DEPARTMENT OF APPLIED MATHEMATICS
AND COMPUTER SCIENCE

02614 - High Performance Computing

1. Matrix Multiplication

Authors:
Maria Garofalaki - s202378
Jorge Sintes Fernández - s202581
Klara Maria Ute Wesselkamp - s202381

Course responsible:
Bernd Dammann

January 8, 2021

Individual responsibilities

The following table shows who has the main responsibility for the different parts in this project:

Part	Student no.	Name
I.	-	Shared
II.	s202382	Klara Wesselkamp
III.	s202378	Maria Garofalaki
IV.	s202581	Jorge Sintes Fernández

Part I. matrix-matrix multiplication

function matmul_nat():

For this task we write the function matmul_nat() whose implementation can be seen in the code section below:

```
void matmult_nat(int m, int n, int k, double **A, double **B, double **C){
    int i1, i2,i3;
    for(i1 = 0; i1 < m; i1++) {
        for(i2 = 0; i2 < n; i2++){
            C[i1][i2] = 0;
        }
    }

    for(i1 = 0; i1 < m; i1++){
        for(i2 = 0; i2 < n; i2++){
            for(i3 = 0; i3 < k; i3++){
                C[i1][i2] += A[i1][i3] * B[i3][i2];
            }
        }
    }
}
```

The function matmul_nat() takes 6 arguments:

- **m** which is row size of matrix **A**
- **n** which is column size of matrix **B**
- **k** which is the size column of matrix **A** and the size of the rows of the matrix **B** accordingly.
- **A** which is a double pointer to matrix A_{mxk}
- **B** which is a double pointer to matrix B_{kxn}
- **C** which is a double pointer to matrix C_{mxn} which is the resulting matrix of the multiplication $C_{mxn} = A_{mxk}xB_{kxn}$

To get the product of matrices **A** and **B** we form three nested for-loops. In the first loop we iterate through all the row elements of **A** matrix. For each row of **A** we take all the column elements (third loop) and we multiply them with the element of the **B** matrix with the same row index as the specific column of **A** matrix and column index which takes it's value from the second nested for loop.

In the third nested loop we add together all the results of the multiplication between the elements of matrix **A** witch have the same column index with the row index of the elements of matrix **B**.The resulting matrix **C** has the same number of rows as the **A** matrix, and the same number of columns as the **B** matrix.

function matmult_lib():

The function matmul_lib() takes the exact same arguments as the function matmul_nat() and it's implementation is in the code section below:

```c
#include "cblas.h"

void matmult_lib(int m, int n, int k, double **A, double **B, double **C){
    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,m,n,k,1.,&A[0][0],k,&B[0][0],n
    ,0., \
    &C[0][0],n);
}
```

In this function we wrap the call of function **cblas_dgemm** which is provided to us by the **ATLAS** library (CBLAS implementation).

The **cblas_dgemm** routine computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices. The operation is defined as:

$$C := alpha * op(A) * op(B) + beta * C \tag{1}$$

In our case $alpha = 0$ and $beta = 1$ because we want $C = A*B$. We set the argument of the function **cblas_dgemm** according to link [1]

- **CblasRowMajor** which Specifies that the two-dimensional array storage is row-major
- **CblasNoTrans** (the first one) which Specifies that we take the **A** matrix as it is (not it's transpose).
- **CblasNoTrans** (the second one) which Specifies that we take the **B** matrix as it is (not it's transpose).
- **m** which is row size of matrix **A**
- **n** which is column size of matrix **B**
- **k** which is the size column of matrix **A** and the size of the rows of the matrix **B** accordingly.
- **1.** Specifies the scalar alpha which as we said above is 1
- **&A[0][0]**The pointer which shows where the matrix **A** is stored
- **k** which is the leading dimension of matrix **A** which in our case (row-major) is the size of column of A
- **&B[0][0]**The pointer which shows where the matrix is stored
- **n** which is the leading dimension of matrix **B** which in our case (row-major) is the size of column of B
- **0.** Specifies the scalar beta which as we said above is 0.
- **&C[0][0]**The pointer which shows where the matrix **C** is stored
- **n** which is the leading dimension of matrix **C** which in our case (row-major) is the size of column of C

comparing performance:

To check the efficiancy of our function matmul_nat() we compare it's performance with function matmult_lib() which we wrap the call of function **cblas_dgemm**.

For that reason we modified the file: **"mmh_bat.sh"**, more specifically I define the value of the variable **"SIZES"** to be SIZES=10 20 30 40 50 75 100 150 200 250 500 750 1000. In that way the .gcc is executed 13 times and in each iteration the parameters defining the matrix sizes(m,n,k) are all equal to the value that the SIZES variable gets.

In addition I change the variable **PERM** to "lib" and "nat" in order to get the output results for both. Finally I print the outputs together for comparison reasons:

