"The best of both worlds : Fast numerical computation in Erlang"

Couplet, Basile ; Brunet, Lylian

ABSTRACT

The paper presents ErlBlas, a library allowing fast numerical computation, and exposing part of the Blas interface in Erlang. ErlBlas uses the Erlang NIFs to call a CBLAS interface, and is able to use the power of multicore systems, via parallel execution of numerical operations.

CITE THIS VERSION

UCLouvain

École polytechnique de Louvain

The best of both worlds

Fast numerical computation in Erlang

Authors: **Lylian Brunet, Basile Couplet**
Supervisor: **Peter Van Roy**
Readers: **Peter Van Roy, Peer Stritzinger, Tom Barbette**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Contents

List of Figures

Chapter 1

Introduction

This paper presents ErlBlas, a library allowing fast numerical computation, and exposing part of the Blas interface in Erlang, using NIFs to call the CBlas interface. Let's start with some context before seeing how ErlBlas solves the problem of fast numerical computation in Erlang.

1.1 Context

Erlang is a programming language historically developed by Ericsson and used in telecommunication. Erlang differs from other general-purpose languages by its very lightweight concurrency, and its possibilities in terms of fault-tolerance and distributed programming. The lightweight concurrency and distributed capabilities are guaranteed by the message-passing between processes and single assignment variables (among other characteristics). [8]

While the power of Erlang resides in its communication and concurrency abilities, the Erlang language lacks the ad hoc data structures and optimizations for fast numerical computations. Writing two simple and straightforward programs for matrix multiplication, respectively in pure Erlang and in C code, and measuring the execution time for increasing sizes of the matrices, we find a speed-up of 10 when running the matrix multiplication on our naive C code. Erlang is therefore non competitive with classical programming languages for any application where numerical computation is used more than marginally. These limitations prevent Erlang to democratize in fields like simulation and data analysis, but also in fields such as machine learning and embedded sensors and data analyser, where the communication and concurrency without race conditions might prove to be very useful.

Fortunately, the Erlang environment includes the "NIFs" abstraction, a simple and fast way to invoke C code directly from function calls in Erlang.[4]

Blas is a well-known set of functions performing the general most-used numerical operations. Originally written in Fortran, the implementations now have

been translated into C, under the name CBlas. We chose to work with the Blas interface (here, CBlas since the NIFs allow us to directly call code written in C), because the implementations of Blas are among the fastest known implementations for solving numerical systems and performing numerical computations. They have been optimized for years and for lots of different architectures. Therefore, calling these implementations instead of implementing them in Erlang, we are nearly guaranteed to obtain the highest speedup possible. [2]

1.2 Contributions

Our contributions sum up in the ErlBlas library, allowing fast numerical computations in Erlang, while keeping its good properties.

1.2.1 CBlas function calls

The first contribution made consists of the extension of the Numerl library, written by Tanguy Losseau [1], into a functional interface from Erlang to CBlas, for the most used CBlas functions. We first added utility functions to translate C matrices to Erlang matrices in list of lists representation, to copy one matrix into a new resource object, and to retrieve the size of the matrix. We also added the functions dscal and daxpy of the Blas interface to the dgemm function already present. Finally, we changed the representation of the matrices from binary objects to resource objects, which allows our Erlang functions to give our C matrices to other NIFs to modify them (see section 2.2.1).

1.2.2 High concurrency properties

In order for the Erlang schedulers to keep their properties, a general rule is that they can't stay idle for more than 1ms. Thus, a NIF call cannot execute for more than 1ms either. [6]

To ensure that our NIF calls don't take more than 1ms to execute, we have introduced a new representation for big matrices, as Erlang matrices whose elements are sub-matrices, stored as resource objects, and whose dimensions cannot exceed a define threshold. This threshold is determined by a benchmark. The benchmark takes the operation with the longest execution time in the CBlas interface, here the dgemm call, and determine the maximum value for the matrix dimensions such that no call to the dgemm NIF given matrices of those dimensions can execute in a time longer than 1ms.

Once no NIF call can execute in more than 1ms, and that the properties of the schedulers are preserved, we need to define the operations on our new representation (as Erlang matrices of C sub-matrices). We thus implemented a series of classical operations, such as add, mult and inversion, on our representation, using the known algorithms for block matrix operations [15].

1.2.3 Exposing Blas interface

But this library wouldn't be called ErlBlas if we didn't give the possibility to the user to call the Blas functions on our matrix representation. We thus implemented the dgemm, dscal and daxpy functions of CBlas, that can be performed directly on our matrices. Computations are done in-place, meaning that the variable pointing to the matrix is the same as before the call, but the values in the matrix will have changed. The user has to be aware that this breaks the single-assignment rule of Erlang, and that race conditions can easily appear if calls to these functions are done concurrently.

1.2.4 Parallelism

Finally, we made all the operations above (listed in the interface in appendix) use concurrent processes during their execution. This means that, if several cores are available on the machine running the operation, the processes will be automatically distributed on the corresponding schedulers, and executed in parallel.

The usage of block-matrix operations is particularly indicated to make the operations parallel, as they are already cut into small pieces. However, in order to obtain results, we had to separate all the memory allocations from the computations, since Erlang executes all the NIFs that perform a memory allocation on the same scheduler, and therefore makes it impossible to parallelize our operations if the NIFs are not in-place functions.

Together, the block-matrix representation of the large matrices, and the separation of the allocation and the computations, allow us to exploit the performance of multicore systems.

The section 6.4 shows the speedup of the concurrent (parallel) functions regarding their sequential counterparts.

1.3 Roadmap

The second chapter of this Master's thesis will expose the background of ErlBlas, that is, a quick description of Erlang and the NIFs, and the Blas interface. The third chapter will present the interface between Erlang and the CBlas functions (written in C), as well as the utility C code used. The fourth chapter will then describe the Erlang code performing the block matrix operations. Finally, the fifth and sixth chapters will respectively present the evaluation of the correctness and of the performance of our implementation.

Chapter 2

Background

2.1 Erlang

Erlang is a programming language that is mainly used in the telecommunication and networking fields, due to some characteristics that differ from the most popular programming languages, giving Erlang interesting properties.[7] [8] Among those characteristics, we can mention the following.

2.1.1 Erlang Characteristics

Concurrent processes

Erlang allows programs to spawn a very high number of different processes that run their own code with their own local variables. The maximum number of such processes is set to 32k by default but can be extended 256M. [9]

These processes are managed by the schedulers of the Erlang VM. These schedulers are assigned processes when the latter are spawned, and can redistribute the processes if one of the schedulers has more of them running than the others. By default, one scheduler is used per processor's core.[11]

Single assignment variables

Erlang uses single assignment, meaning that the value of a variable can never be changed after its first assignment. If a programmer wants to write a new value, they have to declare a new variable to assign the value to it. The variables are dynamically typed, and a variable must start with an upper case letter.[10]

Message-passing

The processes in Erlang communicate via message-passing. Each process is identified by its PID (Process IDentifier). Any process knowing the value of a PID can send messages to the process it identifies, using the keyword "!". The message can consist of any Erlang value, including tuples and lists of other

values. Each process has its dedicated "mailbox", where all the messages sent using its PID are kept until they are retrieved by the process. The messages are treated using the *receive* block. The *receive* block takes a series of patterns and the corresponding actions, and uses pattern matching to retrieve and treat one message in the mailbox. The message retrieved is the first one in the mailbox that matches the first pattern, or one of the following patterns, if no message matches any of the previous patterns.[9]

Here is an example of such a *receive* block.

```
receive
    doNothing ->
        skip;
    {add, X, Y, ReturnPID} ->
        ReturnPID ! {result, X+Y}
end
```

In this example, the receive block will scan the mailbox of the process executing it. It will first look for a message whose value only consists of the *doNothing* atom. If there is any *doNothing* message in the mailbox, the receive is guaranteed to skip even if the is an *add* message in the mailbox, since this is the first pattern of the *receive* block. If no *doNothing* message is present in the mailbox, the receive will search for an *add* message. The pattern is interpreted as such : we are looking for a message that is a tuple containing four values, and whose first value is the *add* atom.
If such a message is found in the mailbox (while no *doNothing* message has been found), the *receive* block will execute the code after the arrow, with the variable X, Y and $ReturnPID$ bound to whatever values that are in the second, third and fourth positions in the tuple message. Of course, we expect X and Y to be numbers, and $ReturnPID$ to be a process identifier, but the message will be retrieved, and the variables bound even if it is not the case. The code following the arrow will send a message whose value will be a tuple containing the *result* atom and the result of the addition of X and Y, to the process identified by $ReturnPID$. Any message that doesn't match any of the patterns will simply be ignored. If the mailbox only contains such non matching messages, or if it contains no message at all, the *receive* will wait until a message matching one of the pattern arrives in the mailbox.

Process registration

Instead of communicating with PIDs, a particular process can be registered with a name, meaning that the assigned name and the PID of the process are bound. These are the only global variables allowed in Erlang. A process can thus be spawned without knowing the identifier of the process it has to send message to, but only knowing the name of this process. The PID attached to a name can be unregistered, then changed, but there can only be one at a time.[12]

Process monitoring

It is also possible for a process to monitor another process, given its PID or name. The monitor will receive a particular message whenever the monitored process crashes or ends. It is also possible for a process to send messages to its monitor(s), without knowing its name or PID. This mechanism easily allows to restart a node that has crashed.[12]

2.1.2 Erlang Properties

Following these characteristics, we can infer some key properties of the language.

- The first one is the high concurrency possibilities. The little overhead of the processes generation allows us to spawn as many unit of work as we want in different processes. The use of single assignment, processes on different schedulers and message passing with passing by value allows multi-threading without race conditions. Along with the little overhead, it provides a possibly unlimited scalability to Erlang programs.
- In addition to this, Erlang allows to give names to different systems running it, possibly on different machines, making them nodes, and with these names allows the message passing between threads of different nodes.
- Given the monitoring possibilities, the lack of race conditions, and the naming of processes, Erlang is designed for fault-tolerance. A particular process' crash doesn't imply the whole system crash. Instead, the monitor of this process can just restart it and giving it the same name. The behaviour of the new process with the new arriving messages is most likely to be the same as if the node had never crashed.
- The ability to run on lots of different communicating machines, and the fault-tolerance explains why the language is particularly indicated to build distributed, fault-tolerant applications, that can include real-time requirements.

2.2 Erlang NIFs

NIF stands for *Native Implemented Function*. The NIFs are an abstraction in Erlang that allows the programmer to call functions in Erlang, with Erlang arguments, and to execute it in C, mostly for performance reasons. The C code can retrieve the arguments given, in C data types variables, with the help of a couple functions available in the *erl_nif* C library. Other functions in *erl_nif* allow the programmer to make Erlang structures and values out of C data types, allocate and return Erlang values, as well as other interactions with the Erlang environment. The C functions are loaded into an Erlang module at runtime.[4]

2.2.1 Resource objects

The *erl_nif* library provides a simple construction to pass C structures to Erlang without doing any translation into Erlang : the resource objects. The resource object is a pointer to a part of the memory. It is opaque to Erlang and, when returned to the Erlang code, the program can only pass it to another NIF, that will be able to retrieve the C structure unmodified. These objects allow to break the single-assignment rule of Erlang : when an Erlang object is a resource object, a NIF called on this object can modify its content, without having to create a new Erlang object. This mechanism allows to write in-place operations in Erlang.[6]

2.2.2 Dirty NIFs

When a NIF call is performed by a process, the scheduler that is managing it interrupts to execute the NIF until the function returns. Therefore, in order to guarantee the good properties of Erlang in terms of concurrency, such that no process is starving, etc., the total duration of execution of a NIFs call has to be less than 1 ms. A native function that fails to terminate within 1 ms is called a dirty NIF. Such dirty NIFs cannot execute on the usual Erlang schedulers. That is why they have to be marked as dirty, and will be executed on an independent set of schedulers, called dirty schedulers. In general, a programmer should avoid to create dirty NIFs, and always try to write its programs such that every call to a NIF can execute in less than 1 ms, for performance and safety reasons.[6]

2.3 Blas

BLAS stands for *Basic Linear Algebra Subproblems*. It consists of a set of routines to perform general linear algebra computations. These routines are divided in 3 levels [2] [3].

2.3.1 Level 1 Blas operations

The first level of routines has been published in 1979, and perform computations using only vectors, such as vector-vector addition, or vector dot product. The typical level 1 operation given as example is "axpy", and performs the general vector-vector addition : given x and y, two vectors, and α, a scalar, put in y the result of the addition of x multiplied by α, and y :

$y \leftarrow \alpha * x + y$.

2.3.2 Level 2 Blas operations

The second level of routines has been developed from 1984 to 1986, and performs matrix-vector computations such as matrix-vector multiplication, or triangular matrix problem solving. The typical level 2 operation given as example is "gemv", and performs the general matrix-vector multiplication : given A, a

matrix, x and y, two vectors, and α and β, two scalars, put in y the result of the addition of two terms. The first term is the result of the multiplication of A and x, multiplied by α. The second term is the vector y, multiplied by β :

$y \leftarrow \alpha * A * x + \beta * y$.

2.3.3 Level 3 Blas operations

The third level of routines has been developed from 1987 to 1988, and performs matrix-matrix computations such as matrix-matrix multiplication. The typical level 3 operation given as example is "gemm", and performs the general matrix-matrix multiplication : given A, B and C, three matrices, and α and β, two scalars, put in C the result of the addition of two terms. The first term is the result of the multiplication of A and B, multiplied by α. The second term is the matrix C, multiplied by β :

$C \leftarrow \alpha * A * B + \beta * C$.

Chapter 3

CBlas interface

In this section we will present the interface used between Erlang and CBlas. This interface is based on Tanguy Losseau's project *Numerl* [1] and can be used independently from ErlBlas. However, using this project independently won't offer any speedup for multicore processors and can possibly run NIFs longer than 1ms.

3.1 The matrix structure

The main data structure we use is Matrix. Its definition in C code is given by figure 3.1.

```
//Gives easier access to an ErlangBinary containing a matrix.
typedef struct{
    int n_rows;
    int n_cols;

    //Content of the matrix, in row major format.
    double* content;

    //Erlang binary containing the matrix.
    ErlNifBinary bin;
} Matrix;
```

Figure 3.1: C code definition of structure Matrix

This structure contains two integers representing the two dimensions of the matrix and an array of doubles representing the content of the matrix. Note that the contents are stored in a 1-dimensional array as opposed to the more obvious

2-dimensional array. The contents are stored line per line but all lines follow each other directly. Knowing the two lengths of the matrix we can very easily get an element from its 2D position to its 1D position. We use this because it simply is the structure that CBlas uses for spacial locality reasons.

3.2 Representation in Erlang

The original Numerl project used binary objects for passing C matrix to Erlang. This worked perfectly for the original project but after using it in our implementation of ErlBlas, using in place operations and concurrency, it showed some limitations. When using binary objects, they are not supposed to be modified. They are treated by Erlang as immutable variables and can thus be either copied or passed by reference to other processes. In the case of binary objects, they are copied when they are less than 64 bytes and passed by reference otherwise. Since the matrix structure contains 2 int of 4 bytes for the size and an array of 8 bytes doubles for the contents, if the array contains less than 8 elements it will be copied and be passed by reference if there are 8 or more elements.

Since we want to compute the different operations with multiple processes, binary objects created bugs as some blocks were modified but other smaller blocks weren't. So we decided to switch to resource objects since those are always passed by reference. With resource objects, our in place operations are correct no matter the size of the blocks. This however introduce an issue if we want to develop the application further. Since we only have references, we cannot share matrices between multiples devices as they cannot be directly copied by Erlang when sending them to another process.

3.3 The *enif_get* method

The *enif_get* method is a utility function written by Tanguy Losseau allowing to retrieve all the functions arguments in one call. It return true if the retrieving was successful and false otherwise. If the function returns false, we know we need to throw a badarg exception. The two first arguments of the function are the env and the argv variables needed for the nif. We then give a string format representing the types of the different arguments. We use "i" for an integer, "n" for a number so a float or an integer and "m" for a matrix. For example, if we have one float then two integers and two matrices we would write "niimm". After that, we add for every argument the pointer to the variable which will contain the argument.

3.4 Calling CBlas

We can see in figure 3.2 a typical example of how a CBlas NIF calls the CBlas method with the arguments given by Erlang.

```
//Performs blas_daxpy
//Input: a number, vectors X and Y
//Output: a vector of same dimension then Y, containing alpha X + Y
//-------------------------------------------------------------------------
ERL_NIF_TERM nif_daxpy(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[]){
    Matrix x,y;
    int n;
    double alpha;

    if(!enif_get(env, argv, "nmm", &alpha, &x, &y)){
        return enif_make_badarg(env);
    }

    n = x.n_cols*x.n_rows;

    if(n != y.n_cols*y.n_rows){
        return enif_make_badarg(env);
    }

    cblas_daxpy(n, alpha, x.content, 1, y.content, 1);

    return atom_true;
}
```

Figure 3.2: daxpy implementation in C interface

We first have to retrieve the arguments : we use the string "nmm" to retrieve one number and two matrices.
Then, we call the *daxpy* function of CBlas on the arguments, considering the two matrices as vectors of numbers, whose length is the number of rows of the matrices multiplied by their number of columns.

3.5 Adding new functions

The advantage of the *enif_get* method is that it allows to easily add new NIFs to the project. Since the library is focused around CBlas, we will describe how to add a new CBlas function. First in the C part, inside the new function definition you only need to :

1. Declare the arguments in C
2. Call *enif_get*
3. Call the CBlas function

4. return true (or anything you want, true being our convention)

You then need to add the definition in the *nif_funcs* array at the end of the file to transmit the definition to Erlang with the name and the number of arguments.

```
ErlNifFunc nif_funcs[] = {
    {"matrix", 1, nif_matrix},
    {"get", 3, nif_get},
    {"at", 2, nif_at},
    {"mtfli", 1, nif_mtfli},
    ...
    }
```

Figure 3.3: The nif_funcs array

Finally you need to add the function definition in the Erlang file with the correct number of arguments and export it. The function in Erlang only needs to return *nif_not_loaded*. The nif definition will overwrite it on build.

```
daxpy(_,_,_)->
    nif_not_loaded.
```

Figure 3.4: daxpy implementation in the Erlang interface

Once everything is added, simply recompiling using *rebar3 compile* will apply the changes so you can test them. If you use the project with ErlBlas, you would need to delete the *_build* folder before recompiling and, if needed, change the target branch/repository inside the *rebar.config* file if you make a fork or create a new branch.

```
{deps, [{numerl, {git, "https://github.com/CoupletB/numerl.git",
    {branch, "resource"}}}]}.
```

Figure 3.5: numerl dependency in *rebar.config*

Chapter 4

Large Matrix Representation

4.1 Matrix Representation

Since the main goal of ErlBlas is the implementation of block matrix operations, we opted for a matrix representation already divided into blocks. Formally, the ErlBlas matrices are Erlang matrices (in list of lists representation), whose elements are C matrices, each represented by a resource object (see section 3.2). This allows us to consider the ErlBlas matrices just as numerical Erlang matrices (in list of lists representation), except that the operations on elements are not the simple operations on numbers, like $+$, $-$, or $*$, but are the C interface operations on the C matrices. The implementation of the ErlBlas matrix creation operations and of the block matrix operations are detailed in section 4.5.

Inside Erlang, an ErlBlas matrix would look like this :

```
[[A11, A12, ..., A1n],
 [A21, A22, ..., A2n]
 ...
 [Am1, Am2, ..., Amn]]
```

Figure 4.1: ErlBlas representation of a matrix

With each A being a resource object representing a C matrix as presented in section 3.1

4.1.1 Benchmark

In order to find the maximum length for the dimensions of our sub-matrices, we need to perform a benchmark of the running system to find the dimensions above which a NIF operation on the sub-matrices can exceed 1 ms execution time. The longest NIF operation possible on matrices of given dimensions is the *dgemm* function of the C interface, so this is the function the benchmark is based on.

The benchmark we implemented has been calibrated on a very unstable system (Linux VirtualBox VM) : in general, the system can perform the *dgemm* NIF on two matrices of dimensions 64*64. However, sometimes (rarely), the time needed to perform the *dgemm* NIF on two matrices of dimensions 2*2 exceeds the 1ms limit. The goal of the benchmark we created is to find a limit that is close enough to the general limit (here, 64) and such that the *dgemm* NIF can be performed under 1 ms nearly everytime.
The final implementation of our benchmark always gives a limit fairly close to the real system limit, with some margin, and such that the 1 ms limit is only exceeded nearly once in a hundred (on the unstable VM).

The implementation is divided in two rounds, using the round_one_benchmark and round_two_benchmark functions, respectively. The first round gives a rough limit, and the second round refines it. The implementation is the following :

- round_one_benchmark : takes a number N as argument and gives a rough limit. It runs the *dgemm* NIF on matrices of dimensions N*N. If the execution time is below 1ms, it calls itself recursively, multiplying N by 2. If the execution time exceeds 1ms, it returns N/2.

- round_two_benchmark : takes a number N as argument and refines the limit. It runs the *dgemm* NIF on matrices of dimensions N*N. If the execution time is below 1ms, it calls itself recursively, multiplying N by 1.2. If the execution time exceeds 1ms, it returns N/1.2.

- benchmark function : returns the limit based on the benchmark. It first takes the minimal value of five executions of the round_one_benchmark function as a rough limit. It then runs 20 times the round_two_benchmark function to refine the limit, and take the second smallest value as final limit, to avoid up to one outsider, since the system is unstable.

4.2 Memory allocation and concurrency

In the functions that need memory allocation, we separated the nif doing the allocation of the new matrix and the computation of the result. There are two main reasons for this.

The first is that allocating memory takes a bit of time, even more if we need to copy a matrix, and since we only have a limited time to execute a nif separating the allocation and the computation can allow to treat larger blocks.

The second is that parallel memory allocation is a difficult problem. Knowing that, Erlang execute all NIFs doing memory allocation on the same scheduler. If we do a memory allocation and a computation in all processes all the computations will also execute on one scheduler. This prevents concurrent computation to be done and thus having a multicore implementation becomes useless.

4.3 Single assignment versus In-place functions

Another goal of ErlBlas was to essentially use CBlas in the NIFs, and partially expose the Blas interface in Erlang. However, no NIF can allocate memory and perform calculation in the same call, since all the NIFs that allocate memory on their code (using malloc) are executed on the same scheduler, and we want the ability to run the NIFs on several schedulers to exploit the possibilities of multicore systems. Also, the Blas interface only consists of in-place operations, considering that the allocation has been done somewhere else. Thus, the Blas functions exposed in Erlang perform in-place calculations, modifying the content of the resource objects contained in the ErlBlas matrices (see section 4.1). Yet, doing in-place operations in Erlang breaks the single-assignment rule, and thus the guarantee of the good properties of the Erlang language (see section 2.1).

This is why we kept, next to the Blas interface, standard matrix operations such as add, mult, etc., that respect the single-assignment rule. These functions usually start by creating a copy of (one of) the matrix, then storing the result in the newly created matrix using the in-place operations (usually from the Blas interface).

4.4 Termination checking

Since the computations are done in place, the processes doing the computations don't need to send any data back. However, the main process still needs to know when all computations are done to be able to continue. The processes doing the computations needs to send a message to the main process when they are all done computing. If all processes send a message to the same process this will create a bottleneck an slow down the program.

In order to reduce the bottleneck we develop a termination checking method to use multiple processes to receive the termination messages. We spawn one process per line of the block matrix which spawns the processes doing the computation for each block. Once a process is done computing, it sends a message to the line process which spawned it. Once the line process has received the termination message from all of its spawned processes, the line process sends a termination message to the main process. Once the main process has received a message from all of the lines, it knows all processes are done computing and can end the function call. Doing it this way allows to use multiple cores to check termination and reduce the bottleneck.

For example, if we have a block matrix of 20x20 blocks, We have 400 processes doing computations. With our method we have 20 processes receiving 20 messages instead of 1 process receiving 400 messages. Since the complexity of finding a message in the mailbox is $O(N)$ and we need to find N messages we have a complexity of $O(N^2)$ to get all messages. Having a huge mailbox is detrimental to the execution time so having multiple small mailboxes is already better in single core and we get the advantage of multicore processing at the same time.

4.5 Implementation

We describe in this section some implementation details worth mentioning for each function, if any.

4.5.1 Matrix creation

As explained in section 2.1, the ErlBlas matrices are represented by an Erlang matrix (a list of lists) of C sub-matrices, represented by resource objects. Neither dimension of these sub-matrices can exceed the *MAX_LENGTH*, determined by the benchmark (see section 2.2). The matrix creation functions are implemented such that a maximum of the sub-matrices have a size of *MAX_LENGTH*MAX_LENGTH*. In the case where N or M (respectively the number of rows and the number of columns of the matrix we want to create) are not multiples of *MAX_LENGTH*, the sub-matrices of the first row and/or the first column of the ErlBlas matrix will have a size that differ from *MAX_LENGTH*MAX_LENGTH*. We can see an example in Figure 4.2 : In case N is not a multiple of *MAX_LENGTH*, the sub-matrices in the first row of the ErlBlas matrix will have a number of rows of $N\%MAX\_LENGTH$; in case M is not a multiple of *MAX_LENGTH*, the sub-matrices in the first column of the ErlBlas matrix will have a number of columns of $M\%MAX\_LENGTH$.

There are three matrix creation functions : *zeros*, that creates a zero-filled ErlBlas matrix; *eye*, that creates a square ErlBlas matrix with ones on the main diagonal, and zeros at every other position; and *matrix*, that takes an Erlang matrix as argument (in list of lists representation), and creates an ErlBlas matrix whose elements are the same as the ones in the Erlang matrix.

Those three functions are recursive : the base case is when N and M are both less than or equal to *MAX_LENGTH*. The creation function thus calls the homonymous NIF, and returns the resource object as the only element of an ErlBlas matrix. For the other cases, the function splits the parameters to obtain multiples of *MAX_LENGTH*, and calls itself recursively on the new arguments. If N and M are already multiples of *MAX_LENGTH*, the function splits the parameters into *MAX_LENGTH* and others (smaller) multiples of *MAX_LENGTH*, and calls itself recursively on the new arguments.

Figure 4.2: Matrix creation : dimensions of sub-matrices

The figure 4.3 shows the code for the creation of zero-filled matrices. ModN and ModM are respectively the number of rows and the number of columns in the upper-left sub-matrix. RowMultiple and ColMultiple are respectively the number of rows and columns in the ErlBlas matrix, minus one (when excluding the first row and column).
We can see the four cases :

- First, the base case : when N and M are less than the maximum length of a sub-matrix. this correspond to have only the upper-left sub-matrix, which we create by calling the corresponding function of the C interface.
- Second case : when M is greater than MAX_LENGTH, but N is not. This corresponds to only have one row of sub-matrices. We call the corresponding function of the C interface with the dimensions of the upper-left sub-matrix, and recursively call the function with the dimensions of the rest of the array, then combine the two matrices.
- Third case : when N is greater than MAX_LENGTH, but M is not. Same as second case, but with a single column.
- Fourth case : when both M and N are greater than MAX_LENGTH. This is the general case. We start by calling the corresponding function of the C interface to create the upper-left matrix, then we recursively call

```
zeros(N,M) ->

    %declare ModN, ModM, RowMultiple, ColMultiple

    if
        N =< MAX_LENGTH, M =< MAX_LENGTH ->
            [[numerl:zeros(N, M)]];
        N =< MAX_LENGTH, M > MAX_LENGTH ->
            A = zeros(N, ModM),
            B = zeros(N, ColMultiple * MAX_LENGTH),
            appendEach(A, B);
        N > MAX_LENGTH, M =< MAX_LENGTH ->
            A = zeros(ModN, M),
            C = zeros(RowMultiple * MAX_LENGTH, M),
            lists:append(A, C);
        N > MAX_LENGTH, M > MAX_LENGTH ->
            A = zeros(ModN, ModM),
            B = zeros(ModN, ColMultiple * MAX_LENGTH),
            C = zeros(RowMultiple * MAX_LENGTH, ModM),
            D = zeros(RowMultiple* MAX_LENGTH, ColMultiple * MAX_LENGTH),
            recompose4(A,B,C,D)
    end
```

Figure 4.3: Zero-filled matrices creation

the function three times : one the dimensions of the first row and one on the dimensions of the first column of sub-matrices, and the last on the dimensions of the ErlBlas matrix, when excluding the first row and first column. Finally, we combine the four matrices into one.

