An introduction to numerical methods using BLAS

Georgios A. Kafanas

University of Luxembourg

January 27, 2024

Abstract

This short introduction to BLAS covers the basic functionality of the library and efficiency considerations that are common in all numerical libraries. Practical aspects of compiling and linking with BLAS libraries are also covered.

Contents

1 Introduction 1

2 Data representation for vectors and matrices 1
- 2.1 A simple data structure for storing vectors . . . 2
- 2.2 A data structure for vectors contained in data structures for matrices . . . 2
  - 2.2.1 A simple data structure for matrices . . . 2
  - 2.2.2 A vector data structure for accessing matrix row vectors . . . 3
- 2.3 A matrix data structure allowing access to sub-matrices . . . 4
- 2.4 Design choices in vector and matrix data structures . . . 5

3 The BLAS library and its interface 5
- 3.1 Naming conventions and data representation . . . 5
- 3.2 Useful extensions of the official BLAS interface . . . 7
- 3.3 Design choices for the functional interface of BLAS . . . 8

4 The effects of caches in BLAS function performance 8

5 Software libraries and BLAS 9
- 5.1 Linking with a dynamic BLAS library . . . 10
- 5.2 Linking with ELF generated from other compilers . . . 11

6 Selecting a BLAS library 11

Bibliography 11

Appendices 12

A The interface of BLAS: quick reference guide 12

List of Figures

1 Example representation of a matrix in memory . . . 2
2 Storage type demonstration . . . 7
3 CPU cache architecture . . . 9
4 Memory allingment . . . 9

List of Tables

1 Matrix type indicators . . . 6

List of Algorithms

1 Simple vector data structure . . . 2
2 Simple vector scaling . . . 2
3 Simple matrix data structure . . . 3
4 Vector scaling . . . 3
5 Vector data structure . . . 4
6 Structure for matrices . . . 5
7 Psedocode for `xAXPY` . . . 8

1 Introduction

Computers are designed to perform numerical computations quickly. Numerical libraries are a method of organizing the algorithms (instructions) that perform the numerical computations in a manner that allows the user to easily find and use the algorithm they require. At the same time, the numerical libraries hide any implementation complexity from the end user.

However, the end user must still be aware of the implementation details of numerical libraries in order to extract the optimal performance. The architecture of modern computers and use of caches in particular, means that mathematically equivalent evaluations in terms of the number of mathematical operations will have different performance. Even though the corresponding algorithms are equivalent in asymptotic performance, the difference in speed and energy consumption is significant for larger problems.

Take for instance a simple matrix-vector multiplication:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix}. \tag{1}$$

Even this simple computation can be performed in 2 different manners, either using dot-product operations and concatenation:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ \begin{pmatrix} 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} \end{pmatrix}, \tag{2}$$

or by scalar-matrix products and matrix addition:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = 1 \begin{pmatrix} 1 \\ 3 \end{pmatrix} + 2 \begin{pmatrix} 2 \\ 4 \end{pmatrix}. \tag{3}$$

Even though these two approaches involve the same number of instructions, the limitations imposed by computer architecture imply that one approach, the one based on scalar-matrix products, is faster for large enough problems!

In this short introduction to numerical libraries we will present BLAS, a library for performing basic linear algebra operations. Among the topics this introduction covers are

- what are numerical libraries and how to use them,
- what kind of functions are implemented in BLAS and their interface, and
- how computer architecture, and more specifically caching, affects the performance of BLAS in different layouts out data.

2 Data representation for vectors and matrices

Understanding how vectors and vectors are represented in computers helps a lot in understanding the interface of BLAS. To understand the representation of vectors and matrices, it is important to understand the limitations of computer memory and how the effect the type of operations performed on these mathematical objects. Let's start with vectors, the simpler of the 2 objects.

2.1 A simple data structure for storing vectors

In general, vectors in computers are stored in arrays. Consider a data structure holding a vector in a language like C. A first approach in the design of such a data structure is given in algorithm 1. The structure hold a pointer to a memory location holding the data of some numerical type T, and the number of elements held in the vector.

Algorithm 1 A simple designing for a data structure holding vectors.

```
structure vector(T)
    data : T*
    n : integer
end structure
```

Consider for instance the function scaling the vector by a scalar in algorithm 2. To scale a vector v by a scalar α, that is perform the operation

$$x \leftarrow \alpha \cdot x, \tag{4}$$

the call to function SCALE is

$$\textsc{scale}(\text{v.n}, \alpha, \text{v.data}), \tag{5}$$

so the data structure for v contains all the data for calling SCALE. However, such a representation has some limitations.

Algorithm 2 Simple function scaling a vector x by scalar α.

```
function SCALE(N, alpha, x)
    for i = 0 : N-1 do
        x[i] ← alpha · x[i]
    end for
end function
```

2.2 A data structure for vectors contained in data structures for matrices

The limitations of the simple vector representation in algorithm 1 appear when we consider how vector scaling is used in the context of problems involving constructs such as matrices. BLAS operates on dense matrices which, like vectors, are represented in the memory by arrays. Arrays are 1-dimensional objects in the memory, unlike mathematical arrays that are 2-dimensional objects. This means that a matrix must be serialized either across its rows or its columns to be stored in an array.

As an example, the 3×2 matrix in fig. 1 is serialized across its columns. The thick lines in the array denote the limits of the array columns, and do not have any physical meaning in the computer memory.

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \longrightarrow \boxed{1}\boxed{2}\,\boxed{3}\boxed{4}\,\boxed{5}\boxed{6}$$

Figure 1: Example representation of a $\mathbb{R}^{3\times 2}$ matrix in an array of doubles in the memory. Delimiters in the array representation denote the row boundaries, without having any physical meaning.