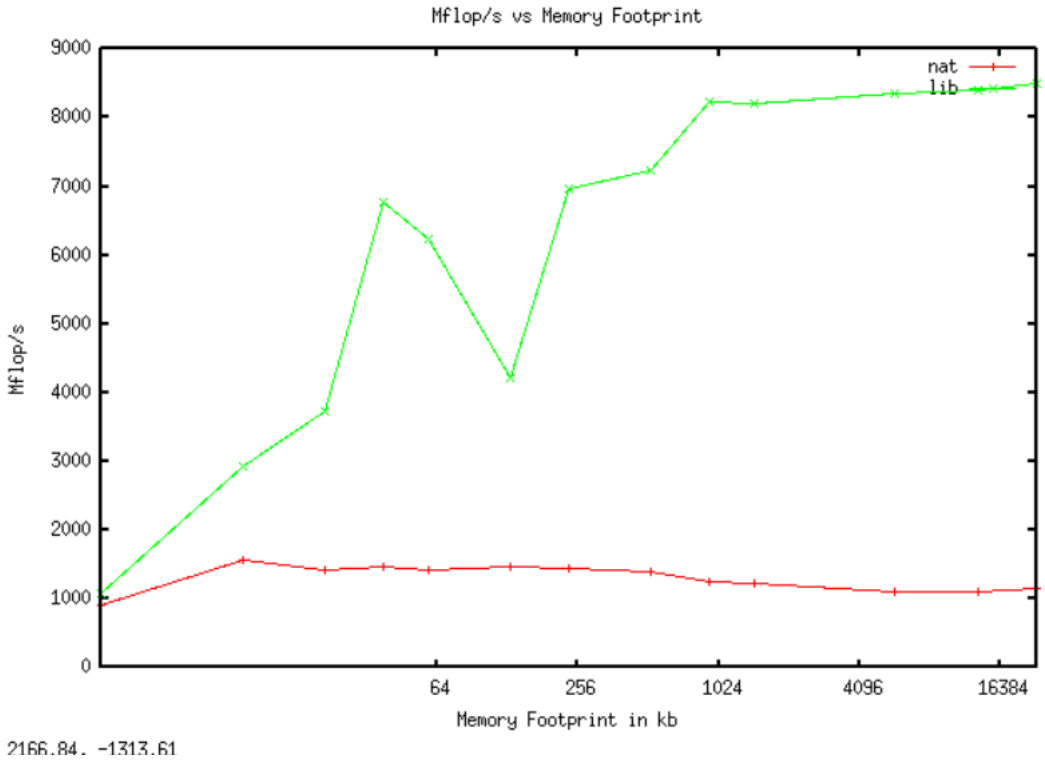


Figure 1: comparing performances between matmul_nat and cblas_dgemm functions

Part II. Loop Permutations

Question Implementation:

As we have seen in Part 1, in order to calculate the multiplied matrix C, we need three loops:

```
for(i1 = 0; i1 < m; i1++){
    for(i2 = 0; i2 < n; i2++){
        for(i3 = 0; i3 < k; i3++){
            C[i1][i2] +=A[i1][i3] * B[i3][i2];
        }
    }
}
```

Listing 1: pointBlock function

Although it may seem counter-intuitive, we can actually freely choose the order in which we perform those loops (if this doesn't seem convincing, consider that we are basically using a double sum, which is commutative). We thus come up with 6 permutations (corresponding to 3! possibilities). We call them by the order in which the dimensions are called, so we end up with the following programs

- matmult_ nmk.c
- matmult_ nkm.c
- matmult_ mnk.c
- matmult_ mkn.c
- matmult_ knm.c
- matmult_ kmn.c

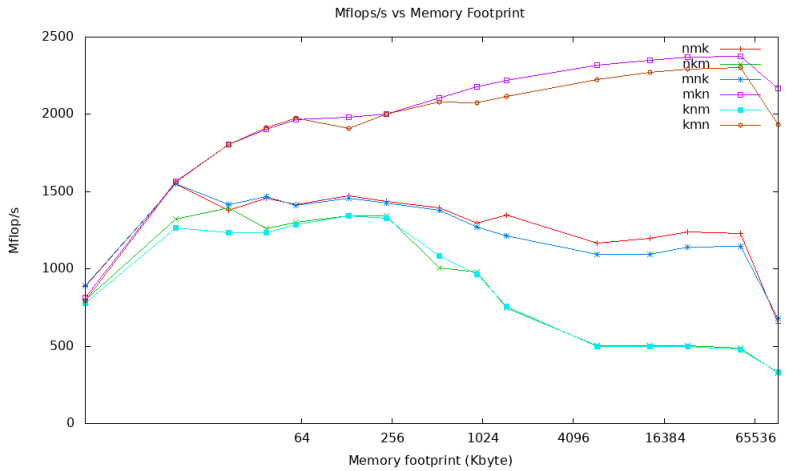
Question Performance Evaluation

In this part, there's two central questions: Which permutation is the fastest (for different matrix sizes) and can this be changed by using better compiler options and optimizers. How well each matrix performs depends on it's size. We know that one of the costliest thing about the operation is retrieving the data from memory. If we're working with small matrices, the entire 3 matrices will fit in the cache, the bigger it is the farther out the data is stored. We'll thus want to evaluate (1) a matrix size that fit's entirely in the cache to evaluate the efficiency of the operations themselves (2) a matrix that needs to be stored in further out memory. (**how to find cache size of assigned server??**)

(1) We chose Matrix Sizes of 20 10 30. We remark that there's no need to optimize loops here, as the entire matrices will fit in the cache. As the execution times are very low for this operation and in order to make the execution times comparable we repeat the same operation several times (100 times).

