Speed-up Techniques in Matrix Computation: a Case Study

Paula Monteiro
Jorge Moura

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
pmonteiro@dsi.uminho.pt
jorge.moura@unue.net

Abstract. Improvements on the execution performance of a given application can be obtained simply by the use of more efficient algorithms, a better codification and by the selection of adequate compiling options. The present study aims to analyse these different scopes, and to apply this methodology to square matrix multiplication. The case study uses C language, the GCC compiler, and a reference algorithm taken from the BLAS library. This study also aims comment the impact the codification and compiling techniques have on the whole process.

1 Introduction

Nowadays, there is a great lack of quality software production. With the upcoming of the Internet, new paradigms were adopted. Quality is being undervalued to get speed, not the software speed, but the production speed itself. This way, we see software being developed in a record time without the slightest concern for the control of exceptions and for the efficiency and speed of execution. Of course, in most of the software, we can disguise the efficiency and speed problems with hardware upgrades but the problem remains there. We aim to analyse different programming techniques and to evaluate the differences of performance between them.

This analysis will be developed with its basis in the architecture of the processor and memory and to study this issue we used the multiplication of two square matrices. To develop this task, we are going to analyse five algorithms with quite different degrees of optimisation and complexity. We are going to codify them and analyse the time span of their resolution. We are going to use the C codification to deal with our problem and use the GCC compiler from GNU. In our study, we are also going to analyse the different degrees of compilation of GCC and use the optimisation option (*O*) of the compiler.
In a deep sense, we aim to justify the higher or lower level of efficiency of some techniques by analysing the architecture of the machine.

2 Methodologies

2.1 Tools

As the main tool for the elaboration of this communication, we chose the C programming language and the GCC compiler from GNU. We also used BLAS libraries[1]. The BLAS are

[1] Basic Linear Algebra Subprograms.

libraries built for the calculation of operations with vectors and matrices. These libraries intend to be a standard whenever one needs to get some mathematical rigor in the calculation of operations with vectors and matrices. We used a C interface from BLAS.

The evaluation of the effectiveness of the algorithms is given in CPE (Cycles per Element) [1]. By element we mean each one of the elements of the calculated matrices. This is the most used metric because, this way, we can compare different matrix dimensions. This is also the most indicated one when we aim to measure the time spent between two points of our program. This is an efficient way to measure the time of execution between two points of our program. The cycles are a great source of inefficiency if they aren't optimised. As all the algorithms in this study are based on loops, this method seems to be the most appropriate. This measurement method is practically independent from the processor. The given value is the number of cycles, which took place between the call of `start_counter` to start the counting and the call of `get_counter` to stop the counting and read. This value does not indicate us if all the measured cycles were used or not by our program. However, since the program takes no longer than a few microseconds, the counter was initiated immediately before the call of the algorithm and finished soon after it and we can almost guarantee that this value was spent in the process under analysis.

2.2 Algorithms

In our study, we used five algorithms. Four of these algorithms[2] were defined in our program while the fifth was used in the BLAS interface. The choice of these algorithms was due to the variety of techniques to compute the results they provided. For each one of the algorithms, we got five results for each given dimension of the matrix and each given degree of optimisation of the compiler.

Before doing the five measurements we ran the algorithm once to "warm-up" the cache for the data. This way we collected 5 homogeneous results. Whenever the program was run, the five algorithms were run and the results kept in a file.

The first algorithm we are studying is the normal algorithm, the basic one. This algorithm was chosen by the fact of being known by everybody.

```
for (i=0;i<N;i++){
        for (j=0;j<N;j++){
                c[i][j] = 0.0;
                for (k=0;k<N;k++){
                        c[i][j] += a[i][k]*b[k][j];
                }
        }
}
```

The second algorithm, the transpose one, was chosen because it already presents a series of techniques that allow us to get interesting results.

```
void transpose(int N){
        int i,j,k;
        double temp;

        for (i=0;i<N;i++){
                for (j=0;j<N;j++){
                        bt[j][i] = b[i][j];
                }
        }
```

[2] Extracted from mm.c by Smotherman, M. http://home.iae.nl/users/mhx/mm.html.

```
        for (i=0;i<N;i++){
                for (j=0;j<N;j++){
                        temp = a[i][0]*bt[j][0];
                        for (k=1;k<N;k++){
                                temp += a[i][k]*bt[j][k];
                        }
                        c[i][j] = temp;
                }
        }
}
```