4.5.2 Generic functions and block-matrix operations

To implement the block-matrix operations, we defined in the *utils* module two generic functions: *matrix_operation*, and *element_wise_op*.

The *matrix_operation* function takes a matrix and a unitary operator, and returns the matrix whose elements are the results of applying the operator on the corresponding elements of the matrix given as argument.

The *element_wise_op* function is a function from the *mat.erl* library [13]. The function takes two matrices and a binary operator, and returns the matrix whose elements are the results of applying the operator on the corresponding elements of the matrices given as arguments, two by two. This function is implemented and used in two flavors : the classical one, from *mat.erl*, returns a new matrix. The second takes an in-place operator, and returns "ok" when the operator has been applied on all the elements of the matrix that must be modified.

We can see on figure 4.4 the code of these two functions. The *matrix_operation* function uses two nested maps to go through the elements of the matrix. The *matrix_operation* function uses the *zipwith* function of the lists library. This function takes as argument a binary operator and two lists, and returns the list whose elements are the results of applying the operator to the elements of the two lists given as arguments, two by two.

To give an example of how we used these function to perform the block-matrix operations, we can look at the *daxpy* function.

4.5.3 daxpy implementation

We can see the code of the function on figure 4.5. This function uses the in-place version of *element_wise_op*. The lambda function given takes the two blocks and calls *numerl:daxpy* with the argument *Alpha* (a scalar) and those two blocks. The user can either call directly the function *daxpy*, or call the *add* and *sub* functions, that respect the single-assignment rule of Erlang, and copy the content of the matrix to be modified before calling daxpy, with *Alpha* set at 1 or -1 for *add* and *sub*, respectively.

4.5.4 copy and copy_shape implementation

These two functions use *matrix_operation*. In this case it applies the functions numerl:copy and numerl:copy_shape, respectively. The first one allocates a new matrix and copies the content of the given one. The second one allocates a new matrix with the same shape of the given one and fills it with zeros.

```
matrix_operation(Op, M) ->
    lists:map(fun(Row) ->
                lists:map(fun(A) -> Op(A) end, Row)
              end,
              M).
```

```
element_wise_op(Op, M1, M2) ->
    lists:zipwith(fun(L1, L2) ->
                      lists:zipwith(fun(E1, E2) ->
                                        Op(E1, E2)
                                    end,
                                    L1,
                                    L2)
                  end,
                  M1,
                  M2).
```

Figure 4.4: Erlang code of the *matrix_operation* and *element_wise_op* generic functions

```
daxpy(Alpha, X, Y) ->
    utils:element_wise_op_ip(fun(A, B) ->
                                 numerl:daxpy(Alpha, A, B)
                             end,
                             X,
                             Y).
```

Figure 4.5: The function *daxpy* of ErlBlas

4.5.5 dscal implementation

This function also uses the *matrix_operation* generic function and gives as argument a one argument function calling *numerl:dscal* with Alpha and the block given to the lambda function.

4.5.6 Transpose implementation

Again, this function uses *matrix_operation* to call *numerl:transpose*, and transpose the elements of the sub-matrices. But here, transposing the contents of the blocks is not enough : we also need to transpose the blocks themselves. To do that we use the *tr* function from *mat.erl*, a simple numerical computation library written in Erlang, with list of lists representations of matrices. This library comes from another thesis written by Kalbusch Sébastien and Verpoten Vincent [13] [13]. The *tr* function from *mat.erl* transposes an Erlang matrix, in list of lists representation. We thus call *tr* to transpose the blocks, then *numerl:transpose* on each block to transpose the contents.

4.5.7 Equals implementation

The *equals* function uses *element_wise_op*. We call *element_wise_op* with *numerl:equals* to check if the blocks are equals. This returns a boolean matrix. We then go through this boolean matrix using the built-in function *lists:all* which returns true if all the elements in the list is evaluated to true, and false otherwise.

4.5.8 daxpy implementation

This function also uses the in-place version of *element_wise_op*. The lambda function given takes the two blocks and calls *numerl:daxpy* with the argument *Alpha* and those two blocks.

4.5.9 dgemm implementation

This function is more complex and using *element_wise_op* is not enough. In order to compute a matricial multiplication on block matrices, we perform do a matricial multiplication on the block matrices the same way as if there were numbers instead of blocks and then use the *dgemm* NIF of our C interface on the blocks that need to be multiplied together. Dgemm also has transposition arguments which needs to be taken care of since, as mentionned previously, transposing the inside of the blocks is not enough, we also need to transpose the blocks themselves. The first step in the dgemm function is to transpose matrix M1 and M2 if needed. We only call *tr* to transpose the blocks and not *transpose* to do a full transposition. We will use the transposition arguments from the *dgemm* function of the interface in order to treat the transposition of the contents of the blocks. However, for the second matrix we will do the opposite of what the the argument says. We transpose the blocks if *BTransp* is

false and don't if it is true. This is to more easily match elements of the line of A and the column of B. This way going through lines of each matrix is like going through the lines of A and the columns of B.

Since we are decomposing the dgemm operation, we are building the result of the blocks of C progressively. Because of that we need to manage the multiplicative argument of C *Beta* outside of the C calls to dgemm. The second step is thus to call *dscal* on C with constant *Beta* if *Beta* is different than 1.

Now we need to start the computations in themselves. The principle is that we will start a new process for each block of C. For each of these blocks we sequentially compute the different dgemm operations with the blocks of the corresponding lines of A and B. This will add the result of each multiplication in the block of C and we will have the final result for that block once the sequential operations are done. The different dgemm operations for each block need to be sequential because otherwise we would have race conditions on the addition in C.

Finally we check for the termination as explained in section 4.4.

4.5.10 Invert implementation

The method to invert a block matrix can be a bit tricky to understand and describing the mathematics behind that is not part of this thesis. We will just show the formula without any mathematical proof of its correctness.

First we will take a look at the general formula for inverting a matrix using blocks. [14]

$$\mathbf{P} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \\ -\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & \left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \end{bmatrix},$$

Figure 4.6: Wikipedia's formula for block matrix inversion

In order for this formula to work, A and D must be squares. Fortunately to invert a matrix, the matrix must be of square shape and using our representation the upper left block and the lower right one are always squares when $N = M$.

In general our block matrices have more than 4 blocks. Recursion is thus involved. Since our representation uses a list of list, we will use the formula recursively until the inversion is called on a single block where we now call the nif function to invert a matrix. Unfortunately, sometimes a matrix can be invertible but its upper left block could not be. This is a problem that could be addressed in some future work just be aware that this function can currently fail in some situations.

As you can see in the formula, there are a couple of operations that are used multiple times such as A^{-1} and $(D - CA^{-1} - B)^{-1}$. In order to avoid recomputing them those blocks be will stored and then reused when necessary. The trade off is that this will improve the performances but will require more memory.

Chapter 5

Correctness test suite

For all the tests that need a particular value for *MAX_LENGTH*, the current value of the parameter is stored at the beginning of the test, then changed, and restored at its previous value when the test is finished.

5.1 General approach

For each of the operation functions of the interface (see appendix B), we create a file which will contain all the related tests. We call these files *name_test_SUITE* where name is the name of the tested function.

5.1.1 Deterministic tests

Each file contains 3 tests for which *MAX_LENGTH* is set to 5, with hand-written matrices. Both the argument matrices and the result matrix are written in Erlang, with chosen values, then translated into the ErlBlas representation. Finally, the result of the operation on the argument matrices is compared to the result matrix. Those tests suppose that the *matrix* function of ErlBlas is working accordingly to its specifications.

The first test is usually done on matrices of dimension $1 * 1$, to verify that the basic operation works. The second test is done on matrices containing four sub-matrices of maximum size, in order to test the implementation of the block-matrix operation on a simple case too. The third test is similar to the second one, except it is done on smaller matrices, and with float values instead of integers, to test the possible differences between the two primitive types.

Figures 5.1 and 5.2 respectively show the code for the first and third such tests for the *add* function of ErlBlas. The first test only performs the addition between two $1 * 1$ matrices, respectively containing 1 and 2 as only elements, and check that the result is the $1 * 1$ matrix containing 3 as only element. The third test performs an addition on two $2 * 6$ matrices containing float values.

```
base_test() ->
    A = [[1]],
    B = [[2]],
    C = [[3]],
    ABlock = erlBlas:matrix(A),
    BBlock = erlBlas:matrix(B),
    CBlock = erlBlas:matrix(C),
    Res = erlBlas:add(ABlock, BBlock),
    ?assert(erlBlas:equals(Res, CBlock)).
```

Figure 5.1: Deterministic basic test for the ErlBlas *add* function

```
float_test() ->
    A = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42, 9.58, 10.013, 69.42]],

    B = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42, 9.58, 10.013, 69.42]],

    ABlock = erlBlas:matrix(A),
    BBlock = erlBlas:matrix(B),
    Res = erlBlas:add(ABlock, BBlock),
    ANum = numerl:matrix(A),
    BNum = numerl:matrix(B),
    Conf = numerl:add(ANum, BNum),
    Expected = numerl:mtfl(Conf),
    Actual = erlBlas:toErl(Res),
    ?assert(mat:'=='(Expected, Actual)).
```

Figure 5.2: Deterministic float-elements test for the ErlBlas *add* function

```
small_random_test() ->
    erlBlas:set_max_length(50),
    Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
    {timeout, 100, fun() -> matrix_test_core(Sizes) end}.

corner_cases_test_() ->
    erlBlas:set_max_length(50),
    Sizes = [49, 50, 51, 99, 100, 101],
    {timeout, 100, fun() -> matrix_test_core(Sizes) end}.

random_test() ->
    erlBlas:set_max_length(50),
    Sizes = [rand:uniform(800) + 200, rand:uniform(800) + 200, rand:uniform(800) + 200],
    {timeout, 100, fun() -> matrix_test_core(Sizes) end}.
```

Figure 5.3: Random tests for the ErlBlas operation functions

5.1.2 Random tests

Each file also contains 3 tests for which *MAX_LENGTH* is set to 50, with random-generated matrices, meaning the elements of these matrices are integers randomly picked between 1 and 1000. For each of these tests, we give a set of sizes. The test will then generate matrices whose dimensions will be all possible combinations of the sizes given as argument, and check the result of the operation against the result of the C interface corresponding operation.

The three tests only differ by the given set of sizes, and each test different cases. The first test performs the operation on matrices containing only a few blocks, to verify that the second deterministic test can be repeated for different values, and different number of blocks. Thus, for the first test, three sizes randomly picked between 1 and 10 are given. The second test performs the operation on matrices whose dimensions are corner cases : as the number of block changes when the number of row/column of the ErlBlas matrix becomes greater than the maximum length of a block (here, 50), the sizes given are {49, 50, 51, 99, 100, 101}. The third test performs operations on big matrices, with a large number of blocks. There are three sizes given, randomly picked between 500 and 1300.

Figure 5.3 shows the code for the three random tests. This code is identical in all the test files of the operation functions of ErlBlas. The code set the maximum length for the blocks' dimensions at 50, and calls the *matrix_test_core* function on the chosen sizes.

Figure 5.4 shows the *matrix_test_core* function for the tests of the operations

```
matrix_test_core(Sizes) ->
    lists:map(
        fun(M) ->
            lists:map(
                fun(N) ->
                    random_test_core(M, N)
                end,
                Sizes
            )
        end,
        Sizes
    ).
```

Figure 5.4: *matrix test core* for operations with two possible dimensions

```
random_test_core(M, N) ->
    A = utils:generateRandMat(M, N),
    B = utils:generateRandMat(M, N),
    ABlock = erlBlas:matrix(A),
    BBlock = erlBlas:matrix(B),
    Res = erlBlas:add(ABlock, BBlock),
    ANum = numerl:matrix(A),
    BNum = numerl:matrix(B),
    Conf = numerl:add(ANum, BNum),
    Expected = numerl:mtfl(Conf),
    Actual = erlBlas:toErl(Res),
    ?assert(mat:'=='(Expected, Actual)).
```

Figure 5.5: *random test core* for the *add* operation

having two possible dimensions for the arguments (for example, there are two possible dimensions for the arguments of the add operation : the number of row and the number of columns of the two matrices). The *matrix_test_core* function consists of two nested *map* functions. As a result, the function *random_test_core* is called with all possible combinations of two dimensions from the sizes given (with repetition). This generates $|Sizes|^2$ combinations of dimensions. If the arguments of the operation can have three possible dimensions (for example, there are three possible dimensions for the arguments of the mult operation : the number of row and the number of columns of the first matrix, and the number of column of the second matrix), there are three nested maps and three dimensions are given to the function *random_test_core*, generating $|Sizes|^3$ combinations of dimensions.

Figure 5.5 shows the code of the function *random_test_core* for the *add* operation of ErlBlas :

- two matrices with random elements are generated with the given dimensions;
- the matrices are translated in both the ErlBlas and the C interface representations;
- the result of the operation is calculated both by the ErlBlas function and the C interface NIF;
- the two results are translated back to Erlang and compared.

It has thus been assumed for these tests that the corresponding NIFs of the C interface were giving results accordingly to their specifications.

5.2 Tests for parallel functions

When the only operations available were sequential, the tests were obviously done on the sequential operations. However, when the concurrent pendants of these operations have been written, the test files have been changed to call these concurrent versions. Testing the concurrent versions on multicore machines with the same tests as for the sequential versions gives fair guarantee that the parallel execution doesn't affect the results of the operations.

However, testing the concurrent versions on the same tests than their sequential pendants were not enough, as the number of available cores could also change the behaviour of the functions. We thus run those tests on different numbers of schedulers (1,2,4,6), by modifying the erlang "+S" argument, to make sure that the number of available cores doesn't change the behaviour of the operations. As the "rebar3 eunit" command doesn't allow to change the arguments, the tests have been run manually from the Erlang CLI.

5.3 Special cases

5.3.1 Matrix Creation

Matrix creation testing is highly dependent on the maximal size of a block. We thus set its value to 5 so that we don't have too big matrices and can test more easily.

To test the correctness in this case, we call the block creation operation on one side and on the other side we call the C interface creation operation on each sub matrix and put them in the intended format. We finally check that the two matrices are equals for the placement, size and content of the blocks.

The creation functions are tested on different sizes, covering the basic and corner cases :

- $1 * 1$ and $2 * 2$ matrices, to test the function for one-block matrices;
- $1 * 10$ and $10 * 1$ matrices, to test the function for several-block, row and column matrices;
- a $10 * 10$ matrix, to test the function for matrices with several blocks on each dimension;
- a $7 * 7$ matrix, to test the function for matrices with several blocks, with some non-full blocks on the first row and column.

5.3.2 Inversion

For the inversion function tests, we also have the six tests from the general approach (section 5.1). Unfortunately, due to what seems to be a bug in the OpenBlas implementation when used on Ubuntu, the inversion NIF crashes if given a matrix with dimension greater than 17. We thus put the MAX_LENGTH parameter to 17 before running the tests. The inverse operation also fails in the case where the upper left block is uninvertible as said in section 4.5.10. There is thus one test case which fails in this module since we specifically test for it.

Chapter 6

Performance evaluation

In terms of performance, we tested our library on different aspects. Different hardware have been used for each test. Each CPU has a different frequency and number of cores. The frequency will affect the maximal size of a block and the number of cores the number of operations that can be done in parallel.

We have divided the performance tests in four parts :

- The first section compares the operations of the CBlas interface to the corresponding operation, implemented in pure Erlang, showing the speedup of using CBlas.
- The second section compares the operations of ErlBlas to the same pure Erlang operations as in the previous section.
- The third section compares the sequential and concurrent operations of ErlBlas, showing the speedup of using parallel execution, tested on different machines.
- The last section evaluates the performance of the benchmark, showing how close the threshold given by the benchmark is to the real limit of 1ms for a dgemm NIF call.

For each graph presented hereafter, a value N on the "Size" axis means that the computations have been performed on square matrices of dimensions $N * N$, thus containing N^2 elements.
All the values in the y axis (time or ratio) are mean values of 40 times the same execution (20 times on the Raspberry pi).

6.1 Hardware description

We used 3 different machines in our test that will be named as follows :

- Desktop : a desktop computer with an AMD Ryzen 5 3600 @ 3.60GHz with a turbo at 4.00GHz when a single core is running, 6 cores and 12 threads running Windows 10 with a Ubuntu 20.04 WSL.
- Laptop : a laptop with an Intel core i3-6006U @ 2.00GHz with 2 cores and 4 threads running Ubuntu 20.04.
- Raspberry pi : a raspberry pi 2 with a 700 MHz 4 cores processor running Raspberry pi OS.

6.2 Comparing pure Erlang implementation to the C interface

In order to compare the pure Erlang code with our C interface, we used the *mat.erl* library [13].

The graphs on figures 6.1 to 6.5 present the results of the comparison for the matrix multiplication. The graphs are organised two by two for each hardware. The first presents the comparison in terms of absolute speed and the second presents the ratio between the time of the pure Erlang implementation and the NIF call. Those are thus the ratio between the speed of execution of the NIF call and the speed of execution of the pure Erlang implementation. We can see that we have a speedup of a several hundreds for big matrices (at least $50 * 50$), for any machine. The drop on the graphs in figures 6.2 and 6.4 is explained by an explosion of the time taken by the NIF call when the matrices exceed exactly $64 * 64$. This explosion is discussed in section 6.5.4. However this, as seen in the next section, this drop doesn't have a big impact on the performances of the ErlBlas operations, thanks to the benchmark.

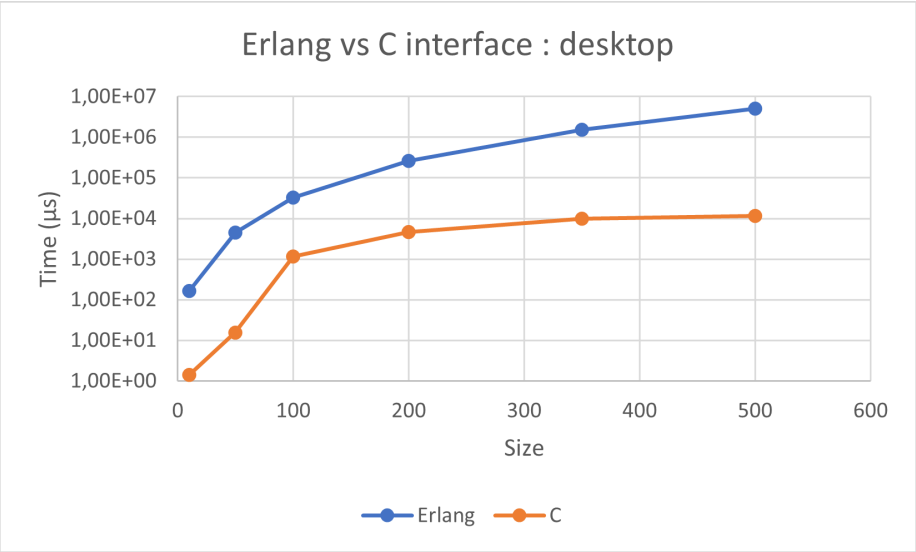

Figure 6.1: Comparing single core pure Erlang to the single core pure C interface on multiplication for desktop

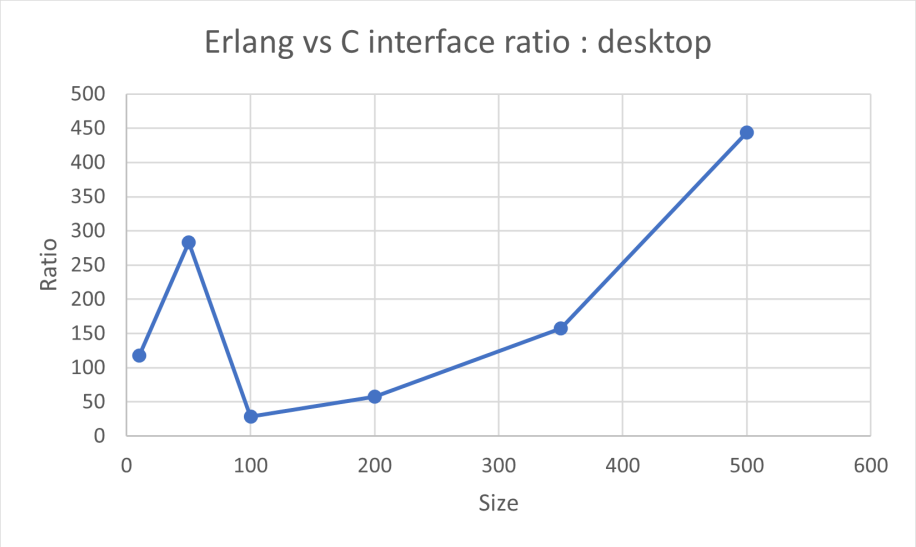

Figure 6.2: Ratio of single core pure Erlang and single core the pure C interface on multiplication for desktop

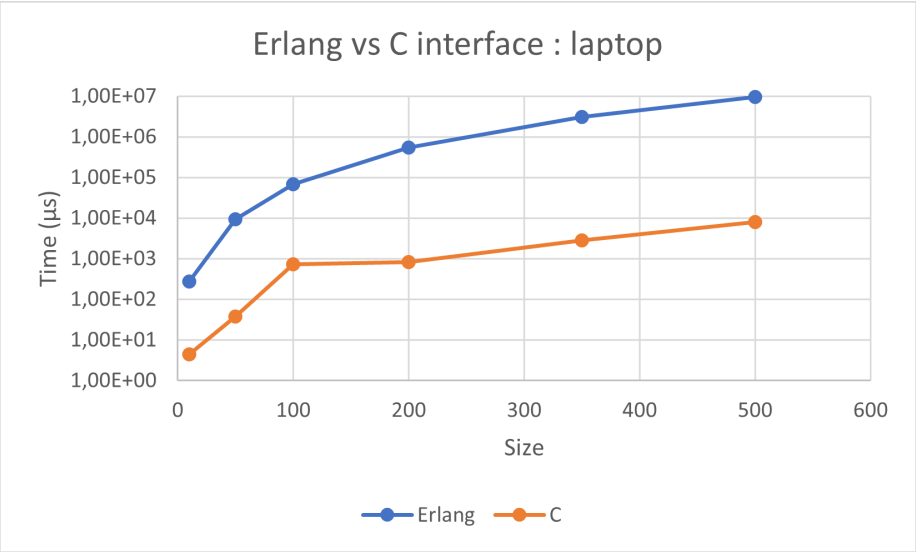

Figure 6.3: Comparing single core pure Erlang to the single core pure C interface on multiplication for laptop

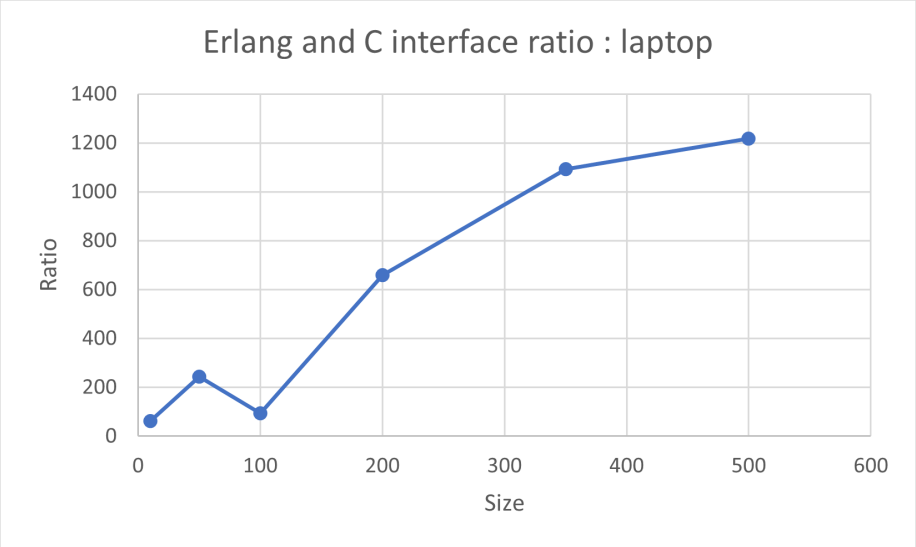

Figure 6.4: Ratio of single core pure Erlang and the single core pure C interface on multiplication for laptop

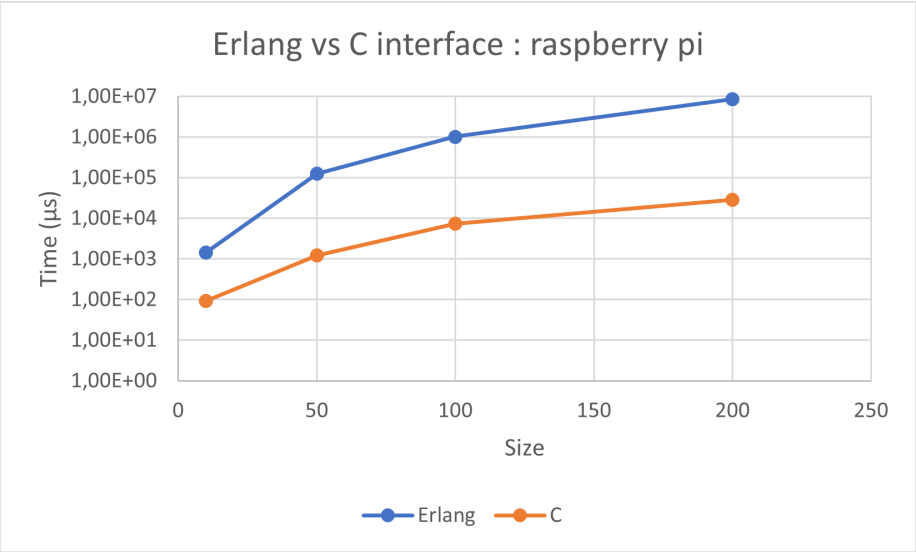

Figure 6.5: Comparing single core pure Erlang to the single core pure C interface on multiplication for raspberry pi

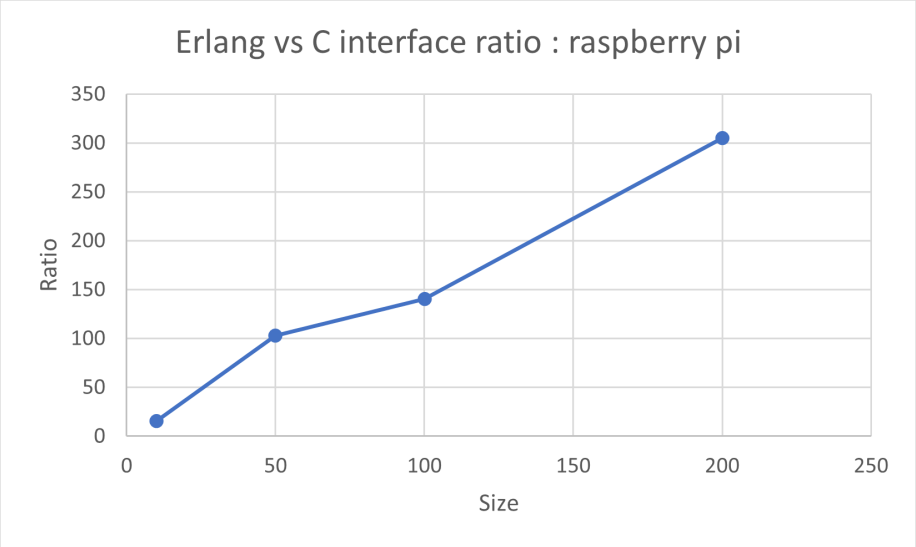

Figure 6.6: Ratio of single core pure Erlang and the single core pure C interface on multiplication for raspberry pi

6.3 Comparing pure Erlang implementation to ErlBlas

Now that we have seen the speedup of the C code regarding a pure Erlang implementation, we have to make sure that our block-matrix representation and computations don't break the speed-up. For this purpose, the pure Erlang has again been compared on the multiplication operation, but this time with the corresponding block matrix operation of ErlBlas.

