries;
for i = 1 to n
    for j = 1 to p
        for k = 1 to p
            c(i,j) += b(i,k)*a(k,j);
        endfor
    endfor
endfor
return c;
```

level-1 is vector/vector operations, the result can be a number(dot product) or a vector (addition/subtraction), or a matrix (outer product). level-2 is matrix/vector operations, the result can be a vector(product). level-3 is matrix/matrix operations, the result can be a number(dot product), or a matrix(multiplication or addition/-subtraction). In this thesis all dense linear algebra will be matrix multiplications done by computers, so in Listing 2.1 an example of a standard matrix multiplication algorithm is given in pseudo-code.

2.5 Autotuning and blocking

When referring to an autotuner in this thesis, the author refers to a program that automatically tries to find the cache size, and then calculates from that what it expects the optimal blocking-size to be. It works at run-time without the user interfering. The found blocking size will be used in a blocking algorithm. Loop blocking algorithms make sure that calculations that use the same information are done in succession, and thereby increases the effectiveness of the caches. In Listing 2.2 a normal matrix multiplication algorithm is shown. It multiplies matrix y by matrix z to get matrix x. In Listing 2.3 the same calculation is done, only with a blocked algorithm. Two extra loops are added, and a parameter B, which stands for blocking size, is now used. The jj- and kk-loops both have$\lceil N/B \rceil$ iterations, and the j- and k-loops have at most B iterations, the i-loop remains the same. This means the parameter B can be used to tune how many different data accesses there are between calls for the same data, which influences cache efficiency. For this algorithm, at every iteration of i, the multiplication in the inner loop will be done $B * B$ times, instead of $N * N$ times. This results in the algorithm using less cache space per i-loop, while still retaining the reusability of the data. For more detail on how this would affect the use of the cache see Section 3.1.3

Listing 2.2: matrix multiplication algorithm before blocking. Algorithm taken from Computer Architecture: A Quantitative Approach, Fifth Edition, page 89

```
Matrix multiplication algorithm before blocking:

for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j + 1)
        {r = 0;
        for (k = 0; k < N; k = k + 1
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
        };
```

Listing 2.3: matrix multiplication algorithm after blocking. Algorithm taken from Computer Architecture: A Quantitative Approach, Fifth Edition, page 90

```
Matrix multiplication algorithm after blocking:

for(jj = 0; jj < N; jj = jj + B)
for(kk = 0; kk < N; kk = kk + B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B,N); j = j + 1)
        {r = 0;
        for (k = kk; k < min(kk+B,N); k = k + 1
            r = r + y[i][k]*z[k][j];
        x[i][j] = x[i][j] + r;
        };
```

Figure taken from Computer Architecture: A Quantitative Approach, FifthEdition, page73

2.6 BLAS and ATLAS

In 1973, 31 years before the first 7 dwarfs were identified and published, scientists already realized the usefulness of having standard optimized ways of implementing common high-level mathematical operations. They started creating low-level implementations for these high level operations. In 1979 this had led to the birth of level-1 BLAS. BLAS [9] stands for Basic Linear Algebra Subprograms, it is set of low-level routines for performing common linear algebra. Level-1 BLAS(1979) consists of of operations that take linear O(n) time, such as dot products and vector normalization. Level-2 BLAS(1988) consists of of operations that take quadratic time O(n^2), like matrix-vector multiplication. Level-3 BLAS(1990), has routines for operations that take cubic time O(n^3), such as matrix multiplication.

ATLAS [10], or Automatically Tuned Linear Algebra Software, is a library for linear algebra that improves the speed of the BLAS library. It is in essence an autotuner for the BLAS library, as it tries to use information about a computer's architecture to make linear algebra run faster on it. It is divided in levels the same way BLAS is. Its optimization process consists of three approaches, parameterization, multiple implementation, and code generation. Parameterization in this context is the search for the relevant parameters such as the parameter space for a function, blocking factor or cache size. Multiple implementation searches for various

approaches to implement a function. Code generation is a program writing another program, by utilizing the knowledge they have about the system. For level-1 and level-2 BLAS optimizing, parameterization and multiple implementation is used, for level-3 code generation is added

Chapter 3

Implementation

In this chapter the methods that are used to gain the necessary data to answer the research question will be discussed. The method used to find the different levels of cache sizes will be explained, as well as how the data on the effectiveness of different blocking sizes will be acquired.

3.1 The cache sizes

3.1.1 First two cache levels

The first goal of the autotuner is to find the cache sizes. The method to do this is to test how fast the processor can get access to various blocks of data of a certain size thousands of times over. The reasoning behind this is that if this block fits inside a certain cache level, it will be done significantly faster than a block that does not fit inside this data cache, even when adjusted for the increase in data. Finding the sizes of the first two cache levels, the local caches, starts by determining a maximum size, in the used program the maximum size is set to be 1MB as no local caches are larger than this. The first test is started at a size that is smaller than the smallest modern cache (2KB), and an exponent of 2. Every time the size to be tested doubles, until it is larger or equal to the maximum size. Finding the size of the shared cache is done differently due to a number of reasons that will be discussed in the next section.

Every time before the cache is tested the program makes sure that no part of the data that will be used in the test is left in the cache, this is done by thrashing the cache. Cache thrashing, in this context, means going from cache line to cache line and filling each with information that is useless for the computations used in the test. The test are done using a test array, a shuffle array, and a random array, all consisting of integers. The size of the test array plus the shuffle array is equal to the size of the test, the random array is negligibly small. The program allocates the memory for the arrays in contiguous blocks aligned to the size of a cache line, this way all the data that is loaded into the cache during a miss for the test array, is data for the test array. The amount of time every test takes, and the amount of data it uses are stored in arrays.

In order to make sure the exact amount of size that is specified for the test is actually used in the test the program jumps from cache line to cache line during the tests. Every 16th index (16 integers = 16 x 4 bytes = 64 bytes = size of cache line) the program tries to access the array, forcing the entire array to be loaded in the cache, cache line by cache line. However, if this were done consecutively (16th, 32nd, 48th, ... etc), the prefetcher would be able to predict the next memory access and the program would load two (or maybe more) useful 64 bit blocks in the cache per miss. In order to see the difference between the data fitting in a cache, or just not fitting in a cache more clearly, the program needs to outsmart the prefetcher. This is why the aforementioned shuffle array is used. The shuffle array contains a permutation of multiplicities of 16, which the program uses to access the test array blocks in random order. Lastly there is the random array, this array contains a few integers with values between and including 0 - 15, and is used to access a random integer in each cache line. This way the prefetcher cannot know what part of the memory will be accessed next.

Every test consists of going through the test array in this way 1000 times, and measuring the time it takes to do this. The reason it is done 1000 times, is to make sure the entire cache is being used by this part of the program alone. At any time a lot of processes that can utilize the cache could share the core, but if the program puts such high demand on it in so little time then the cache is being used almost entirely by the program. This test is repeated 5 times for every size, and the average outcomes of these tests is used to determine the local cache sizes.

First it is calculated, for every test, how much time it took per 2048 bytes of size. For example, if a test of size 524288 bytes took 10ms to complete, then its "time per 2048 bytes" is 10ms/(524288/2048). See Definition 3.1.1.

Definition 3.1.1. Time per x bytes is defined as:
$timeperxbytes = totaltime \ / \ (totalsize \ / \ x)$

The multiplication value of a test is then calculated by taking the "time per 2048 bytes" of the next test, and dividing it by it's own "time per 2048 bytes". See Definition 3.1.2. This gives a measure of how much more time the next test costed.

Definition 3.1.2. multiplication value is defined as:
$multiplication\_value\_test\_a_1 = timeperxbytes\_a_2 \ / \ timeperxbytes\_a_1$

The two sizes with the largest multiplication value are the sizes of the first two cache levels, the smallest one the L1, and the larger one the L2. As there is a small chance interference can create a mishap, this process is repeated 10 times and the most common cache size is chosen.

3.1.2 Last level cache

The last level cache (LLC), also referred to as shared cache or L3 cache, requires a different approach for two reasons. The first being that its size is not necessarily a power of 2. The second is that the LLC could be a smart cache, which means other cores could be in competition for the space in the LLC. For these reasons the

tests are repeated with sizes that are increments larger, instead of twice as large. The LLC size can be expected to be anywhere from 2 to 20 MB, so increments of 128 KB are used starting at 1MB. These increments are small enough to get enough samples for even the smaller LLCs, and large enough to keep the computational time low. This does not completely solve the problem that the program uses only a certain part of the cache depending on its requirements or demand. For this reason the exact computations done in the test, will be done just before the test. This will ensure that the program's data has taken up the necessary space in the LLC, if the space is available. We purposefully do not trash the cache after this, so that the left over data from the rounds just before the test is still in it. This makes it easier to detect when the necessary data does not fit in the cache.