The third algorithm, the "unroll16", unrolls 16 times the inner loop. This method has the particularity of, in each *for* step, performing sixteen operations.

```
void unroll16(int N){
        int i,j,k;
        double temp;

        for (i=0;i<N;i++){
                for (j=0;j<N;j++){
                        temp = 0.0;
                        for (k=0;k<(N-15);k+=16){
                                temp += a[i][k]*b[k][j];
                                temp += a[i][k+1]*b[k+1][j];
                                temp += a[i][k+2]*b[k+2][j];
                                temp += a[i][k+3]*b[k+3][j];
                                temp += a[i][k+4]*b[k+4][j];
                                temp += a[i][k+5]*b[k+5][j];
                                temp += a[i][k+6]*b[k+6][j];
                                temp += a[i][k+7]*b[k+7][j];
                                temp += a[i][k+8]*b[k+8][j];
                                temp += a[i][k+9]*b[k+9][j];
                                temp += a[i][k+10]*b[k+10][j];
                                temp += a[i][k+11]*b[k+11][j];
                                temp += a[i][k+12]*b[k+12][j];
                                temp += a[i][k+13]*b[k+13][j];
                                temp += a[i][k+14]*b[k+14][j];
                                temp += a[i][k+15]*b[k+15][j];
                        }
                        for (;k<N;k++){
                                temp += a[i][k]*b[k][j];
                        }

                        c[i][j] = temp;
                }
        }
}
```

The fourth algorithm, "block", structures the matrix in sub-blocks of size 20.

```
void blocking(int N, int nb) {

        int i,j,k;
        int ii,jj,kk;
        double s00,s01,s10,s11;

        for( ii = 0; ii < N; ii += nb ){
                for( jj = 0; jj < N; jj += nb ){
                        for( i = ii; i < ii + nb ; i++ ){
                                for( j = jj; j < jj + nb ; j++ ){
                                        c[i][j] = 0.0;
                                }
                        }
                        for( kk = 0; kk < N; kk += nb ){
                                for( i = ii; i < ii + nb ; i += 2 ){
                                        for( j = jj; j < jj + nb ; j += 2 ){
                                                s00 = c[i ][j ];
```

```
                                        s01 = c[i ][j+1];
                                        s10 = c[i+1][j ];
                                        s11 = c[i+1][j+1];
                                        for( k = kk; k < kk + nb ; k++ ){

                                        s00 = s00 + a[i ][k]*b[k][j ];
                                        s01 = s01 + a[i ][k]*b[k][j+1];
                                        s10 = s10 + a[i+1][k]*b[k][j ];
                                        s11 = s11 + a[i+1][k]*b[k][j+1];
                                        }

                                        c[i ][j ] = s00;
                                        c[i ][j+1] = s01;
                                        c[i+1][j ] = s10;
                                        c[i+1][j+1] = s11;
                                    }
                                }
                            }
                        }
                    }
                }
```

To use BLAS we ran the function `cblas_dgemm`, the function that computes the square matrices with double precision.

2.3 Evaluation Methodology

The program was tested in a Linux 1133 MHz Celeron® system with 256 MB of main memory. The cache sizes were of 32Kb for L1 and 256Kb for L2.

We ran some test using 3 different matrix sizes. We decide to analyse the performance of the multiplication of two square matrices of dimension 20, 100 and 200. The choice of these matrices sizes aimed to test the cases where the matrices would fit in the L1 cache, would not fit in the L1 cache but fit in the L2 cache, and when they would fit neither the L1 nor the L2 cache. To satisfy the first case we have the size 20 because each matrix has 400 elements that occupy 8 bytes each (double) and, therefore, 6.4Kb. In case we have two matrices of size 100, the two matrices fit L2 cache (160Kb). Finally, there is a case in which with a size 200, they do not fit a L2 cache (640Kb).

Another fact we analyse were the compiler levels of optimisation, O0 and O2. These levels of optimisation are options of GCC compiler, since O0 does not add any optimisation in the code. The option of optimisation O2 performs practically all the supported optimisations, except for loop unrolling and function inlining.

The performance of the algorithms was given in CPE (Cycles per Element), where “element” refers to the matrix elements. In the multiplication of square matrices, the number of clock cycles per element is related to the number of needed operations; for example: in case of two matrices of size NxN we need NxN values for the result matrix, and for each element we need Nx2 element of the source matrix; so we need to do NxNxNx2 operations.

3 Collected Data

To the elaboration of the final results, we averaged the five results of each algorithm for each matrix size. The organization of the results was made under two perspectives: the first one was to compare the performance of the several algorithms; the second was to analyse the improvements of performance through the degrees of optimisation used in the compilation.