The figures 6.7 to 6.12 follow the same structure as in the previous section. The reader can still observe a speed-up of several dozens or hundreds for big matrices, depending on the machine.

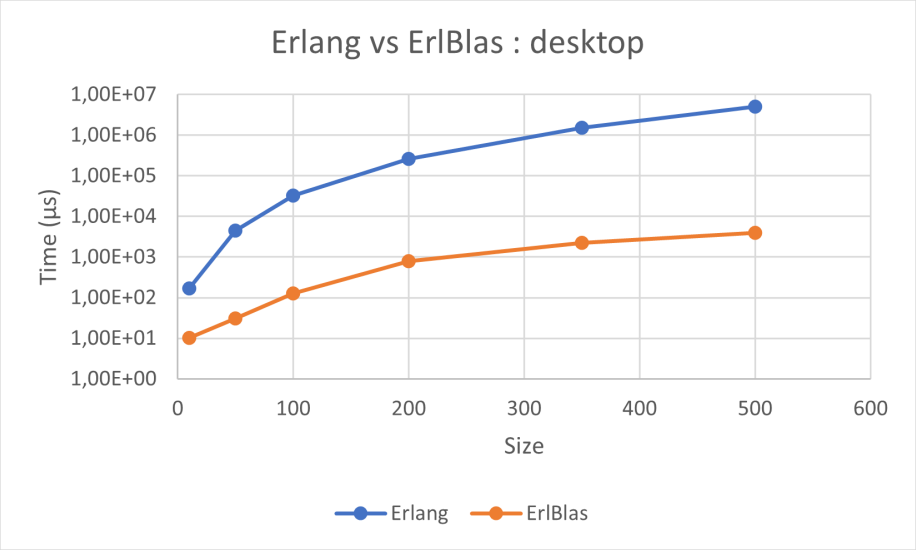

Figure 6.7: Comparing single core pure Erlang to multicore ErlBlas on multiplication for desktop

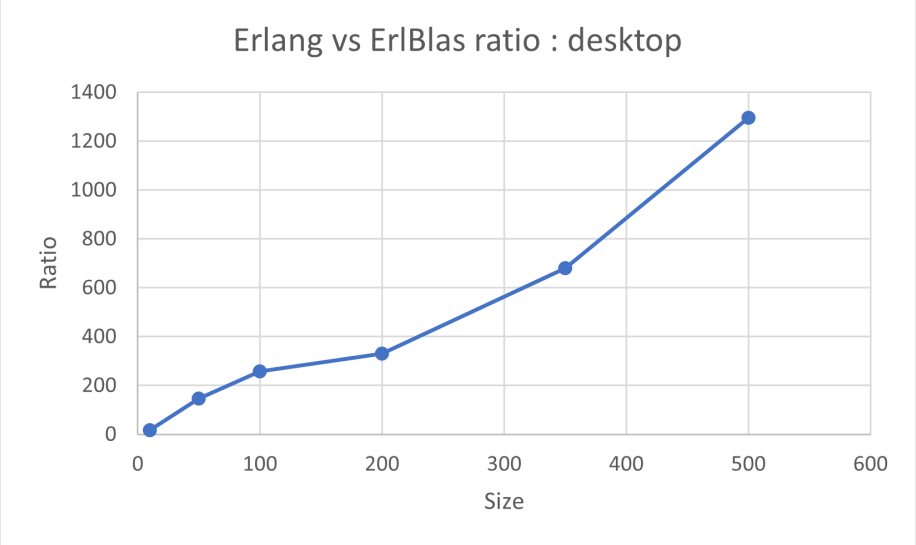

Figure 6.8: Ratio of single core pure Erlang and multicore ErlBlas on multiplication for desktop

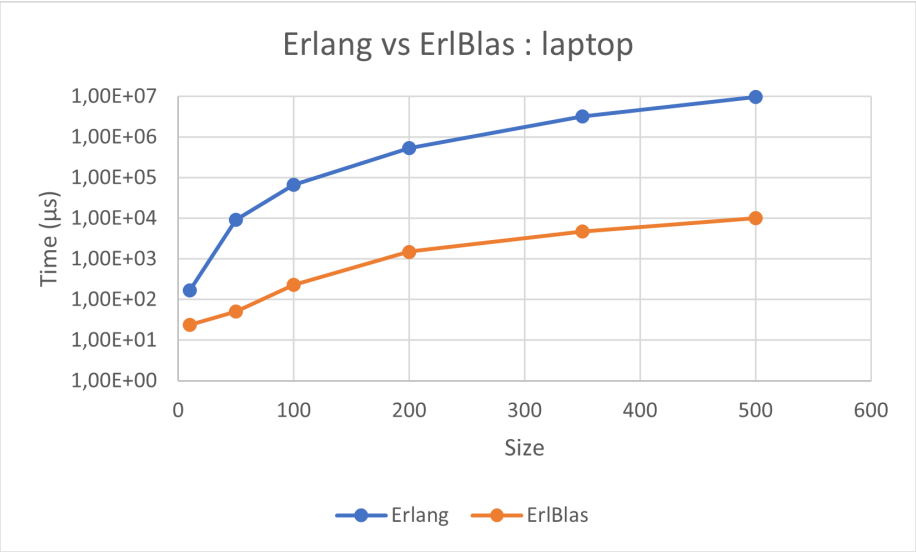

Figure 6.9: Comparing single core pure Erlang to multicore ErlBlas on multiplication for laptop

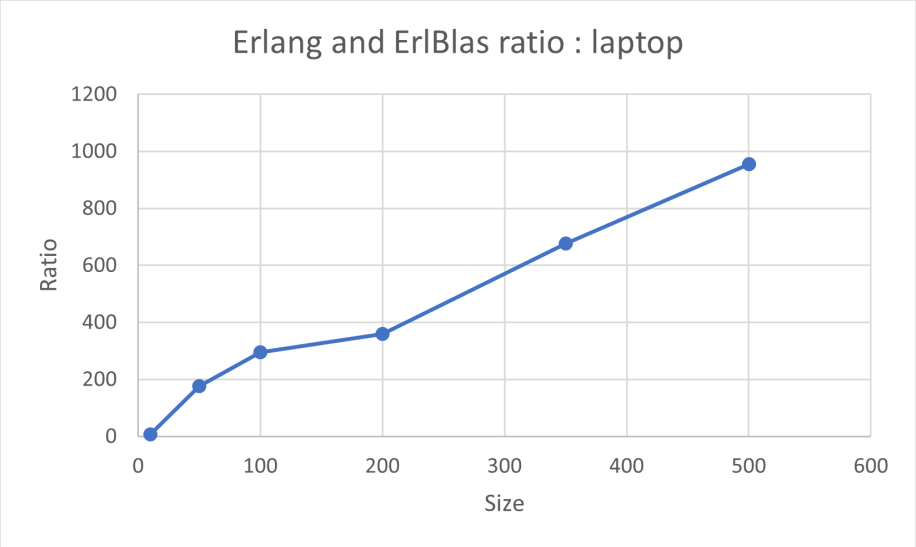

Figure 6.10: Ratio of single core pure Erlang and multicore ErlBlas on multiplication for laptop

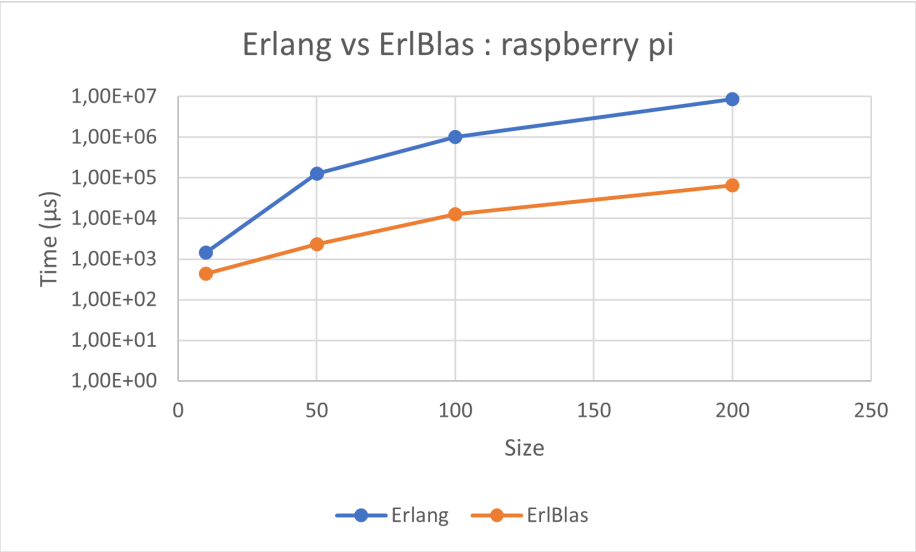

Figure 6.11: Comparing single core pure Erlang to multicore ErlBlas on multiplication for raspberry pi

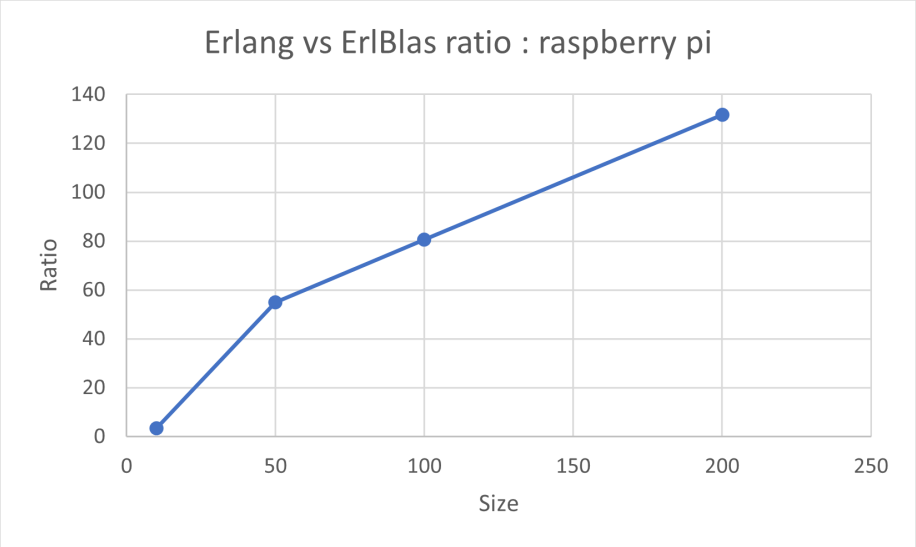

Figure 6.12: Ratio of single core pure Erlang and multicore ErlBlas on multiplication for raspberry pi

6.4 Comparing sequential and parallel versions

In this section we will assess the performance gained by using concurrent operations on processors with several cores, thus allowing parallel execution. We present the results both for the matrix multiplication and for the matrix addition.

The figures 6.13 to 6.24 follow the same structure as in the previous sections, for both the multiplication and the addition.

For the multiplication operation, we can observe a speed-up up to the number of physical cores for the desktop and even more on the laptop. The Raspberry pi seems to have a less predictable speed-up than the other machines... The speedup observed on the laptop is also variable from one run to another. Sometimes the sequential version takes more time for no apparent reason while the parallel version is more consistent in its execution time.

For the add operation, we can observe that there is no speed-up, even for very large matrices. We can even observe that the parallel execution is slower for small matrices. This is easily explained by the block matrix representation. The concurrent implementation of the add operation spawns a process for each line of the ErlBlas matrix and performs all the computations for the given line in a single process then sends the result back to its parent process. However, if the spawn of the process and the delivery of the message take more time than the NIF calls for the line, the parallel implementation can be slower than the sequential implementation. As the benchmark ensures that no call to dgemm on the sub-matrices takes more than 1ms to execute (see section 4.1.1), the execution of the *daxpy* NIF is often extremely fast. We thus have no gain executing them in several different processes.

6.4.1 multiplication

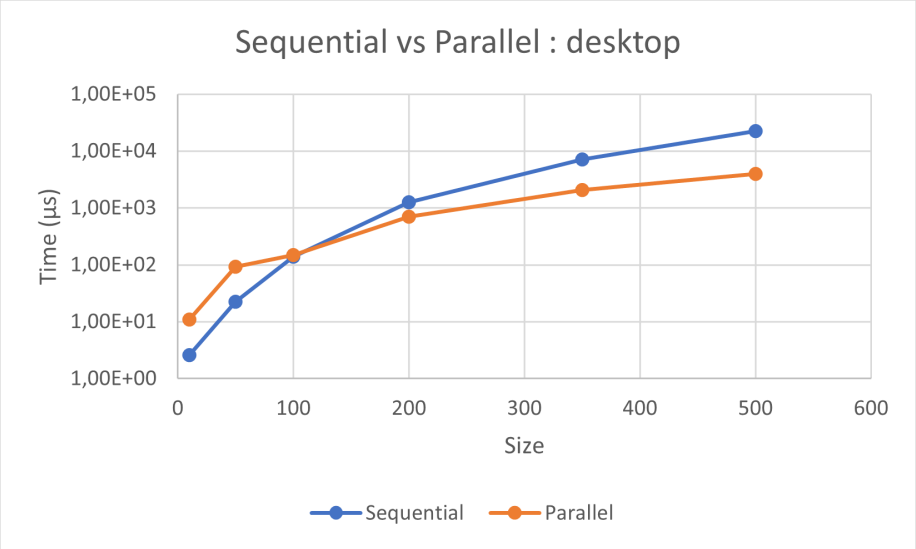

Figure 6.13: Comparing ErlBlas sequential and parallel implementations on multiplication for desktop

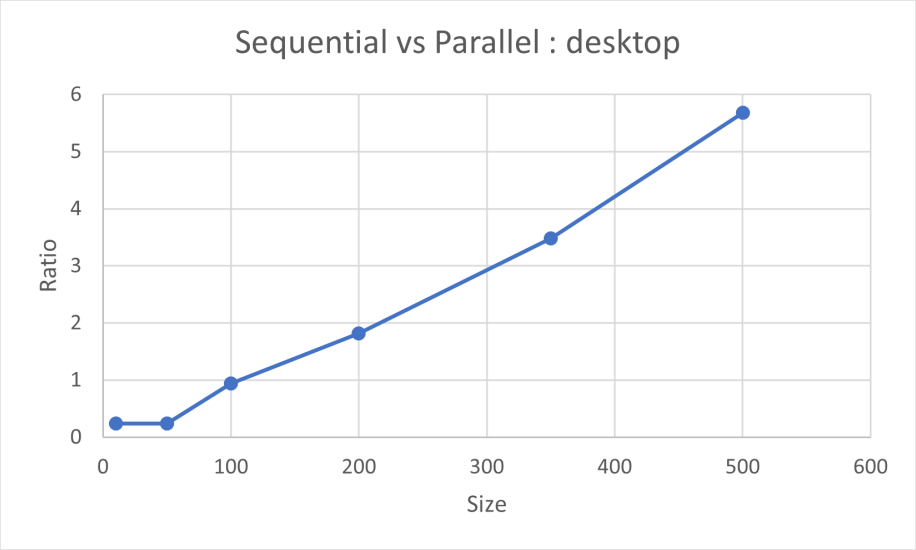

Figure 6.14: Ratio of ErlBlas sequential and parallel implementations on multiplication for desktop

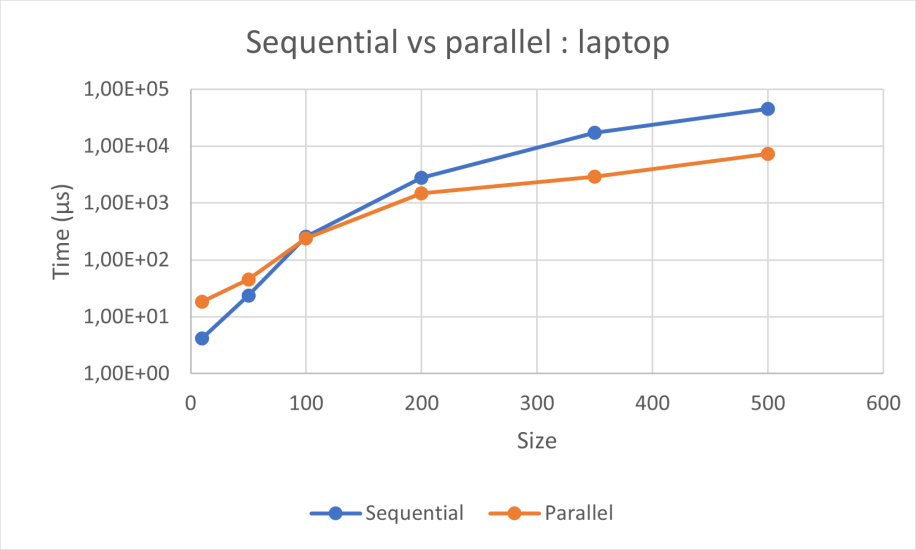

Figure 6.15: Comparing ErlBlas sequential and parallel implementations on multiplication for laptop

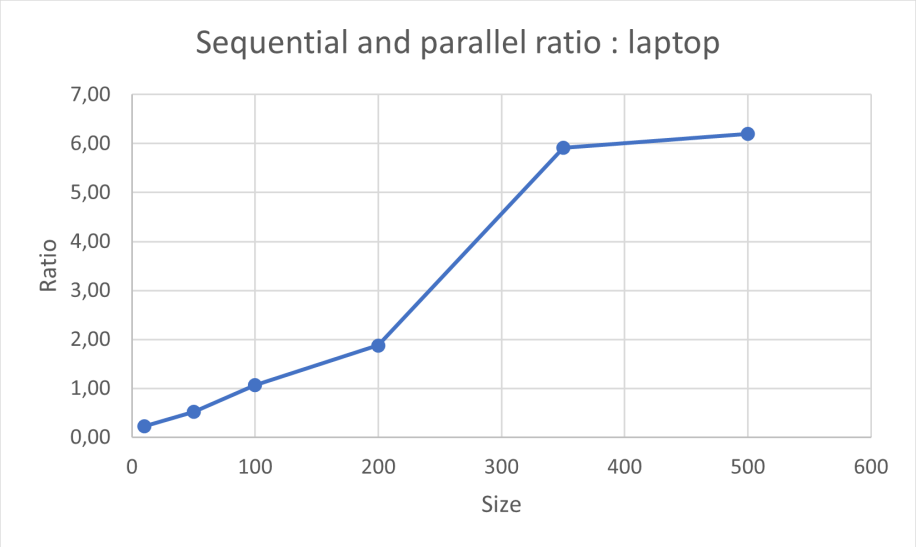

Figure 6.16: Ratio of ErlBlas sequential and parallel implementations on multiplication for laptop

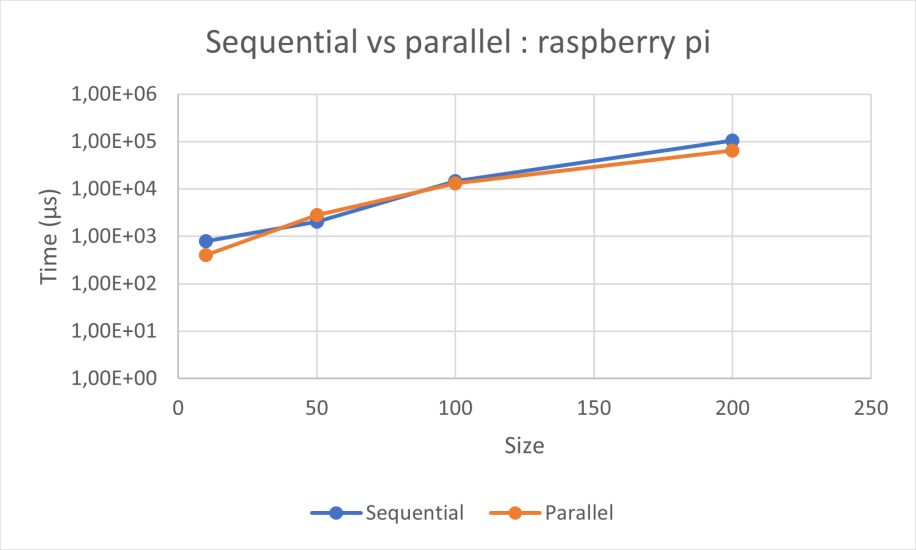

Figure 6.17: Comparing ErlBlas sequential and parallel implementations on multiplication for raspberry pi

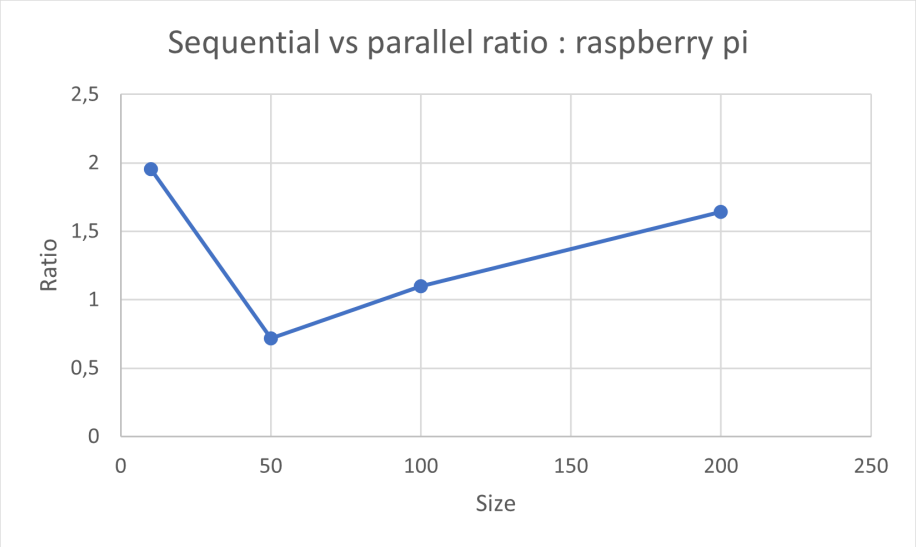

Figure 6.18: Ratio of ErlBlas sequential and parallel implementations on multiplication for raspberry pi

6.4.2 addition

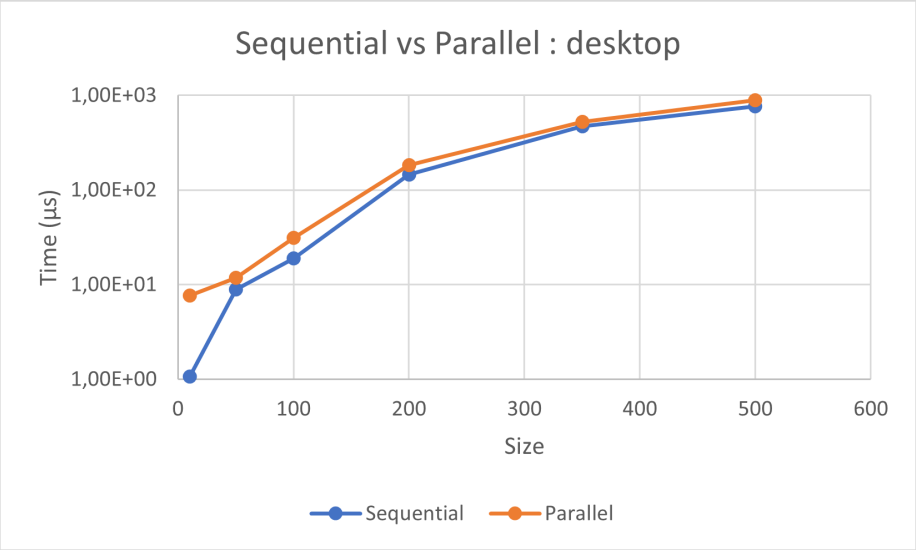

Figure 6.19: Comparing ErlBlas sequential and parallel implementations on addition for desktop

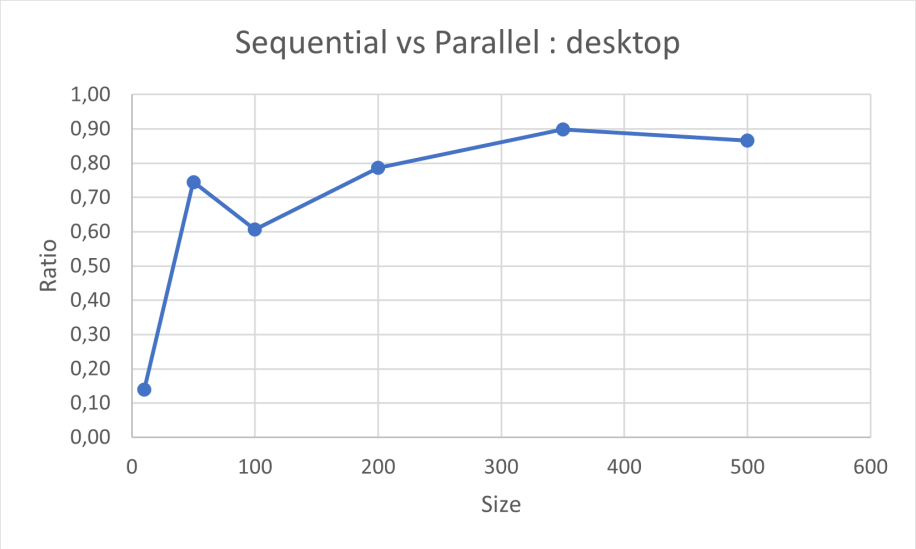

Figure 6.20: Ratio of ErlBlas sequential and parallel implementations on addition for desktop

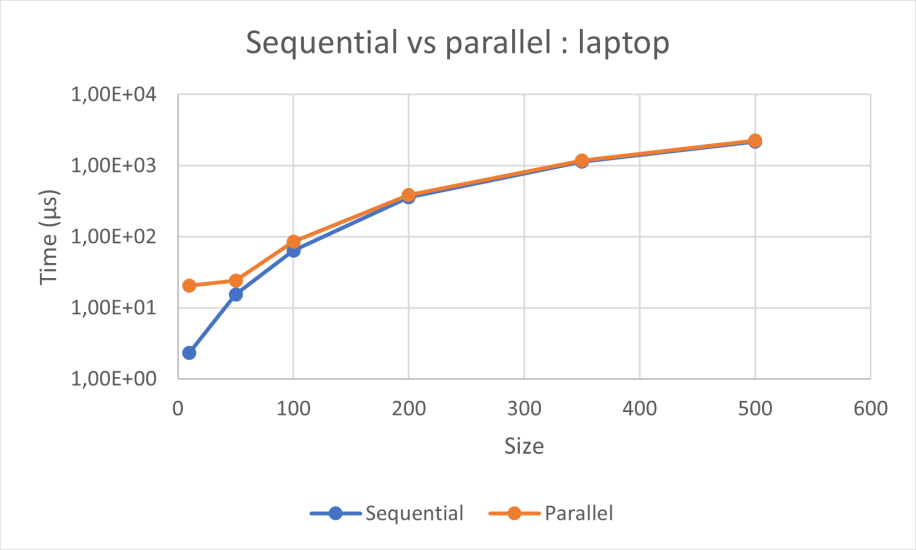

Figure 6.21: Comparing ErlBlas sequential and parallel implementations on addition for laptop