In order to explain how the LLC size is found, two concepts have to be introduced. A candidate and a bump. A candidate is a test where the next 10 tests all have a 1.05 times larger time per x bytes value. This means that all next 10 tests have a worse performance. Keep in mind that every next test, tests a slightly larger size. A bump is a test where the time per x bytes of the next test is at least 1.10 times as large as its own time per x bytes. This means that the next test, even though it uses only a slightly larger size, was significantly less efficient. The definition of these concepts were determined through trial and error testing.

Due to the nature of a smart cache it is likely for a bump to occur when a test uses more size than the program can utilize in the smart cache. But we are interested in the smart cache total size, not the current size being used by the program. When a bump occurs without being near the cache edge, the program will (if other processes allow it) get a larger part of the cache to use. This means that in the following tests, the time per x bytes will drop to normal levels again at some point. So when a bump occurs, but it is not at the cache edge, it will not also be a candidate. What needs to be found is a test size that is both a bump and a candidate, this will be the closest approximation of the actual size of the smart cache.

If no test is both a bump and a candidate, the test that is a candidate and has the lowest test size will be taken as the smart cache size. The reason the candidate using the lowest test size is chosen, is because when the program loads more data than the LLC can handle, the program will have to read from main memory which can also cause candidates to be detected.

The outcome of this smart cache size test is referred to as the "usable L3 cache size", as it tries to find the point where the computations start to become slower. The real LLC size can be found by round up to an MB, as the usable cache size is mostly not far off from the real cache size, and LLCs come in size of whole MB's. Another piece of information that will be stored about the LLC is the most effective cache size. This is the size where the amount of time per x bytes is the lowest. This test is also done 10 times, with the most common size being chosen for every parameter.

3.1.3 Deriving a blocking size

When the cache size is found, a corresponding blocking size for integer dense matrix multiplication must be calculated for each level. There are three loops (i,j and k), the two inner loops (j and k) are used to block

with. Within the inner-most loop (k) there is data being written to one variable and read from two array elements, see Listing 2.3 for the code being referred to. `r += y[i][k] * z[k][j]` is the command that is being executed in the k-loop. The goal is to keep `z[k][j] to z[k+blocking size][j+blocking size]` in the cache throughout all of the j loops a block has. This requires $blockingsize * blockingsize * 4$ bytes of space, as an integer has a size of 4 bytes and the block spans $blockingsize * blockingsize$ integers. For this test we assume the worst case scenario for executing matrix multiplication: a cache that strictly uses a FIFO replacement policy. If the cache could only hold the block of the z array, misses would occur when a new row of `y[i][k] to y[i][k+blocking size]` and the new `x[i][j] to x[i][j+blocking size]` row were accessed. So enough space is needed to accommodate the z array block and enough space to add one k-loop of integers from array y, and one j-loop of integers from array x. To be able to make a program use this information, a formula is needed that can can calculate the corresponding blocking size to any cache size. Theorem 3.1.1 contains this formula and the necessary mathematical proof.

Theorem 3.1.1. When multiplying two matrices containing integers: Under the assumption of a fully associative cache that uses a FIFO replacement algorithm, all non-compulsory misses in a cache with a size of c bytes are eliminated when a blocking size of $\sqrt{(}c/4+1)-1$ is used.

Proof. (i) Let y, z, x be matrices used for matrix multiplication as in Listing 2.3
(ii) A block of B by B (with B the blocking size) of matrix z has to fit in the cache in its entirety, throughout all the two inner loops, or avoidable misses would occur.
(iii) y[i][k] to y[i][k+B] is continuously used as the block from z is used, and x[i][j] to x[i][j+BS] is used once during a full use of the block from z, so these have to fit as well to prevent avoidable misses.
(iv) The block from matrix z takes up $B \cdot B \cdot 4$ bytes of space, as an integer is 4 bytes in size.
(v) The row from x and the row from y both take up 4 * B bytes of space.
(vi) So $B \cdot B \cdot 4 + 8 \cdot B$ bytes are required in the cache.
(vii) This translates to $4B^2 + 8B = c$, with B the blocking size in bytes and a the cache size in bytes.
(viii) $B^2 + 2B = c/4$
(ix) $B^2 + 2B + 1 = c/4 + 1$
(x) $(B+1)^2 = c/4 + 1$
(xi) $B + 1 = \sqrt{(}c/4+1)$
(xii) $B = \sqrt{(}c/4+1) - 1$

□

In the program the code $blockingsize = floor(root(c/4) - 1)$ is used, as the +1 is negligible and the results needs to be an integer that is not larger than the outcome. This formula can also be rewritten to calculate what cache size would be necessary to make a blocking size work best: $cachesize = 4 * blockingsize^2 - 4$.

3.2 Data collection

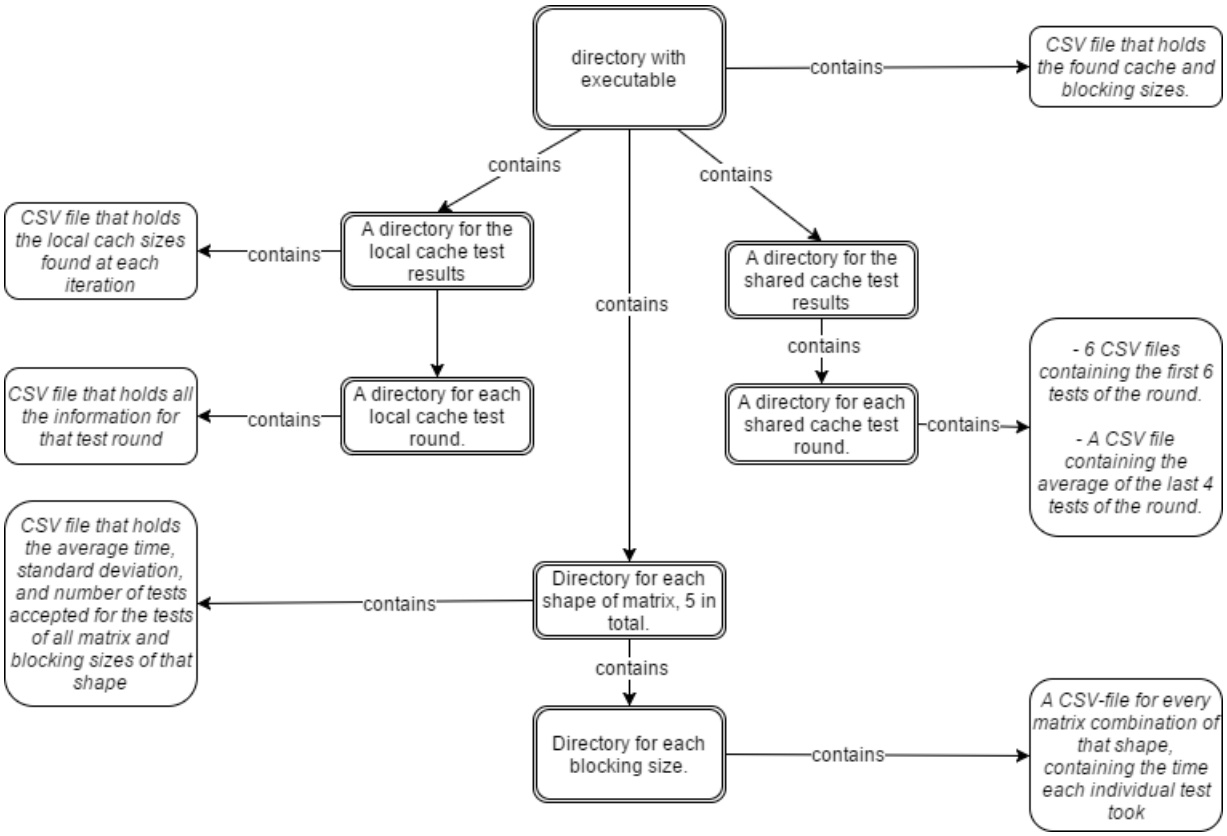
The tests are done with various sizes of matrices combined with various shapes of matrices combined with various blocking sizes. The sizes range from 500 entries to 4 million entries per matrix, needing at most a total of around 45MB of space. The shapes are square, and four kinds of rectangular. There is the rectangular matrix multiplied by another rectangular matrix that have a common length that is larger than the length in which they differ, and the reverse of this. Both these variations have another version where the length to width ratio is different. The reason for checking different kinds of shapes, is because the sizes of the loops change, hopefully leading to interesting results.