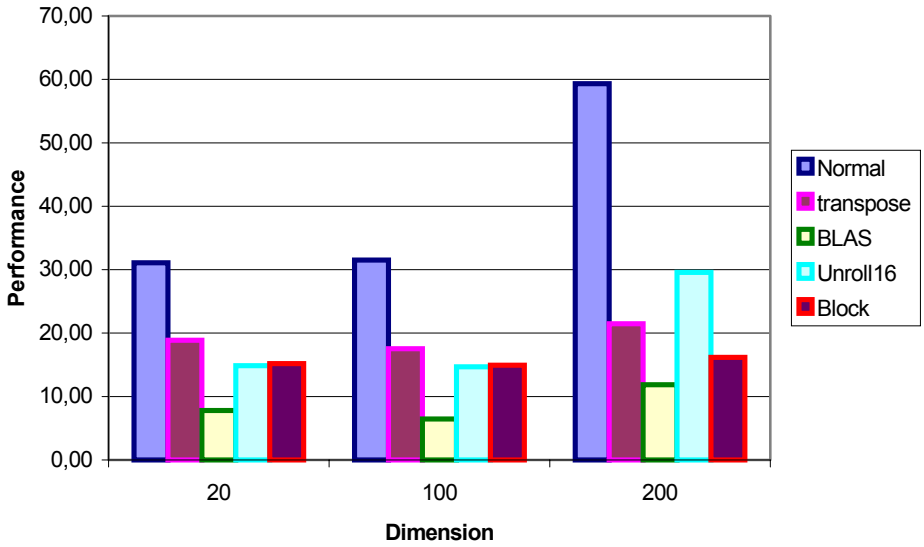

Fig. 1. Comparison between the five algorithms compiled with the GCC O0 optimisation level for the three different matrix sizes

As we can see in Figure 1, without choosing any level of optimisation when compiling, the BLAS algorithm turned to be the most efficient whatever the dimension of the matrices. With a close effectiveness to the BLAS algorithm, we found the Block algorithm and the Transpose. The Normal algorithm is, by far, the one that presents the worst effectiveness. While BLAS algorithm presents a value of about 11,86 CPE for a matrix of a dimension of 200, the Normal algorithm presents a value of about 59,35 CPE (for the same matrix).

There is no significant difference between the values obtained for the matrices of dimension 20 and the matrices of dimension 100, i.e., there is no real difference between cache L1 and L2. This might be due to the used processor [2].

When we analyse the same data but compiled with the O2 level of optimisation, we observe a great change in what concerns the most efficient algorithm (fig. 2.).

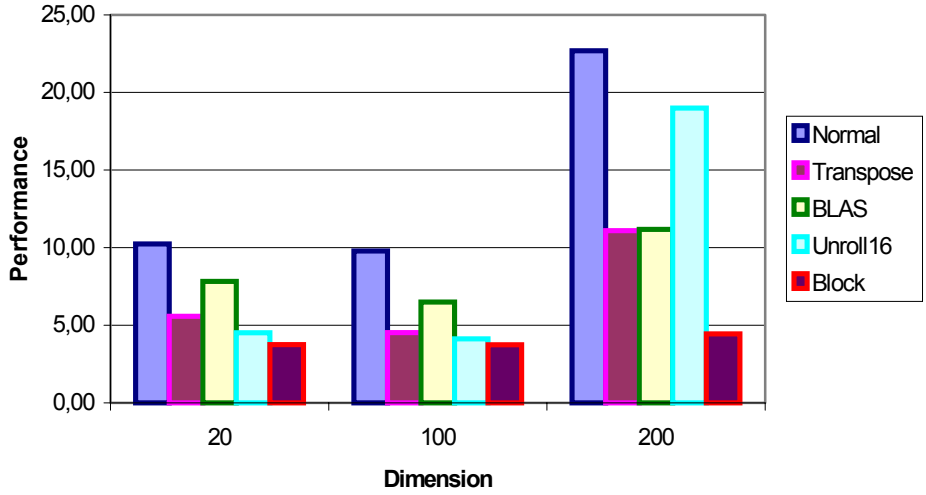

Fig. 2. Comparison between the five algorithms compiled with the GCC O2 optimisation level for the three different matrix sizes

Here, it is the Block algorithm that is the most efficient, needing only about 4,45 CPE for concluding the operation with matrices of a dimension of 200. The value of CPE

decreases for less than a half comparing with the necessary for the BLAS algorithm to conclude the same operation but without any level of optimisation.

After the Block algorithm, practically with the same values for the matrices of a dimension 20 and 100, comes the Unroll16 and the Transpose. A small advantage of the Transpose and the Unroll16 for matrices of small dimensions is noticed while BLAS gains a slight advantage for bigger dimensions. As the less efficient algorithm continues the Normal algorithm, although having a significant increase in the performance, now needing about 22,69 CPE for the conclusion the operation.

