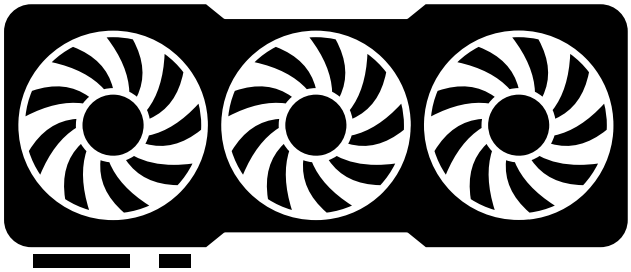

Parallel processing in GPU graphics cards

Center for Computational Simulation (CCS)

Author: Rongshen Wang Lin

Contents

Preface

The following document is intended to be a compilation of the main concepts studied during the collaboration grant sponsored by the Center for Computational Simulation (CCS). It includes links to relevant sources that can be accessed when the cursor is situated on top of the specific word. The important advancements start in section 7.

1. C++ Benchmarks

1.1. Single core and Multi-core Performance

Using only one core of the CPU it took several minutes to perform the 10,000 iterations. In the task manager I wasn't able to see a 100% use of the core, only peaks that sometimes reached 90%, but were mostly around 60%.

Visual Studio C++ supports OpenMP, so a new function was created that did the same operations but in parallel. There was a significant reduction in time because all cores worked at 100% connected to a power supply. When running on battery power it stabilises on 80%.

1.2. Results

The following are the results of the benchmark for our selected case (array dimensions of 5,000 and 10,000 iterations). The processor used is the Intel i5-10210U that uses 4 cores.

Table 1: CPU single and multiple core test in C++.

Single core	**Multi-core**
565.581	102.795

The results seem to be better for OpenMP because the single core test didn't extract all of the performance available. A practical but technically wrong solution would be to use the multi-core test and limit all cores except one. Additionally, I've performed the CPU benchmarks in Fortran, MATLAB and Python.

1.3. Additional Results

Fortran, whether using the `matmul` function or a simple loop never reaches a 100% use on any of the cores. Using `matmul` on Fortran is similar in speed compared with OpenMP on C++.

Table 2: CPU results for MATLAB, Python and Fortran.

Language	**Single core**	**Multi-core**
MATLAB	N/A	109.247901
Python	N/A	46.659596
Fortran	101.297 (`matmul`)	

1.4. Conclusions

MATLAB and Fortran didn't use 100% of CPU but achieved results similar to C++ which did use all cores at full capacity. I don't know if Fortran uses BLAS (Basic Linear Algebra Subprograms) but Python and MATLAB do. Python gives the best results but that might be because MATLAB limits CPU usage. I've looked for information about how BLAS works but it is low level and requires time to understand in detail.

2. Analysis of Theoretical Limits

The main goal of our research is to find whether an expensive processor or a GPU cluster is faster using the same budget. It would be of interest to create a simple model that can predict the maximum speed a GPU cluster can achieve.

2.1. Theoretical Time

In the benchmark we are using as reference we are multiplying several times a matrix with a vector, where the matrix has dimensions N and M. As a first approximation, the theoretical time can be written as the equation shown below.

$$t^* = \frac{4 \times \text{Number of operations}}{\text{Clock Speed} \times \text{Vectorization} \times \text{Number of threads}} \tag{1}$$

Where the number of operations is equal to $2 \times N \times M \times TIMES$, the vectorization argument is 512/32 (using single precision) and the number of threads is 1. We are taking into account that several operations can be done at the same time using a type of instructions available in modern processors[1].

For reference, my laptop has an Intel i5-10210U processor, with a clock speed of 2.1 GHz. The benchmark we are doing has a matrix of size 5000×5000 and the number of iterations is 1000. If we calculate the theoretical time, we get that $t^* = 5.952$ seconds. This time is not very accurate because my processor does not support the AVX-512 extension.

2.2. Column and Row Operations

After writing the `my_matmul` function, we can see it is really slow compared to the intrinsic function. This occurs because the usual way of doing a matrix-vector multiplication is not efficient for the processor, which has to access elements in the memory that are very far apart. If we rewrite the function that does the operation by columns it is way faster.

$$\begin{bmatrix} a_{11} & \cdots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NM} \end{bmatrix} \begin{Bmatrix} x_1 \\ \vdots \\ x_N \end{Bmatrix} = \sum_{j=1}^{M} \begin{bmatrix} a_{1j} \\ \vdots \\ a_{Nj} \end{bmatrix} x_j = \sum_{j=1}^{M} \begin{Bmatrix} b_1^* \\ \vdots \\ b_N^* \end{Bmatrix}_j = \begin{Bmatrix} b_1 \\ \vdots \\ b_N \end{Bmatrix} \tag{2}$$

The difference is that now each column of the matrix gets multiplied by an element of the vector x_j, resulting in a new vector $\{b^*\}_j$. Adding all the vectors together gives the original result, thus being the same as the one obtained in the conventional (naive) way.

```
b = 0
do j = 1, M
    do i = 1, N
        b(i) = b(i) + A(i, j)*x(j)
    end do
end do
```

Figure 1: Simple implementation in Fortran.

Using the previous method, the time was 343.264 seconds. With the new function, we achieve a time of 28.637 seconds, which is a speedup of almost 12. The test was done without any optimization options, because if active, the times would be the same due to the vectorization.

[1] This is reviewed on the following weeks and can have modifications.

3. Size Influence

The matrix size plays a big role in the time it takes to operate. After doing some tests, we came to the conclusion that the larger the matrix, the longer it takes to finish. The main hypothesis is that if the matrix fits in the cache, it will be faster. Once it doesn't fit, it will have to start looking for space elsewhere, thus being slower.