The blocking sizes range from 4 to 1680, resulting in what according to Theorem 3.1.1 would be optimal for a fully associative FIFO cache of $4 * 4^2 - 4 = 60$ bytes to $(4 * 1680^2 - 4)/1024^2 = 10.8$ MB. The maximum blocking size of 1680 has been chosen because the largest matrix used has 2000 by 2000 entries, blocking with blocking sizes too close to this number results in almost no blocking being done. There will be 25 different blocking sizes, of which 5 are taken from tests, and 20 are static hard coded values between 4 and 1680. The 5 blocking sizes taken from tests are the blocking sizes derived from the L1 cache size, L2 cache size, usable LLC size, real LLC size, and most effective LLC size. Usable LLC size and most effective LLC size are explained in Section 3.1.2. There are 20 different matrix sizes per matrix shape.

Every one of these combinations is tested 5 times, this raw data is stored in the folder hierarchy. The outcome of these 5 tests is tested for outliers, by calculating the average, and checking if any of the singular tests are greater 1.5 times the average, or smaller than the average divided by 1.5. If any outliers are taken out, a new average is calculated. The non-outliers are used to calculate the standard deviation, so that one does not need to look at all the raw data to see how close the different tests are. A CSV-file will hold all the averages (after outliers), standard deviations and number of test that these are based on for every test for all the combinations of matrix sizes and blocking sizes per matrix shape.

Results from the local cache tests, LLC tests, and each of the five different matrix shapes will get their own folder in the same directory as the executable. Each of these folders contain folders for each different blocking size, and the file CSV-file that has the averages, standard deviation and number of tests for all the test done with this matrix shape. Within each blocking size folder, there are a number of CSV-files that contain the raw data for each matrix size. See Figure 3.1 for a diagram explaining the folder hierachy.

Figure 3.1: A diagram that visualizes the folder hierarchy, it uses descriptions of what is in the folder as showing all folders would disrupt oversight.



Chapter 4

Experiments

The analysis of the experiments will consist of comparing the data in a number of ways. The main interest of the analysis is to find out if the blocking sizes calculated by the program are the best possible blocking sizes. It will also need to be verified if this is consistent over different matrix sizes, and if the outcome for different matrix shapes is comparable. The conditions of the experiments, the found cache sizes, the best blocking size for local and for shared caches, the stability of the tests, differences between architectures, performance on different shapes of matrices, and performance on different sizes of matrices will all be discussed in this chapter.

4.1 Experimental setup

The experiments were done on two different systems, on a workstation and on a node of the DAS-4. The DAS-4 uses Intel E5620 [1] CPUs on SuperMicro 2U-twins nodes[2]. The nodes determine the power-supply and hardware slots. The most important thing to note for the experiments in this thesis is what CPU is used. The CPU on these nodes is the Intel E5620, which has a base frequency of 2.4 GHz, a max frequency of 2.66 GHz, a L1 cache of 32Kb, a L2 cache of 256Kb, and a LLC of 12Mb. It runs 4 cores with a total of 8 threads. The nodes run on the centos operating system[3]. The workstation runs on ubuntu 12 and uses an Intel Core i7-3770 CPU[4], which also has a L1 cache of 32Kb, a L2 cache of 256Kb, and 8MB of shared smart cache. It also has 4 cores with a total of 8 threads. The base CPU frequency of these cores is 3.40 GHz and the max frequency is 3.90 GHz.

The main differences with the CPU used in the DAS-4 is the system it is embedded in, and that the workstation CPU uses turbo-boost 1.0 whereas the DAS-4 CPU uses turbo-boost 2.0. The exact replacement policies used in the caches are not known. The amount of RAM memory and the rest of the system is not being tested, so this

[1] http://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI
[2] https://www.supermicro.com/products/nfo/2UTwin2.cfm
[3] https://www.centos.org/
[4] http://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz

does not matter for our tests. However the turbo-boost is a collection of algorithms that scale the frequency of the processor to the workload. This means that both systems use a different way of calculating this, which could change the outcomes for different blocking sizes. Another noticeable difference is that the workstation uses a GUI, while the DAS-4 uses a command line interface.

For the DAS-4, the job is simply scheduled and then started when a node is available. For the workstation, some precaution has been taken. The workstation was rebooted, and only the terminal was started from which the autotuner was executed. After this the workstation was left completely untouched until finished.

4.2 Found cache sizes

The program was run three times on the DAS-4 and twice on the workstation. The reason for this is that the second test on the DAS-4 gave a different last level cache size than the actual last level cache size, and another test had to be done to determine if whatever caused this influenced the results of the other tests. The third test instance on the DAS-4 showed similar results to the first and second test instances when concerning the matrix multiplication tests, so the second DAS-4 instance will be used in this thesis.

Table 4.1: table of the found L1 and L2 cache sizes for all test instances

Architecture	test instance	l1 cache size found	in rounds	l2 cache size found	in rounds
DAS-4	1	32767	2 - 10	262143	2 - 10
DAS-4	2	32767	2 & 4 - 10	262143	2 & 4 - 10
DAS-4	3	32767	1 & 4 - 10	262143	1 & 4 - 10
Workstation	1	32767	1 - 10	262143	1 - 10
Workstation	2	32767	1 - 10	262143	1 - 10

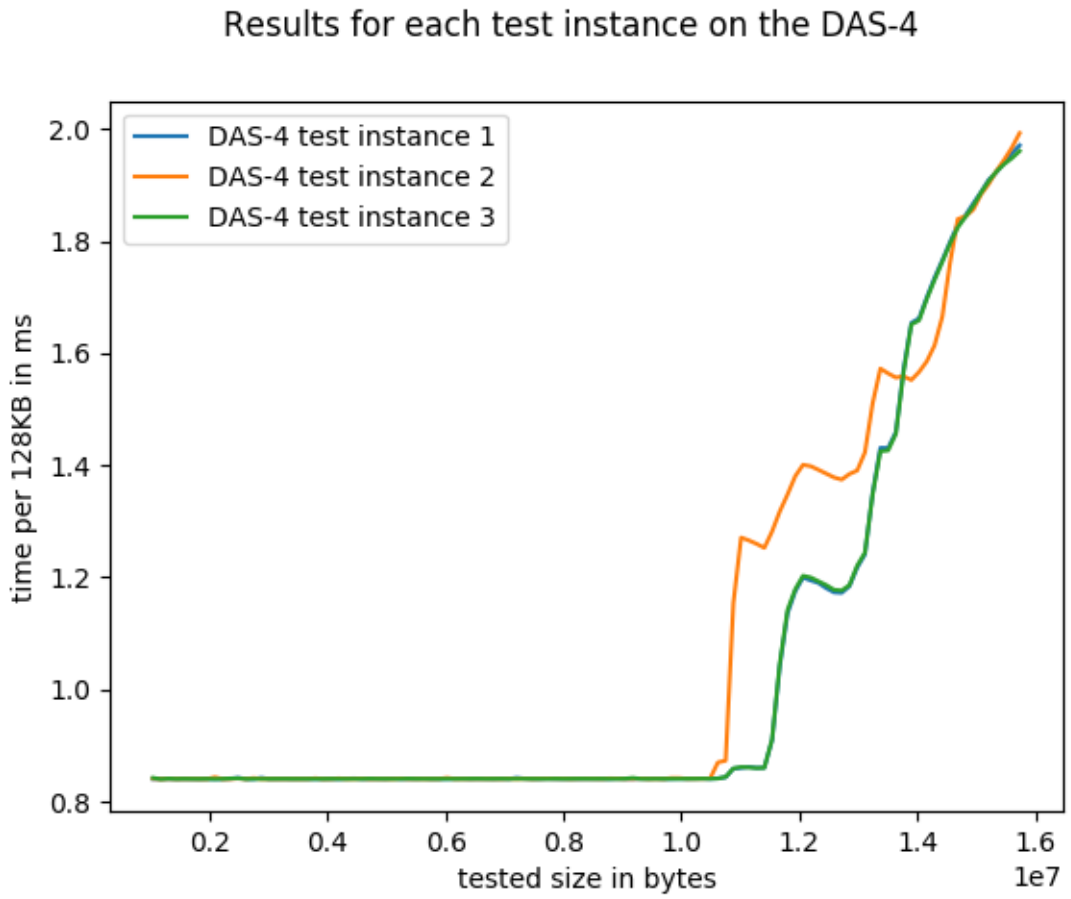
Table 4.2: table of found LLC sizes for all test instances

Architecture	test instance	l3 usable cache size found	l3 real cache size found	in rounds
DAS-4	1	11534338	12582912 (12MB)	4 - 10
DAS-4	2	10747905	11534336 (11MB)	2 - 10
DAS-4	3	11534338	12582912 (12MB)	5 - 10
Workstation	1	7995394	8388608 (8MB)	3 - 10
Workstation	2	7864322	8388608 (8MB)	3 - 4 & 8 - 10

In Tables 4.1 and 4.2 the final found cache sizes of all test instances are shown. A test instance is a full run of the program on a system. A round of cache tests is one of the ten tests that are done for both the local caches, or one of ten tests that is done for the shared caches, as explained in Section 3.1. The table contains information on what size was found for each level cache, and in what rounds it was found. The local cache sizes were all correct for all systems on all test instances. The results of the shared cache tests were correct on the DAS-4 the first time and third time (12MB), but off by 1MB the second time (11MB). This merits a closer look at the graphs of the information that was used to determine the shared cache size of the first and second test instances on the DAS-4. From this information it should become clear if the algorithm that interprets the

data is wrong, or if the data really did indicate the shared cache size to be this large. The cache tests of the workstation will not be discussed in further detail, as all sizes were found correctly.