We can see that BLAS algorithm is the best when it is compiled without any compilation option. But when the level of optimisation is O2 the performance when compared with the other algorithms decrease because we are using an external function from BLAS and therefore, it is not subject to the optimisations that are executed when compiling the program.

Improvements can be found in almost all the algorithms (Fig. 3.) except in the BLAS algorithm, having in this case practically the same performance.

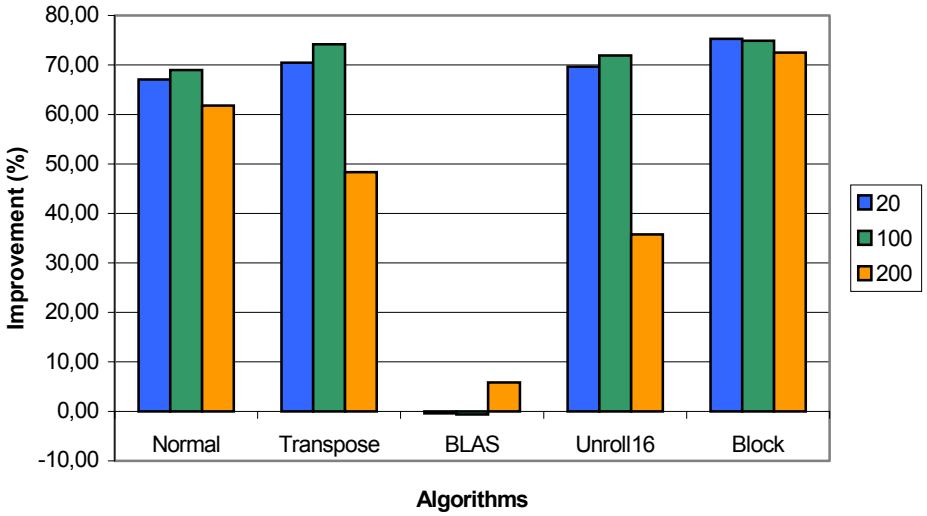

Fig. 3. Improvements from O0 to O2

The algorithm that had a wider increase in its performance to improve was the Normal, presenting an improvement of 67,09% for dimension 20, 68,99% for dimension 100 and 61,77% for dimension 200. The BLAS presents an improvement between 5,82% (for a dimension of 200) and a decrease of 0,39% (for a dimension of 20). By the way, for the matrices of dimension 20 all the remaining algorithms presented improvements in the performance, the Block algorithm improves about 75%, the Transpose about 70% CPE and the Unroll16 about 69% CPE. For the matrices of dimension 200 the performance of the Block algorithm improves about 73%, the Transpose about 48% and the Unroll16 about 36%.

An interesting thing was detected analysing the differences between the different sizes of matrices. Analysing the values obtained in the multiplications of matrices of dimension 100, we see that the results were better than the results of the multiplication of matrices of dimension 20. It was interesting because the matrices of size 20 were in the L1 cache and the matrices of size 100 were in the cache L2. This result is going to be explained in the next chapter.

In conclusion, analysing the results of the measures made in the multiplication of two square matrices of dimension 20, 100 and 200, and using the optimising levels O0 and O2

of the GCC compiler we see that the smallest CPE values were obtained with the optimisation level O2 and with the Block algorithm.

4 Data Analysis

The characteristics of the architecture have influence in the performance particularly the pipeline, the superscalar and has we had already see the memory organization.

The pipeline divides the instruction execution into many parts. This division makes with that an instruction that is needed it could usually be taken from the prefetch buffer (store of set of registers) rather than waiting for a memory read to complete. Each one of these divisions is handled by a dedicated part of hardware and all run in parallel. In fig. 4(a) we see a pipeline with five stages. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs. Stage 3 locates and fetches the operands, either from registers or from memory. Stage 4 does the work of carrying out the instruction, typically by running the operands through the data path of the CPU. And, stage 5 writes the result back to the proper register.