To prove this, we can run several tests for different dimensions. The number of iterations must also vary so that the number of operations remains constant, and thus the theoretical time. Once finished, we can plot the results in a graph.

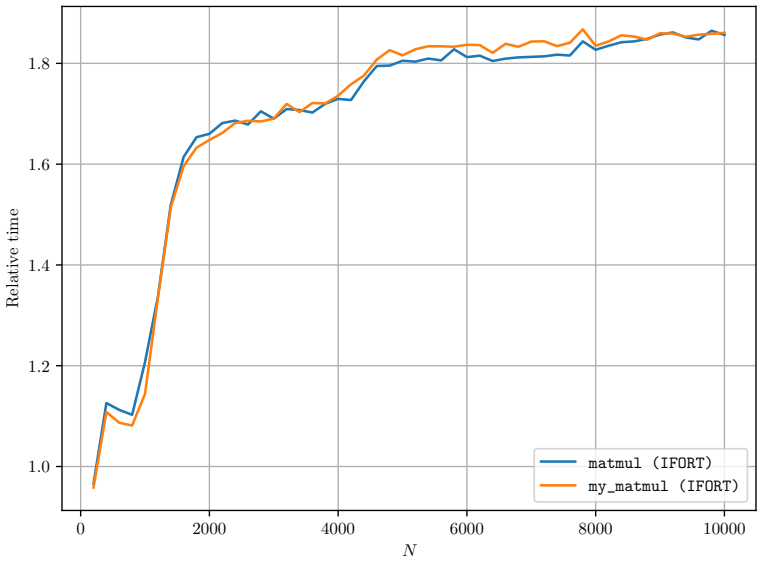

Figure 2: Comparison of speed using different dimensions.

For more resolution, we do the same but using smaller steps. By doing this, we can see the behavior more clearly and find the maximum size which can fit in the cache. In my case, the optimal seems to be around 1,000 but there is some weird behavior for smaller sizes. This behavior will be analyzed in detail in the next section, and has relation with the CPU cache.

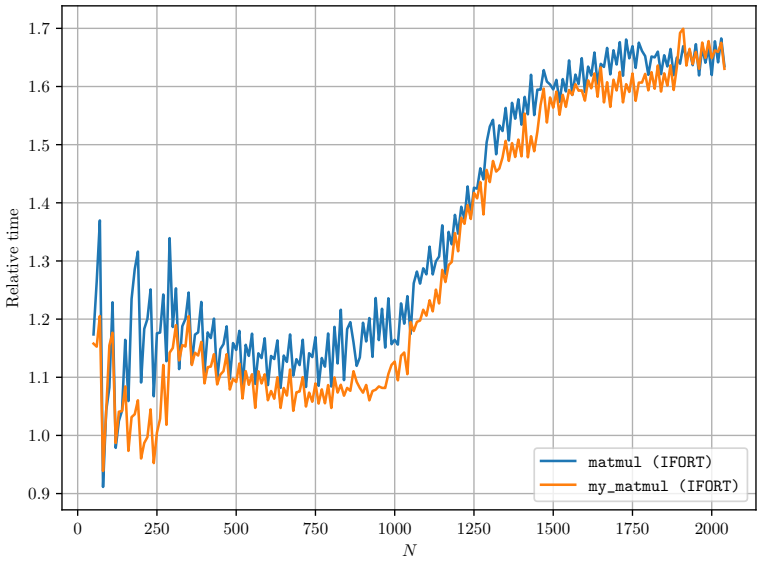

Figure 3: Detailed graph to find the optimal size.

4. Cache and Pipelining

4.1. Block Operations

As explained previously in the previous week, if the matrix fits in the cache memory, we can achieve times close to the theoretical. A strategy that might work is to partition the matrix and the resulting vector horizontally in k blocks.

$$\left[\begin{array}{c} A_1 \\ \hline \vdots \\ \hline A_k \end{array}\right] \left\{\begin{array}{c} x_1 \\ \vdots \\ x_N \end{array}\right\} = \left\{\begin{array}{c} b_1 \\ \hline \vdots \\ \hline b_k \end{array}\right\} \tag{3}$$

$$\{b_i\} = [A_i]\{x\} \qquad i = 1, \ldots, k \tag{4}$$

The implementation in Fortran can be done creating a two new variables that store the blocks we are working with (b_i and A_i) in each iteration. Instead of slicing inside the loop, it can be done outside and then repeat the operation as many times as required. In the code below, Q is the number of blocks and N is the dimension of the square matrix.

```
do i = 0, Q-1
    A_i = A(i*N/Q+1:(i+1)*N/Q, :)
    do k = 0, TIMES
        b_i = matmul(A_i, x)
    end do
    b(i*N/Q+1:(i+1)*N/Q) = b_i
end do
```

Figure 4: Block matrix multiplication in Fortran.

After doing some tests with the code, I found that the optimal is to get blocks that fit in the cache appropriately, using as much as possible. This way it reduces the number of multiplications and maximizes speed.

4.2. BLAS Optimization

The Basic Linear Algebra Subprograms are a set of routines that were programmed in the early 1970s. They are aimed to optimize operations and are widely used to this day. There is an option to force the intrinsic Fortran function `matmul` to use a BLAS subroutine. In GFortran, this can be activated by with the option `-fexternal-blas`.

In the case of the Intel Fortran Compiler (ifort), it can be activated following the next steps: *Properties > Fortran > Optimization > Enable Matrix Multiply Library Call > Yes (/Qopt-matmul)*. The only downside is that it uses all cores and despite that the results are worse than expected. One could also call the `sgemv` routine directly but that is a different topic.