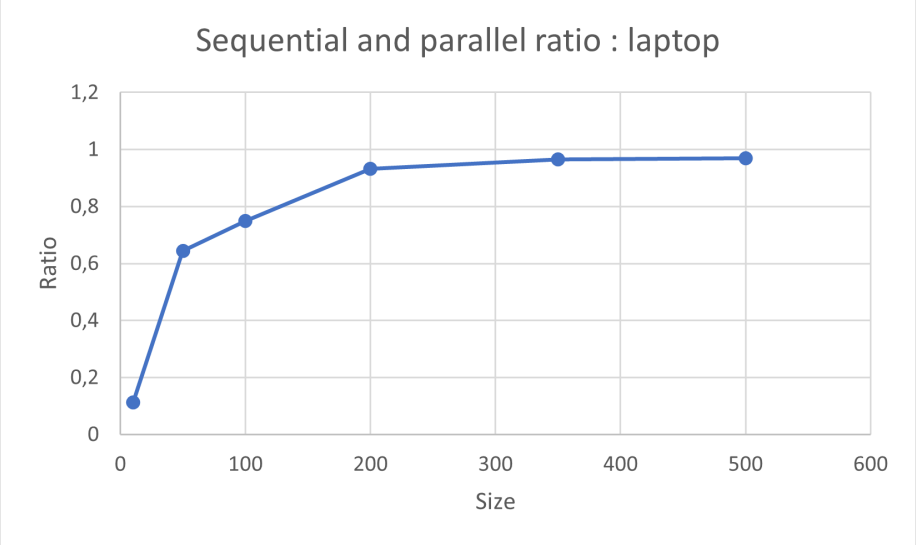

Figure 6.22: Ratio ErlBlas of sequential and parallel implementations on addition for laptop

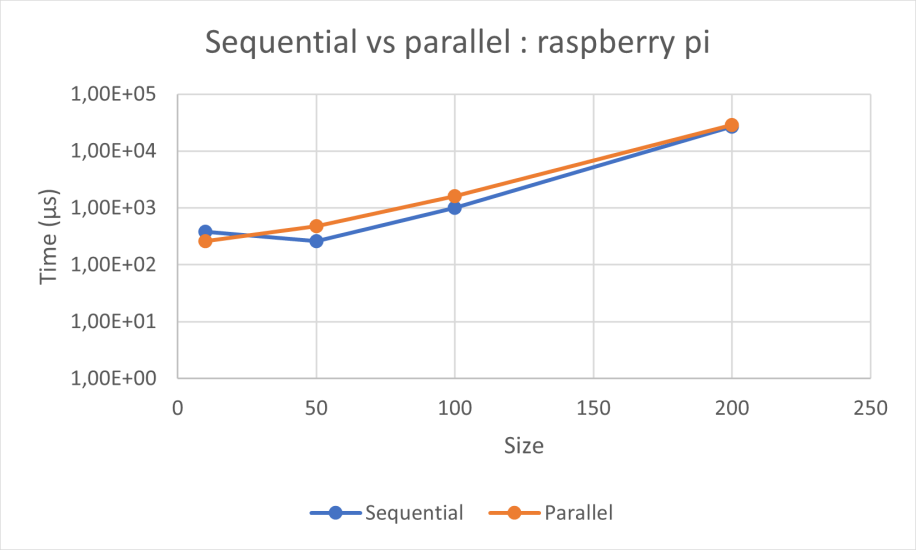

Figure 6.23: Comparing ErlBlas sequential and parallel implementations on addition for raspberry pi

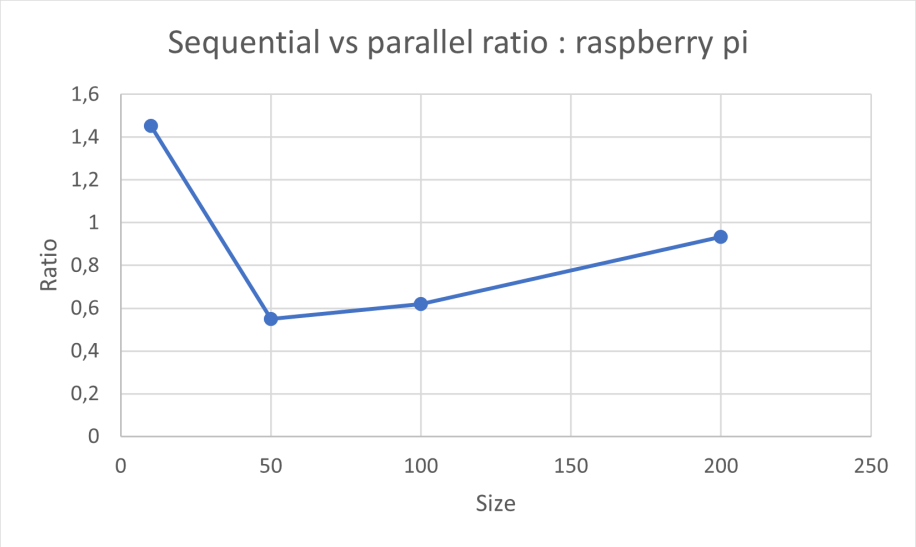

Figure 6.24: Ratio of ErlBlas sequential and parallel implementations on addition for raspberry pi

6.5 Performance of the benchmark function

In this section we will assess how close to the 1ms limit the benchmark bring the maximal block size. Note that since the time taken to execute an operation is a bit variable we would like to have some margin with the true limit to avoid a maximum to exceed 1ms.

The graphs were done by running the same *dgemm* as the benchmark function 40 times for each size and doing an average. This will give taste of what the benchmark sees when testing those sizes and thus let us see where it should end.

6.5.1 Raspberry Pi

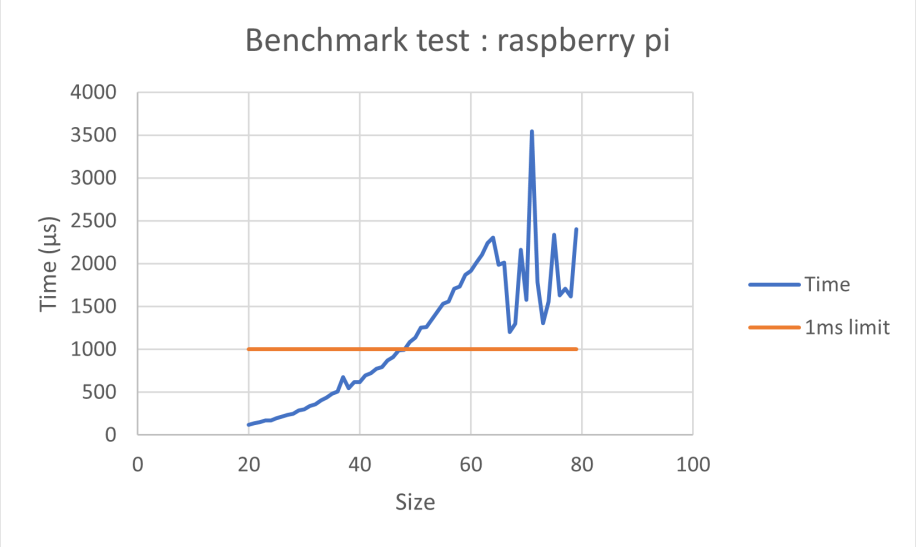

Figure 6.25: Running times for multiplication of pure C on different sizes on Raspberry pi

With the raspberry pi we reach the 1ms limit, symbolised by the orange line, with a matrix size of 49. The benchmark function return a value of 42 which run for 720μs on graph 6.25. This is good for being sure we never go above 1ms but we could want a value a bit higher like 45 to be closer to 1ms while staying quite safe.

We also observe a weird behaviour beginning with a size of 65, which will be discussed further in section 6.5.4.

6.5.2 Desktop

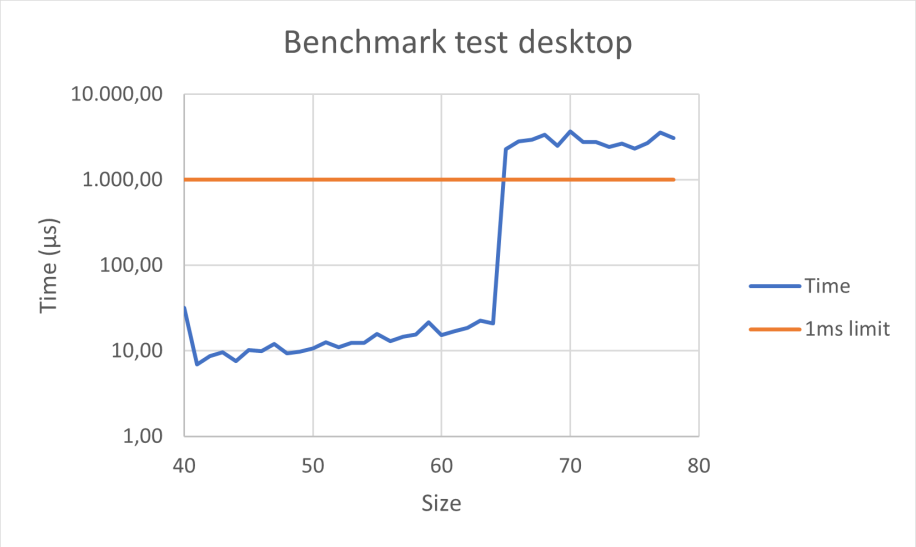

Figure 6.26: Running times for multiplication of pure C on different sizes on desktop

With the desktop computer we are around 20-25μs to execute the multiplication with a size of 64 but it jumps abruptly above 1ms once reaching 65. The benchmark function return a value of 58 which would be good if we really were close to 1ms but it is absolutely not the case.

6.5.3 Laptop

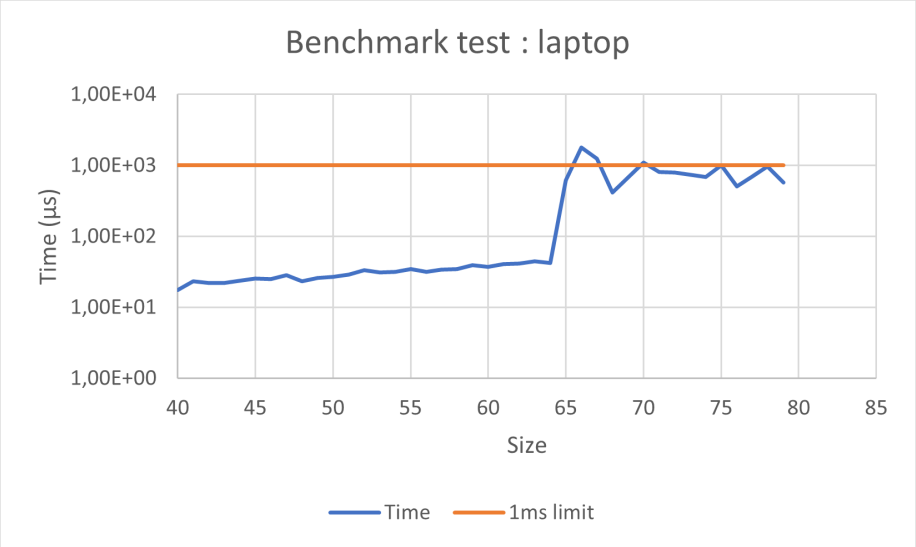

Figure 6.27: Running times for multiplication of pure C on different sizes on laptop

Before 65 we are around 40-45μs for an execution, which is about 2 times slower than the desktop. However, the time explodes lower than the desktop and is around the 1ms limit. This can put the benchmark function in a delicate situation as it can get times under and above 1ms at any point past a size of 65. Fortunately, from experience using this computer, problems rarely arise and the benchmark function almost always returns 58. This is unsurprisingly the same value than the desktop since they both go above 1ms at a size of 65.

6.5.4 Explosion at 65

We don't really know what is causing this behaviour and investigating it is out of the scope of this thesis but we can emit some hypotheses. Since the size of 64 is a nice number for computers, it's an exponent of 2, this could be an optimisation done either on the Erlang side when passing arguments to NIFs or on the CBlas side when doing the computation. This optimisation could only be done when the size of the data is smaller than a certain value which would explain why the time explodes.

The behaviour could also come from cache misses and replacements in the cache causing more cache misses causing a higher execution time.

Making a special for the benchmark in the case when we have a huge difference between 64 and 65 is not a good idea as this behaviour could come from the specific implementation of CBlas used and may not appear in others. It

could also change in the future with an update of CBlas and we would prefer not having to change the implementation when that happens.

Chapter 7

Conclusion

This concludes the presentation of ErlBlas, our newly-written library that allows fast numerical computations in Erlang. The idea of the library is to represent the big matrices as a smaller matrix whose elements are sub-matrices (section 4.1), and to use an extension of the Numerl [1] project as C interface, to call the CBlas function, in order to natively perform the operations on these sub-matrices (chapter 3). The blocks have a maximum size in order to keep the properties of Erlang when calling the natively implemented functions, and to execute these functions in different schedulers (section 2.2.2). This maximum size is determined by a benchmark running at the first call to a function of the library (section 4.1.1).
The interface of ErlBlas contains a subset of the Blas interface, with in-place operations such as *daxpy* or *dgemm*, as well as general-use, single-assignment operations such as *add* or *mult* (appendix B and chapter 4). The in-place operations combined with the block-matrix representation allow to define concurrent versions of these operations (used by default), that can execute in parallel on multicore systems (sections 4.3 and 4.2). Different tests have been performed to assess the correctness of each function exposed to the user (chapter 5). Finally, some performance tests have been presented on *mult* (matrix-matrix multiplication) operation, giving a speed-up of more than one thousand between a pure Erlang naive implementation of the operation and the ErlBlas implementation, performed on a classical desktop processor with six running cores (figure 6.8). The difference of performance between the sequential and the concurrent operations, performed on multicore systems, have also been presented for the *mult* and *add* operations, with *add* showing no gain, and even a little overhead, since the benchmark, and thus the time taken for the NIFs to execute, is fitted on the *mult* function (section 6.4.2).
So far, the library shows quite good results, as the functions are tested on various cases, with successful tests, and as the obtained speed-up can be really high for large matrices. We hope our work can be extensively used by the Erlang community.
However, there is still a long way to go until ErlBlas becomes a perfect Blas

interface for Erlang, so here are some possible improvements and developments.

7.1 Future work

Firstly, as stated in section 4.5.10, we could look for an other formula for the block matrix inversion that would always work if the matrix is invertible. Our current implementation is imperfect and fails when it shouldn't. Having another implementation could be slower depending on the complexity if the formula used, but correctness is more important than speed.

Secondly, we could add more operations in the library. As explained in section 3.5, adding new functions in the CBlas interface is quite easy but adding them inside ErlBlas can be more difficult. You would need to really understand the operation you're adding and find a way to implement a block matrix version. As seen with our inversion operation this is far from trivial and could need some research.

Then, we could add a benchmark specifically for addition operations. The addition being faster then the multiplication, the blocks can be larger before exceeding 1ms runtime. This would allow to reduce the overhead of spawning new processes compared to their execution time. Doing that means that we would either have to make a function that transforms a matrix with a certain MAX_LENGTH to another or have the user specify that it only wants to do additions and no multiplications.

Finally, we could try to develop a distributed version of the library. Currently, due to the usage of resource objects, which are pointers in memory, and in place operations, we cannot distribute our matrix operations on multiples machines. If we would, for example, perform an addition and give each computer a block to compute, the original computer would compute its block and then receive confirmation messages from the other but without the modifications. We would thus end with one correct block and all others would not have been changed. This is without even mentioning that the other computers could segfault due to trying to access some non allocated memory. Even if they don't segfault they would most likely make computations with garbage data since the original data isn't copied.

Bibliography

[1] Tanguy Losseau *Numerl: Efficient Vector and Matrix Computation for Erlang*, UCLouvain. https://dial.uclouvain.be/memoire/ucl/object/thesis:33858

[2] Blas wikipedia page
https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

[3] Blas manual page
http://www.netlib.org/blas/

[4] Nif presentation page
https://www.erlang.org/doc/tutorial/nif.html

[5] C portdriver page
https://www.erlang.org/doc/tutorial/c_portdriver.html

[6] erl_nif C library presentation page
https://www.erlang.org/doc/man/erl_nif.html

[7] Erlang/OTP site
https://www.erlang.org/faq/introduction.html

[8] Erlang Wikipedia page
https://en.wikipedia.org/wiki/Erlang_(programming_language)

[9] Erlang/OTP concurrent programming page
https://www.erlang.org/doc/getting_started/conc_prog.html

[10] Erlang expressions page
https://www.erlang.org/doc/reference_manual/expressions.html

[11] Erlang schedulers page
https://www.erlang.org/doc/man/scheduler.html

[12] Erlang distributed programming page
https://www.erlang.org/doc/reference_manual/distributed.html

[13] Kalbusch Sébastien and Verpoten Vincent. *The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking*. MA thesis. UCLouvain, 2020. `https://dial.uclouvain.be/memoire/ucl/object/thesis:30740`

[14] Block matrix inversion on wikipedia
`https://en.wikipedia.org/wiki/Block_matrix#Block_matrix_inversion`

[15] Block matrix on wikipedia
`https://en.wikipedia.org/wiki/Block_matrix`

Appendix A

Github links

A.1 ErlBlas

https://github.com/zelirio/ErlBlas

A.2 CBlas interface

https://github.com/CoupletB/numerl

Appendix B

ErlBlas interface

B.1 Types definitions

Here are the different new types used in our implementation.

matrix() → [[**number()**]]
Erlang representation of a matrix

numerl_matrix() → **ressource_object()**
A pointer to a matrix to be used in a numerl Nif

block_matrix() → [[**numerl_matrix()**]]
Representation of a block matrix to be used with NumerlPlus' functions

B.2 Specifications

B.2.1 Matrix creation

zeros(N, M) → **block_matrix()**
N = M = integer()
Creates a block matrix filled with 0 representing a N x M matrix

eye(N) → **block_matrix()**
N = integer()
Creates a block matrix filled with a diagonal of 1 of size N x N

matrix(Mat) → **block_matrix()**
Mat = matrix()
Creates a block matrix from a matrix in Erlang representation

copy(M) → **block_matrix()**
M = block_matrix()
Creates a new block matrix with the same content as the given one

copy_shape(M) → block_matrix()
M = block_matrix()
Creates a new block matrix with the same dimensions as the given one but filled with 0

get_shape(M) → integer(), integer()
M = block_matrix()
return the number of lines L and number of columns C of matrix M as a tuple L, C

B.2.2 Arithmetic operations

add(M1, M2) → block_matrix()
M1 = M2 = block_matrix()
Performs the addition of matrices M1 and M2 and return the result in a new matrix

sub(M1, M2) → block_matrix()
M1 = M2 = block_matrix()
Performs the subtraction of matrices M1 and M2 and return the result in a new matrix

mult(M1, M2) → block_matrix()
M1 = M2 = block_matrix()
Performs the matrix multiplication of matrices M1 and M2 and return the result in a new matrix

inv(M1) → block_matrix()
M1 = block_matrix()
returns a new matrix containing the inverse of matrix M1. Warning: it only works for blocks smaller than 17*17

scal(Const, M) → block_matrix()
M = block_matrix()
Const = integer()
Multiplies Matrix M by the constant Const and return the result in a new matrix

transpose(M) → block_matrix()
M = block_matrix()
Returns the transpose of the given matrix

B.2.3 Blas interface

dgemm(ATransp, BTransp, Alpha, A, B, Beta, C) → true
ATransp = BTransp = boolean()
Alpha = Beta = number()
M1 = M2 = C = block_matrix()

Performs the dgemm operation from CBlas with the given arguments [3]. This computes C := Alpha*op(A)*op(B) + Beta*C where op(X) = X or op(X) = X**T depending on the value of ATransp and BTransp. The result is written inside the C matrix.

daxpy(Alpha, X, Y) → true
Alpha = number()
X = Y = block_matrix()
Performs the daxpy operation from CBlas with the given arguments [3]. This function computes Y := Alpha*X + Y. The result is written inside the Y matrix.

dscal(Alpha, X) → true
Alpha = number()
X = block_matrix()
Performs the dscal operation from CBlas with the given arguments [3]. This function multiplies Matrix X by constant Const. The result is written inside the X matrix.

B.2.4 Utility functions

toErl(M) → matrix()
M = block_matrix()
Returns the Erlang representation of block matrix M

equals(M1, M2) → boolean()
M1 = M2 = block_matrix()
Returns true if matrices M1 and M2 are equals with a 10e-6 precision, false otherwise

B.2.5 Benchmark functions

get_max_length() → integer()
Returns maximal size of a block

set_max_length(N) → ?
N = integer()
Sets the maximal size of a block to N. Doesn't change already existing matrices