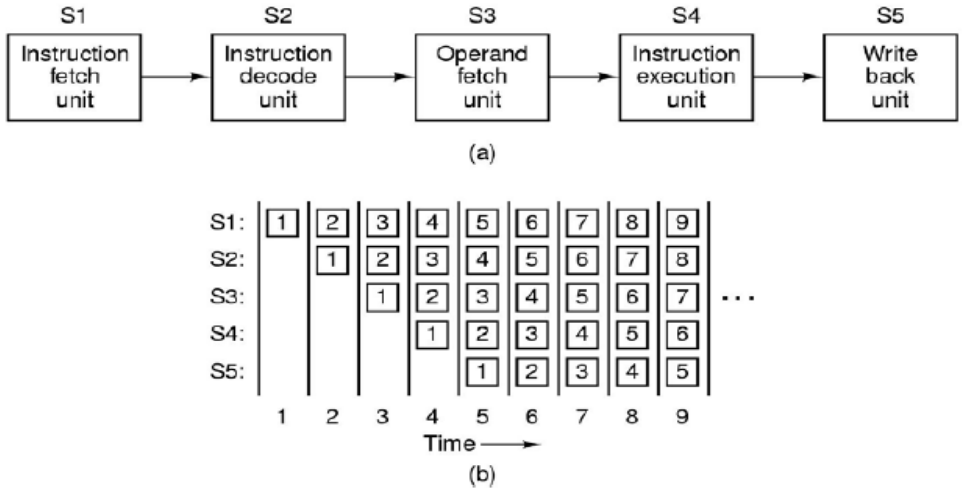

Fig. 4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated. **(**From "Computer Systems - A Programmer's Perspective", with kind permission of Randal E. Bryant and David R. O'Hallaron).

In Fig. 4(b) we see how the pipeline operates in time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction, and stage S3 fetches the third instruction, and so on. Finally, during cycle 5, S5 writes the result of instruction 1 back, and the other stages work on the other instructions.

The superscalar architecture represents an architecture that have not one pipeline but two or more pipelines. We are going to talk only of the dual pipeline CPU. In a dual pipeline CPU we have a unique instruction fetch unit that fetches pairs of instructions and put each one in the corresponding pipeline. But, to allow the run in parallel the two instructions cannot have conflict in the usage of registers, and cannot depend of the result of the other. This situation must be guarantee by the compiler in single pipeline architecture. We can see an example of a dual pipeline in Fig. 5.

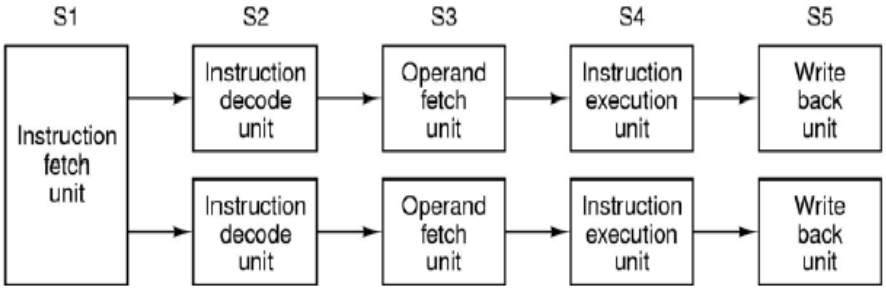

Fig 5 Dual five-stage pipelines with a common instruction fetch unit. **(**From "Computer Systems - A Programmer's Perspective", with kind permission of Randal E. Bryant and David R. O'Hallaron).

But the use of the parallel computation does not have only advantages, has some disadvantages to. In Parallel computation we need lot of registers to hold sums and products. For pointers and loop conditions we need 8 Floating-point registers. We cannot reference more operands than instructions set allows.

As we see the optimisation is dependent of the machine and of the programmer. On the point of view of the machine we see that the optimisation depends of the pointer code, the loop unrolling made by the compilers is not very optimised when we compare with the made by hand, and the instruction level parallelism is very depend of the machine.

The program instructions are kept in memory and the CPU reads them. We also can evaluate the location of a program. The code has a property very important that distinguishes from program data, it cannot be modified in runtime.

The hardware and the software complement each other. This complement is known by memory hierarchy. In Fig. 5. we see the typical memory hierarchy.

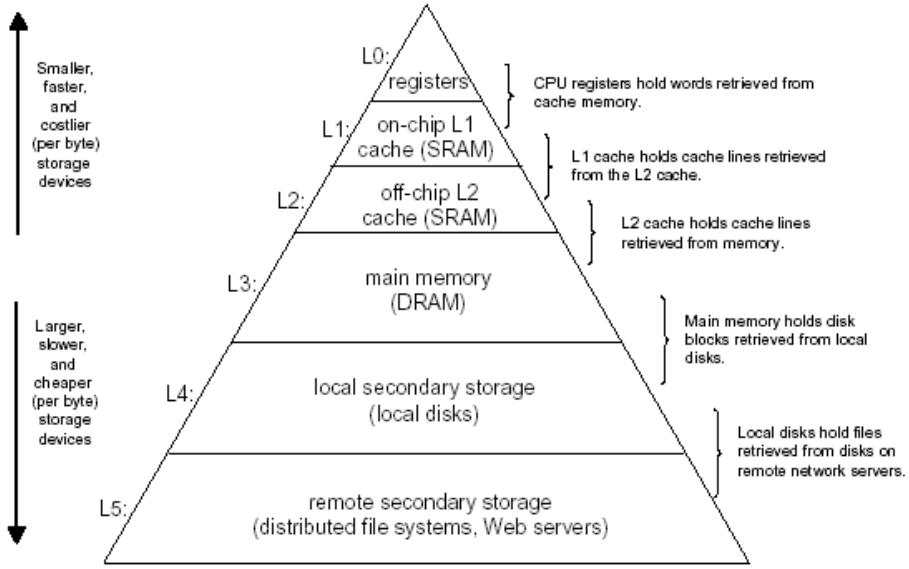

Fig. 5. The memory hierarchy**. (**From Computer Systems - A Programmer's Perspective, with kind permission of the Randal E. Bryant and David R. O'Hallaron)