Figure 4.1: The results of the average of the last round for all three test instances of the DAS-4.



In Figure 4.1 it is visible that the algorithm that interprets the data has determined the correct size for the usable cache size at test instance 2, as the efficiency drops immensely just after $1.0e7$. Table 4.2 also shows that this size was found consistently over different rounds. So it can be concluded that the fault does not lie with the program's interpretation of the data, and will have to be found somewhere else.

The algorithm for determining the shared cache size simply assumes that the usable cache size will be less than 1MB smaller than the real shared cache size. This rule has been determined through trial and error, and can therefore not be expected to be true 100 percent of the time. The reason behind this 1MB figure coming out of the trial and error tests is that the test is done in an environment where only the necessary processes for the OS are running besides it. It can therefore use a large amount of the cache, but cannot use all of it as there will always be some system calls that take up space in the caches. From the collected data it can only be speculated that the OS has been more active in the second test than in the first, or that another process interfered. The fact that the graphs of the first and third test instance are similar to such a degree that they are almost exactly the same, and the second test instance being significantly different also points towards interference in the second test instance. What can be said with certainty is that the maximum amount of shared cache space that the process could possibly get at the time was found.

The goal of these tests is to find the cache sizes at run-time, without making OS based calls to simply get the information about the processor. These limitations have resulted in a factor that cannot be predicted with certainty, the amount of interference on the system. For this reason the found "real cache size" will in some instances be slightly lower than the actual cache size, but the usable amount of shared cache size at run-time will then still be found. The data about matrix multiplication that have come out of this test are also still valid. All that has to be considered when looking at the results of the second DAS-4 round is that the one blocking size out of 25 tested that is said to represent the real cache size, does not represent the real cache size in this instance.

4.3 Test stability

Every piece of data within a test is gathered from a test that was repeated 5 times, as mentioned in Chapter 3. A 5 times repeat of the same test will be referred to as a test set, and one of the 5 times will be referred to as a test in this section. A test set is accepted completely if none of the test results are greater than 1.5 times the average of its test set result, and if none of the test results are smaller than the the average of its test set result divided by 1.5. If a test set is not accepted completely, only the tests that fall within the acceptance range will be accepted, and the new average will be calculated from those. It is also possible that all tests fall outside the acceptance range, this will mean the entire test set is discarded.

Table 4.3 shows the amount of total tests and total test sets for the first test instance for square matrix multiplication of the DAS-4 and the workstation. The amount and percentage of accepted test, fully accepted test sets, partially accepted test sets, and fully discarded test sets is also shown for both. The percentage of fully accepted tests sets is very high, however this is only designed to disregard test sets that are too inconsistent to even consider using, so the threshold is low.

Table 4.4 shows the coëfficient of variation, calculated on the tests that remain. For the DAS-4, Almost all tests stay within the 0 to 5 percent range of standard deviation, only very few (around 10 out of the 500 data points) are above this range. With the exception of the "100 by 100 x 100 by 100" multiplication the workstation results are even more stable. This means that the data used in this thesis is likely to predict future test instances, as the variation is low.

Table 4.3: table that shows the acceptance percentages of tests and test sets

Architecture	test instance	% accepted tests	% fully accepted test sets	% fully discarded test sets
DAS-4	1	99.36	98.00	0.2
DAS-4	2	99.24	98.40	0.4
DAS-4	3	99,48	98.40	0.2
Workstation	1	98.44	97.80	0.2
Workstation	2	99.16	97.60	0.4

Table 4.4: tables that shows the coëfficient of variation over various sets of results

Metric	DAS-4 instance 1	DAS-4 instance 2	DAS-4 instance 3
average CV over all	0.499	0.264	0.256
minimum CV over all blocking sizes	0.067	0.046	0.033
maximum CV over all blocking sizes	2.165	1.036	1.060
minimum CV over all matrix sizes	0.018	0.010	0.005
maximum CV over all matrix sizes	3.557	3.445	3.788

Metric	WS instance 1	WS instance 2
average CV over all	0.591	0.628
minimum CV over all blocking sizes	0.200	0.136
maximum CV over all blocking sizes	2.438	2.360
minimum CV over all matrix sizes	0.039	0.051
maximum CV over all matrix sizes	3.547	4.353

4.4 Best blocking size for the last level cache

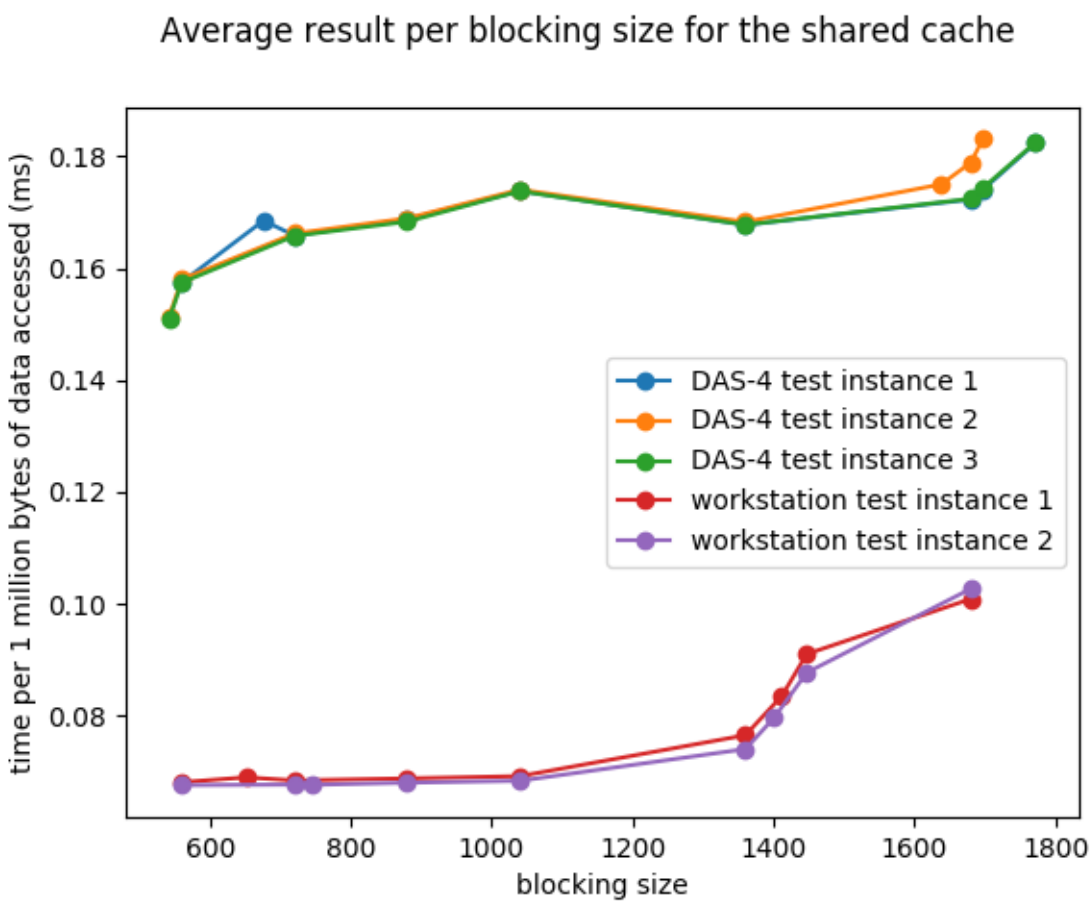
This section will attempt to identify the best blocking size for the LLC, which will be done by comparing the results that have blocking sizes that are determined to be optimal for caches over 1MB according to Theorem 3.1.1. A problem that arises when analyzing the results of blocking for the shared cache is that the smallest blocking size is larger than some of smaller matrices. This results in no blocking being done. Additionally, blocking for the shared cache only makes sense for larger matrices that do not fit in the shared cache. For these reasons the best shared cache blocking sizes will only be determined from the larger matrices. This means only matrix multiplication with 2 square matrices of 1800x1800, 1900x1900, and 2000x2000 will be used to determine the best blocking size, as these have a larger height and width than the maximum blocking size. This results in only the matrix multiplications with square matrices being considered for the shared cache, as these are the only ones with multiplications where both dimensions are large enough.