Function	nmk	nkm	kmn	knm	mnk	mkn
CPU-Time (s)	2,91	3,23	2,89	3,32	3,39	2,82

(2) Here we chose the matrix size 200 100 300, which takes up a memory size (using double precision) of 880K Bytes, which is well above the L1-Cache Size.
The intuition is that the fastest options are those, which can operate on an entire cache line at once, in other words, first fixing a row of A (that is m) and a row of B(that is k, which would make the matmult_mkn and matmult_mnk-Permutation the fastest. We receive a first confirmation of this by the plotting of the memory footprint (that is the size of the matrix) and the Mflops/s:



following CPU-Time Comparison for the different functions. The compiler options were at first left unchanged:

```
gcc -g -O3
```

Listing 2: basic compilation

Function	nmk	nkm	kmn	knm	mnk	mkn
CPU-Time	9.086	11.668	5.564	12.439	8.916	5.364

describe options, what is difference between Ofast and ffast-math? The compiler option -Ofast is optimized for speed, thus disregarding the strict standard compliances. Amongst others it turns on the -ffast-math mode. [2] The -ffast-math mode again enables among others '-funsafe-math-optimizations', '-ffinite-math-only', '-fno-rounding-math', '-fno-signaling-nans', '-fcx-limited-range' and '-fexcess-precision=fast'. A lot of the options here rely on the security that there will be no NANs or missing values in our data. This can be assumed, because the matrices are fully initialized and at this size an overflow seems unlikely. The CPU-Time comparison for different functions and matrices with sizes 200 100 300 and the -Ofast -option enabled:

```
gcc -g -Ofast
```

Listing 3: -Ofast compilation

Function	nmk	nkm	kmn	knm	mnk	mkn
CPU-Time	9.146	11.708	5.584	12.549	9.497	5.364

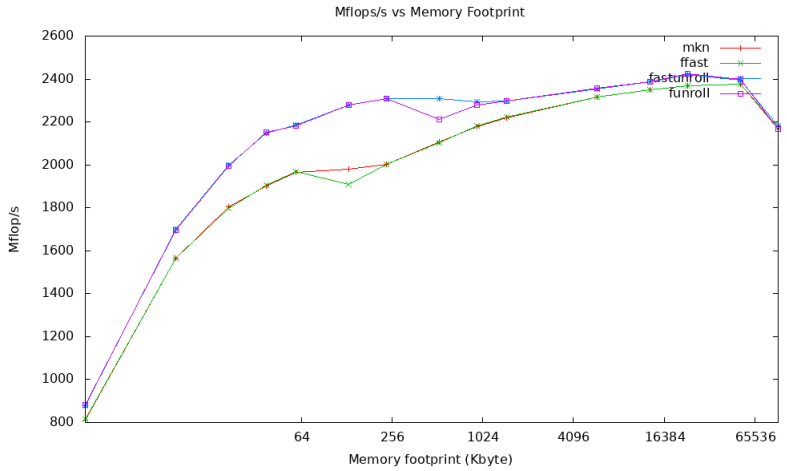
Surprisingly enough this option does not seem to have had a significant effect on the performance. This might be due to the fact that the maths in our code are very basic, not requiring a lot of optimization. We are therefore now going to examine the performance of one of the most important loop-tuning options: **-funroll-loops**. Loop-unrolling is an option that reduces the number of iterations in the loop by doing several consecutive commands in one iteration (incrementing the index) if that's possible. This is promising in our case. CPU-Time Comparison for different functions and matrices with sizes 200 100 300.

```
gcc -g -O3 -funroll-loops
```

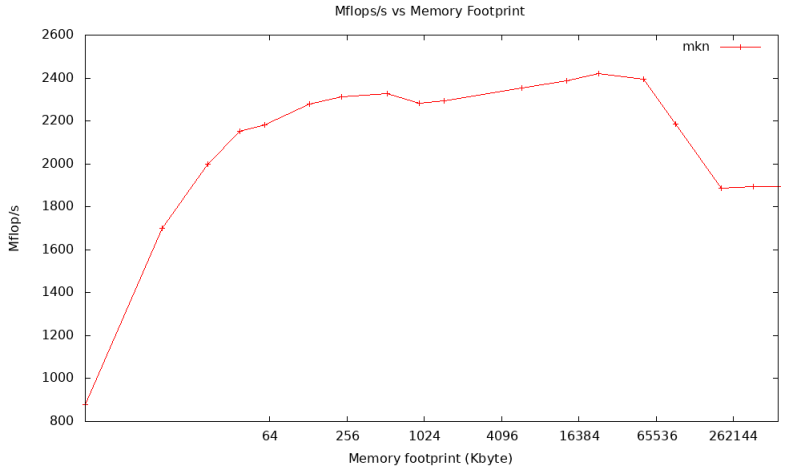
Listing 4: unrolled compilation

Function	nmk	nkm	kmn	knm	mnk	mkn
CPU-Time	8.126	11.918	5.354	12.669	8.286	5.234

In order to compare the different compiler options we chose to focus on our best permutation, matmult_mkn and compare the 3 different aforementioned versions of compilation, as well as a combination of -ffast-math and -funroll-loops:



Finally, we are going to present the behaviour of the function for increasingly larger memory. We see that we can see the beginning of the classic step-function which indicates that the data is now in a further-out memory.



Part III. Analysis

For analysing the differences in performances between different versions and measure specific characteristics we used the er_print command to extract data from the experiments which we get by using the given collect_batch.sh file and changing the field "PERM" according to the version we wanted to test.

After the execution of the er_print on the above cases we get the following results:

Characteristics for kmn:

Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 D-cache Hits	Excl. L2 D-cache Misses
5.354	11661683504	464145149	451341142	12012015

Characteristics for knm:

Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 D-cache Hits	Excl. L2 D-cache Misses
12.669	27197248096	2647227832	2051841642	595595601

Characteristics for mnk:

Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 D-cache Hits	Excl. L2 D-cache Misses
8.286	17467500179	653004208	240075075	412412413

Characteristics for mkn:

Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 D-cache Hits	Excl. L2 D-cache Misses
5.234	11541563150	281688091	278487087	2002004

Characteristics for nkm:

Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 D-cache Hits	Excl. L2 D-cache Misses
11.918	26876927227	2906508916	2832885887	73073076

Characteristics for nmk:

Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 D-cache Hits	Excl. L2 D-cache Misses
8.126	17827861196	86427030	73623023	11011017