Appendix C

ErlBlas code

C.1 erlBlas.erl

```
-module(erlBlas).

-compile({no_auto_import, [get/1, put/2]}).

-import(persistent_term, [get/1, put/2]).

-on_load(benchmark/0).

-import(
    utils,
    [
        split4/1,
        recompose4/4,
        appendEach/2,
        appendEachList/1,
        appendList/1,
        generateRandMat/2,
        splitLine/4,
        split4/3,
        element_wise_op/3,
        lineSum/1,
        element_wise_op_conc/3,
        matrix_operation/2
    ]
).

% user interface : single assignment operations and utility
    functions
-export([
    add/2,
    sub/2,
    mult/2,
    inv/1,
    zeros/2,
    matrix/1,
    eye/1,
```

```
36      transpose/1,
37      toErl/1,
38      equals/2,
39      get_max_length/0,
40      set_max_length/1,
41      copy/1,
42      copy_shape/1,
43      get_shape/1
44  ]).
45
46  % user interface : Blas in-place operations
47  -export([dgemm/7, daxpy/3, dscal/2]).
48
49  %%%%%%%%%%%%%%%%%%%%%%%%%%
50  %%% MATRIX GENERATION %%%%
51  %%%%%%%%%%%%%%%%%%%%%%%%%%
52
53  % Creates a zero matrix of size (N x M)
54  zeros(N, M) ->
55      MAX_LENGTH = get(max_length),
56
57      if
58          N rem MAX_LENGTH == 0 ->
59              ModN = MAX_LENGTH,
60              RowMultiple = N div MAX_LENGTH - 1;
61          true ->
62              ModN = N rem MAX_LENGTH,
63              RowMultiple = N div MAX_LENGTH
64      end,
65
66      if
67          M rem MAX_LENGTH == 0 ->
68              ModM = MAX_LENGTH,
69              ColMultiple = M div MAX_LENGTH - 1;
70          true ->
71              ModM = M rem MAX_LENGTH,
72              ColMultiple = M div MAX_LENGTH
73      end,
74
75      if
76          N =< MAX_LENGTH, M =< MAX_LENGTH ->
77              [[numerl:zeros(N, M)]];
78          N =< MAX_LENGTH, M > MAX_LENGTH ->
79              A = [[numerl:zeros(N, ModM)]],
80              B = zeros(N, ColMultiple * MAX_LENGTH),
81              appendEach(A, B);
82          N > MAX_LENGTH, M =< MAX_LENGTH ->
83              A = [[numerl:zeros(ModN, M)]],
84              C = zeros(RowMultiple * MAX_LENGTH, M),
85              lists:append(A, C);
86          N > MAX_LENGTH, M > MAX_LENGTH ->
87              A = [[numerl:zeros(ModN, ModM)]],
88              B = zeros(ModN, ColMultiple * MAX_LENGTH),
89              C = zeros(RowMultiple * MAX_LENGTH, ModM),
90              D = zeros(RowMultiple * MAX_LENGTH, ColMultiple *
    MAX_LENGTH),
91              recompose4(A, B, C, D)
```

```
92      end.
93
94  % Returns an Identity matrix of size (N x N)
95  eye(N) ->
96      MAX_LENGTH = get(max_length),
97
98      if
99          N rem MAX_LENGTH == 0 ->
100             ModN = MAX_LENGTH,
101             RowMultiple = N div MAX_LENGTH - 1;
102         true ->
103             ModN = N rem MAX_LENGTH,
104             RowMultiple = N div MAX_LENGTH
105     end,
106
107     if
108         N =< MAX_LENGTH ->
109             [[numerl:eye(N)]];
110         N > MAX_LENGTH ->
111             A = [[numerl:eye(ModN)]],
112             B = zeros(ModN, RowMultiple * MAX_LENGTH),
113             C = zeros(RowMultiple * MAX_LENGTH, ModN),
114             D = eye(RowMultiple * MAX_LENGTH),
115             recompose4(A, B, C, D)
116     end.
117
118 % Take a numeric matrix in Erlang format (list of lists of numbers)
      and returns a matrix in numerlplus format (list of lists of
    numerl submatrices)
119 matrix(Mat) ->
120     N = length(Mat),
121     M = length(lists:nth(1, Mat)),
122     matrix(Mat, N, M).
123
124 matrix(Mat, N, M) ->
125     MAX_LENGTH = get(max_length),
126
127     if
128         N rem MAX_LENGTH == 0 ->
129             ModN = MAX_LENGTH,
130             RowMultiple = N div MAX_LENGTH - 1;
131         true ->
132             ModN = N rem MAX_LENGTH,
133             RowMultiple = N div MAX_LENGTH
134     end,
135
136     if
137         M rem MAX_LENGTH == 0 ->
138             ModM = MAX_LENGTH,
139             ColMultiple = M div MAX_LENGTH - 1;
140         true ->
141             ModM = M rem MAX_LENGTH,
142             ColMultiple = M div MAX_LENGTH
143     end,
144
145     if
146         N =< MAX_LENGTH, M =< MAX_LENGTH ->
```

```
147             [[numerl:matrix(Mat)]];
148         N =< MAX_LENGTH, M > MAX_LENGTH ->
149             {L, R} = splitLine(Mat, [], [], ModM),
150             A = [[numerl:matrix(L)]],
151             B = matrix(R, N, ColMultiple * MAX_LENGTH),
152             appendEach(A, B);
153         N > MAX_LENGTH, M =< MAX_LENGTH ->
154             {U, L} = lists:split(ModN, Mat),
155             A = [[numerl:matrix(U)]],
156             C = matrix(L, RowMultiple * MAX_LENGTH, M),
157             lists:append(A, C);
158         N > MAX_LENGTH, M > MAX_LENGTH ->
159             {UL, UR, LL, LR} = split4(Mat, ModN, ModM),
160             A = [[numerl:matrix(UL)]],
161             B = matrix(UR, ModN, ColMultiple * MAX_LENGTH),
162             C = matrix(LL, RowMultiple * MAX_LENGTH, ModM),
163             D = matrix(LR, RowMultiple * MAX_LENGTH, ColMultiple *
    MAX_LENGTH),
164             recompose4(A, B, C, D)
165     end.
166
167 copy(M) ->
168     matrix_operation(fun numerl:copy/1, M).
169
170 copy_shape(M) ->
171     matrix_operation(fun numerl:copy_shape/1, M).
172
173 get_shape(M) ->
174     H = get_height(M, 0),
175     W = get_width(lists:nth(1, M), 0),
176     {H, W}.
177
178 get_height(M, N) ->
179     case M of
180         [] ->
181             N;
182         [H | T] ->
183             {L, _} =
184                 numerl:get_shape(
185                     lists:nth(1, H)
186                 ),
187             get_height(T, N + L);
188         _ ->
189             error("get_height: invalid matrix")
190     end.
191
192 get_width(M, N) ->
193     case M of
194         [] ->
195             N;
196         [H | T] ->
197             {_, W} = numerl:get_shape(H),
198             get_width(T, N + W);
199         _ ->
200             error("get_width: invalid matrix")
201     end.
202
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% NUMERIC OPERATIONS %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Returns the result of the addition of the two matrices in
    argument, in numerlplus format
add(M1, M2) ->
    M3 = copy(M1),
    daxpy(1.0, M2, M3),
    M3.

% Returns the result of the substraction of the two matrices in
    argument, in numerlplus format
sub(M1, M2) ->
    M3 = copy(M1),
    daxpy(-1.0, M2, M3),
    M3.

% Returns the result of the multiplication of the two matrices in
    argument, in numerlplus format
mult(M1, M2) ->
    {R1, C1} = get_shape(M1),
    {C1, C2} = get_shape(M2),
    M3 = zeros(R1, C2),
    dgemm(false, false, 1.0, M1, M2, 0.0, M3),
    M3.

tr(M) ->
    tr(M, []).

tr([[] | _], Rows) ->
    lists:reverse(Rows);
tr(M, Rows) ->
    {Row, Cols} = tr(M, [], []),
    tr(Cols, [Row | Rows]).

tr([], Col, Cols) ->
    {lists:reverse(Col), lists:reverse(Cols)};
tr([[H | T] | Rows], Col, Cols) ->
    tr(Rows, [H | Col], [T | Cols]).

% Returns the result of the multiplication of the matrix M by the
    scalar Const
scal(Const, M) ->
    R = copy(M),
    dscal(Const, R),
    R.

% Returns the inverse of the matrix given, in numerlplus format
inv(M1) ->
    Len = length(M1),
    if
        Len > 1 ->
            {A, B, C, D} = split4(M1),
            InvA = inv(A),
            InvAB = mult(InvA, B),
            CInvA = mult(C, InvA),
```

```
256             % inv(D-C InvA B)
257             C4 = inv(sub(D, mult(C, InvAB))),
258             Big = mult(C4, CInvA),
259             C3 = scal(-1, Big),
260             C2 = scal(-1, mult(InvAB, C4)),
261             C1 = add(InvA, mult(InvAB, Big)),
262             recompose4(C1, C2, C3, C4);
263         true ->
264             [[Bin]] = M1,
265             [[numerl:inv(Bin)]]
266     end.
267
268 % Returns the transpose of the matrix given, in numerlplus format
269 transpose(M) ->
270     Tr = tr(M),
271     matrix_operation(fun numerl:transpose/1, Tr).
272
273 % Takes a matrix in numerlplus format and return the same matrix in
      erlang format (list of lists of numbers)
274 toErl(M) ->
275     appendList(
276         lists:map(
277             fun(Row) ->
278                 appendEachList(lists:map(fun(Elem) -> numerl:mtfl(
    Elem) end, Row))
279             end,
280             M
281         )
282     ).
283
284 % Returns true if the elements of the two matrix given (in
    numerlplus format) are all the same, false otherwise.
285 equals(M1, M2) ->
286     Eq = element_wise_op(fun numerl:equals/2, M1, M2),
287     lists:all(fun(Row) -> lists:all(fun(B) -> B end, Row) end, Eq).
288
289 %%%%%%%%%%%%%%%%%%%%%%%%%%%
290 %%% BENCHMARK %%%%%%%%%%%%%
291 %%%%%%%%%%%%%%%%%%%%%%%%%%%
292
293 get_max_length() ->
294     get(max_length).
295
296 set_max_length(N) ->
297     put(max_length, N).
298
299 benchmark() ->
300     First_max = lists:min([round_one_benchmark(5) || _ <- lists:seq
    (1, 5)]),
301     Second_max =
302         lists:nth(2, lists:sort([round_two_benchmark(First_max) ||
    _ <- lists:seq(1, 20)])),
303     put(max_length, Second_max).
304
305 round_one_benchmark(N) ->
306     Passed = test_time(N, false),
307     if
```

```
308         Passed ->
309             round_one_benchmark(N * 2);
310         true ->
311             N div 2
312     end.
313
314 round_two_benchmark(N) ->
315     Passed = test_time(N, false),
316     if
317         Passed ->
318             round_two_benchmark(round(N * 1.2));
319         true ->
320             round(N / 1.2)
321     end.
322
323 test_time(N, true) ->
324     timer:sleep(100),
325     erlang:display(hello),
326     M = generateRandMat(N, N),
327     Mat = numerl:matrix(M),
328     TimesValues = [timer:tc(numerl, dot, [Mat, Mat]) || _ <- lists:
    seq(1, 50)],
329     Times = [Time || {Time, _} <- TimesValues],
330     Test = lists:search(fun(Time) -> Time > 1000 end, Times),
331     case Test of
332         {value, _} ->
333             false;
334         _ ->
335             true
336     end;
337 test_time(N, false) ->
338     M = generateRandMat(N, N),
339     M2 = generateRandMat(N, N),
340     Mat = numerl:matrix(M),
341     Mat2 = numerl:matrix(M2),
342     C = numerl:zeros(N, N),
343     {Time, _} = timer:tc(numerl, dgemm, [1, 1, 2.5, Mat, Mat2, 3.5,
     C]),
344     if
345         Time < 1000 ->
346             true;
347         true ->
348             false
349     end.
350
351 %%%%%%%%%%%%%%%%%%%%%%%%%%
352 %%% BLAS INTERFACE %%%%%%%
353 %%%%%%%%%%%%%%%%%%%%%%%%%%
354
355 dgemm(ATransp, BTransp, Alpha, M1, M2, Beta, C) ->
356     if
357         ATransp ->
358             A = tr(M1);
359         true ->
360             A = M1
361     end,
362     if
```

```
363         BTransp ->
364             B = M2;
365         true ->
366             B = tr(M2)
367     end,
368     if
369         Beta /= 1.0 ->
370             dscal(Beta, C);
371         true ->
372             skip
373     end,
374     PID = self(),
375     PIDs =
376         lists:zipwith(
377             fun(RowA, RowC) ->
378                 spawn(fun() ->
379                     ParentPID = self(),
380                     PidList =
381                         lists:zipwith(
382                             fun(RowB, ElemC) ->
383                                 spawn(fun() ->
384                                     lists:zipwith(
385                                         fun(ElemA, ElemB) ->
386                                             numerl:dgemm(
387                                                 if
388                                                     ATransp -> 1;
389                                                     true -> 0
390                                                 end,
391                                                 if
392                                                     BTransp -> 1;
393                                                     true -> 0
394                                                 end,
395                                                 Alpha,
396                                                 ElemA,
397                                                 ElemB,
398                                                 1.0,
399                                                 ElemC
400                                             )
401                                         end,
402                                         RowA,
403                                         RowB
404                                     ),
405                                     ParentPID ! {finished, self()}
406                                 end)
407                             end,
408                             B,
409                             RowC
410                         ),
411                     lists:map(
412                         fun(Elem) ->
413                             receive
414                                 {finished, Elem} ->
415                                     ok
416                             end
417                         end,
418                         PidList
419                     ),
```

```
420                     PID ! {finished, self()}
421                 end)
422             end,
423             A,
424             C
425         ),
426
427     lists:map(
428         fun(Elem) ->
429             receive
430                 {finished, Elem} ->
431                     ok
432             end
433         end,
434         PIDs
435     ).
436
437 daxpy(Alpha, X, Y) ->
438     element_wise_op_conc(fun(A, B) -> numerl:daxpy(Alpha, A, B) end
    , X, Y).
439
440 dscal(Alpha, X) ->
441     matrix_operation(fun(A) -> numerl:dscal(Alpha, A) end, X).
```

C.2 utils.erl

```
1  -module(utils).
2
3  -export([
4      generateRandMat/2,
5      append/1,
6      appendEach/2,
7      appendEachList/1,
8      appendList/1,
9      splitLine/3,
10     split4/1,
11     recompose4/4, recompose4/1,
12     split4/3,
13     splitLine/4,
14     lineSum/1
15 ]).
16
17 -export([
18     element_wise_op/3,
19     element_wise_op_conc2/3,
20     element_wise_op_conc/3,
21     matrix_operation/2
22 ]).
23
24 generateRandMat(0, _) ->
25     [];
26 generateRandMat(Dim1, Dim2) ->
27     [generateRandVect(Dim2) | generateRandMat(Dim1 - 1, Dim2)].
28
29 generateRandVect(0) ->
```

```
    [];
generateRandVect(Dim2) ->
    [rand:uniform(1000) | generateRandVect(Dim2 - 1)].

splitLine(M1, Acc1, Acc2) ->
    case M1 of
        [] ->
            {lists:reverse(Acc1), lists:reverse(Acc2)};
        [H | T] ->
            {A1, A2} = lists:split(trunc(length(H) / 2), H),
            splitLine(T, [A1 | Acc1], [A2 | Acc2])
    end.

splitLine(M1, Acc1, Acc2, N) ->
    case M1 of
        [] ->
            {lists:reverse(Acc1), lists:reverse(Acc2)};
        [H | T] ->
            {A1, A2} = lists:split(N, H),
            splitLine(T, [A1 | Acc1], [A2 | Acc2], N)
    end.

split4(M1) ->
    {A1, A2} = splitLine(M1, [], []),
    {Xa, Xc} = lists:split(trunc(length(A1) / 2), A1),
    {Xb, Xd} = lists:split(trunc(length(A2) / 2), A2),
    {Xa, Xb, Xc, Xd}.

split4(M1, N, M) ->
    {A1, A2} = splitLine(M1, [], [], M),
    {Xa, Xc} = lists:split(N, A1),
    {Xb, Xd} = lists:split(N, A2),
    {Xa, Xb, Xc, Xd}.

recompose4(M1, M2, M3, M4) ->
    lists:append(appendEach(M1, M2), appendEach(M3, M4)).

recompose4({Pid, ID}) ->
    receive
        {a, A} ->
            receive
                {b, B} ->
                    receive
                        {c, C} ->
                            receive
                                {d, D} ->
                                    Result = recompose4(A, B, C, D)
    ,
                                    Pid ! {ID, Result}
                            end
                    end
            end
    end.

append({Pid, ID}) ->
    receive
        {a, A} ->
```

```
86              receive
87                  {b, B} ->
88                      Result = lists:append(A, B),
89                      Pid ! {ID, Result}
90              end
91      end.
92
93  appendEach(M1, M2) ->
94      case {M1, M2} of
95          {[], []} ->
96              [];
97          {[H1 | T1], [H2 | T2]} ->
98              [lists:append(H1, H2) | appendEach(T1, T2)]
99      end.
100
101 appendEachList(L) ->
102     case L of
103         [H] ->
104             H;
105         [H | T] ->
106             appendEach(H, appendEachList(T))
107     end.
108
109 appendList(L) ->
110     case L of
111         [H] ->
112             H;
113         [H | T] ->
114             lists:append(H, appendList(T))
115     end.
116
117 element_wise_op(Op, M1, M2) ->
118     lists:zipwith(
119         fun(L1, L2) -> lists:zipwith(fun(E1, E2) -> Op(E1, E2) end,
     L1, L2) end,
120         M1,
121         M2
122     ).
123
124 element_wise_op_conc2(Op, M1, M2) ->
125     ParentPID = self(),
126     PidList = lists:zipwith(
127         fun(Row1, Row2) ->
128             spawn(fun() ->
129                 Result =
130                     lists:zipwith(
131                         fun(Elem1, Elem2) ->
132                             ParPID = self(),
133                             Pidipid =
134                                 spawn(fun() ->
135                                     Res = Op(Elem1, Elem2),
136                                     ParPID ! {Res, self()}
137                                 end),
138                             receive
139                                 {Res, Pidipid} ->
140                                     Res
141                             end
```

```
142                             end,
143                             Row1,
144                             Row2
145                         ),
146                     ParentPID ! {Result, self()}
147                 end)
148         end,
149         M1,
150         M2
151     ),
152     lists:map(
153         fun(Pid) ->
154             receive
155                 {Result, Pid} ->
156                     Result
157             end
158         end,
159         PidList
160     ).
161
162 element_wise_op_conc(Op, M1, M2) ->
163     ParentPID = self(),
164     PidMat =
165         lists:zipwith(
166             fun(L1, L2) ->
167                 spawn(fun() ->
168                     Result = lists:zipwith(fun(E1, E2) -> Op(E1, E2
    ) end, L1, L2),
169                     ParentPID ! {Result, self()}
170                 end)
171             end,
172             M1,
173             M2
174         ),
175     lists:map(
176         fun(LinePid) ->
177             receive
178                 {Result, LinePid} ->
179                     Result
180             end
181         end,
182         PidMat
183     ).
184
185 matrix_operation(Op, M) ->
186     lists:map(fun(Row) -> lists:map(fun(A) -> Op(A) end, Row) end,
    M).
187
188 lineSum([H | T]) ->
189     lineSum(T, H).
190
191 lineSum(List, Acc) ->
192     case List of
193         [H | T] ->
194             lineSum(T, numerl:add(Acc, H));
195         [] ->
196             Acc
```

```
197     end.
```

C.3 sequential.erl

```
1  -module(sequential).
2
3  -export([dgemm/7, add/2, daxpy/3, dscal/2, mult/2]).
4
5  dgemm(ATransp, BTransp, Alpha, M1, M2, Beta, C) ->
6      if
7          ATransp ->
8              A = tr(M1);
9          true ->
10             A = M1
11     end,
12     if
13         BTransp ->
14             B = M2;
15         true ->
16             B = tr(M2)
17     end,
18     if
19         Beta /= 1.0 ->
20             dscal(Beta, C);
21         true ->
22             skip
23     end,
24     lists:zipwith(
25         fun(RowA, RowC) ->
26             lists:zipwith(
27                 fun(RowB, ElemC) ->
28                     lists:zipwith(
29                         fun(ElemA, ElemB) ->
30                             numerl:dgemm(
31                                 if
32                                     ATransp -> 1;
33                                     true -> 0
34                                 end,
35                                 if
36                                     BTransp -> 1;
37                                     true -> 0
38                                 end,
39                                 Alpha,
40                                 ElemA,
41                                 ElemB,
42                                 1.0,
43                                 ElemC
44                             )
45                         end,
46                         RowA,
47                         RowB
48                     )
49                 end,
50                 B,
51                 RowC
```

```
52                  )
53          end,
54          A,
55          C
56      ).
57
58  add(M1, M2) ->
59      M3 = erlBlas:copy(M1),
60      daxpy(1.0, M2, M3),
61      M3.
62
63  daxpy(Alpha, X, Y) ->
64      utils:element_wise_op(fun(A, B) -> numerl:daxpy(Alpha, A, B)
        end, X, Y).
65
66  dscal(Alpha, X) ->
67      utils:matrix_operation(fun(A) -> numerl:dscal(Alpha, A) end, X)
        .
68
69  tr(M) ->
70      tr(M, []).
71
72  tr([[] | _], Rows) ->
73      lists:reverse(Rows);
74  tr(M, Rows) ->
75      {Row, Cols} = tr(M, [], []),
76      tr(Cols, [Row | Rows]).
77
78  tr([], Col, Cols) ->
79      {lists:reverse(Col), lists:reverse(Cols)};
80  tr([[H | T] | Rows], Col, Cols) ->
81      tr(Rows, [H | Col], [T | Cols]).
82
83  mult(M1, M2) ->
84      multT(M1, tr(M2)).
85
86  multT(M1, M2) ->
87      [
88          [
89              utils:lineSum(
90                  lists:zipwith(fun(A, B) -> numerl:dot(A, B) end, Li
        , Cj)
91              )
92           || Cj <- M2
93          ]
94       || Li <- M1
95      ].
```

Appendix D

Test code

D.1 add_seq_test_SUITE.erl

```
-module(add_seq_test_SUITE).

-include_lib("stdlib/include/assert.hrl").
-include_lib("eunit/include/eunit.hrl").

base_test() ->
    A = [[1]],
    B = [[1]],
    C = [[2]],
    ABlock = erlBlas:matrix(A),
    BBlock = erlBlas:matrix(B),
    CBlock = erlBlas:matrix(C),
    Res = sequential:add(ABlock, BBlock),
    ?assert(erlBlas:equals(Res, CBlock)).

max_size_blocks_test() ->
    A = [
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    ],

    B = [
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
```

```
37          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
38          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
39          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
40          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
41      ],
42
43      C = [
44          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
45          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
46          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
47          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
48          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
49          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
50          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
51          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
52          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
53          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
54      ],
55
56      ABlock = erlBlas:matrix(A),
57      BBlock = erlBlas:matrix(B),
58      CBlock = erlBlas:matrix(C),
59      Res = sequential:add(ABlock, BBlock),
60      ?assert(erlBlas:equals(Res, CBlock)).
61
62  float_test() ->
63      A = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
        9.58, 10.013, 69.42]],
64
65      B = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
        9.58, 10.013, 69.42]],
66
67      ABlock = erlBlas:matrix(A),
68      BBlock = erlBlas:matrix(B),
69      Res = sequential:add(ABlock, BBlock),
70      ANum = numerl:matrix(A),
71      BNum = numerl:matrix(B),
72      Conf = numerl:add(ANum, BNum),
73      Expected = numerl:mtfl(Conf),
74      Actual = erlBlas:toErl(Res),
75      ?assert(mat:'=='(Expected, Actual)).
76
77  small_random_test() ->
78      Max = erlBlas:get_max_length(),
79      erlBlas:set_max_length(50),
80      Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
81      {timeout, 100, fun() ->
82          matrix_test_core(Sizes),
83          erlBlas:set_max_length(Max)
84      end}.
85
86  corner_cases_test_() ->
87      Max = erlBlas:get_max_length(),
88      erlBlas:set_max_length(50),
89      Sizes = [49, 50, 51, 99, 100, 101],
90      {timeout, 100, fun() ->
91          matrix_test_core(Sizes),
```

```
92          erlBlas:set_max_length(Max)
93      end}.
94
95  random_test() ->
96      Max = erlBlas:get_max_length(),
97      erlBlas:set_max_length(50),
98      Sizes = [rand:uniform(800) + 500, rand:uniform(800) + 500, rand
        :uniform(800) + 500],
99      {timeout, 100, fun() ->
100         matrix_test_core(Sizes),
101         erlBlas:set_max_length(Max)
102     end}.
103
104 matrix_test_core(Sizes) ->
105     lists:map(
106         fun(M) ->
107             lists:map(
108                 fun(N) ->
109                     random_test_core(M, N)
110                 end,
111                 Sizes
112             )
113         end,
114         Sizes
115     ).
116
117 random_test_core(M, N) ->
118     A = utils:generateRandMat(M, N),
119     B = utils:generateRandMat(M, N),
120     ABlock = erlBlas:matrix(A),
121     BBlock = erlBlas:matrix(B),
122     Res = sequential:add(ABlock, BBlock),
123     ANum = numerl:matrix(A),
124     BNum = numerl:matrix(B),
125     Conf = numerl:add(ANum, BNum),
126     Expected = numerl:mtfl(Conf),
127     Actual = erlBlas:toErl(Res),
128     ?assert(mat:'=='(Expected, Actual)).
```

D.2 add_test_SUITE.erl

```
1  -module(add_test_SUITE).
2  -import(erlBlas, []).
3
4  -include_lib("stdlib/include/assert.hrl").
5  -include_lib("eunit/include/eunit.hrl").
6
7  base_test() ->
8      A = [[1]],
9      B = [[1]],
10     C = [[2]],
11     ABlock = erlBlas:matrix(A),
12     BBlock = erlBlas:matrix(B),
13     CBlock = erlBlas:matrix(C),
14     Res = erlBlas:add(ABlock, BBlock),
```

```
15      ?assert(erlBlas:equals(Res, CBlock)).
16
17  max_size_blocks_test() ->
18      A = [
19          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
20          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
21          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
22          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
23          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
24          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
25          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
26          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
27          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
28          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
29      ],
30
31      B = [
32          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
33          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
34          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
35          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
36          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
37          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
38          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
39          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
40          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
41          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
42      ],
43
44      C = [
45          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
46          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
47          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
48          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
49          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
50          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
51          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
52          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
53          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
54          [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
55      ],
56
57      ABlock = erlBlas:matrix(A),
58      BBlock = erlBlas:matrix(B),
59      CBlock = erlBlas:matrix(C),
60      Res = erlBlas:add(ABlock, BBlock),
61      ?assert(erlBlas:equals(Res, CBlock)).
62
63  float_test() ->
64      A = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
        9.58, 10.013, 69.42]],
65
66      B = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
        9.58, 10.013, 69.42]],
67
68      ABlock = erlBlas:matrix(A),
69      BBlock = erlBlas:matrix(B),
```

```
70      Res = erlBlas:add(ABlock, BBlock),
71      ANum = numerl:matrix(A),
72      BNum = numerl:matrix(B),
73      Conf = numerl:add(ANum, BNum),
74      Expected = numerl:mtfl(Conf),
75      Actual = erlBlas:toErl(Res),
76      ?assert(mat:'=='(Expected, Actual)).
77
78  small_random_test() ->
79      Max = erlBlas:get_max_length(),
80      erlBlas:set_max_length(50),
81      Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
82      {timeout, 100, fun() ->
83          matrix_test_core(Sizes),
84          erlBlas:set_max_length(Max)
85      end}.
86
87  corner_cases_test_() ->
88      Max = erlBlas:get_max_length(),
89      erlBlas:set_max_length(50),
90      Sizes = [49, 50, 51, 99, 100, 101],
91      {timeout, 100, fun() ->
92          matrix_test_core(Sizes),
93          erlBlas:set_max_length(Max)
94      end}.
95
96  random_test() ->
97      Max = erlBlas:get_max_length(),
98      erlBlas:set_max_length(50),
99      Sizes = [rand:uniform(800) + 500, rand:uniform(800) + 500, rand
        :uniform(800) + 500],
100     {timeout, 100, fun() ->
101         matrix_test_core(Sizes),
102         erlBlas:set_max_length(Max)
103     end}.
104
105 matrix_test_core(Sizes) ->
106     lists:map(
107         fun(M) ->
108             lists:map(
109                 fun(N) ->
110                     random_test_core(M, N)
111                 end,
112                 Sizes
113             )
114         end,
115         Sizes
116     ).
117
118 random_test_core(M, N) ->
119     A = utils:generateRandMat(M, N),
120     B = utils:generateRandMat(M, N),
121     ABlock = erlBlas:matrix(A),
122     BBlock = erlBlas:matrix(B),
123     Res = erlBlas:add(ABlock, BBlock),
124     ANum = numerl:matrix(A),
125     BNum = numerl:matrix(B),
```

```
126     Conf = numerl:add(ANum, BNum),
127     Expected = numerl:mtfl(Conf),
128     Actual = erlBlas:toErl(Res),
129     ?assert(mat:'=='(Expected, Actual)).
130 
131 performance_test_() ->
132     Max = erlBlas:get_max_length(),
133     {timeout, 1000, fun() ->
134         MaxLengths = [5, 10, 50, 100, 200, 500],
135         lists:map(
136             fun(MaxLength) ->
137                 erlBlas:set_max_length(MaxLength),
138                 erlang:display({max_length, erlBlas:get_max_length
    ()}),
139                 performance(),
140                 performance_conc(),
141                 erlBlas:set_max_length(Max)
142             end,
143             MaxLengths
144         )
145     end}.
146 
147 performance_conc() ->
148     timer:sleep(100),
149     %add_exec_time(10,100),
150 
151     %,1000,2000],
152     Sizes = [10, 50, 100, 200, 350, 500],
153     Results =
154         lists:map(
155             fun(Size) ->
156                 Times = add_conc_exec_time(40, Size),
157                 stats(Times)
158             end,
159             Sizes
160         ),
161     erlang:display({conc, Results}).
162 
163 add_conc_exec_time(N, Size) ->
164     M1 = utils:generateRandMat(Size, Size),
165     M2 = utils:generateRandMat(Size, Size),
166     Mat1 = erlBlas:matrix(M1),
167     Mat2 = erlBlas:matrix(M2),
168     {Time, _} = timer:tc(erlBlas, add, [Mat1, Mat2]),
169     if
170         N == 1 ->
171             [Time];
172         true ->
173             [Time | add_conc_exec_time(N - 1, Size)]
174     end.
175 
176 performance() ->
177     timer:sleep(100),
178     %add_exec_time(10,100),
179 
180     %,1000,2000],
181     Sizes = [10, 50, 100, 200, 350, 500],
```

```
182     Results =
183         lists:map(
184             fun(Size) ->
185                 Times = add_exec_time(40, Size),
186                 stats(Times)
187             end,
188             Sizes
189         ),
190     erlang:display({seq, Results}).
191
192 add_exec_time(N, Size) ->
193     M1 = utils:generateRandMat(Size, Size),
194     M2 = utils:generateRandMat(Size, Size),
195     Mat1 = erlBlas:matrix(M1),
196     Mat2 = erlBlas:matrix(M2),
197     {Time, _} = timer:tc(sequential, add, [Mat1, Mat2]),
198     if
199         N == 1 ->
200             [Time];
201         true ->
202             [Time | add_exec_time(N - 1, Size)]
203     end.
204
205 stats(List) ->
206     case List of
207         [X] ->
208             {X, 0};
209         _ ->
210             {mean(List), var(List, mean(List))}
211     end.
212
213 mean(List) ->
214     %erlang:display(List),
215     lineSum(List) / length(List).
216
217 var(List, Mean) ->
218     lineSum(lists:map(fun(Elem) -> (Elem - Mean) * (Elem - Mean)
        end, List)) / length(List).
219
220 lineSum([H | T]) ->
221     lineSum(T, H).
222
223 lineSum(List, Acc) ->
224     case List of
225         [H | T] ->
226             lineSum(T, Acc + H);
227         [] ->
228             Acc
229     end.
```

D.3 benchmark_test_SUITE.erl

```
1 -module(benchmark_test_SUITE).
2
3 -include_lib("stdlib/include/assert.hrl").
```

```
-include_lib("eunit/include/eunit.hrl").

benchmark_test() ->
    erlang:display(erlBlas:get_max_length()).

max_test() ->
    {timeout, 1000, run(20, 40, [])}.

run(80, 0, _) ->
    ok;
run(N, 0, Acc) ->
    erlang:display(N),
    erlang:display(stats(Acc)),
    run(N + 1, 40, []);
run(N, T, Acc) ->
    M = utils:generateRandMat(N, N),
    Mat = numerl:matrix(M),
    M2 = utils:generateRandMat(N, N),
    Mat2 = numerl:matrix(M2),
    C = numerl:zeros(N, N),
    {Time, _} = timer:tc(numerl, dgemm, [1, 1, 1.5, Mat, Mat2, 2.0,
     C]),
    run(N, T - 1, [Time | Acc]).

stats(List) ->
    case List of
        [X] ->
            {X, 0};
        _ ->
            {mean(List), var(List, mean(List))}
    end.

mean(List) ->
    lineSum(List) / length(List).

var(List, Mean) ->
    lineSum(lists:map(fun(Elem) -> (Elem - Mean) * (Elem - Mean)
    end, List)) / length(List).

lineSum([H | T]) ->
    lineSum(T, H).

lineSum(List, Acc) ->
    case List of
        [H | T] ->
            lineSum(T, Acc + H);
        [] ->
            Acc
    end.
```