4.3. CPU Cache

A cache is a small, fast memory which stores data that the processor thinks it will be frequently used. There is a hierarchy of cache levels, commonly L1, L2 and L3. Each core has its own L1 and L2 cache, while the L3 cache is shared between all cores. L1 is the fastest level, followed by L2 and L3.

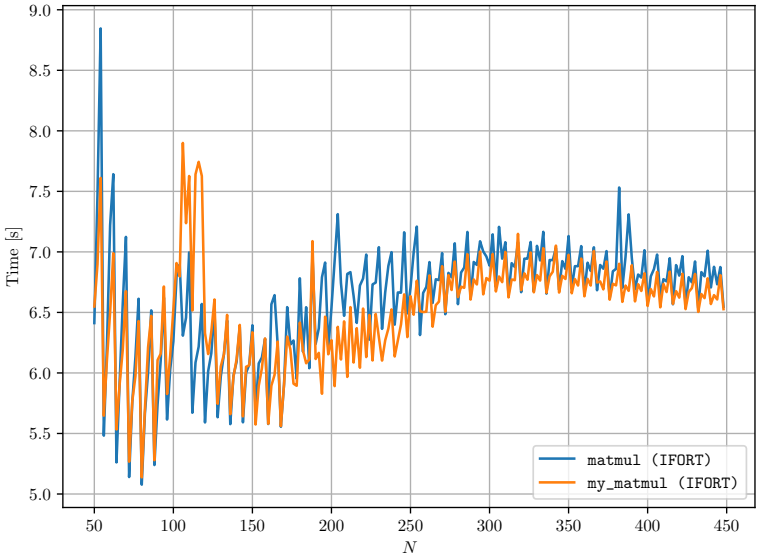

Figure 5: Small dimension results.

5. Understanding BLAS

Before diving deeper and doing tests without a clear idea, we must understand how BLAS and its improved versions (LAPACK, MKL, GotoBLAS, OpenBLAS) optimize matrix multiplications. For that purpose, during this week I analyzed a couple of papers and documents.

5.1. Fortran Performance Best Practices

5.1.1. Memory Accesses

1. Loop/Array Ordering: Fortran uses column-major ordering so the innermost index should be the one that varies in the inner loop.
2. Indirect Addressing: sometimes the memory access is not known so it can be better to create a dimension that forces contiguous accesss in memory.
3. Repeated Array Accesses: if an element in a loop is accessed repeatedly by using `x(i,j)`, it can be beneficial to create a temporal variable to store that element. This prevents access to DRAM every time though most compilers can store it in cache.
4. Allocatable, pointer, target, aliasing, contiguous arrays: in Fortran, pointers are rarely used since the introduction of `allocatable` arrays. Pointers aren't contiguous and can also have the problem of aliasing.
5. Array Layout: there are two things to consider: cache and vectorization.
   (a) To maximize cache performance, it is useful to organize the data in a related manner, to make future access quicker.
   (b) I don't understand this point.

5.1.2. Vectorization

To make it easy for the compiler to vectorize loops, it is ideal to follow some recommendations. If-statements should be outside the innermost loops, as well as print statements. There is also an important case that is called loop-carried dependence. For example if there is an accumulation operation, it can't do SIMD parallelism. The solution is to separate the loops, so some of them can be vectorized.

5.1.3. Miscellaneous

1. Array Allocation: it is recommended to limit the use of dynamic arrays and use stack arrays as much as possible. The problem is that we have very large arrays (heap).
2. Array Slicing: this feature should be avoided as it causes a big performance loss. It could only be used to move contiguous chunk of data, without operations. Otherwise, it is better to write things as loops when operations are involved and to fuse loops whenever possible.
3. Array Temporaries: when calling a subroutine where array slicing is involved, the performance degrades. This is because it creates a temporary array that it's constantly allocating and deallocating. To solve this, it is better to avoid the slicing altogether.
4. Register Spilling: sometimes it is hard for the compiler to keep the relevant local variables in registers and it spills some onto the stack. For this reason, it is recommended to reduce to distance between the assignation of the variable and its use.

5.1.4. Floating Point Issues

When using floating points, there are some issues to be aware of. Division is more expensive than multiplication. This means when there is a division inside a loop, it is better to replace this by the inverse of the value and then multiply inside the loop. Additionally, it is useful to take advantage of Fused Multiply and Accumulate (FMA) that will compute `a*x + b` for the cost of a multiplication. This feature in included on AVX2 and CUDA.

5.2. Optimizing Matrix-matrix Multiplication on Intel AVX

5.2.1. Introduction

Intel's AVX is a type of parallel processing wherein single instruction is applied to several data streams simultaneously or in parallel. It has 16 registers (YMM0-YMM15) in 64-bit mode. Intel's AVX is a 256-bit prescript that extends the capabilities of Intel Streaming SIMD Extensions (SSE) and is developed for the application of intensive floating-point.

5.2.2. Optimization Techniques

1. Loop Unrolling: it involves reducing the number of iterations by processing several instructions in the inner loop at the same time. Even though it increases the complexity, it reduces the amount of cache accesses by keeping data in the registers.
2. Blocking: large matrices are broken down into smaller fixed size matrices. In the article, the idea it to divide matrix A and C into a $1 \times BS$ row and matrix B into $BS \times BS$ blocks. The block size is chosen so that the submatrix of B and the row of C can fit in the cache.

```
for (kb = 0; kb < N; kb += BS) {
    for (jb = 0; jb < N; jb += BS) {
        for (i = 0; i < N; i++) {
            for (k = kb; k < kb + BS; k++) {
                acc1 = a[i][k]
                for (j = jb; j < jb + BS; j++) {
                    c[i][j] += acc1 * b[k][j];
                }
            }
        }
    }
}
```

Figure 6: Blocked matrix-matrix multiplication in C.