By inspecting the tables above we can get in the following intuitions:

- **mkn case** has great performance just 5.234 Total CPU (sec). That is totally reasonable us the function matmult_mkn performs the multiplication of matrices A and B by accessing them both row-wise. That is the reason we have the lower number in L2 D-cache Misses only 2002004
- **kmn case** has the good performance too only 5.354 Total CPU (sec). That is also reasonable us the function matmult_kmn performs the multiplication of matrices A and B by taking the whole row of matrix A(two outer loops) and multiply it with every element of a specified column of B which is accessed row wise(inner loop). In that way the elements of matrix B which we want to acquire more often as they are in the inner loop they belong to a continuous row of memory and this is the reason we have reduced number of D-cache Misses.
- **nmk case** has medium performance 8.126 Total CPU (sec) because the function matmult_nmk performs the multiplication of matrices A and B by accessing A row-wise and B column-wise so we have a big loss of performance in accession of B matrix and this is the reason we have more D-cache Misses than in the previous cases.

- **mnk case** has medium performance 8.286 Total CPU (sec) because the function matmult_mnk performs the multiplication of matrices A and B by accessing A row-wise and B column-wise so we have a big loss of performance in accession of B matrix and this is the reason we have more D-cache Misses than in the previous cases.
- **nkm case** does not have such a good performance, 11.918 Total CPU (sec). That is totally reasonable us the function matmult_nkm performs the multiplication of matrices A and B by accessing them both column-wise and that is the reason we have such an increased number of D-cache Misses which has a big effect in the total performance.
- **knm case** has the worst performance of all, 12.669 Total CPU (sec). That is happening because in the function matmult_knm where we perform the multiplication between the matrix A and B we accessing matrix A column wise (which as we mention before increases the number of D-cache Misses) and that is the reason we are getting such a big number in L2 D-cache Misses=595595601.

Moreover from the results above we come to the obvious conclusion that the why we are acceding our matrices have a major role in the performance of our code. That is the reason that when we are acceding our matrices row-wise we have such a reduce number of cache Misses. So it should be a major priority to avoid cache misses by acceding all our matrices row-wise. Finally if we have to choose the best case that it would be the **mkn case** for all the reasons we mentioned above.

We can also analyse the functions using the Oracle Analyzer Tool.

Total CPU Time INCLUSIVE sec.	/zhome/51/5/153768/assignment1/matmult_mkn.c
	1. #include <stdio.h>
	2.
	3. void
	<Function: matmult_mkn>
0.	4. matmult_mkn(int m, int n, int k, double **A, double **B, double **C) {
	5.
	6. int i1, i2,i3;
0.040	7. for(i1 = 0; i1< m; i1++){
	8. for(i2 = 0; i2 < n; i2++){
0.	9. C[i1][i2]=0;
	10. }
	11. }
	12.
	13.
	14.
0.	15. for(i1=0; i1<m; i1++){
0.020	16. for(int i3=0; i3<k; i3++){
0.070	17. for(i2 = 0; i2 < n; i2++){
5.104	18. C[i1][i2]+=A[i1][i3]*B[i3][i2];
	19. }
	20. }
	21. }
	22.
0.	23. }

Total CPU Time INCLUSIVE sec.	/zhome/51/5/153768/assignment1/matmult_knm.c	
	1.	#include <stdio.h>
	2.	
	3.	void
		<Function: matmult_knm>
0.	4.	matmult_knm(int m, int n, int k, double **A, double **B, double **C) {
	5.	
	6.	int i1, i2,i3;
0.030	7.	for(i1 = 0; i1< m; i1++){
	8.	for(i2 = 0; i2 < n; i2++){
0.	9.	C[i1][i2]=0;
	10.	}
	11.	}
	12.	
	13.	
	14.	
0.	15.	for(i3=0; i3<k; i3++){
0.	16.	for(int i2=0; i2<n; i2++){
0.590	17.	for(i1 = 0; i1 <m ; i1++){
12.038	18.	C[i1][i2]+=A[i1][i3]*B[i3][i2];
	19.	}
	20.	}
	21.	}
	22.	
0.	23.	}

We have the worst and the best performing function here. In both cases - as was to be expected - the bottleneck of the function is the calculation, while the loops (where the loop condition needs to be checked) takes only a negligible amount of time.

Another thing the analyzer allows us to see the effect of the -funroll-loops-Option: We remember the CPU-Times in the knm-Code and look at the following times, compiled without named option:

Total CPU Time INCLUSIVE sec.	/zhome/51/5/153768/assignment1/matmult_knm.c
	1. #include <stdio.h>
	2.
	3. void
	<Function: matmult_knm>
0.	4. matmult_knm(int m, int n, int k, double **A, double **B, double **C) {
	5.
	6. int i1, i2,i3;
0.	7. for(i1 = 0; i1< m; i1++){
	8. for(i2 = 0; i2 < n; i2++){
0.050	9. C[i1][i2]=0;
	10. }
	11. }
	12.
	13.
	14.
0.	15. for(i3=0; i3<k; i3++){
0.	16. for(int i2=0; i2<n; i2++){
1.691	17. for(i1 = 0; i1 <m ; i1++){
10.687	18. C[i1][i2]+=A[i1][i3]*B[i3][i2];
	19. }
	20. }
	21. }
	22.
0.	23. }

Part IV. Block Implementation

For the last part, we need to write a blocked version of our matrix-matrix multiplication function, called `matmult_blk()`. We'll modify the permutation `matmult_mkn()` so that it works using square blocks of size `bs`. It will work in the following way:

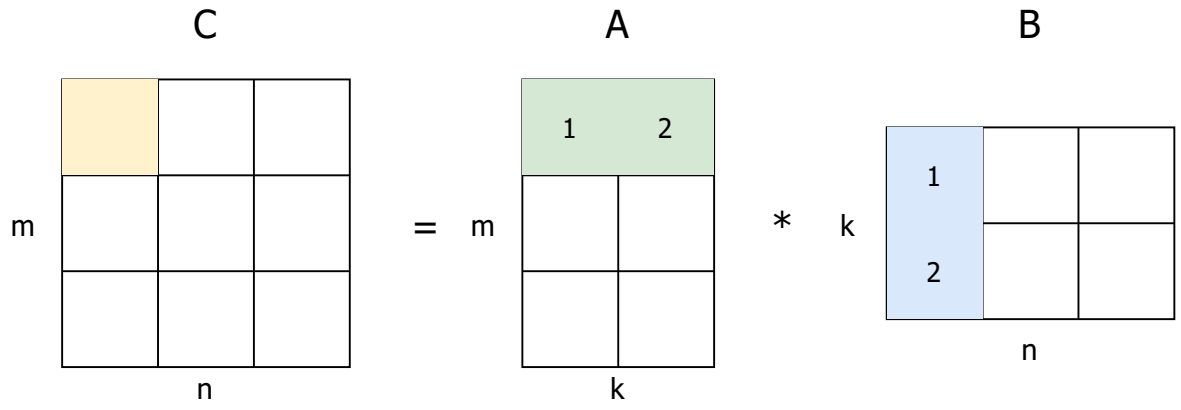


Figure 2: Block Matrix Multiplication

Each square of the matrices in figure 2 has to be understood as a block matrix of size `bs` × `bs`. That way, the yellow block of matrix C will be the result of the operation block A1 × block B1 + block A2 × block B2. Note that every multiplication of the block is in itself a matrix multiplication. Also, we're not taking into account yet that the sizes `m`, `n` and `k` may not be a mutiple of `bs`. We'll deal with that later.

The first problem we run into is that, in order to implement this way of calculating the product, we need somehow to slice the arrays indexing them, which may seem difficult to do in C at first glance. First, we thought of using a new double pointer `**Z` that took advantage of how pointers and arrays relate in C:

```c
void pointBlock(int row, int col, int bs, double **A, double **Z){
    /* This function takes Z, which is a matrix bs*bs and makes the pointers stored at
    Z[0], Z[1]... to point at the locations &A[row][col], &A[row+1][col].
    In this way we make Z point at the values of A we want for the block.
    CAREFUL: Z needs to be declared and allocated in memory with:
    Z = malloc(bs * sizeof(double *)); before using this function. */

    int i;
    for(i = 0; i < bs; i++){
        Z[i] = &A[row+i][col];
    }
}
```

Listing 5: pointBlock function

This function definitely works, and one can access the contents of the block matrix by using `Z[0][1]` as a normal 2d array in C. However, if we stop and think for a bit, we can see that in this implementation the contents of Z will not be stored contiguously in memory, but every row will be located in a different part of it.

The purpose of the algorithm is making sure that both block matrices can fit on the cache at the same time to make the product calculation faster. If the processor stores data in the cache linearly as we've been told in class, obviously this approach is not good at all. We need a way to store data in the Z matrix contiguously. So, we came up with this other function:

```c
double ** malloc_2d(int m, int n){
  /* initializes a matrix of size m*n dynamically, storing the matrix contiguously in
    memory. */
    int i, j;
    double **A;

    A = malloc(m * sizeof(double *));
    A[0] = malloc(m * n * sizeof(double));
    for(i = 1; i < m; i++){
        A[i] = A[0] + i*n;
    }
    return A;
}

void getBlock(int row, int col, int bs, int m, int n, double **A, double **Z){
    /* This functions copies the values from A we want for the block into a matrix Z
    of size bs x bs i.e. make a bs*bs block starting in position A[row][col].
    Z needs to be initialized with: Z = malloc_2d(bs, bs); before being used. */

    int i, j;

    for(i = 0; i < bs; i++){
        for(j = 0; j < bs; j++){
            Z[i][j] = 0;
        }
    }
}
```

Listing 6: malloc_2d and getBlock functions

In this function, we are storing the values of the block in an array `Z` we are initializing using `malloc_2d`. Thus, we are making sure the block matrices are going to be stored in memory as we want them. Figure 3 illustrates the difference between both approaches pretty well.

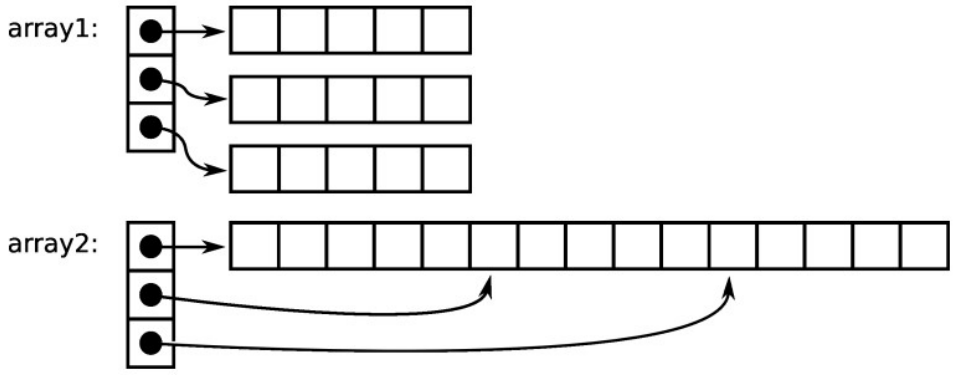


Figure 3: Difference between both approaches [3]