D.4 daxpy_test_SUITE.erl

```
-module(daxpy_test_SUITE).

-include_lib("stdlib/include/assert.hrl").
```

```
-include_lib("eunit/include/eunit.hrl").

base_test() ->
    A = [[1]],
    B = [[1]],
    C = [[2]],
    ABlock = erlBlas:matrix(A),
    BBlock = erlBlas:matrix(B),
    CBlock = erlBlas:matrix(C),
    erlBlas:daxpy(1.0, ABlock, BBlock),
    ?assert(erlBlas:equals(BBlock, CBlock)).

max_size_blocks_test() ->
    A = [
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    ],

    B = [
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
    ],

    C = [
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
        [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
    ],

    ABlock = erlBlas:matrix(A),
    BBlock = erlBlas:matrix(B),
    CBlock = erlBlas:matrix(C),
    erlBlas:daxpy(1.0, ABlock, BBlock),
    ?assert(erlBlas:equals(BBlock, CBlock)).
```

```
61
62 float_test() ->
63     A = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
       9.58, 10.013, 69.42]],
64
65     B = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
       9.58, 10.013, 69.42]],
66
67     ABlock = erlBlas:matrix(A),
68     BBlock = erlBlas:matrix(B),
69     erlBlas:daxpy(1.0, ABlock, BBlock),
70     ANum = numerl:matrix(A),
71     BNum = numerl:matrix(B),
72     numerl:daxpy(1.0, ANum, BNum),
73     Expected = numerl:mtfl(BNum),
74     Actual = erlBlas:toErl(BBlock),
75     ?assert(mat:'=='(Expected, Actual)).
76
77 small_random_test() ->
78     Max = erlBlas:get_max_length(),
79     erlBlas:set_max_length(50),
80     Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
81     {timeout, 100, fun() ->
82         matrix_test_core(Sizes),
83         erlBlas:set_max_length(Max)
84     end}.
85
86 corner_cases_test_() ->
87     Max = erlBlas:get_max_length(),
88     erlBlas:set_max_length(50),
89     Sizes = [49, 50, 51, 99, 100, 101],
90     {timeout, 100, fun() ->
91         matrix_test_core(Sizes),
92         erlBlas:set_max_length(Max)
93     end}.
94
95 random_test() ->
96     Max = erlBlas:get_max_length(),
97     erlBlas:set_max_length(50),
98     Sizes = [rand:uniform(800) + 500, rand:uniform(800) + 500, rand
       :uniform(800) + 500],
99     {timeout, 100, fun() ->
100         matrix_test_core(Sizes),
101         erlBlas:set_max_length(Max)
102     end}.
103
104 matrix_test_core(Sizes) ->
105     lists:map(
106         fun(M) ->
107             lists:map(
108                 fun(N) ->
109                     random_test_core(M, N),
110                     random_positive_test_core(M, N),
111                     random_negative_test_core(M, N)
112                 end,
113                 Sizes
114             )
```

```
115         end,
116         Sizes
117     ).
118
119 random_test_core(M, N) ->
120     A = utils:generateRandMat(M, N),
121     B = utils:generateRandMat(M, N),
122     ABlock = erlBlas:matrix(A),
123     BBlock = erlBlas:matrix(B),
124     erlBlas:daxpy(1.0, ABlock, BBlock),
125     ANum = numerl:matrix(A),
126     BNum = numerl:matrix(B),
127     numerl:daxpy(1.0, ANum, BNum),
128     Expected = numerl:mtfl(BNum),
129     Actual = erlBlas:toErl(BBlock),
130     ?assert(mat:'=='(Expected, Actual)).
131
132 random_negative_test_core(M, N) ->
133     A = utils:generateRandMat(M, N),
134     B = utils:generateRandMat(M, N),
135     ABlock = erlBlas:matrix(A),
136     BBlock = erlBlas:matrix(B),
137     erlBlas:daxpy(-2.5, ABlock, BBlock),
138     ANum = numerl:matrix(A),
139     BNum = numerl:matrix(B),
140     numerl:daxpy(-2.5, ANum, BNum),
141     Expected = numerl:mtfl(BNum),
142     Actual = erlBlas:toErl(BBlock),
143     ?assert(mat:'=='(Expected, Actual)).
144
145 random_positive_test_core(M, N) ->
146     A = utils:generateRandMat(M, N),
147     B = utils:generateRandMat(M, N),
148     ABlock = erlBlas:matrix(A),
149     BBlock = erlBlas:matrix(B),
150     erlBlas:daxpy(2.5, ABlock, BBlock),
151     ANum = numerl:matrix(A),
152     BNum = numerl:matrix(B),
153     numerl:daxpy(2.5, ANum, BNum),
154     Expected = numerl:mtfl(BNum),
155     Actual = erlBlas:toErl(BBlock),
156     ?assert(mat:'=='(Expected, Actual)).
```

D.5 dgemm_perf_test_SUITE.erl

```
1 -module(dgemm_perf_test_SUITE).
2
3 -include_lib("stdlib/include/assert.hrl").
4 -include_lib("eunit/include/eunit.hrl").
5
6 -import(mat, ['*'/2, '*  '/2, tr/1, '+'/2, '-'/2, '=='/2, eval/1]).
7
8 performance_test_() ->
9     {timeout, 100, fun() ->
10         performance(),
```

```
11          performance_conc()
12      end}.
13
14  performance_conc() ->
15      timer:sleep(100),
16      %dgemm_conc_exec_time(10,100),
17      Sizes = [10, 50, 100, 200, 350, 500],
18      Results =
19          lists:map(
20              fun(Size) ->
21                  Times = dgemm_conc_exec_time(10, Size),
22                  stats(Times)
23              end,
24              Sizes
25          ),
26      erlang:display({conc, Results}).
27
28  dgemm_conc_exec_time(N, Size) ->
29      M1 = utils:generateRandMat(Size, Size),
30      M2 = utils:generateRandMat(Size, Size),
31      M3 = utils:generateRandMat(Size, Size),
32      Mat1 = erlBlas:matrix(M1),
33      Mat2 = erlBlas:matrix(M2),
34      Mat3 = erlBlas:matrix(M3),
35      {Time, _} = timer:tc(erlBlas, dgemm, [false, false, 1.0, Mat1,
        Mat2, 1.0, Mat3]),
36      if
37          N == 1 ->
38              [Time];
39          true ->
40              [Time | dgemm_conc_exec_time(N - 1, Size)]
41      end.
42
43  performance() ->
44      timer:sleep(100),
45      %dgemm_exec_time(10,100),
46      Sizes = [10, 50, 100, 200, 350, 500],
47      Results =
48          lists:map(
49              fun(Size) ->
50                  Times = dgemm_exec_time(10, Size),
51                  stats(Times)
52              end,
53              Sizes
54          ),
55      erlang:display({seq, Results}).
56
57  dgemm_exec_time(N, Size) ->
58      M1 = utils:generateRandMat(Size, Size),
59      M2 = utils:generateRandMat(Size, Size),
60      M3 = utils:generateRandMat(Size, Size),
61      Mat1 = erlBlas:matrix(M1),
62      Mat2 = erlBlas:matrix(M2),
63      Mat3 = erlBlas:matrix(M3),
64      {Time, _} = timer:tc(sequential, dgemm, [false, false, 1.0,
        Mat1, Mat2, 1.0, Mat3]),
65      if
```

```
66          N == 1 ->
67              [Time];
68          true ->
69              [Time | dgemm_exec_time(N - 1, Size)]
70      end.
71
72  stats(List) ->
73      case List of
74          [X] ->
75              {X, 0};
76          _ ->
77              {mean(List), var(List, mean(List))}
78      end.
79
80  mean(List) ->
81      %erlang:display(List),
82      lineSum(List) / length(List).
83
84  var(List, Mean) ->
85      lineSum(lists:map(fun(Elem) -> (Elem - Mean) * (Elem - Mean)
        end, List)) / length(List).
86
87  lineSum([H | T]) ->
88      lineSum(T, H).
89
90  lineSum(List, Acc) ->
91      case List of
92          [H | T] ->
93              lineSum(T, Acc + H);
94          [] ->
95              Acc
96      end.
```

D.6 dgemm_test_SUITE.erl

```
1  -module(dgemm_test_SUITE).
2
3  -include_lib("stdlib/include/assert.hrl").
4  -include_lib("eunit/include/eunit.hrl").
5
6  -import(mat, ['*'/2, '*  '/2, tr/1, '+'/2, '-'/2, '=='/2, eval/1]).
7  -import(erlBlas, [dgemm/7]).
8
9  base_test() ->
10     A = [[1]],
11     B = [[1]],
12     C = [[1]],
13     R = [[2]],
14     ABlock = erlBlas:matrix(A),
15     BBlock = erlBlas:matrix(B),
16     CBlock = erlBlas:matrix(C),
17     RBlock = erlBlas:matrix(R),
18     erlBlas:dgemm(false, true, 1.0, ABlock, BBlock, 1.0, CBlock),
19     ?assert(erlBlas:equals(RBlock, CBlock)).
20
```

```
21 max_size_blocks_test() ->
22     A = [
23         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
24         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
25         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
26         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
27         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
28         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
29         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
30         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
31         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
32         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
33     ],
34
35     B = [
36         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
37         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
38         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
39         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
40         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
41         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
42         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
43         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
44         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
45         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
46     ],
47
48     C = [
49         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
50         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
51         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
52         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
53         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
54         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
55         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
56         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
57         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
58         [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100]
59     ],
60
61     ABlock = erlBlas:matrix(A),
62     BBlock = erlBlas:matrix(B),
63     CBlock = erlBlas:zeros(10, 10),
64     RBlock = erlBlas:matrix(C),
65     erlBlas:dgemm(false, false, 1, ABlock, BBlock, 0, CBlock),
66     ?assert(erlBlas:equals(RBlock, CBlock)).
67
68 float_test() ->
69     A = [[-1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
       -9.58, 10.013, 69.42]],
70
71     B = [
72         [1.2, 2.5],
73         [3.6, 4.7],
74         [-5.69, 42.69],
75         [6.24, -7.77],
76         [8.42, 9.58],
```

```
77          [-10.013, 69.42]
78      ],
79
80      ABlock = erlBlas:matrix(A),
81      BBlock = erlBlas:matrix(B),
82      CBlock = erlBlas:zeros(2, 2),
83      erlBlas:dgemm(false, false, 1, ABlock, BBlock, 0, CBlock),
84      ANum = numerl:matrix(A),
85      BNum = numerl:matrix(B),
86      CNum = numerl:zeros(2, 2),
87      numerl:dgemm(0, 0, 1, ANum, BNum, 0, CNum),
88      Expected = numerl:mtfl(CNum),
89      Actual = erlBlas:toErl(CBlock),
90      ?assert(mat:'=='(Expected, Actual)).
91
92  small_random_test() ->
93      Max = erlBlas:get_max_length(),
94      erlBlas:set_max_length(50),
95      Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
96      {timeout, 100, fun() ->
97          matrix_test_core(Sizes),
98          erlBlas:set_max_length(Max)
99      end}.
100
101 corner_cases_test_() ->
102     Max = erlBlas:get_max_length(),
103     erlBlas:set_max_length(50),
104     Sizes = [49, 50, 51, 99, 100, 101],
105     {timeout, 100, fun() ->
106         matrix_test_core(Sizes),
107         erlBlas:set_max_length(Max)
108     end}.
109
110 random_test() ->
111     Max = erlBlas:get_max_length(),
112     erlBlas:set_max_length(50),
113     Sizes = [rand:uniform(800) + 200, rand:uniform(800) + 200, rand
        :uniform(800) + 200],
114     {timeout, 100, fun() ->
115         matrix_test_core(Sizes),
116         erlBlas:set_max_length(Max)
117     end}.
118
119 matrix_test_core(Sizes) ->
120     lists:map(
121         fun(K) ->
122             lists:map(
123                 fun(M) ->
124                     lists:map(
125                         fun(N) ->
126                             matrix_test_basic(K, M, N),
127                             matrix_test_transpose(K, M, N),
128                             matrix_test_transpose_and_scale(K, M, N
        )
129                         end,
130                         Sizes
131                     )
```

```
132                 end,
133                 Sizes
134             )
135         end,
136         Sizes
137     ).
138
139 matrix_test_basic(K, M, N) ->
140     A = generateRandMat(K, M),
141     B = generateRandMat(M, N),
142     C = generateRandMat(K, N),
143     ANumerl = erlBlas:matrix(A),
144     BNumerl = erlBlas:matrix(B),
145     CNumerl = erlBlas:matrix(C),
146     erlBlas:dgemm(false, false, 1, ANumerl, BNumerl, 1, CNumerl),
147     ?assert('=='(erlBlas:toErl(CNumerl), eval([A, '*', B, '+', C]))
        ).
148
149 matrix_test_transpose(K, M, N) ->
150     A = generateRandMat(M, K),
151     B = generateRandMat(M, N),
152     C = generateRandMat(K, N),
153     ANumerl = erlBlas:matrix(A),
154     BNumerl = erlBlas:matrix(B),
155     CNumerl = erlBlas:matrix(C),
156     erlBlas:dgemm(true, false, 1, ANumerl, BNumerl, 1, CNumerl),
157     ?assert('=='(erlBlas:toErl(CNumerl), eval([tr(A), '*', B, '+',
        C]))),
158     A2 = generateRandMat(K, M),
159     B2 = generateRandMat(N, M),
160     C2 = generateRandMat(K, N),
161     ANumerl2 = erlBlas:matrix(A2),
162     BNumerl2 = erlBlas:matrix(B2),
163     CNumerl2 = erlBlas:matrix(C2),
164     erlBlas:dgemm(false, true, 1, ANumerl2, BNumerl2, 1, CNumerl2),
165     ?assert('=='(erlBlas:toErl(CNumerl2), eval([A2, '*', tr(B2), '+
        ', C2]))),
166     A3 = generateRandMat(M, K),
167     B3 = generateRandMat(N, M),
168     C3 = generateRandMat(K, N),
169     ANumerl3 = erlBlas:matrix(A3),
170     BNumerl3 = erlBlas:matrix(B3),
171     CNumerl3 = erlBlas:matrix(C3),
172     erlBlas:dgemm(true, true, 1, ANumerl3, BNumerl3, 1, CNumerl3),
173     ?assert('=='(erlBlas:toErl(CNumerl3), eval([tr(A3), '*', tr(B3)
        , '+', C3]))).
174
175 matrix_test_transpose_and_scale(K, M, N) ->
176     A = generateRandMat(M, K),
177     B = generateRandMat(M, N),
178     C = generateRandMat(K, N),
179     ANumerl = erlBlas:matrix(A),
180     BNumerl = erlBlas:matrix(B),
181     CNumerl = erlBlas:matrix(C),
182     Num1 = rand:uniform(10) / 9,
183     erlBlas:dgemm(true, false, Num1, ANumerl, BNumerl, 1, CNumerl),
184     ?assert('=='(erlBlas:toErl(CNumerl), eval([Num1, '*', tr(A), '*
```

```
    ', B, '+', C]))),
185     A2 = generateRandMat(K, M),
186     B2 = generateRandMat(N, M),
187     C2 = generateRandMat(K, N),
188     ANumerl2 = erlBlas:matrix(A2),
189     BNumerl2 = erlBlas:matrix(B2),
190     CNumerl2 = erlBlas:matrix(C2),
191     Num2 = rand:uniform(10) / 9,
192     Num3 = rand:uniform(10) / 9,
193     erlBlas:dgemm(false, true, Num2, ANumerl2, BNumerl2, Num3,
        CNumerl2),
194     Afois = '*'(Num2, A2),
195     Cfois = '*'(Num3, C2),
196     AB = '*'(Afois, tr(B2)),
197     Result = '+'(AB, Cfois),
198     ?assert('=='(erlBlas:toErl(CNumerl2), Result)),
199     A3 = generateRandMat(M, K),
200     B3 = generateRandMat(N, M),
201     C3 = generateRandMat(K, N),
202     ANumerl3 = erlBlas:matrix(A3),
203     BNumerl3 = erlBlas:matrix(B3),
204     CNumerl3 = erlBlas:matrix(C3),
205     Num4 = rand:uniform(10) / 9,
206     Cfois2 = '*'(Num4, C3),
207     AB2 = '*'(tr(A3), tr(B3)),
208     Result2 = '+'(AB2, Cfois2),
209     erlBlas:dgemm(true, true, 1, ANumerl3, BNumerl3, Num4, CNumerl3
        ),
210     ?assert('=='(erlBlas:toErl(CNumerl3), Result2)).
211
212 generateRandMat(0, _) ->
213     [];
214 generateRandMat(Dim1, Dim2) ->
215     [generateRandVect(Dim2) | generateRandMat(Dim1 - 1, Dim2)].
216
217 generateRandVect(0) ->
218     [];
219 generateRandVect(Dim2) ->
220     [rand:uniform(5) | generateRandVect(Dim2 - 1)].
```

D.7 dscal_test_SUITE.erl

```
1  -module(dscal_test_SUITE).
2
3  -include_lib("stdlib/include/assert.hrl").
4  -include_lib("eunit/include/eunit.hrl").
5
6  base_test() ->
7      A = [[1]],
8      B = [[2]],
9      ABlock = erlBlas:matrix(A),
10     BBlock = erlBlas:matrix(B),
11     erlBlas:dscal(2.0, ABlock),
12     ?assert(erlBlas:equals(ABlock, BBlock)).
13
```

```
14 max_size_blocks_test() ->
15     A = [
16         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
17         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
18         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
19         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
20         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
21         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
22         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
23         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
24         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
25         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
26     ],
27
28     B = [
29         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
30         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
31         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
32         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
33         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
34         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
35         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
36         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
37         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
38         [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
39     ],
40
41     ABlock = erlBlas:matrix(A),
42     BBlock = erlBlas:matrix(B),
43     erlBlas:dscal(2.0, ABlock),
44     ?assert(erlBlas:equals(ABlock, BBlock)).
45
46 float_test() ->
47     A = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
       9.58, 10.013, 69.42]],
48
49     ABlock = erlBlas:matrix(A),
50     erlBlas:dscal(2.3, ABlock),
51     ANum = numerl:matrix(A),
52     numerl:dscal(2.3, ANum),
53     Expected = numerl:mtfl(ANum),
54     Actual = erlBlas:toErl(ABlock),
55     ?assert(mat:'=='(Expected, Actual)).
56
57 small_random_test() ->
58     Max = erlBlas:get_max_length(),
59     erlBlas:set_max_length(50),
60     Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
61     {timeout, 100, fun() ->
62         matrix_test_core(Sizes),
63         erlBlas:set_max_length(Max)
64     end}.
65
66 corner_cases_test_() ->
67     Max = erlBlas:get_max_length(),
68     erlBlas:set_max_length(50),
69     Sizes = [49, 50, 51, 99, 100, 101],
```

```
70      {timeout, 100, fun() ->
71          matrix_test_core(Sizes),
72          erlBlas:set_max_length(Max)
73      end}.
74
75  random_test() ->
76      Max = erlBlas:get_max_length(),
77      erlBlas:set_max_length(50),
78      Sizes = [rand:uniform(800) + 500, rand:uniform(800) + 500, rand
        :uniform(800) + 500],
79      {timeout, 100, fun() ->
80          matrix_test_core(Sizes),
81          erlBlas:set_max_length(Max)
82      end}.
83
84  matrix_test_core(Sizes) ->
85      lists:map(
86          fun(M) ->
87              lists:map(
88                  fun(N) ->
89                      random_test_core(M, N),
90                      random_positive_test_core(M, N),
91                      random_negative_test_core(M, N)
92                  end,
93                  Sizes
94              )
95          end,
96          Sizes
97      ).
98
99  random_test_core(M, N) ->
100     A = utils:generateRandMat(M, N),
101     ABlock = erlBlas:matrix(A),
102     erlBlas:dscal(1.0, ABlock),
103     Expected = A,
104     Actual = erlBlas:toErl(ABlock),
105     ?assert(mat:'=='(Expected, Actual)).
106
107 random_negative_test_core(M, N) ->
108     A = utils:generateRandMat(M, N),
109     ABlock = erlBlas:matrix(A),
110     erlBlas:dscal(-2.5, ABlock),
111     ANum = numerl:matrix(A),
112     numerl:dscal(-2.5, ANum),
113     Expected = numerl:mtfl(ANum),
114     Actual = erlBlas:toErl(ABlock),
115     ?assert(mat:'=='(Expected, Actual)).
116
117 random_positive_test_core(M, N) ->
118     A = utils:generateRandMat(M, N),
119     ABlock = erlBlas:matrix(A),
120     erlBlas:dscal(2.5, ABlock),
121     ANum = numerl:matrix(A),
122     numerl:dscal(2.5, ANum),
123     Expected = numerl:mtfl(ANum),
124     Actual = erlBlas:toErl(ABlock),
125     ?assert(mat:'=='(Expected, Actual)).
```

D.8 eye_test_SUITE.erl

```
-module(eye_test_SUITE).

-include_lib("stdlib/include/assert.hrl").
-include_lib("eunit/include/eunit.hrl").

% test done for MAX_LENGTH = 5

check_matrix(M1, M2) ->
    lists:zipwith(
        fun(Row1, Row2) ->
            lists:zipwith(
                fun(Elem1, Elem2) -> ?assert(numerl:equals(Elem1, Elem2)) end,
                Row1,
                Row2
            )
        end,
        M1,
        M2
    ).

base_test() ->
    Max = erlBlas:get_max_length(),
    erlBlas:set_max_length(5),
    Num = numerl:eye(1),
    Block = erlBlas:eye(1),
    % also checks the block matrix format is correct
    [[Binary]] = Block,
    ?assert(numerl:equals(Num, Binary)),
    erlBlas:set_max_length(Max).

small_test() ->
    Max = erlBlas:get_max_length(),
    erlBlas:set_max_length(5),
    Num = numerl:eye(2),
    Block = erlBlas:eye(2),
    [[Binary]] = Block,
    ?assert(numerl:equals(Num, Binary)),
    erlBlas:set_max_length(Max).

max_size_blocks_test() ->
    Max = erlBlas:get_max_length(),
    erlBlas:set_max_length(5),
    Block = erlBlas:eye(10),
    BlockResult =
        [[numerl:eye(5), numerl:zeros(5, 5)], [numerl:zeros(5, 5), numerl:eye(5)]],
    check_matrix(Block, BlockResult),
    erlBlas:set_max_length(Max).

rest_test() ->
    Max = erlBlas:get_max_length(),
    erlBlas:set_max_length(5),
    Block = erlBlas:eye(7),
    BlockResult =
```

```
54          [[numerl:eye(2), numerl:zeros(2, 5)], [numerl:zeros(5, 2),
        numerl:eye(5)]],
55      check_matrix(Block, BlockResult),
56      erlBlas:set_max_length(Max).
```

D.9 inv_test_SUITE.erl

```
1  -module(inv_test_SUITE).
2
3  -include_lib("stdlib/include/assert.hrl").
4  -include_lib("eunit/include/eunit.hrl").
5
6  base_test() ->
7      M = [[1]],
8      Num = numerl:matrix(M),
9      Block = erlBlas:matrix(M),
10     Res = numerl:inv(Num),
11     Inv = erlBlas:inv(Block),
12     ErlRes = numerl:mtfl(Res),
13     ErlInv = erlBlas:toErl(Inv),
14     ?assert(mat:'=='(ErlInv, ErlRes)).
15
16 max_size_blocks_test() ->
17     M = [
18          [1, 27, 31, 45, 54, 6, 7, 8, 9, 10],
19          [10, 25, 38, 4, 5, 6, 7, 8, 9, 11],
20          [1, 20, 34, 4, 58, 6, 72, 78, 98, 12],
21          [11, 22, 32, 4, 5, 60, 7, 8, 9, 13],
22          [1, 2, 37, 4, 51, 61, 71, 8, 97, 14],
23          [10, 22, 30, 4, 5, 68, 7, 8, 89, 15],
24          [11, 2, 34, 4, 5, 67, 7, 80, 9, 16],
25          [1, 25, 53, 47, 5, 6, 79, 8, 97, 17],
26          [18, 27, 3, 48, 5, 6, 74, 84, 9, 18],
27          [1, 20, 36, 4, 5, 65, 71, 87, 9, 19]
28     ],
29     Num = numerl:matrix(M),
30     NumResult = numerl:inv(Num),
31     ErlNum = numerl:mtfl(NumResult),
32     Block = erlBlas:matrix(M),
33     BlockResult = erlBlas:inv(Block),
34     ErlBlock = erlBlas:toErl(BlockResult),
35     ?assert(mat:'=='(ErlBlock, ErlNum)).
36
37 %% M is invertible but not its upper left block
38 univertible_upper_left_test() ->
39     Max = erlBlas:get_max_length(),
40     erlBlas:set_max_length(2),
41     M = [
42          [1, 2, 3, 4],
43          [1, 2, 7, 13],
44          [9, 10, 11, 12],
45          [13, 14, 15, 16]
46     ],
47     MatInv = mat:inv(M),
48     Block = erlBlas:matrix(M),
```

```
49      Inv = erlBlas:inv(Block),
50      ErlInv = erlBlas:toErl(Inv),
51      ?assert(mat:'=='(ErlInv, MatInv)),
52      erlBlas:set_max_length(Max).
53
54  small_random_test() ->
55      Max = erlBlas:get_max_length(),
56      erlBlas:set_max_length(50),
57      Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
58      {timeout, 100, fun() ->
59          matrix_test_core(Sizes),
60          erlBlas:set_max_length(Max)
61      end}.
62
63  corner_cases_test_() ->
64      Max = erlBlas:get_max_length(),
65      erlBlas:set_max_length(17),
66      Sizes = [16, 17, 18, 33, 34, 35],
67      {timeout, 100, fun() ->
68          matrix_test_core(Sizes),
69          erlBlas:set_max_length(Max)
70      end}.
71
72  random_test() ->
73      Max = erlBlas:get_max_length(),
74      erlBlas:set_max_length(17),
75      Sizes = [rand:uniform(800) + 500, rand:uniform(800) + 500, rand
        :uniform(800) + 500],
76      {timeout, 100, fun() ->
77          matrix_test_core(Sizes),
78          erlBlas:set_max_length(Max)
79      end}.
80
81  matrix_test_core(Sizes) ->
82      lists:map(
83          fun(N) ->
84              random_test_core(N)
85          end,
86          Sizes
87      ).
88
89  random_test_core(N) ->
90      M = utils:generateRandMat(N, N),
91      Block = erlBlas:matrix(M),
92      BlockResult = erlBlas:inv(Block),
93      ErlBlock = erlBlas:toErl(BlockResult),
94      Res = mat:inv(M),
95      ?assert(mat:'=='(ErlBlock, Res)).
```