In the highest level we have a small number of CPU registers that CPU can access in a single clock cycle. Next we have two small SRAM-based (Static RAM) cache memories that can be accessed in a little number of CPU clock cycles. Following are the main memory, a large DRAM-based (Dynamic RAM) that can be accessed in tens to hundreds of clock cycles.

The two small SRAM-based cache memories are known by L1 cache and L2 cache. The performance of cache is evaluated by a number of metrics, like: Miss rate (fraction of memory references during the execution of a program, or a part of a program, that miss), Hit rate (fraction of memory references that hit), Hit time (time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection) and Miss penalty (additional time required because of a miss).

Because the cycles are one of the biggest sources of inefficiency, in case they are not optimised, it is here that the compilers focus their attention. Through the collected data, we can present some ideas on the functioning of the CPU and memory.

The two main factors that influence in the performance of the applications are the techniques of writing the programs and the behaviour of the memory. We go to start for analysing the normal algorithm. For the execution of this algorithm, considering the dimension of 20, goes to be done 8000 multiplications, 8000 additions and two readings of memory for interaction. All this basic operations consume cycles of clock. Thus, latency is the total number of necessary cycles to complete this operation. Issue time is the number of cycles between successive independent operations. The number of clock cycles varies according to the different processors. Beside the clock cycles required by the operations described above, we still have to deal with the cache access.

One program that has a good performance tends to show a good Locality. The principle of Locality is: data that had been recently acceded are serious candidates to be acceded again (temporal locality) and data close to acceded data has good probabilities to be acceded (spatial locality). Programs that fill these characteristics tend to be more efficient.

Cache Hits means that when we need a value of cache of level k+1, this value already meets in cache of inferior level, k. Thus, prevents the access to cache k+1. Cache misses is always that the intended value is not in cache. This value now passes to be also in cache of level k.

As we see a program with a good locality will have lower miss rates and programs with lower miss rates will run faster.

The transpose algorithm has as more value the fact to have a minor miss rate therefore when operating the transposed matrix, is to operate for lines and not for columns, as in the normal algorithm. Thus, when acceding to the first element and as it is not in cache has to be read (miss). But as it also reads the following elements for cache, the reference to these is well succeed (hits). In the normal algorithm this does not happen because when acceding to the first element of the second matrix it also loads the next elements (of the array). Only that the next element to be acceded is n following positions. Thus, it has that to read the value of the memory, what it implies another miss.

As we may see from the results, the Unrolling 16 is one of the best algorithms for matrices with dimension 20 and 100. The reason for this good result is the fact of increasing the quantity of values calculated by loop step. This will reduce the efforts of, in each interaction, increase the index, test the loop condition and make the loop. This way, the loop jump is of k steps and by each step are calculated k elements (in our case k=16). However, this algorithm has its limitations namely if we have a great latency which is our case (multiplication). In the case that we have a great latency level in the Unrolling 16 performance there is a great damage for the process because as the parallelism is limited, we cannot have many operations in parallel. As the operations are dependent from the previous ones we have a latency level worth noting. Another problem of this algorithm takes place when we have a wide dimension. If the quantity of intermediate results surpasses the cache dimension we need to make some accesses to the memory, raising CPE.

Assembly Code for Unroll16

```
…
for (k=0;k<(N-15);k+=16){
        temp += a[i][k]*b[k][j];
        temp += a[i][k+1]*b[k+1][j];
        temp += a[i][k+2]*b[k+2][j]
…
```

```
...
movl   20(%esp), %eax
leal   0(,%eax,4), %ecx
movl   a, %edx
movl   12(%esp), %eax
leal   0(,%eax,8), %esi
movl   (%edx,%ecx), %edi
movl   12(%esp), %eax
leal   0(,%eax,4), %ebx
movl   b, %ecx
movl   16(%esp), %eax
leal   0(,%eax,8), %edx
movl   (%ecx,%ebx), %eax
fldl   (%edi,%esi)
fmull  (%eax,%edx)
fldl   (%esp)
faddp  %st, %st(1)
fstpl  (%esp)
...
```

Fig. 6. Example of the code assembly. First line of cycle k

As we can see from figure 6, for each line from k cycle we make a series of memory readings (leal). In the case both matrices are not completely in Cache L2 then we must read from the main memory, what results in more clock cycles for each element that is calculated. This is the reason because the performance of Unroll16 algorithm is not so good with dimension 200.