However, there's another inconvenience. What happens when `m`, `n` or `k` are not multiples of `bs`? We will have the following case:

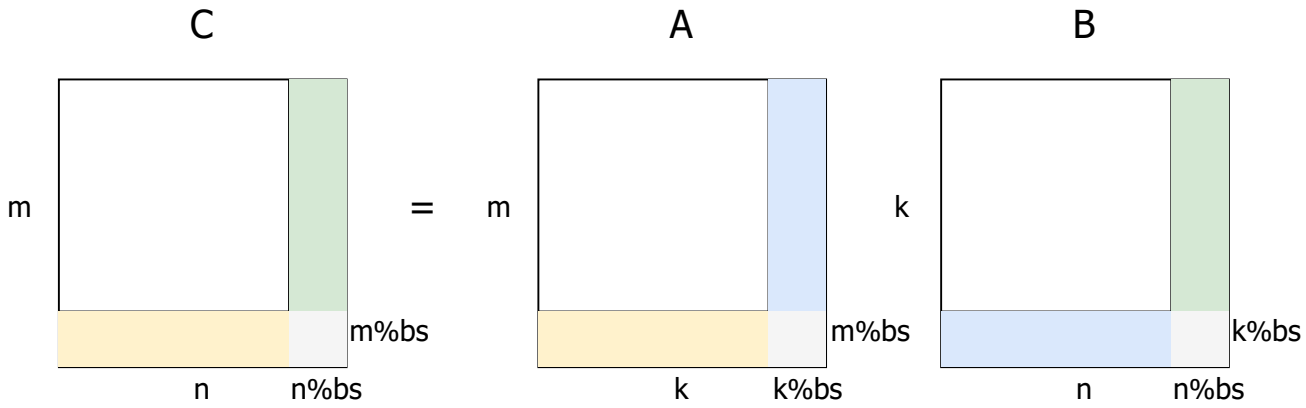


Figure 4: Possible remainders in the matrices

We'll deal with this problem by adding rows and columns of 0s to all the matrices in the following way, until we make every dimension a multiple of `bs`. The result we will be getting is correct, as from the properties of matrix multiplication it follows:

$$\left[\begin{array}{c|c} C & 0 \\ \hline 0 & 0 \end{array}\right] = \left[\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right] * \left[\begin{array}{c|c} B & 0 \\ \hline 0 & 0 \end{array}\right].$$

To implement this we need to modify the `getBlock()` function so we don't trigger a segmentation fault when trying to access parts of A and B which are out of bounds (these are the places where we will allocate the 0s). The final implementation will look like this:

```
void getBlock(int row, int col, int bs, int m, int n, double **A, double **Z){
    /* This functions copies the values from A we want for the block into a matrix Z
    of size bs x bs i.e. make a bs*bs block starting in position A[row][col].
    Z needs to be initialized with: Z = malloc_2d(bs, bs); before being used. */

    int i, j, limi, limj;

    for(i = 0; i < bs; i++){
        for(j = 0; j < bs; j++){
            Z[i][j] = 0;
        }
    }

    limi = bs;
```

```
    limj = bs;

    if(row >= m - m % bs){
        limi = m % bs;
    }

    if(col >= n- n % bs){
        limj = n % bs;
    }

    for(i = 0; i < limi; i++){
        for(j = 0; j < limj; j++){
            Z[i][j] = A[row + i][col + j];
        }
    }
}
```

Listing 7: getBlock() modified

```
void matmult_blk_inside(int row, int col, int bs, double **ZA, double **ZB, double **C){
    /* Calculates the multiplication of the block matrices ZA*ZB and stores it in the
    corresponding block for   matrix C. */

    int i1, i2, i3;
    for(i1 = 0; i1 < bs; i1++){
        for(i3 = 0; i3 < bs; i3++){
            for(i2 = 0; i2 < bs; i2++){
                C[row + i1][col + i2] += ZA[i1][i3] * ZB[i3][i2];
            }
        }
    }
}

void matmult_blk(int m, int n, int k, double **A, double **B, double **C, int bs){
    int i1, i2, i3;
    double **ZA, **ZB;

    // Initialize C to be a 0 matrix.
    for(i1 = 0; i1 < m; i1++){
        for(i2 = 0; i2 < n; i2++){
            C[i1][i2] = 0;
        }
    }

    // First we calculate all entries that can be subdivided in blocks of bs x bs.
    // We allocate in memory the two matrices were we will be storing the blocks.
    // printf("%d", bs);
    ZA = malloc_2d(bs, bs);
    ZB = malloc_2d(bs, bs);

    for(i1 = 0; i1 < m; i1 += bs){
        for(i3 = 0; i3 < k; i3 += bs){
            for(i2 = 0; i2 < n; i2 += bs){
                getBlock(i1, i3, bs, m, k, A, ZA);
                getBlock(i3, i2, bs, k, n, B, ZB);
                matmult_blk_inside(i1, i2, bs, ZA, ZB, C);
            }
        }
    }
}
```

Listing 8: matmlt_blk()