3. Prefetching: there are two types, hardware and software. The latter can be created automatically or manually and makes use of PREFETCH instructions. Using this, a compiler or programmer can insert prefetch code into the system. It can be implemented at different levels of memory hierarchy.

The blocking algorithm in fig. 6 can be better visualized if it represented graphically, instead of loops. To do this, it is important to follow every loop index. To better explain the algorithm, we first need to define some terms:

- The block size n.
- The matrices have a size of N.
- Matrices A and C are partitioned twice (for clarity), first partition will have an asterisk and the second will be written in small case, e.g. A becomes $A*$ and then it is split into a.

$$\left[\begin{array}{c|c} C^* & \cdots \end{array}\right] = \left[\begin{array}{c|c} A^* & \cdots \end{array}\right] \left[\begin{array}{c|c} b & \cdots \\ \hline \vdots & \ddots \end{array}\right] \tag{5}$$

$$\left[\begin{array}{c} C^* \end{array}\right] = \left[\begin{array}{c} c \\ \hline \vdots \end{array}\right] \qquad \left[\begin{array}{c} A^* \end{array}\right] = \left[\begin{array}{c} a \\ \hline \vdots \end{array}\right] \tag{6}$$

In the equations above, b is a $n \times n$ block whilst c and a are $1 \times n$ arrays. Once partitioned, all that is left is to visualize how the loops are performing the different operations.

$$\left[\begin{array}{ccc} c_1 & \cdots & c_n \end{array}\right] = \left[\begin{array}{ccc} a_1 & \cdots & a_n \end{array}\right] \left[\begin{array}{ccc} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{array}\right] \tag{7}$$

$$c_j = \sum_{k=0}^{n} \sum_{j=0}^{n} a_k b_{kj} \tag{8}$$

These last equations describe the inner loop in a mathematical way, though there is an index missing. In each iteration, an element of array a is multiplied with an entire row of block b. Each product ($a_k b_{kj}$) contributes to an element of the c array. Index k indicates that after finishing operations with the first row of b, the second element of a and the second row of b again contribute to the same array c.

In conclusion, the blocking algorithm described is a simple implementation that can increase performance by partitioning in blocks that can fit in cache. To implement a similar algorithm in Fortran, several changes should be made, mainly due to the code being written in C which follows a row-major order instead of column-major order.

5.3. Anatomy of High-Performance Matrix Multiplication

This paper is written by Robert van de Geijn and Kazushige Goto, author of GotoBLAS library. It presents the basic principles of the matrix-matrix multiplication used in the aforementioned library.

5.3.1. Notation

Matrices are partitioned into blocks of columns or rows. The original matrices are of size $m \times n$ and typically $m = n$. Every partitioned operation has been named depending on the size of the blocks involved. The basic idea is that the large operation (GEMM) is broken down into several calls to smaller operations such as GEPP, GEMP, GEPM and finally decomposed into GEBP, GEPB and GEPDOT kernels.

It is important to mention that the GEPB kernel is very similar to the blocking algorithm described in the previous paper. The authors of this paper have decided to implement GEBP because it has superior performance. Blocking is made choosing several variables: k_c, n_r, m_c and m_r for register blocking. The selection process needs to be understood properly.

6. Modern BLAS Algorithms

After reading about how modern implementations of BLAS achieve such great performance, during this week we try to copy the results that were achieved in the paper mentioned last week. During this week, the concept that I understood was that modern implementations of BLAS mainly optimize the loop that goes to the registers, the often called Micro Kernel.

6.1. Blocking Technique

We first tried the blocking algorithm, which is the 5 index nested loop that partitions the operations in blocks that fit in the cache. After adapting the algorithm to Fortran which uses column-major, the results are poor. In some cases it is similar to the intrinsic `matmul` function but many times slower than `sgemm`.

6.2. Implementation in C

In this link there is an example of a pure C implementation of the algorithm that shows the speed compared to the original BLAS reference. It is the method that most modern implementations such as GotoBLAS and MKL are based of. The code listed in the example is divided in several steps and functions.

1. Packing matrices into buffers.
2. Micro Kernel that multiplies panels from A and B.
3. Macro Kernel which multiplies blocks of A and B, previously packed into buffers.

6.2.1. Packing and Buffers

First, it is important to pack the elements of the matrices. Usually the values of an array are organised with padding, thus not completely filling the memory. In C, there are two functions for each matrix: `pack_MRxk` and `pack_A`.

The buffers are arrays that store temporal data, packed for size and quick access. The sizes of these arrays are defined as `MC`, `KC` and `NC`. Additionally `MR` and `NR` are defined for the `_C` buffer. In the first part of the algorithm, `mb`, `nb` and `kb` are defined as the main indices. Additionally, the remainder of the divisions is defined as `_mc`, `_nc` and `_kc`.

7. Theoretical Limits

After several months of tweaking code, reading documentation and doing a variety of tests, we have understood the processor and its capabilities. General linear algebra is better done with external optimized libraries, writing the code is too complex and not worth it.

These libraries such as Intel's Math Kernel Library (MKL) and OpenBLAS are written to optimize the vectorization capabilities of modern processors such as AVX-512 and FMA instructions. It also uses the micro-operations concept (processing several instructions at once). The core of these libraries and where all the speed is achieved is written in assembly language which is very low level programming[2] for the average user.