D.10 matrix_test_SUITE.erl

```
1  -module(matrix_test_SUITE).
2
3  -include_lib("stdlib/include/assert.hrl").
4  -include_lib("eunit/include/eunit.hrl").
```

```
5
6  % test done for MAX_LENGTH = 5
7
8  check_matrix(M1, M2) ->
9      lists:zipwith(
10         fun(Row1, Row2) ->
11             lists:zipwith(
12                 fun(Elem1, Elem2) -> ?assert(numerl:equals(Elem1,
       Elem2)) end,
13                 Row1,
14                 Row2
15             )
16         end,
17         M1,
18         M2
19     ).
20
21 base_test() ->
22     Max = erlBlas:get_max_length(),
23     erlBlas:set_max_length(5),
24     M = [[1]],
25     Num = numerl:matrix(M),
26     Block = erlBlas:matrix(M),
27     % also checks the block matrix format is correct
28     [[Binary]] = Block,
29     ?assert(numerl:equals(Num, Binary)),
30     erlBlas:set_max_length(Max).
31
32 small_test() ->
33     Max = erlBlas:get_max_length(),
34     erlBlas:set_max_length(5),
35     M = [[1, 2], [3, 4]],
36     Num = numerl:matrix(M),
37     Block = erlBlas:matrix(M),
38     [[Binary]] = Block,
39     ?assert(numerl:equals(Num, Binary)),
40     erlBlas:set_max_length(Max).
41
42 line_test() ->
43     Max = erlBlas:get_max_length(),
44     erlBlas:set_max_length(5),
45     M = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]],
46     Sub1 = [[1, 2, 3, 4, 5]],
47     Sub2 = [[6, 7, 8, 9, 10]],
48     Block = erlBlas:matrix(M),
49     BlockResult = [[numerl:matrix(Sub1), numerl:matrix(Sub2)]],
50     check_matrix(Block, BlockResult),
51     erlBlas:set_max_length(Max).
52
53 column_test() ->
54     Max = erlBlas:get_max_length(),
55     erlBlas:set_max_length(5),
56     M = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]],
57     Sub1 = [[1], [2], [3], [4], [5]],
58     Sub2 = [[6], [7], [8], [9], [10]],
59     Block = erlBlas:matrix(M),
60     BlockResult = [[numerl:matrix(Sub1)], [numerl:matrix(Sub2)]],
```

```
61      check_matrix(Block, BlockResult),
62      erlBlas:set_max_length(Max).
63
64  max_size_blocks_test() ->
65      Max = erlBlas:get_max_length(),
66      erlBlas:set_max_length(5),
67      M = [
68          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
69          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
70          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
71          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
72          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
73          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
74          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
75          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
76          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
77          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
78      ],
79      Sub1 =
80          [[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2,
    3, 4, 5], [1, 2, 3, 4, 5]],
81      Sub2 =
82          [
83              [6, 7, 8, 9, 10],
84              [6, 7, 8, 9, 10],
85              [6, 7, 8, 9, 10],
86              [6, 7, 8, 9, 10],
87              [6, 7, 8, 9, 10]
88          ],
89      Block = erlBlas:matrix(M),
90      BlockResult =
91          [[numerl:matrix(Sub1), numerl:matrix(Sub2)], [numerl:matrix
    (Sub1), numerl:matrix(Sub2)]],
92      check_matrix(Block, BlockResult),
93      erlBlas:set_max_length(Max).
94
95  rest_test() ->
96      Max = erlBlas:get_max_length(),
97      erlBlas:set_max_length(5),
98      M = [
99          [1, 2, 3, 4, 5, 6, 7],
100         [1, 2, 3, 4, 5, 6, 7],
101         [1, 2, 3, 4, 5, 6, 7],
102         [1, 2, 3, 4, 5, 6, 7],
103         [1, 2, 3, 4, 5, 6, 7],
104         [1, 2, 3, 4, 5, 6, 7],
105         [1, 2, 3, 4, 5, 6, 7]
106     ],
107     Sub1 = [[1, 2], [1, 2]],
108     Sub2 = [[3, 4, 5, 6, 7], [3, 4, 5, 6, 7]],
109     Sub3 = [[1, 2], [1, 2], [1, 2], [1, 2], [1, 2]],
110     Sub4 =
111         [[3, 4, 5, 6, 7], [3, 4, 5, 6, 7], [3, 4, 5, 6, 7], [3, 4,
    5, 6, 7], [3, 4, 5, 6, 7]],
112     Block = erlBlas:matrix(M),
113     BlockResult =
114         [[numerl:matrix(Sub1), numerl:matrix(Sub2)], [numerl:matrix
```

```
    (Sub3), numerl:matrix(Sub4)]],
    check_matrix(Block, BlockResult),
    erlBlas:set_max_length(Max).
```

D.11 mult_conc_test_SUITE.erl

```
-module(mult_conc_test_SUITE).

-include_lib("stdlib/include/assert.hrl").
-include_lib("eunit/include/eunit.hrl").

performance_test_() ->
    Max = erlBlas:get_max_length(),
    {timeout, 1000, fun() ->
        MaxLengths = [50],
        lists:map(
            fun(MaxLength) ->
                erlBlas:set_max_length(MaxLength),
                erlang:display({max_length, erlBlas:get_max_length
    ()}),
                performance(),
                performance_conc(),
                erlBlas:set_max_length(Max)
            end,
            MaxLengths
        )
    end}.

performance_conc() ->
    timer:sleep(100),
    %mult_conc_exec_time(10,100),

    %,1000,2000],
    Sizes = [10, 50, 100, 200, 350, 500],
    Results =
        lists:map(
            fun(Size) ->
                Times = mult_conc_exec_time(10, Size),
                stats(Times)
            end,
            Sizes
        ),
    erlang:display({conc, Results}).

mult_conc_exec_time(N, Size) ->
    M1 = utils:generateRandMat(Size, Size),
    M2 = utils:generateRandMat(Size, Size),
    Mat1 = erlBlas:matrix(M1),
    Mat2 = erlBlas:matrix(M2),
    {Time, _} = timer:tc(erlBlas, mult, [Mat1, Mat2]),
    if
        N == 1 ->
            [Time];
        true ->
            [Time | mult_conc_exec_time(N - 1, Size)]
```

```
49      end.
50
51  performance() ->
52      timer:sleep(100),
53      %mult_exec_time(10,100),
54
55      %,1000,2000],
56      Sizes = [10, 50, 100, 200, 350, 500],
57      Results =
58          lists:map(
59              fun(Size) ->
60                  Times = mult_exec_time(10, Size),
61                  stats(Times)
62              end,
63              Sizes
64          ),
65      erlang:display({seq, Results}).
66
67  mult_exec_time(N, Size) ->
68      M1 = utils:generateRandMat(Size, Size),
69      M2 = utils:generateRandMat(Size, Size),
70      Mat1 = erlBlas:matrix(M1),
71      Mat2 = erlBlas:matrix(M2),
72      {Time, _} = timer:tc(sequential, mult, [Mat1, Mat2]),
73      if
74          N == 1 ->
75              [Time];
76          true ->
77              [Time | mult_exec_time(N - 1, Size)]
78      end.
79
80  stats(List) ->
81      case List of
82          [X] ->
83              {X, 0};
84          _ ->
85              {mean(List), var(List, mean(List))}
86      end.
87
88  mean(List) ->
89      %erlang:display(List),
90      lineSum(List) / length(List).
91
92  var(List, Mean) ->
93      lineSum(lists:map(fun(Elem) -> (Elem - Mean) * (Elem - Mean)
        end, List)) / length(List).
94
95  lineSum([H | T]) ->
96      lineSum(T, H).
97
98  lineSum(List, Acc) ->
99      case List of
100         [H | T] ->
101             lineSum(T, Acc + H);
102         [] ->
103             Acc
104     end.
```

D.12 mult_erl_test_SUITE.erl

```
-module(mult_erl_test_SUITE).

-include_lib("stdlib/include/assert.hrl").
-include_lib("eunit/include/eunit.hrl").

performance_test_() ->
    {timeout,
     1000,
     fun() ->
        performance_erl(),
        performance_C()
     end}.

performance_C() ->
    timer:sleep(100),
    Sizes = [10, 50, 100, 200, 350, 500],%,1000,2000],
    Results =
        lists:map(fun(Size) ->
                      Times = mult_C_exec_time(20, Size),
                      stats(Times)
                  end,
                  Sizes),
    erlang:display({"C", Results}).

mult_C_exec_time(N, Size) ->
    M1 = utils:generateRandMat(Size, Size),
    M2 = utils:generateRandMat(Size, Size),
    Mat1 = numerl:matrix(M1),
    Mat2 = numerl:matrix(M2),
    {Time, _} = timer:tc(numerl, dot, [Mat1, Mat2]),
    if N == 1 ->
            [Time];
       true ->
            [Time | mult_C_exec_time(N - 1, Size)]
    end.

performance_erl() ->
    timer:sleep(100),
    Sizes = [10, 50, 100, 200, 350, 500],%,1000,2000],
    Results =
        lists:map(fun(Size) ->
                      Times = mult_erl_exec_time(20, Size),
                      stats(Times)
                  end,
                  Sizes),
    erlang:display({erl, Results}).

mult_erl_exec_time(N, Size) ->
    M1 = utils:generateRandMat(Size, Size),
    M2 = utils:generateRandMat(Size, Size),
    {Time, _} = timer:tc(mat, '*', [M1, M2]),
    if N == 1 ->
            [Time];
       true ->
            [Time | mult_erl_exec_time(N - 1, Size)]
```

```
56      end.
57
58  stats(List) ->
59      case List of
60          [X] ->
61              {X, 0};
62          _ ->
63              {mean(List), var(List, mean(List))}
64      end.
65
66  mean(List) ->
67      %erlang:display(List),
68      lineSum(List) / length(List).
69
70  var(List, Mean) ->
71      lineSum(lists:map(fun(Elem) -> (Elem - Mean) * (Elem - Mean)
        end, List)) / length(List).
72
73  lineSum([H | T]) ->
74      lineSum(T, H).
75
76  lineSum(List, Acc) ->
77      case List of
78          [H | T] ->
79              lineSum(T, Acc + H);
80          [] ->
81              Acc
82      end.
```

D.13 mult_test_SUITE.erl

```
1   -module(mult_test_SUITE).
2
3   -include_lib("stdlib/include/assert.hrl").
4   -include_lib("eunit/include/eunit.hrl").
5
6   base_test() ->
7       A = [[1]],
8       B = [[1]],
9       C = [[1]],
10      ABlock = erlBlas:matrix(A),
11      BBlock = erlBlas:matrix(B),
12      CBlock = erlBlas:matrix(C),
13      Res = erlBlas:mult(ABlock, BBlock),
14      ?assert(erlBlas:equals(Res, CBlock)).
15
16  max_size_blocks_test() ->
17      A = [
18          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
19          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
20          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
21          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
22          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
23          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
24          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
```

```
25          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
26          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
27          [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
28      ],
29
30      B = [
31          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
32          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
33          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
34          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
35          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
36          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
37          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
38          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
39          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
40          [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
41      ],
42
43      C = [
44          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
45          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
46          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
47          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
48          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
49          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
50          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
51          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
52          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100],
53          [110, 220, 330, 440, 550, 660, 770, 880, 990, 1100]
54      ],
55
56      ABlock = erlBlas:matrix(A),
57      BBlock = erlBlas:matrix(B),
58      CBlock = erlBlas:matrix(C),
59      Res = erlBlas:mult(ABlock, BBlock),
60      ?assert(erlBlas:equals(Res, CBlock)).
61
62  float_test() ->
63      A = [[-1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
        -9.58, 10.013, 69.42]],
64
65      B = [
66          [1.2, 2.5],
67          [3.6, 4.7],
68          [-5.69, 42.69],
69          [6.24, -7.77],
70          [8.42, 9.58],
71          [-10.013, 69.42]
72      ],
73
74      ABlock = erlBlas:matrix(A),
75      BBlock = erlBlas:matrix(B),
76      Res = erlBlas:mult(ABlock, BBlock),
77      ANum = numerl:matrix(A),
78      BNum = numerl:matrix(B),
79      Conf = numerl:dot(ANum, BNum),
80      Expected = numerl:mtfl(Conf),
```

```
81      Actual = erlBlas:toErl(Res),
82      ?assert(mat:'=='(Expected, Actual)).
83
84  small_random_test() ->
85      Max = erlBlas:get_max_length(),
86      erlBlas:set_max_length(50),
87      Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
88      {timeout, 100, fun() ->
89          matrix_test_core(Sizes),
90          erlBlas:set_max_length(Max)
91      end}.
92
93  corner_cases_test_() ->
94      Max = erlBlas:get_max_length(),
95      erlBlas:set_max_length(50),
96      Sizes = [49, 50, 51, 99, 100, 101],
97      {timeout, 100, fun() ->
98          matrix_test_core(Sizes),
99          erlBlas:set_max_length(Max)
100     end}.
101
102 random_test() ->
103     Max = erlBlas:get_max_length(),
104     erlBlas:set_max_length(50),
105     Sizes = [rand:uniform(800) + 500, rand:uniform(800) + 500, rand
        :uniform(800) + 500],
106     {timeout, 100, fun() ->
107         matrix_test_core(Sizes),
108         erlBlas:set_max_length(Max)
109     end}.
110
111 matrix_test_core(Sizes) ->
112     lists:map(
113         fun(K) ->
114             lists:map(
115                 fun(M) ->
116                     lists:map(
117                         fun(N) ->
118                             random_test_core(K, M, N)
119                         end,
120                         Sizes
121                     )
122                 end,
123                 Sizes
124             )
125         end,
126         Sizes
127     ).
128
129 random_test_core(K, M, N) ->
130     A = utils:generateRandMat(K, M),
131     B = utils:generateRandMat(M, N),
132     ABlock = erlBlas:matrix(A),
133     BBlock = erlBlas:matrix(B),
134     Res = erlBlas:mult(ABlock, BBlock),
135     ANum = numerl:matrix(A),
136     BNum = numerl:matrix(B),
```

```
137     Conf = numerl:dot(ANum, BNum),
138     Expected = numerl:mtfli(Conf),
139     Actual = erlBlas:toErl(Res),
140     ?assert(mat:'=='(Expected, Actual)).
```

D.14 sub_test_SUITE.erl

```
1  -module(sub_test_SUITE).
2
3  -include_lib("stdlib/include/assert.hrl").
4  -include_lib("eunit/include/eunit.hrl").
5
6  base_test() ->
7      A = [[1]],
8      B = [[1]],
9      C = [[0]],
10     ABlock = erlBlas:matrix(A),
11     BBlock = erlBlas:matrix(B),
12     CBlock = erlBlas:matrix(C),
13     Res = erlBlas:sub(ABlock, BBlock),
14     ?assert(erlBlas:equals(Res, CBlock)).
15
16 float_test() ->
17     A = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
       9.58, 10.013, 69.42]],
18
19     B = [[1.2, 2.5, 3.6, 4.7, 5.69, 42.69], [6.24, 7.77, 8.42,
       9.58, 10.013, 69.42]],
20
21     ABlock = erlBlas:matrix(A),
22     BBlock = erlBlas:matrix(B),
23     Res = erlBlas:sub(ABlock, BBlock),
24     ANum = numerl:matrix(A),
25     BNum = numerl:matrix(B),
26     Conf = numerl:sub(ANum, BNum),
27     Expected = numerl:mtfl(Conf),
28     Actual = erlBlas:toErl(Res),
29     ?assert(mat:'=='(Expected, Actual)).
30
31 random_square_test() ->
32     N = 13,
33     A = utils:generateRandMat(N, N),
34     B = utils:generateRandMat(N, N),
35     ABlock = erlBlas:matrix(A),
36     BBlock = erlBlas:matrix(B),
37     Res = erlBlas:sub(ABlock, BBlock),
38     ANum = numerl:matrix(A),
39     BNum = numerl:matrix(B),
40     Conf = numerl:sub(ANum, BNum),
41     Expected = numerl:mtfl(Conf),
42     Actual = erlBlas:toErl(Res),
43     ?assert(mat:'=='(Expected, Actual)).
44
45 random_rectangle_test() ->
46     N = 13,
```

```
47      M = 7,
48      A = utils:generateRandMat(N, M),
49      B = utils:generateRandMat(N, M),
50      ABlock = erlBlas:matrix(A),
51      BBlock = erlBlas:matrix(B),
52      Res = erlBlas:sub(ABlock, BBlock),
53      ANum = numerl:matrix(A),
54      BNum = numerl:matrix(B),
55      Conf = numerl:sub(ANum, BNum),
56      Expected = numerl:mtfl(Conf),
57      Actual = erlBlas:toErl(Res),
58      ?assert(mat:'=='(Expected, Actual)).
59
60  small_random_test() ->
61      Max = erlBlas:get_max_length(),
62      erlBlas:set_max_length(50),
63      Sizes = [rand:uniform(10), rand:uniform(10), rand:uniform(10)],
64      {timeout, 100, fun() ->
65          matrix_test_core(Sizes),
66          erlBlas:set_max_length(Max)
67      end}.
68
69  corner_cases_test_() ->
70      Max = erlBlas:get_max_length(),
71      erlBlas:set_max_length(50),
72      Sizes = [49, 50, 51, 99, 100, 101],
73      {timeout, 100, fun() ->
74          matrix_test_core(Sizes),
75          erlBlas:set_max_length(Max)
76      end}.
77
78  random_test() ->
79      Max = erlBlas:get_max_length(),
80      erlBlas:set_max_length(50),
81      Sizes = [rand:uniform(800) + 500, rand:uniform(800) + 500, rand
        :uniform(800) + 500],
82      {timeout, 100, fun() ->
83          matrix_test_core(Sizes),
84          erlBlas:set_max_length(Max)
85      end}.
86
87  matrix_test_core(Sizes) ->
88      lists:map(
89          fun(M) ->
90              lists:map(
91                  fun(N) ->
92                      random_test_core(M, N)
93                  end,
94                  Sizes
95              )
96          end,
97          Sizes
98      ).
99
100 random_test_core(M, N) ->
101     A = utils:generateRandMat(M, N),
102     B = utils:generateRandMat(M, N),
```

```
103     ABlock = erlBlas:matrix(A),
104     BBlock = erlBlas:matrix(B),
105     Res = erlBlas:sub(ABlock, BBlock),
106     ANum = numerl:matrix(A),
107     BNum = numerl:matrix(B),
108     Conf = numerl:sub(ANum, BNum),
109     Expected = numerl:mtfl(Conf),
110     Actual = erlBlas:toErl(Res),
111     ?assert(mat:'=='(Expected, Actual)).
```

D.15 zeros_test_SUITE.erl

```
1  -module(zeros_test_SUITE).
2
3  -include_lib("stdlib/include/assert.hrl").
4  -include_lib("eunit/include/eunit.hrl").
5
6  % test done for MAX_LENGTH = 5
7
8  check_matrix(M1, M2) ->
9      lists:zipwith(
10         fun(Row1, Row2) ->
11             lists:zipwith(
12                 fun(Elem1, Elem2) -> ?assert(numerl:equals(Elem1,
       Elem2)) end,
13                 Row1,
14                 Row2
15             )
16         end,
17         M1,
18         M2
19     ).
20
21 base_test() ->
22     Max = erlBlas:get_max_length(),
23     erlBlas:set_max_length(5),
24     Num = numerl:zeros(1, 1),
25     Block = erlBlas:zeros(1, 1),
26     % also checks the block matrix format is correct
27     [[Binary]] = Block,
28     ?assert(numerl:equals(Num, Binary)),
29     erlBlas:set_max_length(Max).
30
31 small_test() ->
32     Max = erlBlas:get_max_length(),
33     erlBlas:set_max_length(5),
34     Num = numerl:zeros(2, 2),
35     Block = erlBlas:zeros(2, 2),
36     [[Binary]] = Block,
37     ?assert(numerl:equals(Num, Binary)),
38     erlBlas:set_max_length(Max).
39
40 line_test() ->
41     Max = erlBlas:get_max_length(),
42     erlBlas:set_max_length(5),
```

```
43      Block = erlBlas:zeros(1, 10),
44      BlockResult = [[numerl:zeros(1, 5), numerl:zeros(1, 5)]],
45      check_matrix(Block, BlockResult),
46      erlBlas:set_max_length(Max).
47
48  column_test() ->
49      Max = erlBlas:get_max_length(),
50      erlBlas:set_max_length(5),
51      Block = erlBlas:zeros(10, 1),
52      BlockResult = [[numerl:zeros(5, 1)], [numerl:zeros(5, 1)]],
53      check_matrix(Block, BlockResult),
54      erlBlas:set_max_length(Max).
55
56  max_size_blocks_test() ->
57      Max = erlBlas:get_max_length(),
58      erlBlas:set_max_length(5),
59      Block = erlBlas:zeros(10, 10),
60      BlockResult =
61          [[numerl:zeros(5, 5), numerl:zeros(5, 5)], [numerl:zeros(5,
        5), numerl:zeros(5, 5)]],
62      check_matrix(Block, BlockResult),
63      erlBlas:set_max_length(Max).
64
65  rest_test() ->
66      Max = erlBlas:get_max_length(),
67      erlBlas:set_max_length(5),
68      Block = erlBlas:zeros(7, 7),
69      BlockResult =
70          [[numerl:zeros(2, 2), numerl:zeros(2, 5)], [numerl:zeros(5,
        2), numerl:zeros(5, 5)]],
71      check_matrix(Block, BlockResult),
72      erlBlas:set_max_length(Max).
```

Appendix E

Numerl code

E.1 numerl.erl

```
-module(numerl).
-on_load(init/0).
-export([ eval/1, eye/1, zeros/2, equals/2, add/2, sub/2,mult/2,
    divide/2, matrix/1, rnd_matrix/1, get/3, at/2, mtfli/1, mtfl/1,
     row/2, col/2, transpose/1, inv/1, nrm2/1, vec_dot/2, dot/2,
    daxpy/3, dscal/2, dgemm/7, copy/1, copy_shape/1, get_shape/1]).

%Matrices are represented as such:
%-record(matrix, {n_rows, n_cols, bin}).

init()->
  Dir = case code:priv_dir(numerl) of
              {error, bad_name} ->
                  filename:join(
                    filename:dirname(
                      filename:dirname(
                        code:which(?MODULE))), "priv");
              D -> D
          end,
    SoName = filename:join(Dir, atom_to_list(?MODULE)),
    erlang:load_nif(SoName, 0).

%Creates a random matrix.
rnd_matrix(N)->
    L = [[rand:uniform(20) || _ <- lists:seq(1,N) ] || _ <- lists:
    seq(1,N)],
    matrix(L).

%Combine multiple functions.
eval([L,O,R|T])->
    F = fun numerl:O/2,
    eval([F(L,R) |T]);
eval([Res])->
    Res.

%%Creates a matrix.
```

```
33 %List: List of doubles, of length N.
34 %Return: a matrix of dimension MxN, containing the data.
35 matrix(_) ->
36     nif_not_loaded.
37
38 %%Returns the Nth value contained within Matrix.
39 at(_Matrix,_Nth)->
40     nif_not_loaded.
41
42 %%Returns the matrix as a flattened list of ints.
43 mtfli(_mtrix)->
44     nif_not_loaded.
45
46 %%Returns the matrix as a flattened list of doubles.
47 mtfl(_mtrix)->
48     nif_not_loaded.
49
50 %%Returns a value from a matrix.
51 get(_,_,_) ->
52     nif_not_loaded.
53
54 %%Returns requested row.
55 row(_,_) ->
56     nif_not_loaded.
57
58
59 %%Returns requested col.
60 col(_,_) ->
61     nif_not_loaded.
62
63
64 %%Equality test between matrixes.
65 equals(_, _) ->
66     nif_not_loaded.
67
68
69 %%Addition of matrix.
70 add(_, _) ->
71     nif_not_loaded.
72
73
74 %%Subtraction of matrix.
75 sub(_, _) ->
76     nif_not_loaded.
77
78
79 %% Matrix multiplication.
80 mult(A,B) when is_number(B) -> '*_num'(A,B);
81 mult(A,B) -> '*_matrix'(A,B).
82
83 '*_num'(_,_)->
84     nif_not_loaded.
85
86 '*_matrix'(_, _)->
87     nif_not_loaded.
88
89 %Matrix division by a number
```

```
90 divide(_,_)->
91     nif_not_loaded.
92
93
94 %% build a null matrix of size NxM
95 zeros(_, _) ->
96     nif_not_loaded.
97
98 %%Returns an Identity matrix NxN.
99 eye(_)->
100     nif_not_loaded.
101
102 %Returns the transpose of the given square matrix.
103 transpose(_)->
104     nif_not_loaded.
105
106 %Returns the inverse of asked square matrix.
107 inv(_)->
108     nif_not_loaded.
109
110
111 %-------CBLAS---------
112
113 %nrm2
114 %Calculates the squared root of the sum of the squared contents.
115 nrm2(_)->
116     nif_not_loaded.
117
118 % : dot product of two vectors
119 % Arguments: vector x, vector y.
120 %   x and y are matrices
121 % Returns the dot product of all the coordinates of X,Y.
122 vec_dot(_, _)->
123     nif_not_loaded.
124
125 % dgemm: A dot B
126 % Arguments: Matrix A, Matrix B.
127 %   alpha, beta: numbers (float or ints) used as doubles.
128 %   A,B,C: matrices.
129 % Returns the matrice resulting of the operations alpha * A * B +
      beta * C.
130 dot(_,_)->
131     nif_not_loaded.
132
133 daxpy(_,_,_)->
134     nif_not_loaded.
135
136 dscal(_,_) ->
137     nif_not_loaded.
138
139 dgemm(_,_,_,_,_,_,_,_,_) ->
140     nif_not_loaded.
141
142 copy(_) ->
143     nif_not_loaded.
144
145 copy_shape(_) ->
```

```
146     nif_not_loaded.
147
148 get_shape(_) ->
149     nif_not_loaded.
```