However, to use this function we need to initialize the matrix C as
`C = malloc_2d(m + (bs - m % bs), n + (bs - n % bs));`
otherwise we will get a segmentation error when trying to access the places outside C. As in the assignment we need to use the file `matmult_c.gcc` to compare our code, and we cannot change the declaration of C inside it (we don't have the source code) we will need to make a little change to our function to be able to run it without segmentation faults:

```
void matmult_blk(int m, int n, int k, double **A, double **B, double **D, int bs){
    int i1, i2, i3;
    double **ZA, **ZB, **C;

    // Create C to work with it in this function
    C = malloc_2d(m + (bs - m % bs), n + (bs - n % bs));

    // Initialize C to be a 0 matrix.
    for(i1 = 0; i1 < m; i1++){
        for(i2 = 0; i2 < n; i2++){
            C[i1][i2] = 0;
        }
    }

    // First we calculate all entries that can be subdivided in blocks of bs x bs.
    // We allocate in memory the two matrices were we will be storing the blocks.
    // printf("%d", bs);
    ZA = malloc_2d(bs, bs);
    ZB = malloc_2d(bs, bs);

    for(i1 = 0; i1 < m; i1 += bs){
        for(i3 = 0; i3 < k; i3 += bs){
            for(i2 = 0; i2 < n; i2 += bs){
                getBlock(i1, i3, bs, m, k, A, ZA);
                getBlock(i3, i2, bs, k, n, B, ZB);
                matmult_blk_inside(i1, i2, bs, ZA, ZB, C);
            }
        }
    }

    // Copy the results over to D, the matrix we are passing as argument.
    for(i1 = 0; i1 < m; i1++){
        for(i2 = 0; i2 < n; i2++){
            D[i1][i2] = C[i1][i2];
        }
    }
}
```

Listing 9: matmlt_blk() modified

Now we're ready to find the optimal block size. We'll make a little bash script to try different matrix sizes with different block sizes and store the performance of the calculation. In this script `m = n = k = size`.

```
#!/bin/bash

#BSUB -J find_blocksize
#BSUB -W 10
#BSUB -q hpcintro

rm -f *.txt

msizes="76 112 149 161 172 200"
bsizes="2 5 7 10 15 20 30 35 40 50 70 100 120 150 180"

for size in $msizes
```

```
do
    for block in $bsizes
    do
        if (($block < $size)); then
            echo $size and $block
            matmult_c.gcc blk $size $size $size $block >> "./($size)_output.txt"
        fi
    done
done

echo All done!
```

Listing 10: Bash script to test block sizes modified

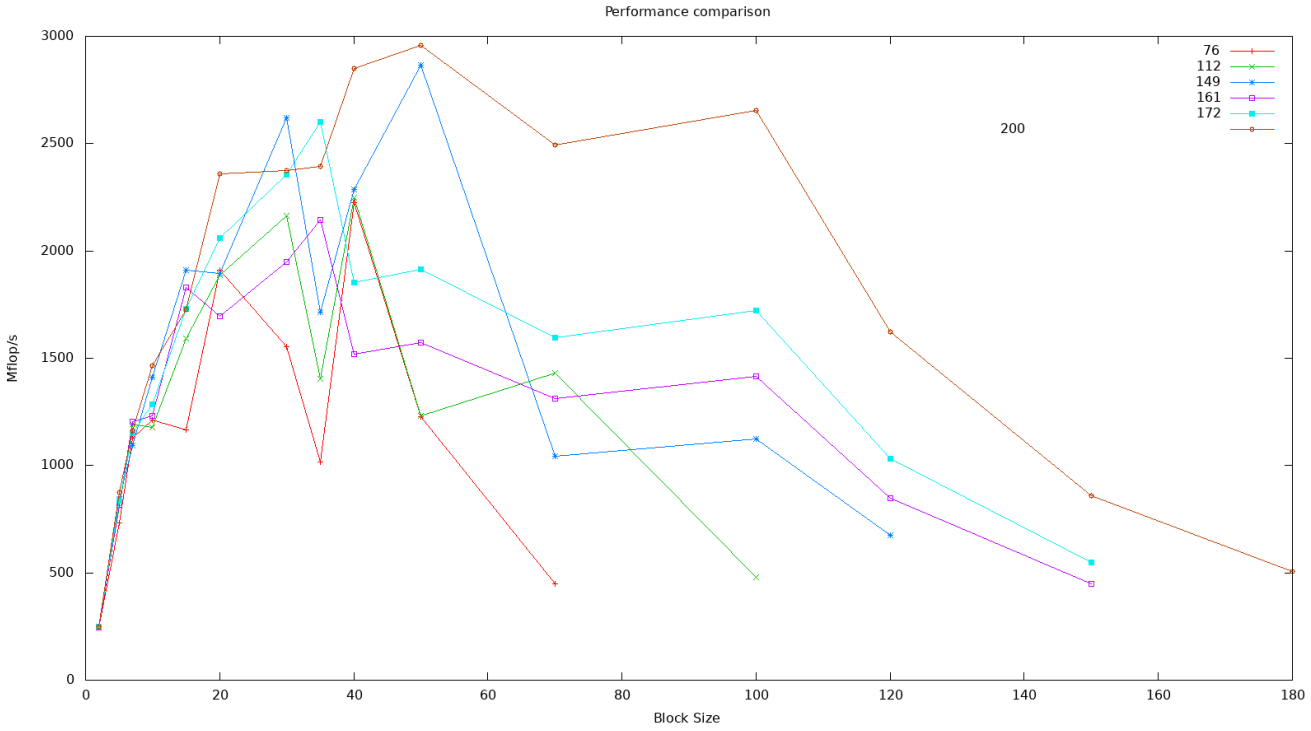


Figure 5: Performance test for different block sizes in matrices with different sizes

In figure 5 we can observe that the algorithm behaves badly when the blocksize is small or when it gets close to the actual size of the matrix, and peaks performance somewhere where the blocksize is 40% of the size of the matrix.