After a lot of time spent understanding the fundamentals of the processor and the matrix multiplication algorithms, we can make a model that establishes the theoretical limits the processor can achieve. The main operation is to perform a matrix multiplication of matrices $C = A \times B$, where the three matrices are of size $N \times N$. This is done for a number of iterations ($TIMES$). The number of operations that the processor has to perform is:

$$N_{ops} = 2 \times N^3 \times TIMES \tag{9}$$

A matrix multiplication operation involves 4 instructions:

1. `load(A)`
2. `load(B)`
3. `multiply(A, B)`
4. `load(C)`

So to be precise, a factor of 4 has to be added to the number of operations.

7.1. Vectorization

Modern processors have a 512-bit registers and are capable of using vectorization to perform operations simultaneously. A single-precision real number (float) is 32-bit, which means the processor can get a factor of $512/32 = 16$ due to vectorization.

7.2. Micro-operations

Although not well understood, a processor can also perform multiple instructions in a single clock-cycle. This concept is related to pipelining and throughput. To simplify, this new factor involved in the theoretical limit can take values ranging from 4 to 8.

7.3. Theoretical CPU Time

Taking into account all of the aforementioned, we can make an estimate of the theoretical time with the following formula:

$$t_1 = \frac{4 \times N_{ops}}{F_{vec} \times G_1 \times F_\mu \times C_1} \tag{10}$$

- F_{vec} is the vectorization factor, which is 16 in most cases.
- F_μ is the micro-ops factor, which can be 4, 6 or 8.
- G is the clock speed (use Hz to get t_1 seconds). C stands for the number of cores.

[2]Low level refers to the level of abstraction, which means how similar the code is to human language.

7.4. GPU Theoretical Time

Like the processor, a graphics card has several cores but the number is much larger. The downside is that it does not have vectorization and other properties. A formula for the theoretical time of a GPU can be written as follows.

$$t_2 = \frac{N_{ops}}{G_2 \times C_2} \tag{11}$$

G_2 stands for the clock speed of the GPU in Hz.

7.5. CPU vs. GPU

The main goal established back in May was to find out the potential speedup of a GPU against a CPU. After understanding and making a rough model of each of the devices, we can fuse them together to estimate the speedup of a GPU against a CPU. The formula refers with subscript 1 to CPU and with 2 for GPU.

$$\frac{t_1}{t_2} = \frac{4 \times G_2 \times C_2}{G_1 \times F_{vec} \times F_{\mu} \times C_1} \tag{12}$$

This formula gives a rough estimate of the speedup achieved using a GPU against a CPU. It is important to mention that the real speedup should be lower because this model does not take into account memory transfer between CPU and GPU.

8. BLIS Library

There are several different modern implementations of BLAS. A few of the most well known are Intel MKL (Math Kernel Library) and OpenBLAS. An improvement of OpenBLAS is BLIS (BLAS-like Library Instantiation Software) and for AMD processors a forge of the latter is AOCL (AMD Optimizing CPU Libraries).

The difference between the original version of BLAS and these modern versions is the optimization of the processor's cache. As mentioned in the previous section, the main idea is to optimize as much as possible the kernel that performs the operations in the registers. This is done by taking advantage of vectorization and micro-operations.

Many programming languages use the previously mentioned libraries by default to perform linear algebra. For example, MATLAB uses MKL and the NumPy package (Python) uses MKL or OpenBLAS (it is usually the latter, but it depends on the installation). As the aim is to get the best performance from the processor, a series of tests have been done using BLIS and AOCL, together with MKL.

With the exception of MKL, the installation process of these libraries is not simple in Windows, so the best option is to use Linux. The reason behind this is that in Linux environments, packages are installed with commands and configuration is almost automatic.

8.1. Installing Linux

This section describes a way of installing Ubuntu (a very popular distribution of Linux) on a Windows machine. You must have a USB drive (with at least 8GB) to do the installation. The first thing is to head to the download page. You can either download the LTS version or the latest one, but the tests were performed with the latter. Another program that is necessary is Rufus, get the Standard version.

Once you have the `.iso` file and Rufus, you can continue the installation process following the steps in this tutorial. Make sure you have enough disk space to make a partition before proceeding. Once you have Ubuntu installed in your machine, get Visual Studio Code to write and run the code.

8.2. BLIS Configuration

After installing and configuring Ubuntu, it is time to install BLIS. Linux uses commands to install packages so most of the installation process is done with the Terminal. To open a new Terminal press the following key combination: `control + alt + T`. Before getting BLIS, install first the GCC compiler and all essential tools by running the following commands:

- `sudo apt update`
- `sudo apt install build-essential`

Press `Y` when asked for confirmation.

Next, follow the instructions mentioned in this link to install BLIS. After downloading the package with the `git clone` command, run the following command: `cd blis`. This is done to have access to the inside packages and continue the installation. In the second step of the above mentioned page, run `./configure auto` instead of any other command specified because it is faster and simpler. The rest of the process is correctly explained in the link. Whenever the system shows you a failure to install a package, try adding `sudo` at the beginning of your command, e.g: `sudo make install`. This allows you to run commands as a superuser.

8.3. Testing with BLIS

To run the tests using BLIS, first install the C/C++ extensions of Visual Studio Code. After this, download and extract the following `.zip` file from GitHub. Open the `sc_blis_test.c` file and compile it with the following key shortcut `control + shift + B`. To run the compiled file, in the VS Code terminal type the following: `./sc_blis_test` and press `enter`. Please note that the compiled file has no extension and cannot be visualized with VS Code.