As we see a program with a good locality will have lower miss rates and programs with lower miss rates will run faster. So the idea that we want to leave is that the Block algorithm is an algorithm that has a good locality comparing with the other algorithms.

Besides, the Block algorithm is better located and the using of blocking will decrease the execution time. Making a block is organizing the data structure in a program into large blocks. After structuring the program, a block will be loaded to the L1 cache, all the reads and writes to that block are made, and then this block is discard and the next one is load, and so on. Doing this to matrices is dividing a matrix in sub matrices and manipulated the matrices as scalars. Blocking make the code harder to read but improves the performance.

The optimisation of a certain algorithm does not need to go through the creation of a total new algorithm or by the development of the compiler optimisation options. Sometimes small adjustments produce great increments in the performance. We are going to see if how we can improve the Normal algorithm with small changes. In the first modified version, instead of the result of the multiplication being immediately added to the writing variable, we chose to introduce a local temporary variable to where the result goes after the multiplications. Only at the end of cycle k can we copy the temporary variable to the writing variable.

```
void normal_1(int N) {
  int i,j,k;
  double temp;

  for (i=0;i<N;i++){
   for (j=0;j<N;j++){
     c[i][j] = 0.0;
     for (k=0;k<N;k++){
          temp += a[i][k]*b[k][j];
     }
     c[i][j]=temp;
    }
  }
}
```

The advantage of this algorithm is that the temporary variable is stored in the register. This way is much quicker to access this variable than to access the writing variable because this is, in the best of the chances, in cache L1. This is precisely what we can see in Figure 7 where we have a piece of cycle K from the assembly code of the two algorithm versions. We can see that the normal algorithm makes a series of calls to the memory, namely to read the value of c[i][j] and, at the end, to save it. In the modified algorithm this does not take place.

Algorithm Normal	Algorithm Normal 1
…	…
movl 24(%esp), %eax	movl 20(%esp), %eax
leal 0(,%eax,4), %ecx	leal 0(,%eax,4), %ecx
movl c, %edx	movl a, %edx
movl 20(%esp), %eax	movl 12(%esp), %eax
sall $3, %eax	leal 0(,%eax,8), %esi
movl %eax, 12(%esp)	movl (%edx,%ecx), %edi
movl (%edx,%ecx), %ecx	movl 12(%esp), %eax
movl %ecx, 8(%esp)	leal 0(,%eax,4), %ebx
movl 24(%esp), %eax	movl b, %ecx
leal 0(,%eax,4), %ecx	movl 16(%esp), %eax
movl c, %edx	leal 0(,%eax,8), %edx
movl 20(%esp), %eax	movl (%ecx,%ebx), %eax
sall $3, %eax	fldl (%edi,%esi)
movl %eax, 4(%esp)	fmull (%eax,%edx)
movl (%edx,%ecx), %ecx	fldl (%esp)
movl %ecx, (%esp)	faddp %st, %st(1)
movl 24(%esp), %eax	fstpl (%esp)
leal 0(,%eax,4), %ecx	leal 12(%esp), %eax
movl a, %edx	incl (%eax)
movl 16(%esp), %eax	jmp .L52
leal 0(,%eax,8), %esi	…
movl (%edx,%ecx), %edi	
movl 16(%esp), %eax	
leal 0(,%eax,4), %ebx	
movl b, %ecx	
movl 20(%esp), %eax	
leal 0(,%eax,8), %edx	
movl (%ecx,%ebx), %eax	
fldl (%edi,%esi)	
fmull (%eax,%edx)	
movl 4(%esp), %eax	
movl (%esp), %edx	
fldl (%edx,%eax)	
faddp %st, %st(1)	
movl 12(%esp), %eax	
movl 8(%esp), %edx	
fstpl (%edx,%eax)	
leal 16(%esp), %eax	
incl (%eax)	
jmp .L39	
…	

Fig. 7. Assembly code from loop k from two versions

As we can see in Figure 8, this method has a significant increase in the performance, when compiled with the level of optimization O0.