Lastly, we need to compare this block implementation with the original `mkn` algorithm, to see if we can get an actual improvement in performance. To do this, we'll also try different compiler options for the library. The bash script we use to implement the different tries onto the `hpcintro` is the following:

```
#!/bin/bash

#BSUB -J comparison
#BSUB -W 10
#BSUB -q hpcintro

rm -f mkn_output.txt blk_output.txt

msizes="30 50 76 112 149 161 172 200 250 300 350 400 500 800 1000 2000 3000 4000"

for size in $msizes
do
    echo $size
```

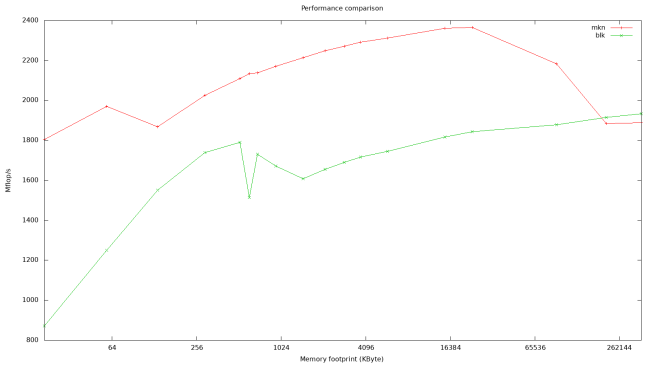
```
    block=$((40*$size/100))
    matmult_c.gcc mkn $size $size $size >> "./mkn_output.txt"
    matmult_c.gcc blk $size $size $size $block >> "./blk_output.txt"

done

echo All done!
```

Listing 11: Bash script to compare mkn and blk

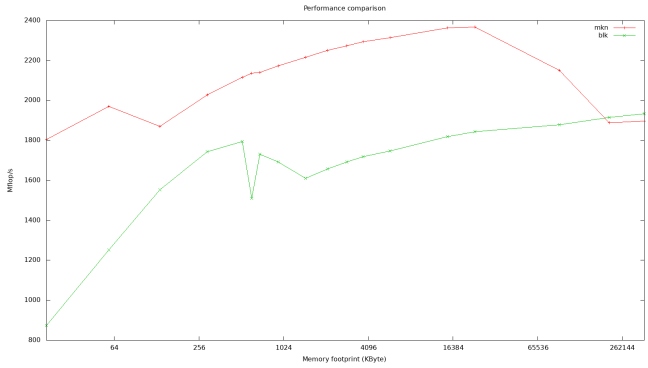
In figure 6 we can observe the results of this tests. Overall, for low values of memory i.e. when working with small matrices, the block function `blk` performs worse than the `mkn` implementation of the algorithm. It's not until we get to the higher values of memory storage, corresponding to calculations of 4000x4000 matrices.

When implementing different compiler options, we get the same behaviour we already saw in part II: the option `-ffast-math` makes practically no notable change. However, we do see an improvement in both functions with `-funroll-loops` , specially in the block matrix.

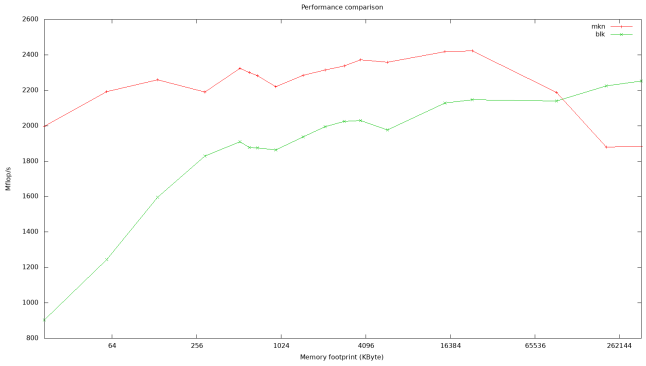
It would be interesting to see how the performance behaves further to the right, with bigger matrices. Also, we should find the optimal block size to implement in those high regions, as it seems that the `blk` function could be useful when working with these huge matrices. Sadly, we have no time to make further analysis and checks.



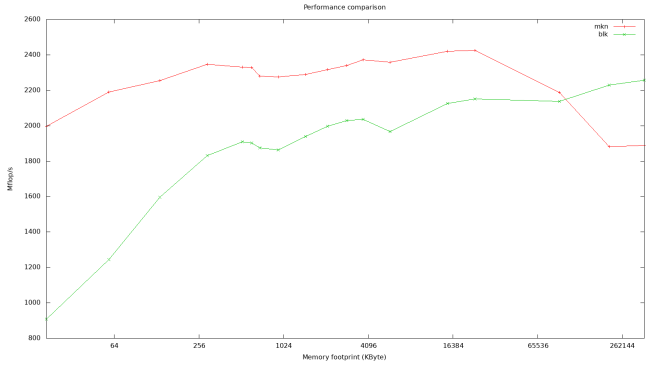
a) With `-O3`



b) With `-O3 -ffast-math`



c) With `-O3 -funroll-loops`



d) With `-O3 -ffast-math -funroll-loops`

Figure 6: Comparison between mkn and blk implementation using different compiler options

References

[1] Intel. *Developer Reference for Intel® oneAPI Math Kernel Library - C*. 2020. URL: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/blas-and-sparse-blas-routines/blas-routines/blas-level-3-routines/cblas-gemm.html?fbclid=IwAR2EHNuWvsXWgfxAjxsIn_Uu_p5DZaBMLvAHZzay-PP41psTpgQPO4PttLg.

[2] Richard M. Stallman. *GNU - Optimizer Documentation*. 2003. URL: https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gcc/Optimize-Options.html#Optimize-Options.

[3] Steve Summit. *comp.lang.c Frequently Asked Questions*. 1995. URL: http://c-faq.com/aryptr/dynmuldimary.html.