8.4. Multi-core Benchmarks in Julia

BLIS supports multithreading but the configuration is not simple. An alternative is to use the Julia programming language to test the multi-core performance. Multithreading can be done in two ways:

- Running the parallel version of `gemm`.
- Transforming the loop into a parallel loop, with `gemm` staying in a single thread.

The second option is the one that gives us better results for a large number of iterations which may be due to the parallel version of `gemm` not being optimized to achieve maximum performance.

Installing Julia is simple and fast. The following is a list of instructions to install Julia with MKL and BLIS libraries.

- Open a new terminal with the `control + alt + T` shortcut.
- `pip install jill`

  If the system does not find `pip`, install it with `sudo apt install python3-pip`. In the case that another error appears try the following:

  - `sudo apt install pipx`
  - `pipx install jill`
  - `pipx ensurepath`
- `jill install`
- `julia`
- Press `alt gr + ]` to enter package installation mode in Julia.
- `add MKL`
- `add BLISBLAS`
- `add DelimitedFiles`
- Open VS Code and install the Julia extension.

Upon finishing the installation, single and multi-core tests in Julia can be done with Intel MKL and BLIS. To do so, download the following folder in GitHub. Even though its as simple as Python, Julia is a compiled language. To run the program, open the `.jl` file and point the cursor to the first line of code. Once there, continuously press the following key combination: `shift + enter`. This tells Julia to compile and run every single line of code in succession.

Comments in the code include instructions to change between libraries and number of threads. All of the results are saved in a `.csv` file.

9. MKL Benchmarks

The Intel oneAPI for Visual Studio that is used to run Fortran and also includes the MKL library. To use the library, copy the `.f90` file the GitHub branch to a new solution. Once done, do the following:

- Go to the Solution Explorer and right click on the solution.
- Click on *Properties* and open the Fortran section.
- Select *Libraries* and in *Use Intel Math Kernel Library* select *Sequential (/Qmkl:sequential)*.
- To use OpenMP to perform parallel loops, go to *Language* and in *Process OpenMP Directives* turn it into *Generate Parallel Code (/Qopenmp)*.
- On the top bar, in *Debug*, select from the drop down menu *Configuration Manager* and check that the options are *Release* and *x64*.

The `.f90` file is written so that you can change between running in single core and in multiple cores. Check the fig. 7 to know which option is being selected.

```
!£omp parallel do private(i)
    do i = 1, TIMES
        call sgemm("N", "N", N, N, N, 1e0, A, N, B, N, 0e0, C, N)
    end do
!£omp end parallel do
```

Figure 7: Inner loop that calls the `gemm` algorithm.

OpenMP directives are not comments but begin with an exclamation mark. If there are two exclamation marks, then it is commented so it uses one core. In the other case, when there is only one mark, it runs the parallel loop, dividing the loop into several threads and using the full potential of the processor. In fig. 7 all cores are being used.

9.1. Fortran Results and Conclusions

Using the Fortran code mentioned above, tests were performed for different dimensions, always mantaininng the same number of operations. The graphs listed below were done with the following N series:

- From $N = 4$ to $N = 260$ with a step of 2.
- $N = 276$ to $N = 2484$ with a step of 16.
- From TIMES = 64 to TIMES = 1 with a step of -1. N is calculated solving eq. (10).

Even though the file uploaded to GitHub has a different N series, what matters it the asymptotic behavior. Figure 8 shows the results for very small dimensions. As the figure suggests, for small dimensions the processor is not performing at its best. This might be due to the registers not being full enough, thus not taking advantage of the vectorization. Starting around $N = 400$, results appear to remain constant, which suggests the processor is working at its maximum.

The aforementioned behavior can be seen in fig. 9, which is a comparison between multi-core and single core performance. The increase of time for the larger dimensions is result of TIMES being smaller each time. When TIMES is small, it cannot be divided into several threads so the result is equivalent to doing the test with a single core.

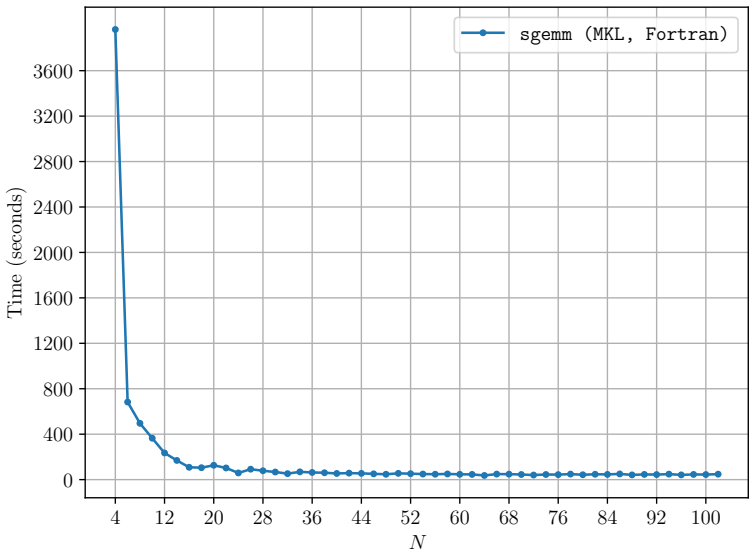

Figure 8: Results for small dimensions.