The second modified normal algorithm has, in relation to the original algorithm, the order of the cycles changed. Thus, as we can see below, firstly the cycle k is fixed, followed by cycle i and finally by cycle j.

```
void normal2(int N) {
  int i,j,k;
  double r;

  for (k=0;k<N;k++){
   for (i=0;i<N;i++){
      r = a[i][k];
      for (j=0;j<N;j++){
           c[i][j] += r*b[k][j];
      }
    }
  }
}
```

This algorithm can seem strange but, however, it has better results than the original algorithm. It is even better, for big dimensions, in relation to the first modified algorithm. The advantage of this algorithm is to have low miss rate. It is true that it has to make a store operation but as the reading of matrix B is made by line this leads it to have a low miss rate. For each element that goes to be read in the cache it immediately gets the following ones to be loaded. However, as we are running matrix B by line, the next values to be read are already loaded in the lowest level of memory. Thus, it does not have to go to the following level of cache again.

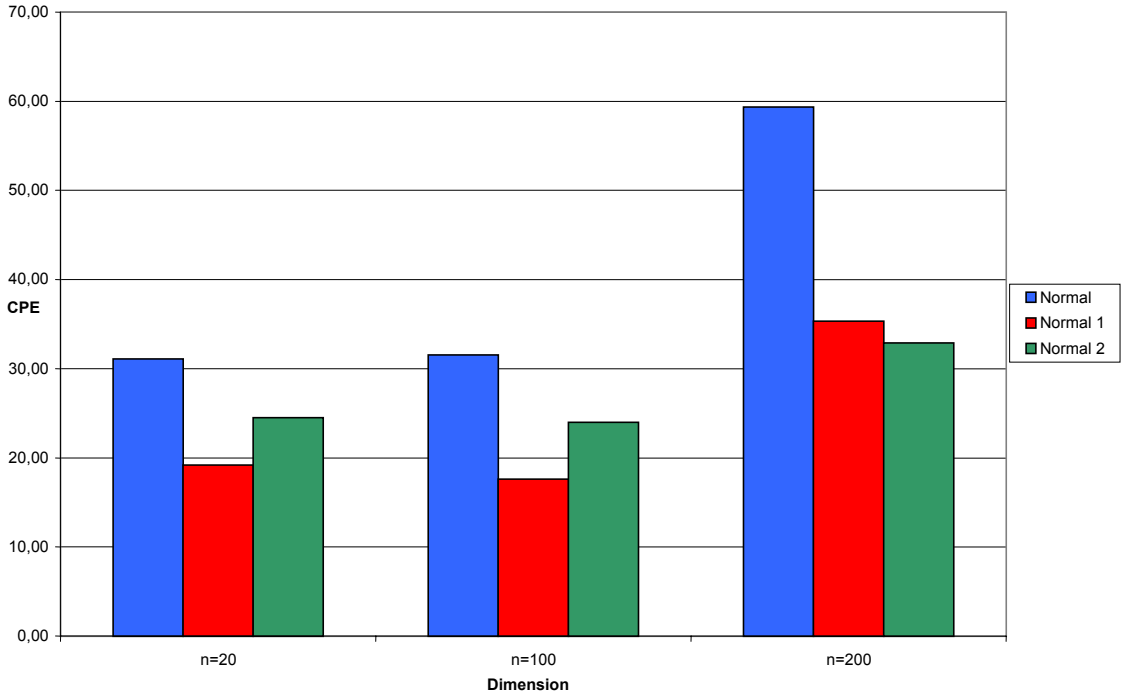

Fig. 8. Comparison between the original Normal and Transpose Algorithm and manual optimised algorithms

These were little examples that show how small adjustments can influence the performance of our algorithms.

5 Conclusions

This work did not reach all the objectives we determined for this research. All the tests were done in only one machine, what did not allow us to have a wider scope of results to

analyse. Another aspect that was not developed was the analysis of the code assembly generated by the compilation with different optimisation levels. This analysis will be important to effectively verify how the optimisation is made by the compiler according to the compilation options.

About the results, we must say that we expected the algorithm of the BLAS to be the most efficient. It was not this that was verified at least when we performed the compilation with optimisation. However, a certain tendency in BLAS was noticed to work better with matrices of a greater dimension, with a better precision of the results, being therefore the most indicated one when we intend to use a more mathematical source. This work intends to arise the sensibility of programmers towards questions concerning issues more associated with hardware and the ways you can and should explore the hardware.

It is not only necessary to dominate the diverse programming languages but also to understand the architecture of the CPU and the memory in a better way in order to be able to explore the capacities of the machine.

This work has raw material to go further towards several directions and to analyse a series of intervening factors in the performance of a determined program.

References

[1] Bryant, O'Hallaron: Computer Systems: A Programmers Perspective. Prentice Hall. (2002)

[2] Intel: Mobile Intel® Celeron® Processor (0.13µ) in Micro-FCBGA and Micro-FCPGA Packages. DataSheet, (2003) 13