In Figure 4.2 a graph of the average results per blocking size of the relevant blocking sizes is shown for every test instance. What is immediately visible is that the largest blocking sizes have the worst performance, despite the largest blocking sizes being closest to what was expected to be optimal for the LLC according to Theorem 3.1.1. The best performing blocking size is surprisingly always the lowest blocking size tested. Upon closer inspection the four best blocking sizes in order of best to worst for the DAS-4 are 542, 560, 720, and 1360, and for the workstation these are 560, 720, 880 and 745.

The lowest blocking size being the best blocking size is not in line with Theorem 3.1.1, and the bottom graph in Figure 4.1.However, to get this graph many circumstances had to be controlled in order to make the LLC behave as needed, such as having already done the exact calculations just before the test without trashing the cache in between. This is described in detail in Section 3.1. Testing the performance of the different blocking sizes has been done without any of these circumstances, so the behaviour of the LLC is now different. In order to make sense of why the results in Figure 4.2 are what they are, the graphs of the first test of the first round of shared cache tests will have to be discussed.

In order to compare the results for the matrix tests with the shared cache tests, the amounts of space needed by the program for the blocking sizes to perform optimally has to be calculated, using the formula in Theorem

Figure 4.2: The average result per blocking size for the blocking sizes that are expected to be optimal for caches of various sizes over 1MB, for every test instance.

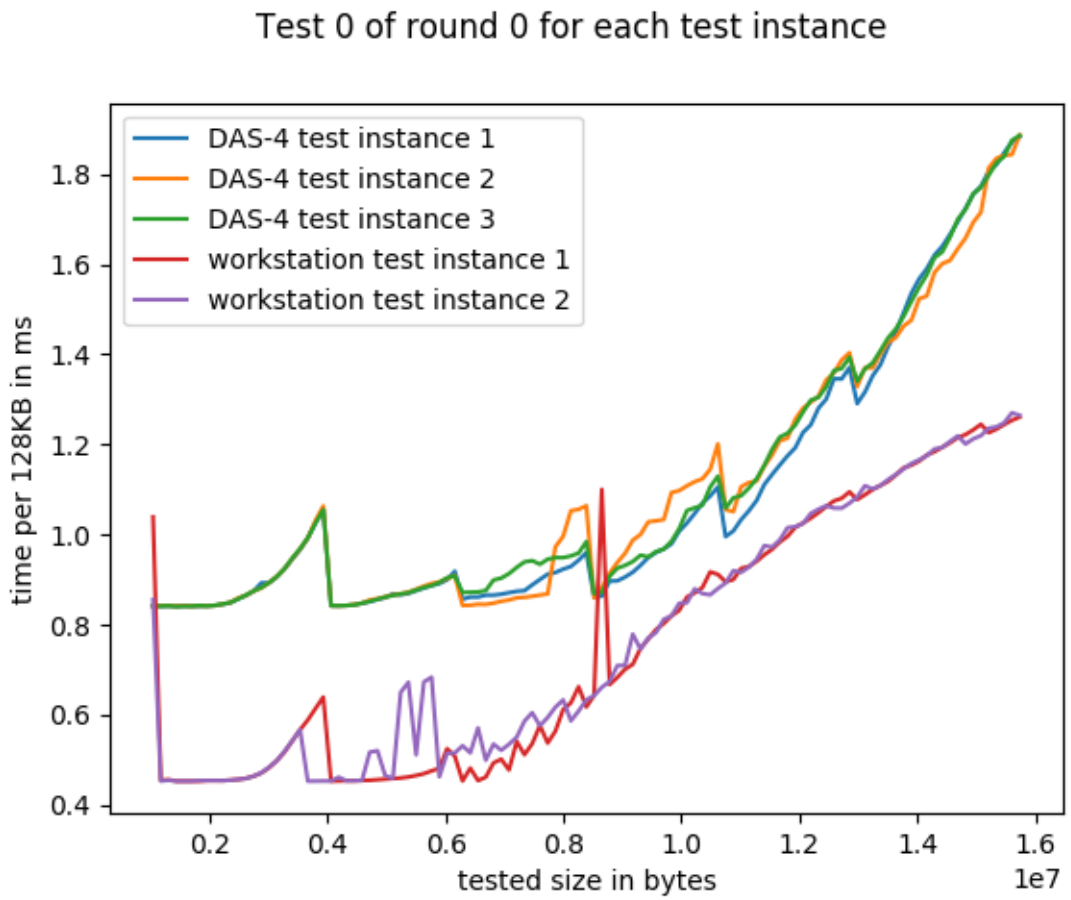


3.1.1. The blocking sizes that work best for the DAS-4: 542, 560, 720, and 1360 correspond to an amount of cache size needed for optimal being optimal of around 1.12MB, 1.18MB, 1.98MB and 7 MB respectively. The blocking sizes that work best for the workstation: 560, 720, 880, 745 correspond to an amount of data access of around 1.18MB, 1.98MB, 2.95MB and 2.12MB respectively.

Figure 4.3 shows the very first test in the very first round of each test instance of the DAS-4 and the workstation. Starting at 1MB to a little over 2MB the efficiency remains the same on all test instances. As the amount of data that is accessed increases further, the efficiency of the shared cache decreases at first, but than suddenly pops back to normal again. This repeats until the cache edge is encountered, after which the efficiency keeps decreasing. This corresponds well with the blocking sizes that were found to be optimal, almost all of these correspond to the 1MB to 2MB range. The explanation of why blocking for these blocking sizes gives consistently good results is simply that the process is always able to use an amount of space that is optimal for these blocking sizes.

There is one outlier left who's good performance remains unexplained, and that is the performance of blocking size 1360 on the DAS-4, corresponding to being optimal for a cache that is 7MB in size. The reason for this outlier is most likely that the process just jumped back to normal. The explanation for why the smallest cache sizes performed the best remains conjecture, as data on the amount of cache size that the process could use at

Figure 4.3: The results of the first tests of the first rounds of all test instances. X axis shows the tested size in bytes, Y axis shows how many ms per 128Kb it took to access this size.



the time of each test is not available. For now it is simply an explanation that fits all the found data in these experiments.

The best blocking size for the LLC for the matrix sizes tested is a blocking size that corresponds to being optimal for cache sizes ranging from 1MB to a little over 2MB. It is possible that a larger blocking size could perform comparably, but it is unlikely for it to perform well consistently due to the unpredictable performance of the LLC in these specific situations.

4.5 Finding the best overall performing blocking size for the local caches

The goal of analyzing the experiments done for this thesis is to find the best blocking size through the results of one particular experiment, and see if it holds up in different settings. Therefore the discussion on what is the best blocking size will start by analyzing tests for square matrices on the DAS-4, and the square matrix tests for the workstation. After these tests have been analyzed and the findings on each processor have been compared to each other in this section, it will be discussed if the results of this analysis hold over different matrix shapes and different test instances in the following sections. The best blocking size for the local caches will be determined from comparing the results that have blocking sizes that are determined to be optimal for

caches under 1MB according to Theorem 3.1.1

Figure 4.4 shows the average performance of each local blocking size for each test instance, of both architectures. According to Theorem 3.1.1 the L1 cache on both systems corresponds to a blocking size of 89, and the L2 cache on both systems corresponds to a blocking size of 254. Noticeable in the graph is that the smallest blocking sizes perform the very worst, these are blocking sizes of a size so low that the overhead of the added loops creates too much work for it to be optimal. The very largest blocking sizes, over 254, tested can also be discarded as the best overall performer according to the graph. This is within expectations as these correspond to being optimal for caches larger than the L2 cache. There are differences in the trends between the architectures, and the remaining blocking sizes have very similar performance, so in order to determine the best overall blocking size the averages of all test instances per architecture have been taken and the top 5 in performance are shown in Table 4.5.

89 is the blocking size derived from the L1 cache, and as seen in Table 4.5 it is the best performing blocking size for the DAS-4 and the second best, with only a small difference, on the workstation. The blocking size derived from the L2 cache is not in the top 5 performers for any architecture. The two architectures both have the same cache size, but do not have the same blocking size performing the best overall, as the best performing blocking size on the workstation is 32. So the blocking size corresponding to the L1 cache is an overall good performer on both architectures, but only the very best on the DAS-4.

Figure 4.4: Graph that shows the average time per blocking size for each test instance.