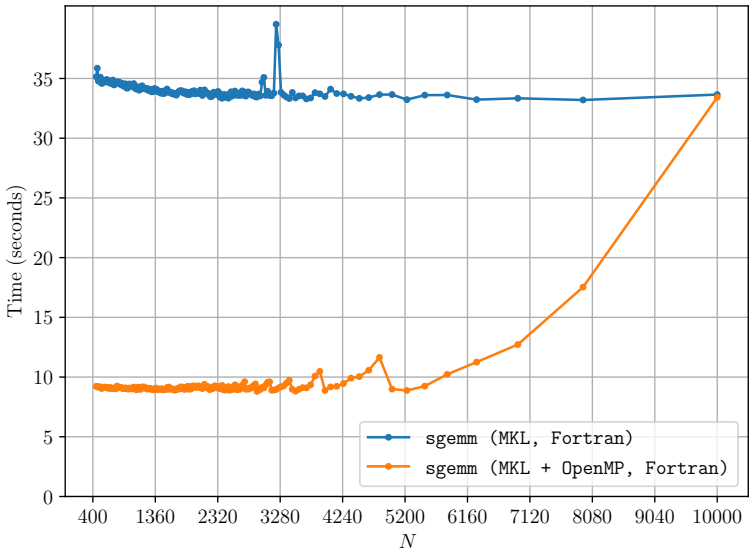

Figure 9: Comparison between single and multi-core performance.

9.1.1. Speedup

To find out the speedup, it is as simple as dividing the single core results by the multi-core results. The results, as shown in fig. 10, are very promising, as almost a factor of 4 (the number of cores) is achieved.

9.1.2. Comments on MATLAB

The same benchmarks were performed in MATLAB but are not relevant due to the calculations being performed in double precision. Changing this setting is somewhat complex as changing the variables to single precision does not reduce the time to perform the operations.

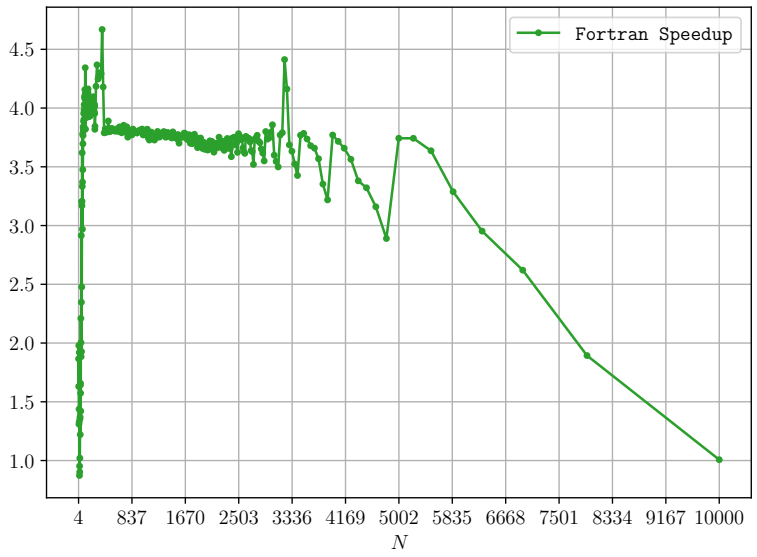

Figure 10: Speedup achieved using parallel OpenMP loops.

10. BLIS Benchmarks

After installing Ubuntu and BLIS and following all the steps mentioned in section 8, tests were carried on in single and multi-core. Results are surprising because the predicted behavior was that MKL and BLIS should be similar but results show the opposite.

The single core test was performed in C, whereas the multi-core was done using Julia. Using different programming languages does not make a difference because BLIS stays the same. As for multithreading, Julia uses a similar system as OpenMP so results should be coherent.

Multi-core tests were performed in Ubuntu using Julia. Configuration was done as specified in section 8, resulting in the following folder uploaded to GitHub. For an easy and quick downlaod, go the branch and press the `Code` button, downloading it as a `.zip`. Once downloaded, extract it and open the `mkl_blis_julia` folder with Visual Studio.

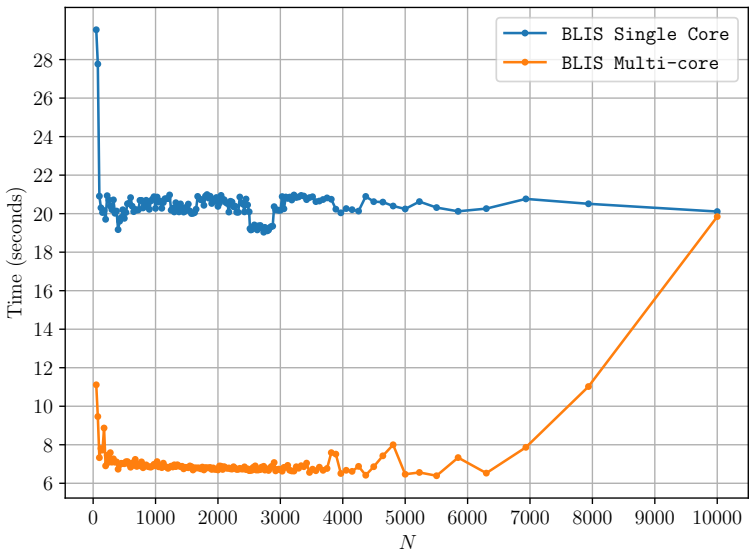

Figure 11: Comparison between single and multi-core using BLIS.