E.2 numerl.c

```
1  #include <erl_nif.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <math.h>
5  #include <cblas.h>
6
7
8  /*
9  ------------------------------------------------------------|
10      ____________________________                            |
11     |          LAPACKE          |                           |
12     |___________________________|                           |
13 ------------------------------------------------------------|
14 */
15
16 void dgetrf_(int* M, int *N, double* A, int* lda, int* IPIV, int*
       INFO);
17 void dgetri_(int* N, double* A, int* lda, int* IPIV, double* WORK,
       int* lwork, int* INFO);
18
19
20 /*
21 ------------------------------------------------------------|
22      ____________________________                            |
23     |          INIT FC          |                           |
24     |___________________________|                           |
25 ------------------------------------------------------------|
26 */
27
28
29 ERL_NIF_TERM atom_nok;
30 ERL_NIF_TERM atom_true;
31 ERL_NIF_TERM atom_false;
32 ERL_NIF_TERM atom_matrix;
33 ErlNifResourceType *mat_resource;
34 ErlNifResourceType *doubleArray;
35
36 ErlNifResourceType *MULT_YIELDING_ARG = NULL;
37
38 int load(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM load_info){
39     atom_nok = enif_make_atom(env, "nok\0");
40     atom_true = enif_make_atom(env, "true\0");
41     atom_false = enif_make_atom(env, "false\0");
42     atom_matrix = enif_make_atom(env, "matrix\0");
43
44     mat_resource = enif_open_resource_type(env, "numerl", "mat
       resource", NULL, ERL_NIF_RT_CREATE, NULL);
```

```
45      doubleArray = enif_open_resource_type(env, "numerl", "double
        array", NULL, ERL_NIF_RT_CREATE, NULL);
46      return 0;
47  }
48
49  int upgrade(ErlNifEnv* caller_env, void** priv_data, void**
        old_priv_data, ERL_NIF_TERM load_info){
50      return 0;
51  }
52
53  //Gives easier access to an ErlangBinary containing a matrix.
54  typedef struct{
55      int n_rows;
56      int n_cols;
57
58      //Content of the matrix, in row major format.
59      double* content;
60
61      //Erlang binary containing the matrix.
62      //ErlNifBinary bin;
63  } Matrix;
64
65
66  //Access asked coordinates of matrix
67  double* matrix_at(int col, int row, Matrix *m){
68      return &(m->content[row*m->n_cols + col]);
69  }
70
71  //Allocates memory space of for matrix of dimensions n_rows, n_cols
        .
72  //The matrix_content can be modified, until a call to array_to_erl.
73  //Matrix content is stored in row major format.
74  static Matrix *matrix_alloc(int n_rows, int n_cols){
75
76      Matrix *matrix = enif_alloc_resource(mat_resource, sizeof(
        double *)+2*sizeof(int));
77      double *content = enif_alloc_resource(doubleArray, n_rows*
        n_cols*sizeof(double));
78
79      matrix->n_cols = n_cols;
80      matrix->n_rows = n_rows;
81      matrix->content = content;
82
83      return matrix;
84  }
85
86  //Creates a duplicate of a matrix.
87  //This duplicate can be modified until uploaded.
88  Matrix *matrix_dup(Matrix *m){
89      Matrix *d = matrix_alloc(m->n_rows, m->n_cols);
90      memcpy(d->content, m->content, m->n_rows*m->n_cols*sizeof(
        double));
91      return d;
92  }
93
94  //Free an allocated matrix that was not sent back to Erlang.
95  void matrix_free(Matrix *m){
```

```
96      enif_release_resource(m);
97  }
98
99  //Constructs a matrix Erlang term.
100 //No modifications can be made afterwards to the matrix.
101 ERL_NIF_TERM matrix_to_erl(ErlNifEnv* env, Matrix *m){
102     ERL_NIF_TERM term = enif_make_tuple2(env,enif_make_resource(env
        , m), enif_make_resource(env, m->content));
103     enif_release_resource(m);
104     return term;
105 }
106
107 int enif_is_matrix(ErlNifEnv* env, ERL_NIF_TERM term){
108     Matrix *dest;
109     int arity;
110     ERL_NIF_TERM *content;
111     if(!enif_get_tuple(env, term, &arity, &content))
112         return 0;
113     if(!enif_get_resource(env,content[0],mat_resource,&dest))
114         return 0;
115     if(!enif_get_resource(env,content[1],doubleArray,&dest->content
        ))
116         return 0;
117     return 1;
118 }
119
120 //Reads an Erlang term as a matrix.
121 //As such, no modifications can be made to the read matrix.
122 //Returns true if it was possible to read a matrix, false otherwise
123 int enif_get_matrix(ErlNifEnv* env, ERL_NIF_TERM term, Matrix **
        dest){
124     int arity;
125     ERL_NIF_TERM *content;
126     if(!enif_get_tuple(env, term, &arity, &content))
127         return 0;
128     if(!enif_get_resource(env,content[0],mat_resource, dest))
129         return 0;
130     if(!enif_get_resource(env,content[1],doubleArray,&((*dest)->
        content)))
131         return 0;
132     return 1;
133 }
134
135 //Used to translate at once a number of ERL_NIF_TERM.
136 //Data types are inferred via content of format string:
137 //  n: number (int or double) translated to double.
138 //  m: matrix
139 //  i: int
140 int enif_get(ErlNifEnv* env, const ERL_NIF_TERM* erl_terms, const
        char* format, ...){
141     va_list valist;
142     va_start(valist, format);
143     int valid = 1;
144
145     while(valid && *format != '\0'){
146         switch(*format++){
147             case 'n':
```

```
148                 //Read a number as a double.
149                 ;
150                 double *d = va_arg(valist, double*);
151                 int i;
152                 if(!enif_get_double(env, *erl_terms, d))
153                     if(enif_get_int(env, *erl_terms, &i))
154                         *d = (double) i;
155                     else valid = 0;
156                 break;
157 
158             case 'm':
159                 //Reads a matrix.
160                 valid = enif_get_matrix(env, *erl_terms, va_arg(
    valist, Matrix**));
161                 break;
162 
163             case 'i':
164                 //Reads an int.
165                 valid = enif_get_int(env, *erl_terms, va_arg(valist
    , int*));
166                 break;
167 
168             case 'c':
169                 valid = enif_get_atom(env, *erl_terms, va_arg(
    valist, char*), 3, ERL_NIF_LATIN1);
170                 break;
171 
172             default:
173                 //Unknown type... give an error.
174                 valid = 0;
175                 break;
176         }
177         erl_terms ++;
178     }
179 
180     va_end(valist);
181     return valid;
182 }
183 
184 
185 //-------------------------------------------------------------|
186 //      ___________________________________                    |
187 //      |                NIFS              |                   |
188 //      ___________________________________                    |
189 //-------------------------------------------------------------|
190 
191 
192 //@arg 0: List of Lists of numbers.
193 //@return: Matrix of dimension
194 ERL_NIF_TERM nif_matrix(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
195     unsigned n_rows, line_length, dest = 0, n_cols = -1;
196     ERL_NIF_TERM list = argv[0], row, elem;
197     Matrix *m;
198 
199     //Reading incoming matrix.
200     if(!enif_get_list_length(env, list, &n_rows) && n_rows > 0)
```

```
201         return enif_make_badarg(env);
202
203     char str[100];
204
205     for(int i = 0; enif_get_list_cell(env, list, &row, &list); i++)
    {
206         if(!enif_get_list_length(env, row, &line_length))
207             return enif_make_badarg(env);
208
209         if(n_cols == -1){
210             //Allocate binary, make matrix accessor.
211             n_cols = line_length;
212             m = matrix_alloc(n_rows,n_cols);
213         }
214
215         if(n_cols != line_length)
216             return enif_make_badarg(env);
217
218         for(int j = 0; enif_get_list_cell(env, row, &elem, &row); j
    ++){
219             if(!enif_get_double(env, elem, &(m->content[dest]))){
220                 int i;
221                 if(enif_get_int(env, elem, &i)){
222                     m->content[dest] = (double) i;
223                 }
224                 else{
225                     return enif_make_badarg(env);
226                 }
227             }
228             dest++;
229         }
230     }
231
232     return matrix_to_erl(env, m);
233 }
234
235 #define PRECISION 10
236
237 //Used for debug purpose.
238 void debug_write(char info[]){
239     FILE* fp = fopen("debug.txt", "a");
240     fprintf(fp, info);
241     fprintf(fp, "\n");
242     fclose(fp);
243 }
244
245 void debug_write_matrix(Matrix *m){
246     char *content = enif_alloc(sizeof(char)*((2*m->n_cols-1)*m->
    n_rows*PRECISION + m->n_rows*2 + 3));
247     content[0] = '[';
248     content[1] = '\0';
249     char converted[PRECISION];
250
251     for(int i=0; i<m->n_rows; i++){
252         strcat(content, "[");
253         for(int j = 0; j<m->n_cols; j++){
254             snprintf(converted, PRECISION-1, "%.5lf", m->content[i*
```

```
    m->n_cols+j]);
255             strcat(content, converted);
256             if(j != m->n_cols-1)
257                 strcat(content, " ");
258         }
259         strcat(content, "]");
260     }
261     strcat(content, "]");
262     debug_write(content);
263     enif_free(content);
264
265 }
266
267
268 //@arg 0: matrix.
269 //@arg 1: int, coord m: row
270 //@arg 2: int, coord n: col
271 //@return: double, at cord Matrix(m,n).
272 ERL_NIF_TERM nif_get(ErlNifEnv * env, int argc, const ERL_NIF_TERM
    argv[]){
273     int m,n;
274     Matrix *matrix;
275
276     if(!enif_get(env, argv, "iim", &m, &n, &matrix))
277         return enif_make_badarg(env);
278
279     m--, n--;
280
281     if(m < 0 || m >= matrix->n_rows || n < 0 || n >= matrix->n_cols
    )
282         return enif_make_badarg(env);
283
284     int index = m*matrix->n_cols+n;
285     return enif_make_double(env, matrix->content[index]);
286 }
287
288 ERL_NIF_TERM nif_at(ErlNifEnv * env, int argc, const ERL_NIF_TERM
    argv[]){
289     int n;
290     Matrix *matrix;
291
292     if(!enif_get(env, argv, "mi", &matrix, &n))
293         return enif_make_badarg(env);
294     n--;
295
296     if( n < 0 || n >= matrix->n_cols * matrix->n_rows)
297         return enif_make_badarg(env);
298
299     return enif_make_double(env, matrix->content[n]);
300 }
301
302 //Matrix to flattened list of ints
303 ERL_NIF_TERM nif_mtfli(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
304     Matrix *M;
305     if(!enif_get(env, argv, "m", &M)){
306         return enif_make_badarg(env);
```

```
307     }
308
309     ERL_NIF_TERM *mat = enif_alloc(sizeof(ERL_NIF_TERM)*M->n_rows);
310     int index = 0;
311     for (int i=0;i<M->n_rows;i++){
312         ERL_NIF_TERM *row = enif_alloc(sizeof(ERL_NIF_TERM)*M->
    n_cols);
313         for (int j=0;j<M->n_cols;j++){
314             row[j] = enif_make_int(env, (int)M->content[index]);
315             index++;
316         }
317         mat[i]=enif_make_list_from_array(env, row, M->n_cols);
318     }
319
320     ERL_NIF_TERM result = enif_make_list_from_array(env, mat, M->
    n_rows);
321     enif_free(mat);
322     return result;
323 }
324
325 //Matrix to flattened list of ints
326 ERL_NIF_TERM nif_mtfl(ErlNifEnv * env, int argc, const ERL_NIF_TERM
     argv[]){
327     Matrix *M;
328     if(!enif_get(env, argv, "m", &M)){
329         return enif_make_badarg(env);
330     }
331
332     ERL_NIF_TERM *mat = enif_alloc(sizeof(ERL_NIF_TERM)*M->n_rows);
333     int index = 0;
334     for (int i=0;i<M->n_rows;i++){
335         ERL_NIF_TERM *row = enif_alloc(sizeof(ERL_NIF_TERM)*M->
    n_cols);
336         for (int j=0;j<M->n_cols;j++){
337             row[j] = enif_make_double(env, (double) M->content[
    index]);
338             index++;
339         }
340         mat[i]=enif_make_list_from_array(env, row, M->n_cols);
341     }
342
343     ERL_NIF_TERM result = enif_make_list_from_array(env, mat, M->
    n_rows);
344     enif_free(mat);
345     return result;
346 }
347
348 //Equal all doubles
349 //Compares whether all doubles are approximately the same.
350 int equal_ad(double* a, double* b, int size){
351     for(int i = 0; i<size; i++){
352         if(fabs(a[i] - b[i])> 1e-6)
353             return 0;
354     }
355     return 1;
356 }
357
```

```
358 //@arg 0: Array.
359 //@arg 1: Array.
360 //@return: true if arrays share content, false if they have
        different content || size..
361 ERL_NIF_TERM nif_equals(ErlNifEnv *env, int argc, const
        ERL_NIF_TERM argv[]){
362     Matrix *a,*b;
363
364     if(!enif_get(env, argv, "mm", &a, &b))
365         return atom_false;
366
367     //Compare number of columns and rows
368     if((a->n_cols != b->n_cols || a->n_rows != b->n_rows))
369         return atom_false;
370
371     //Compare content of arrays
372     if(!equal_ad(a->content, b->content, a->n_cols*a->n_rows))
373         return atom_false;
374
375     return atom_true;
376 }
377
378
379 //@arg 0: int.
380 //@arg 1: Array.
381 //@return: returns an array, containing requested row.
382 ERL_NIF_TERM nif_row(ErlNifEnv * env, int argc, const ERL_NIF_TERM
        argv[]){
383     int row_req;
384     Matrix *matrix;
385     if(!enif_get(env, argv, "im", &row_req, &matrix))
386         return enif_make_badarg(env);
387
388     row_req--;
389     if(row_req<0 || row_req >= matrix->n_rows)
390         return enif_make_badarg(env);
391
392     Matrix *row = matrix_alloc(1, matrix->n_cols);
393     memcpy(row->content, matrix->content + (row_req * matrix->
        n_cols), matrix->n_cols*sizeof(double));
394
395     return matrix_to_erl(env, row);
396 }
397
398
399 //@arg 0: int.
400 //@arg 1: Array.
401 //@return: returns an array, containing requested col.
402 ERL_NIF_TERM nif_col(ErlNifEnv * env, int argc, const ERL_NIF_TERM
        argv[]){
403     int col_req;
404     Matrix *matrix;
405     if(!enif_get(env, argv, "im", &col_req, &matrix))
406         return enif_make_badarg(env);
407
408     col_req--;
409     if(col_req<0 || col_req >= matrix->n_cols)
```

```
410         return enif_make_badarg(env);
411
412
413     Matrix *col = matrix_alloc(matrix->n_rows, 1);
414
415     for(int i = 0; i < matrix->n_rows; i++){
416         col->content[i] = matrix->content[i * matrix->n_rows +
    col_req];
417     }
418
419     return matrix_to_erl(env, col);
420 }
421
422 //@arg 0: Matrix.
423 //@return: the dimensions of the matrix as {n_rows, n_cols}.
424 ERL_NIF_TERM nif_get_shape(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
425     Matrix *matrix;
426     if(!enif_get(env, argv, "m", &matrix))
427         return enif_make_badarg(env);
428
429     return enif_make_tuple2(env, enif_make_int(env, matrix->n_rows)
    , enif_make_int(env, matrix->n_cols));
430 }
431
432
433 //@arg 0: int.
434 //@arg 1: int.
435 //@return: empty Matrix of requested dimension..
436 ERL_NIF_TERM nif_zeros(ErlNifEnv *env, int argc, const ERL_NIF_TERM
     argv[]){
437     int m,n;
438     if(!enif_get(env, argv, "ii", &m, &n))
439         return enif_make_badarg(env);
440
441     Matrix *a = matrix_alloc(m,n);
442     memset(a->content, 0, sizeof(double)*m*n);
443     return matrix_to_erl(env, a);
444 }
445
446 //@arg 0: int.
447 //@arg 1: int.
448 //@return: eye matrix of dimension [arg 0, arg 1]..
449 ERL_NIF_TERM nif_eye(ErlNifEnv *env, int argc, const ERL_NIF_TERM
    argv[]){
450     int m;
451     if(!enif_get_int(env, argv[0], &m))
452         return enif_make_badarg(env);
453
454     if(m <= 0)
455         return enif_make_badarg(env);
456
457     Matrix *a = matrix_alloc(m,m);
458     memset(a->content, 0, sizeof(double)*m*m);
459     for(int i = 0; i<m; i++){
460         a->content[i*m+i] = 1.0;
461     }
```

```
462     return matrix_to_erl(env, a);
463 }
464 
465 //Either element-wise multiplication, or multiplication of a matrix
        by a number.
466 ERL_NIF_TERM nif_mult(ErlNifEnv *env, int argc, const ERL_NIF_TERM
        argv[]){
467 
468     Matrix *a,*b;
469     double c;
470 
471     if(enif_get(env, argv, "mn", &a, &c)){
472         Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
473 
474         for(int i = 0; i<a->n_cols*a->n_rows; i++){
475             d->content[i] = a->content[i] * c;
476         }
477         return matrix_to_erl(env, d);
478     }
479     else
480     if(enif_get(env, argv, "mm", &a, &b)
481             && a->n_rows*a->n_cols == b->n_rows*b->n_cols){
482 
483         Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
484 
485         for(int i = 0; i < a->n_rows*a->n_cols; i++){
486             d->content[i] = a->content[i] * b->content[i];
487         }
488         return matrix_to_erl(env, d);
489     }
490 
491     return enif_make_badarg(env);
492 }
493 
494 
495 //Either element-wise addition, or addition of a matrix by a number
        .
496 ERL_NIF_TERM nif_add(ErlNifEnv *env, int argc, const ERL_NIF_TERM
        argv[]){
497 
498     Matrix *a,*b;
499     double c;
500 
501     if(enif_get(env, argv, "mn", &a, &c)){
502         Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
503 
504         for(int i = 0; i<a->n_cols*a->n_rows; i++){
505             d->content[i] = a->content[i] + c;
506         }
507         return matrix_to_erl(env, d);
508     }
509     else
510     if(enif_get(env, argv, "mm", &a, &b)
511             && a->n_rows*(a->n_cols) == b->n_rows*b->n_cols){
512 
513         Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
514 
```

```
515         for(int i = 0; i < a->n_rows*a->n_cols; i++){
516             d->content[i] = a->content[i] + b->content[i];
517         }
518         return matrix_to_erl(env, d);
519     }
520 
521     return enif_make_badarg(env);
522 }
523 
524 
525 //Either element-wise subtraction, or subtraction of a matrix by a
    number.
526 ERL_NIF_TERM nif_sub(ErlNifEnv *env, int argc, const ERL_NIF_TERM
    argv[]){
527 
528     Matrix *a,*b;
529     double c;
530 
531     if(enif_get(env, argv, "mn", &a, &c)){
532         Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
533 
534         for(int i = 0; i<a->n_cols*a->n_rows; i++){
535             d->content[i] = a->content[i] - c;
536         }
537         return matrix_to_erl(env, d);
538     }
539     else
540     if(enif_get(env, argv, "mm", &a, &b)
541             && a->n_rows*a->n_cols == b->n_rows*b->n_cols){
542 
543         Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
544 
545         for(int i = 0; i < a->n_rows*a->n_cols; i++){
546             d->content[i] = a->content[i] - b->content[i];
547         }
548         return matrix_to_erl(env, d);
549     }
550 
551     return enif_make_badarg(env);
552 }
553 
554 
555 //Either element-wise division, or division of a matrix by a number
    .
556 ERL_NIF_TERM nif_divide(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
557 
558     Matrix *a,*b;
559     double c;
560 
561     if(enif_get(env, argv, "mn", &a, &c)){
562 
563         if(c!=0.0){
564             Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
565 
566             for(int i = 0; i<a->n_cols*a->n_rows; i++){
567                 d->content[i] = a->content[i] / c;
```

```
568             }
569             return matrix_to_erl(env, d);
570         }
571     }
572     else if(enif_get(env, argv, "mm", &a, &b)
573             && a->n_rows*a->n_cols == b->n_rows*b->n_cols){
574 
575         Matrix *d = matrix_alloc(a->n_rows, a->n_cols);
576 
577         for(int i = 0; i < a->n_rows*a->n_cols; i++){
578             if(b->content[i] == 0.0){
579                 return enif_make_badarg(env);
580             }
581             else{
582                 d->content[i] = a->content[i] / b->content[i];
583             }
584         }
585         return matrix_to_erl(env, d);
586     }
587 
588     return enif_make_badarg(env);
589 }
590 
591 
592 //Transpose of a matrix.
593 Matrix *tr(Matrix *a){
594     Matrix *result = matrix_alloc(a->n_cols ,a->n_rows);
595 
596     for(int j = 0; j < a->n_rows; j++){
597         for(int i = 0; i < a->n_cols; i++){
598             result->content[i*result->n_cols+j] = a->content[j*a->
    n_cols+i];
599         }
600     }
601     return result;
602 }
603 ERL_NIF_TERM nif_transpose(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
604 
605     Matrix *a;
606     if(!enif_get_matrix(env, argv[0], &a))
607         return enif_make_badarg(env);
608 
609     return matrix_to_erl(env, tr(a));
610 }
611 
612 
613 //arg0: Matrix.
614 ERL_NIF_TERM nif_inv(ErlNifEnv *env, int argc, const ERL_NIF_TERM
    argv[]){
615 
616     Matrix *a;
617     if(!enif_get_matrix(env, argv[0], &a))
618         return enif_make_badarg(env);
619 
620     Matrix *inv = matrix_dup(a);
621 
```

```
622     int N = a->n_rows;
623     int *IPIV = enif_alloc(sizeof(int)*N);
624     int LWORK = N*N;
625     double* WORK = enif_alloc(sizeof(int)*N*N);
626     int INFO1, INFO2;
627 
628     dgetrf_(&N,&N,inv->content,&N,IPIV,&INFO1);
629     dgetri_(&N,inv->content,&N,IPIV,WORK,&LWORK,&INFO2);
630 
631     enif_free(IPIV);
632     enif_free(WORK);
633     ERL_NIF_TERM result;
634 
635     if(INFO1 > 0 || INFO2 > 0){
636         result = enif_raise_exception(env, enif_make_atom(env, "
        nif_inv: could not invert singular matrix->"));
637         matrix_free(inv);
638     }
639     else if(INFO1 < 0 || INFO2 < 0){
640         result = enif_raise_exception(env, enif_make_atom(env, "
        nif_inv: LAPACK error->"));
641         matrix_free(inv);
642     }
643     else result = matrix_to_erl(env, inv);
644 
645 
646     return result;
647 }
648 
649 //-------------------------------------------------------------|
650 //       ___________________________________                   |
651 //       |                CBLAS             |                  |
652 //       ___________________________________                   |
653 //-------------------------------------------------------------|
654 //Some CBLAS wrappers.
655 
656 //Calculates the norm of input vector/matrix, aka the square root
        of the sum of its composants.
657 ERL_NIF_TERM nif_dnrm2(ErlNifEnv *env, int argc, const ERL_NIF_TERM
         argv[]){
658     Matrix *x;
659 
660     if(!enif_get(env, argv, "m", &x)){
661         return enif_make_badarg(env);
662     }
663 
664     double result = cblas_dnrm2(x->n_cols*x->n_rows, x->content, 1)
        ;
665 
666     return enif_make_double(env, result);
667 }
668 
669 
670 //Performs blas_ddot
671 //Input: two vectors / matrices
672 ERL_NIF_TERM nif_ddot(ErlNifEnv *env, int argc, const ERL_NIF_TERM
        argv[]){
```

```
673     Matrix *x,*y;
674
675     if(!enif_get(env, argv, "mm", &x, &y)){
676         return enif_make_badarg(env);
677     }
678
679     int n = fmin(x->n_rows*x->n_cols, y->n_rows*y->n_cols);
680     if(n <= 0){
681         return enif_make_badarg(env);
682     }
683
684     double result = cblas_ddot(n, x->content, 1, y->content, 1);
685
686     return enif_make_double(env, result);
687 }
688
689
690 //Performs blas_daxpy
691 //Input: a number, vectors X and Y
692 //Output: a vector of same dimension then Y, containing alpha X + Y
693 //
    ------------------------------------------------------------------------

694 ERL_NIF_TERM nif_daxpy(ErlNifEnv *env, int argc, const ERL_NIF_TERM
        argv[]){
695     Matrix *x,*y;
696     int n;
697     double alpha;
698
699     if(!enif_get(env, argv, "nmm", &alpha, &x, &y)){
700         return enif_make_badarg(env);
701     }
702
703     n = x->n_cols*x->n_rows;
704
705     if(n != y->n_cols*y->n_rows){
706         return enif_make_badarg(env);
707     }
708
709     cblas_daxpy(n, alpha, x->content, 1, y->content, 1);
710
711     return atom_true;
712 }
713
714 // Arguments: alpha, A, x, beta, y
715 // Performs alpha*A*x + beta*y.
716 // alpha, beta are numbers
717 // A, x, y are matrices (x and y being vectors)
718 // x and y are expected to have a length of A->n_cols
719 ERL_NIF_TERM nif_dgemv(ErlNifEnv *env, int argc, const ERL_NIF_TERM
        argv[]){
720     Matrix *A,*x,*y;
721     double alpha, beta;
722
723     if(!enif_get(env, argv, "nmmmm", &alpha, &A, &x, &beta, &y)){
724         enif_make_badarg(env);
725     }
```

```
726
727     //Check dimensions compatibility
728     int vec_length = fmin(fmax(x->n_cols, x->n_rows), fmax(y->
        n_cols, y->n_rows));
729     if(vec_length < A->n_cols || fmin(x->n_cols, x->n_rows) != 1 ||
         fmin(y->n_cols, y->n_rows) != 1){
730         enif_make_badarg(env);
731     }
732
733     Matrix *ny = matrix_dup(y);
734
735     cblas_dgemv(CblasRowMajor, CblasNoTrans, A->n_rows, A->n_cols,
        alpha, A->content, A->n_rows, x->content, 1,  beta, ny->content
        , 1);
736
737     return matrix_to_erl(env, ny);
738 }
739
740
741
742 //Arguments: double alpha, matrix A, matrix B, double beta, matrix
        C
743 ERL_NIF_TERM nif_dgemm(ErlNifEnv *env, int argc, const ERL_NIF_TERM
         argv[]){
744     Matrix *A, *B;
745     double alpha = 1.0;
746     double beta = 0.0;
747
748     if(!enif_get(env, argv, "mm", &A, &B)
749             || A->n_cols != B->n_rows){
750
751         return enif_make_badarg(env);
752     }
753
754     Matrix *C = matrix_alloc(A->n_rows, B->n_cols);
755
756
757
758     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
759     A->n_rows, B->n_cols, A->n_cols,
760     alpha, A->content, A->n_cols, B->content, B->n_cols,
761     beta, C->content, C->n_cols);
762
763     return matrix_to_erl(env, C);
764 }
765
766 //Arguments: double alpha, matrix A, matrix B, double beta, matrix
        C
767 ERL_NIF_TERM nif_dgemm2(ErlNifEnv *env, int argc, const
        ERL_NIF_TERM argv[]){
768     int TransA, TransB;
769     Matrix *A, *B, *C;
770     double alpha;
771     double beta;
772
773     if(!enif_get(env, argv, "iinmmnm", &TransA, &TransB, &alpha, &A
        , &B, &beta, &C)){
```

```
774
775         return enif_make_badarg(env);
776     }
777
778     cblas_dgemm(CblasRowMajor,
779     TransA ? CblasTrans : CblasNoTrans,
780     TransB ? CblasTrans : CblasNoTrans,
781     TransA ? A->n_cols : A->n_rows,
782     TransB ? B->n_rows : B->n_cols,
783     TransA ? A->n_rows : A->n_cols,
784     alpha,
785     A->content,
786     A->n_cols,
787     B->content,
788     B->n_cols,
789     beta, C->content, C->n_cols);
790
791     return atom_true;
792 }
793
794 ERL_NIF_TERM nif_dscal(ErlNifEnv *env, int argc, const ERL_NIF_TERM
       argv[]){
795     Matrix *X;
796     double alpha;
797
798     if(!enif_get(env, argv, "nm", &alpha, &X)){
799         return enif_make_badarg(env);
800     }
801
802     int n = X->n_cols*X->n_rows;
803     cblas_dscal(n, alpha, X->content, 1);
804     return atom_true;
805 }
806
807 ERL_NIF_TERM nif_copy(ErlNifEnv *env, int argc, const ERL_NIF_TERM
      argv[]){
808     Matrix *X;
809
810     if(!enif_get(env, argv, "m", &X)){
811         return enif_make_badarg(env);
812     }
813
814     Matrix *Y = matrix_dup(X);
815     return matrix_to_erl(env, Y);
816 }
817
818 ERL_NIF_TERM nif_copy_shape(ErlNifEnv *env, int argc, const
      ERL_NIF_TERM argv[]){
819     Matrix *X;
820
821     if(!enif_get(env, argv, "m", &X)){
822         return enif_make_badarg(env);
823     }
824
825     Matrix *a = matrix_alloc(X->n_rows,X->n_cols);
826     memset(a->content, 0, sizeof(double)*X->n_rows*X->n_cols);
827     return matrix_to_erl(env, a);
```

```
828 }
829
830 ErlNifFunc nif_funcs[] = {
831     {"matrix", 1, nif_matrix},
832     {"get", 3, nif_get},
833     {"at", 2, nif_at},
834     {"mtfli", 1, nif_mtfli},
835     {"mtfl", 1, nif_mtfl},
836     {"equals", 2, nif_equals},
837     {"row", 2, nif_row},
838     {"col", 2, nif_col},
839     {"zeros", 2, nif_zeros},
840     {"eye", 1, nif_eye},
841     {"mult", 2, nif_mult},
842     {"add",  2, nif_add},
843     {"sub",  2, nif_sub},
844     {"divide", 2, nif_divide},
845     {"transpose", 1, nif_transpose},
846     {"inv", 1, nif_inv},
847     {"copy", 1, nif_copy},
848     {"copy_shape", 1, nif_copy_shape},
849     {"get_shape", 1, nif_get_shape},
850
851     //--- BLAS----------
852     {"nrm2", 1, nif_dnrm2},
853     {"vec_dot", 2, nif_ddot},
854     {"dot", 2, nif_dgemm},
855     {"daxpy", 3, nif_daxpy},
856     {"dscal", 2, nif_dscal},
857     {"dgemm", 7, nif_dgemm2}
858 };
859
860 ERL_NIF_INIT(numerl, nif_funcs, load, NULL, upgrade, NULL)
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**