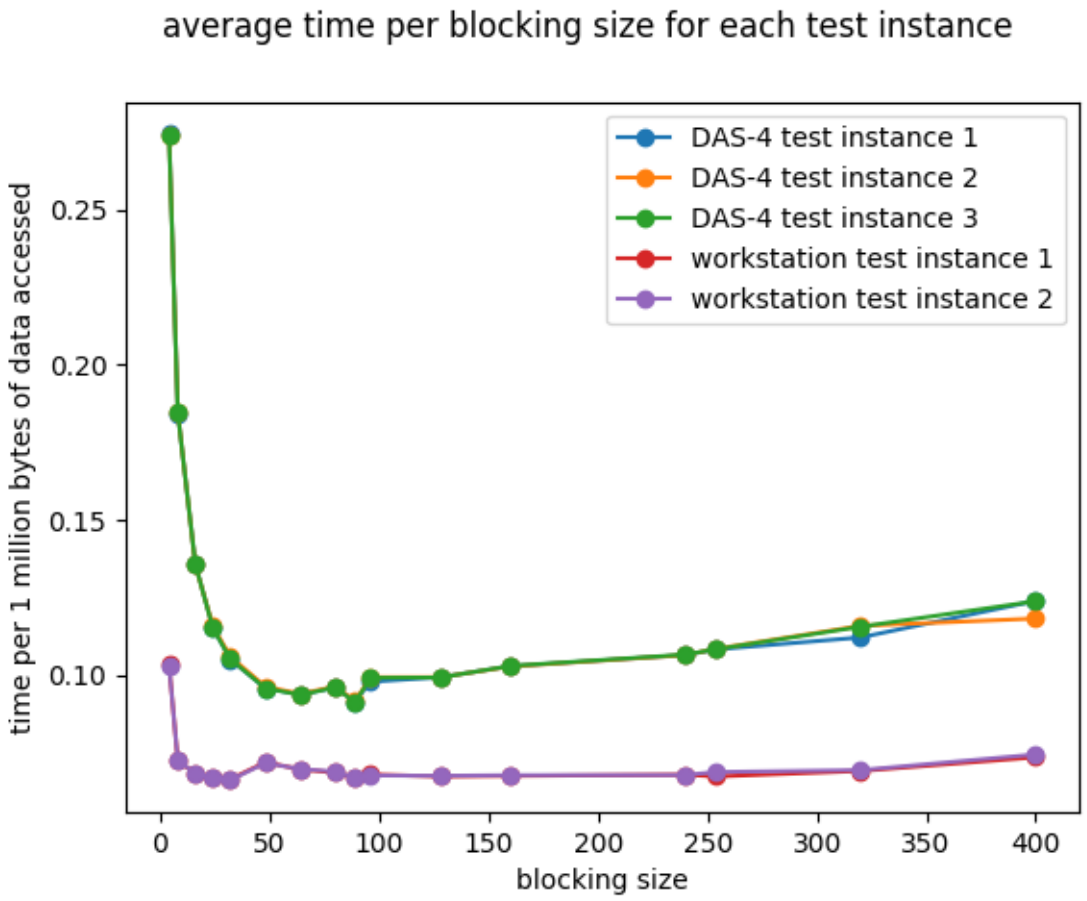


Table 4.5: Table that shows the fastest blocking sizes for the DAS-4 and the workstation, taken from the average of all the test instances for each

Description	blocking size DAS-4	time DAS-4	blocking size workstation	time workstation
number 1	89	2.565	32	0.797
number 2	64	2.631	89	0.803
number 3	48	2.693	24	0.803
number 4	80	2.697	128	0.811
number 5	96	2.766	160	0.813

4.6 Performance on matrices with different shapes

This section will determine if there is a difference between the efficiency of local blocking sizes between different shapes of matrices. Figures 4.5, 4.6 and 4.7 show the different shapes of matrices and their results. The matrix multiplications have been converted to amount of data that needs to be accessed to complete the calculation, this is shown on the x-axis. The y-axis shows the time in ms per 64Kb accessed. The reason for using amount of data accessed instead of the matrix dimensions is to better compare the graphs, as the matrix with the largest amount of entries used is not the same for each shape. It is also possible that the amount of total data accessed for a multiplication could also have an effect on the performance.

The graph for the ”2x by 2x matrix * 2x by 2x matrix” multiplication (Figure 4.5) and the graph for the ”2x by x matrix * x by 2x matrix” multiplication (Figure 4.6) show the same trends and rankings for the performance of various blocking sizes. The graphs for ”x by 2x matrix * 2x by x matrix” multiplication (Figure 4.7) do not show the same trends and rankings for most blocking sizes as the other two. The blocking sizes corresponding to the L1 and L2 cache perform considerably worse for this shape, especially at a higher amount of inner iterations. Both of these phenomena hold over different architectures and test instances.

The writer believes it would require further research to determine the cause of this phenomena, as the available data does not have enough information for this. What can be concluded is that the shape of the matrix can influence the effectiveness of blocking sizes, and the best blocking size that will be determined for the local caches holds for multiplying two square matrices and for ”2x by x matrix * x by 2x matrix” multiplication, but cannot be said to hold for ”x by 2x matrix * 2x by x matrix” multiplication.

Figure 4.5: Graph of the performance on matrices of ”2x by 2x * 2x by 2x” matrix multiplication on the DAS-4, test instance 1.

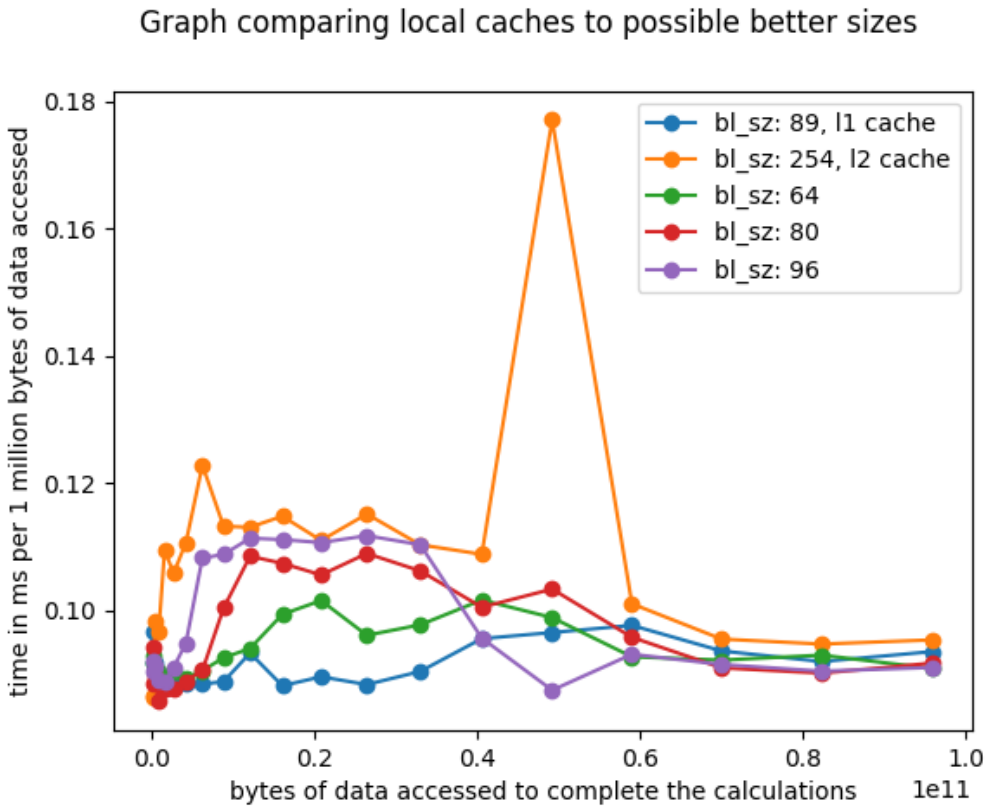


Figure 4.6: Graph of the performance on matrices of ”2x by x * x by 2x” matrix multiplications on the DAS-4, test instance 1.