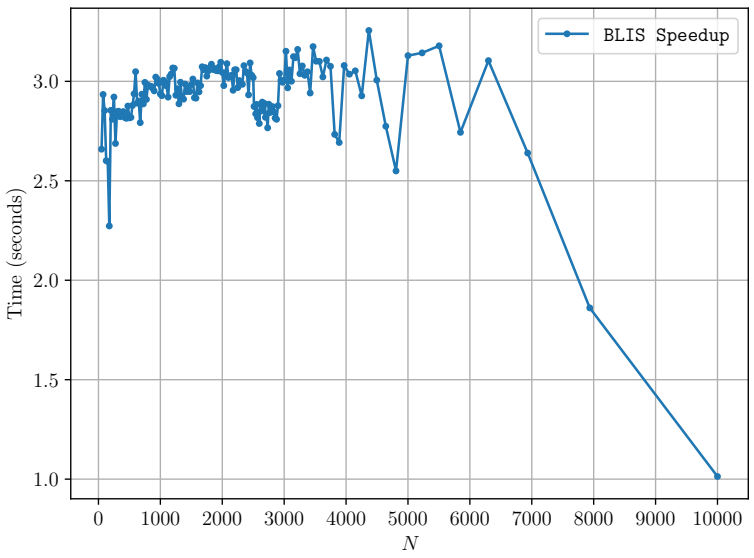

Figure 12: Speedup obtained using BLIS and Julia multithreading.

10.1. Julia and C Results

The speedup achieved of around 3 is not as good as MKL but still very good. This can be due to many different reasons such as computer performance in Ubuntu, multithreading in Julia not being optimized, etc.

Figure 11 shows a similar behavior as fig. 9 but the times are lower. Whereas in Fortran MKL (single core) the asymptotic time is around 34 seconds, in BLIS it is around 20 seconds. In the case of multi-core, Fortran MKL achieves around 9 seconds, whereas BLIS gets to 7 seconds.

This is not the expected result because other tests carried out with different computers show the same behavior in MKL and BLIS. The main hypothesis is related to the Windows operating system. When performing the MKL tests on Windows, the laptop used did not heat up at all. In Ubuntu, after a couple of loops, the laptop got really hot, requiring pauses to cool down and avoid damage to the hardware. To run out of doubts, MKL tests were performed in Ubuntu using the previously mentioned file.

10.2. MKL Results in Julia

The following graphs show the results obtained using Julia to run MKL in Ubuntu. It confirms the theory that MKL and BLIS have a very similar performance regardless of the computer. MKL seems to be a bit quicker in the single core test but for multi-core they are identical.

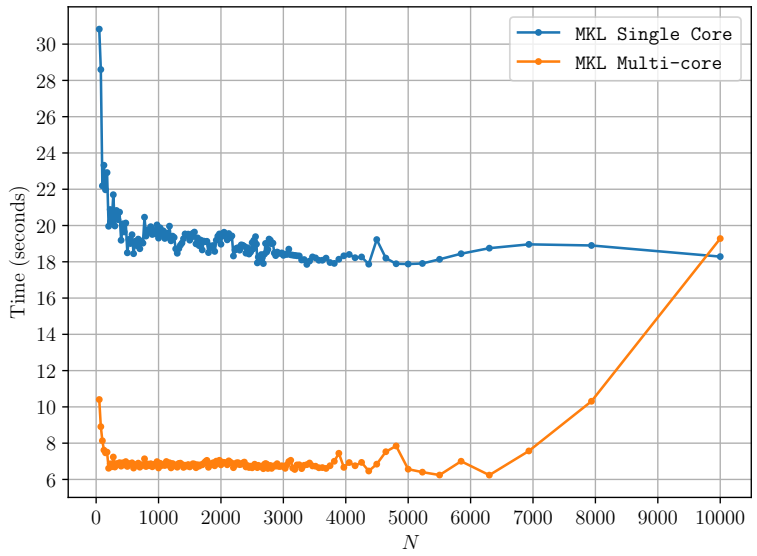

Figure 13: Comparison between single and multi-core using MKL in Julia.

If we look at fig. 13 and compare with fig. 11, we can see that the results are almost the same. All of the tests (BLIS and MKL in Ubuntu) were performed following this series of N:

- From $N = 50$ to $N = 2500$ with a step size of 25.
- From TIMES = 63 to TIMES = 1 with a step of -1. Again as specified before, N is calculated solving eq. (10).

As for the speedup shown in fig. 14, understandably is also similar to the ones achieved with BLIS (see fig. 12). Due to the single core being faster but multi-core staying the same as BLIS, the speedup appears to be a fraction lower but it is not very relevant. To sum things up, one can use BLIS or MKL interchangeably and results should be very similar. For ease of use, MKL included with oneAPI for Visual Studio offers good results when working with desktops. When it comes to laptops it depends on how Windows manages the CPU temperatures.

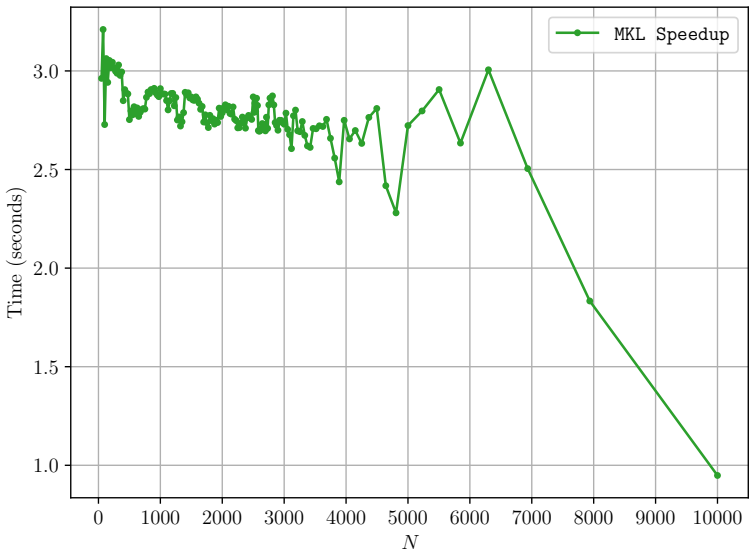

Figure 14: Speedup obtained using MKL and Julia multithreading.