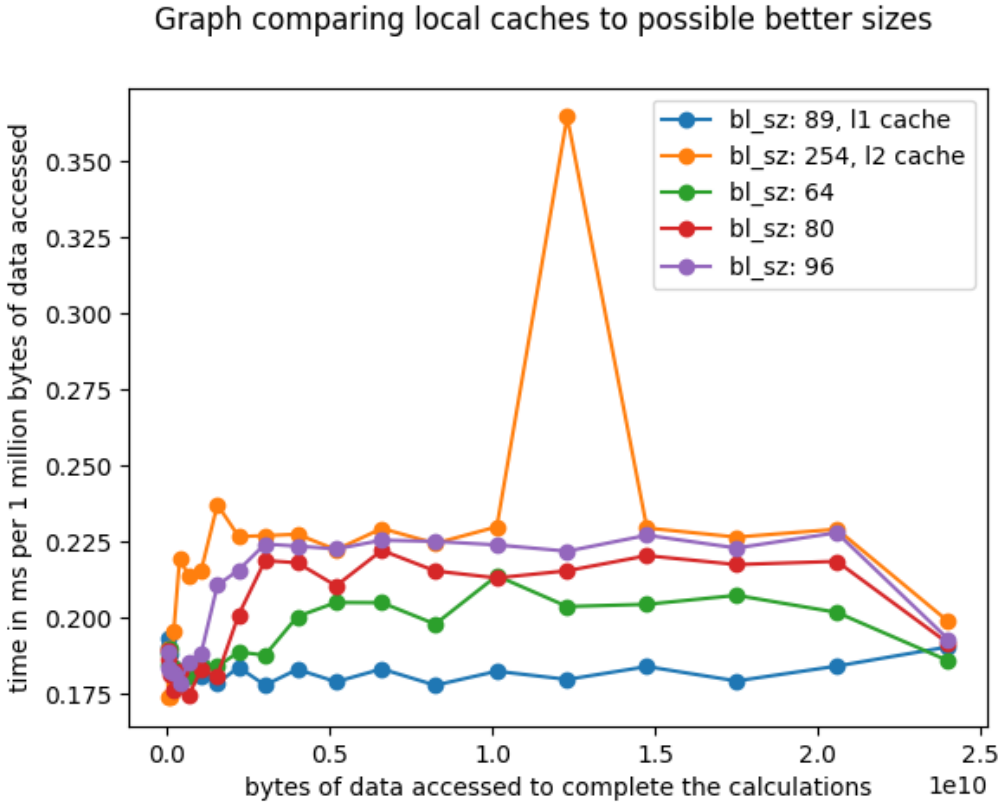
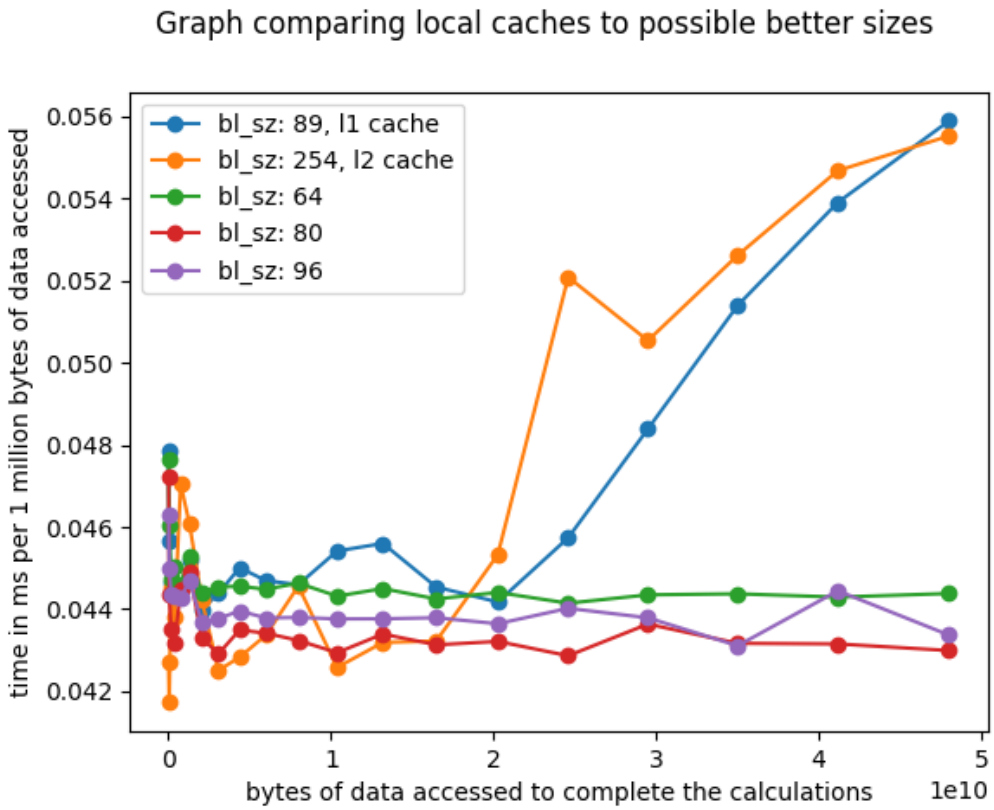


Figure 4.7: Graph of the performance on matrices of ”x by 2x * 2x by x” matrix multiplications on the DAS-4, test instance 1.



4.7 Performance per matrix size

This section will explore if the size of the matrices being multiplied affects the performance of blocking sizes. For this section only the square matrix multiplications of the first test instance of the DAS-4 and the square matrix multiplications of the workstation will be used. In Figure 4.8 the performance of a selection of blocking sizes is shown for each matrix size of the first test instance of the DAS-4 is shown, and in Figure 4.9 the same is shown for the first test instance of the workstation. These graphs show that the blocking size corresponding to the L1 cache (89) performs well across all matrix sizes. The other blocking sizes shown are all larger than 89 but smaller or equal to 254, which is the blocking size corresponding to the L2 cache size. These blocking sizes generally perform worse than 89 for the smaller matrices, but outperform 89 for the matrices over 1600 by 1600. So larger blocking sizes seem to perform better on larger matrices than the overall best performing blocking size.

In Table 4.6 the blocking sizes that were most often the best performing are shown, ranked for being the best for most matrix sizes only being the best once. Blocking sizes that were not the best for any matrix multiplication size are left out. The results for the DAS-4 show that the overall best performing blocking size was best for 8 out of the 20 tested matrix sizes, and the results for the workstation show that the overall best performing blocking size was best for 5 out of the 20 tested matrix sizes. So the overall best performing

blocking sizes were the best blocking sizes for more matrix sizes than any other, but were not the best for more than half the of the matrix sizes. This means the best overall performing blocking size cannot be said to be the best performing blocking size on all matrix multiplication sizes, although it performs best on the largest amount out of all sizes tested.

Figure 4.8: Graphs showing the performance of blocking sizes over different matrix sizes for the DAS4, test instance 1

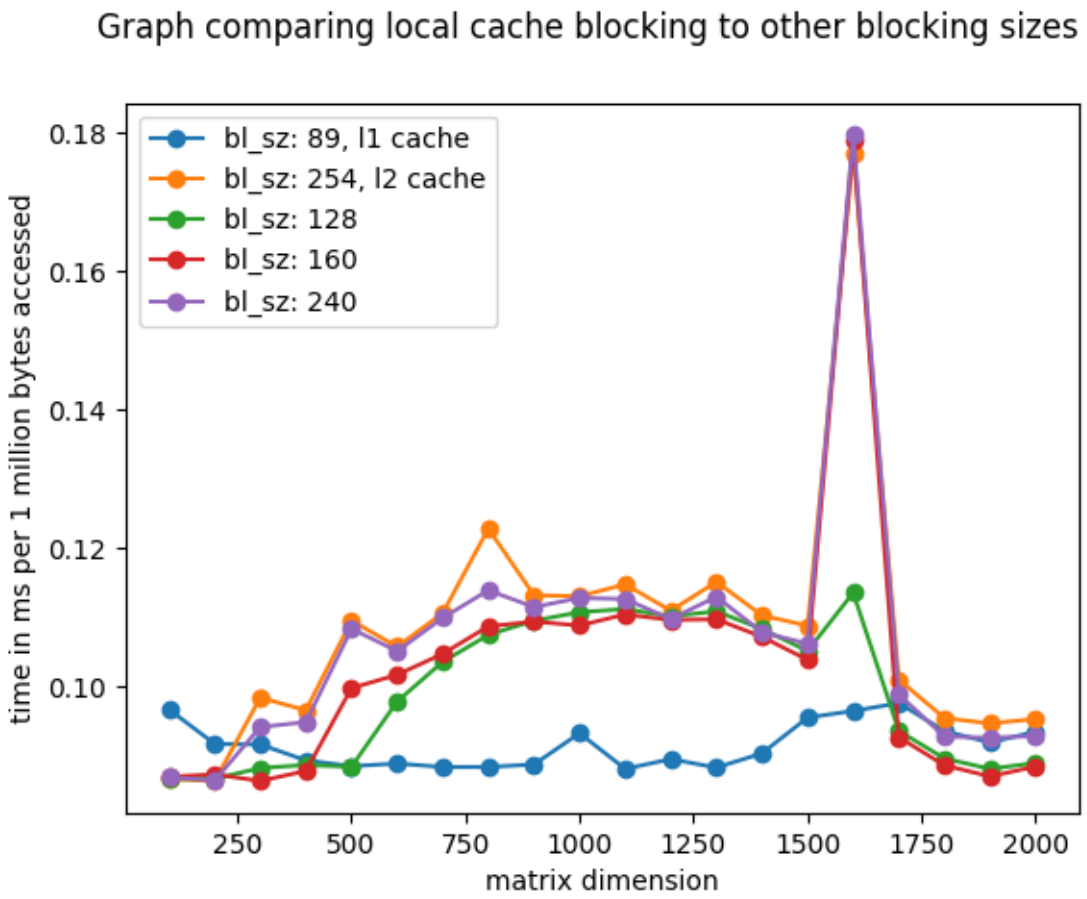


Figure 4.9: Graphs showing the performance of blocking sizes over different matrix sizes for the workstation, test instance 1

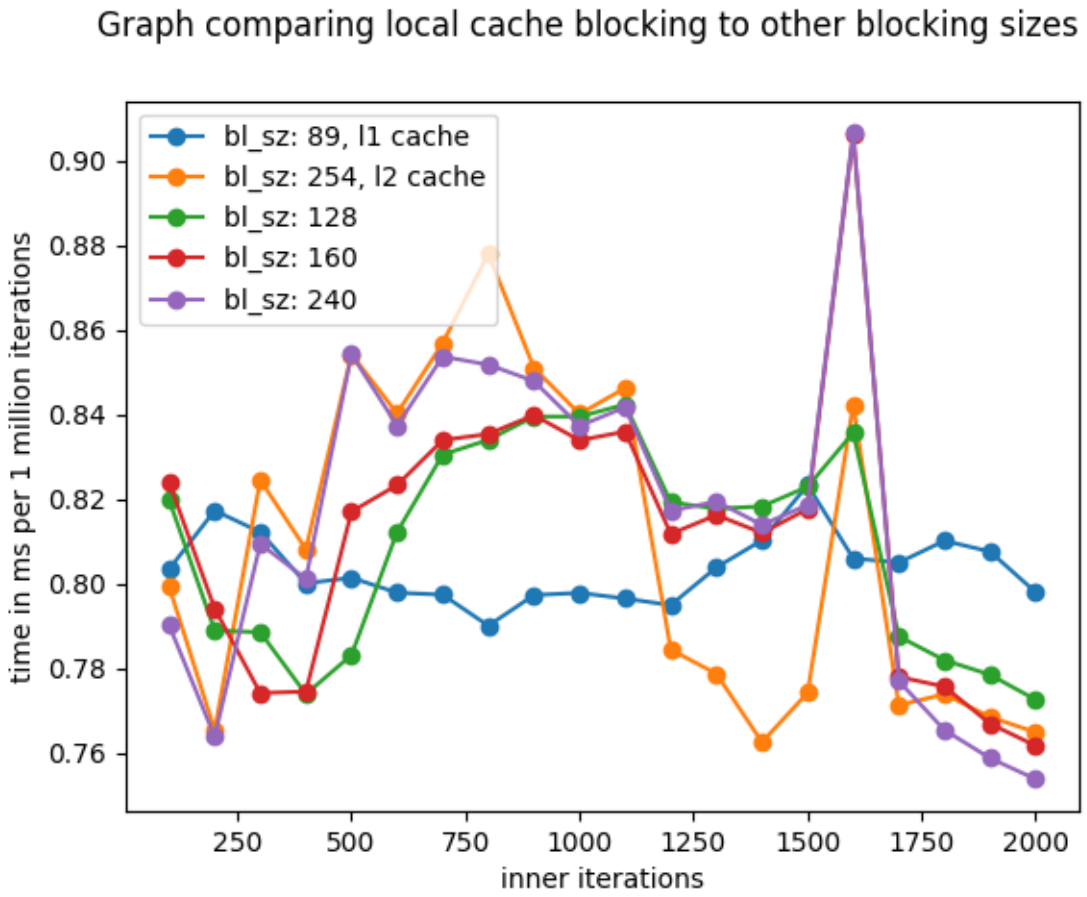


Table 4.6: This table shows how many times a blocking size was the best performing for a certain matrix size for each architecture. The data for the first test instance of each architecture was used.

BS DAS-4	# of matrix sizes it was best for	BS workstation	# of matrix sizes it was best for
89	8	32	5
160	4	254	4
80	3	240	4
64	1	128	2
96	1	80	1
320	1	89	1
400	1	160	1
1697	1	880	1
none	none	1447	1

Chapter 5

Conclusions

5.1 Conclusion

The conclusions drawn in this section are for single level matrix multiplication blocking algorithms, for matrices between 5000 and 4 million entries.

Does using the model of a fully associative cache with a FIFO replacement policy give the best overall performing blocking size for every architecture? Using this model on the L1 cache did result in finding the best overall performing blocking size for the DAS-4 local caches, but only the second best on the workstation local caches. The blocking size calculated from the L2 cache was not the best overall blocking size in any test instance. The blocking that this model got from the shared smart cache size resulted in worse performance than most of the other blocking sizes that were tested for the shared smart cache. So the model does not give the very best overall performing blocking size for every architecture.

Is it then possible to extract the best possible blocking size by just knowing the cache sizes of a processor with another model? The research in this thesis suggests that this is not possible for the local caches (L1 and L2), as both architectures had the same local cache sizes but different blocking sizes performing the best overall. In this thesis a range of blocking sizes was found in which the blocking algorithm was most effective (1-2MB), but the variety in smart cache sizes tested (8MB and 12MB) is too small to say if this would hold up for much smaller sizes such as smart caches 3MB in size. It could be possible for a model to be created, but not from the data in this thesis.

Will the overall best performing blocking size perform the best over different shapes of matrices? The data used in this thesis suggests that some shapes will have blocking sizes perform the same as square matrices, while others perform differently. So the overall best performing blocking size does not perform the best for all different shapes of matrices being multiplied.

Does the overall best performing blocking size perform the best for all different matrix sizes being multiplied? The overall best performing blocking size did not perform the best for all different matrix sizes, but did

perform the best for most matrix sizes tested. Larger matrices had better performance under larger blocking sizes that still mapped to a size smaller or equal to the local caches. Some blocking sizes had good performance on some matrices, but poor performance overall.

The experiments of the test for this thesis are not guaranteed to give the same results in a setting where many different demanding processes run on the same core or system.

5.2 Further research

Further research could be done into looking for the most effective blocking size when blocking for larger matrices. The data in this thesis seems to indicate that matrix multiplications with a higher amount of total data accessed for the calculations benefit from a higher blocking size than those with a lower amount of total data accessed for the calculations. The amount of data accesses at which this starts to show is above $0.6e11$ bytes, corresponding to the square matrix multiplications with matrices larger than 1600 by 1600. The difference is most noticeable between blocking sizes that correspond to being under the size of the L1 cache, and blocking sizes corresponding to being between the size of the L1 cache and the size of the L2 cache. An experiment that would test this could consist of matrix multiplications of various sizes, some below $0.6e11$ bytes of total amount of data accessed and some above $0.6e11$ bytes of total amount of data accessed, with various blocking sizes that correspond to the aforementioned ranges for the architecture that was tested on.

This thesis is not conclusive on using the size of the shared smart cache to find the best blocking size. It has been established through the tests that find the shared smart cache size, that a single process repeatedly doing calculations that require a lot of cache space, will eventually lead to a larger part of the cache being used by that process. So further research could be done into seeing if matrix multiplications multiple times the size of the largest matrix multiplication done in this thesis do benefit from blocking for the shared cache.

Bibliography

[1] David C. Lay *Linear Algebra and its applications 5th edition*. Pearson, 2016

[2] F. S. Hill, Jr. and Stephen M. Kelley *Computer graphics using openGL, third edition*. Pearson, 2006, section 5.3.1 page 24-26

[3] Michael Wolfe *More iteration space tiling*. Proceedings of the 1989 ACM/IEEE conference on Supercomputing, p.655-664, November 12-17, 1989, Reno, Nevada, United States

[4] Lakshminarayanan Renganarayanan, Dae Gon Kim, Sanjay Rajopadhye, Michelle Mills Strout *Parameterized Tiled Loops for Free*. Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, Pages 405-414

[5] Wm A Wulf, Sally A McKee. *Hitting the memory wall: implications of the obvious*. ACM SIGARCH computer architecture news, 1995/3/1

[6] Phillip Colella *Software requirements for scientific computing, presentation slides* `https://www.krellinst.org/doecsgf/conf/2013/pres/pcolella.pdf` [Accessed at July 14th 2017]

[7] Berkeley University *Dwarf mine, overview* `http://view.eecs.berkeley.edu/wiki/Dwarf_Mine` [Accessed at July 14th 2017]

[8] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff: *"A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term"*. IEEE Computer, Vol. 49, No. 5, pp. 54-63, May 2016.

[9] BLAS information and documentation, `http://www.netlib.org/blas/` [Accessed at July 14th 2017]

[10] ATLAS information and documentation, `http://math-atlas.sourceforge.net/` [Accessed at July 14th 2017]