2.2.1 A simple data structure for matrices

A simple design for a structure storing a matrix is presented in algorithm 3. Simply storing the entries of a matrix in an array removes all shape information, so two extra integers, one for the

number of rows (m) and one for the number of columns (n) are also stored. The number N of element of type T allocated for the array data is

$$N = \mathrm{m} \cdot \mathrm{n} \tag{6}$$

so N is not stored explicitly.

Algorithm 3 A simple design for a data structure storing matrices.

```
structure matrix(T)
    data : T*
    m : integer
    n : integer
end structure
```

The matrix can be split in the conventional manner in row and column vectors. Given a matrix $A \in \mathbb{R}^{m \times n}$, we use the notation

- $A[i, j]$ to denote the element in position (i, j), and
- $A[i_0 : i_1, j_0 : j_1]$ to denote a section of A between the row i_0 and i_1 and the columns j_0 and j_1.

In case there is a single element in a range, for instance $A[i_0 : i_1, j : j]$, we simplify the notation to $A[i_0 : i_1, j]$.

2.2.2 A vector data structure for accessing matrix row vectors

Consider some problem where we would like to scale in place a row vector of some matrix. This is a fairly typical operation that appears in problems such as Gaussian elimination. In the example of fig. 1 we can observe that the row vectors are not stored in a consecutive manner. The algorithm 2 however works only on sequential vectors. We fix this problem in algorithm 4, by adding an extra parameter that describes the step between vector elements.

Algorithm 4 Function scaling a vector x by scalar α that supports vectors stored with periodic increments.

```
function SCALE(N, alpha, x, incx)
    for i = 0 : N-1 do
        x[i · incx] ← alpha · x[i · incx]
    end for
end function
```

The scaling function in algorithm 4 can now scale the kth row of a matrix A with a call

$$\textsc{scale}(N = \mathrm{A.n}, \mathrm{alpha} = \alpha, \mathrm{x} = \mathrm{A.data} + k, incx = \mathrm{A.m}). \tag{7}$$

Of course, we could have copied the data in a consecutive range in memory, use the simple scale function in algorithm 2 and copy the result back in the array. The advantage of the implementation supporting the increment parameter is that it performs the operation in place, eliminating the need for data copying.

Our scaling function now supports vectors with periodic increments that are different than one, however, the vector data structure in algorithm 1 can not hold the information to describe vectors with periodic increments. To rectify this disparity, we extend the vector data structure by adding an increment parameter in algorithm 5. Then, the vector v for the kth row of a matrix $A \in \mathbb{R}^{m \times n}$ is

$$A[k, 1 : n] \equiv \mathrm{v} \leftarrow \mathrm{vector}(\mathrm{data} = \mathrm{A.data} + k, \mathrm{n} = \mathrm{A.n}, \mathrm{incx} = \mathrm{A.m}), \tag{8}$$

and the call to scale the vector by α is

$$\textsc{scale}(\mathrm{N} = \mathrm{v.n}, \mathrm{alpha} = \alpha, \mathrm{x} = \mathrm{v.data}, \mathrm{incx} = \mathrm{v.incr}). \tag{9}$$

Thus, the new vector representation in algorithm 5 contains all the information to call the function scaling vector with a periodic increment between entries different than 1.

Algorithm 5 A data structure holding vectors with periodic increments.

```
structure vector(T)
    data : T*
    n : integer
    incr : integer
end structure
```

2.3 A matrix data structure allowing access to sub-matrices

We now have a data structure that allows access to vectors in a more complex context. Is there the need to access matrix elements in contexts more complex than the data structure in algorithm 3 can support? Consider for instance again the Gaussian elimination operating (row wise) on the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{pmatrix}. \tag{10}$$

Denoting the state of the matrix before and after the 1st elimination step by $A^{(0)}$ and $A^{(1)}$ respectively, the result of a single step of the algorithms is

$$A^{(0)} = \left(\begin{array}{|ccc} \hline 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{array}\right) \sim A^{(1)} = \left(\begin{array}{c|cc} 1 & 1 & 1 \\ \cline{2-3} 0 & 1 & 1 \\ 0 & 1 & 2 \end{array}\right) \tag{11}$$

where the demarcated lower-right area is the area on which the next step of the algorithm will operate. In a recursive implementation of the Gaussian elimination, will ideally operate in place. However, looking into the column-wise linearization of the arrays,

$$A^{(0)} \equiv \left(\begin{array}{ccc|ccc|ccc} 1 & 1 & 1 & 1 & 2 & 2 & 1 & 2 & 3 \end{array} \right) \tag{12}$$

and

$$A^{(1)} \equiv \left(\begin{array}{ccc|ccc|ccc} 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 \end{array} \right), \tag{13}$$

the columns of $A^{(1)}$ are no longer stored in a consecutive manner.

The situation is similar to the increment parameter in the vector data structure, and it is rectified by the introduction of a parameter to capture the constant spacing between initial entries of consecutive columns. The parameter is called the leading dimension of the matrix (ld) because in many computations replaces the column length. The new data structure can be seen in algorithm 6. Assume that a data structure of the new matrix type M_0 stores the data for $A^{(0)}$, then the data structure M_1 for $A^{(1)}$ is derived by

$$\mathrm{M}_1 \leftarrow \mathrm{matrix}(\mathrm{data} = \mathrm{data.M} + 1 \cdot \mathrm{ld} + 1, \mathrm{ld} = 3, \mathrm{m} = 2, \mathrm{n} = 2). \tag{14}$$

Algorithm 6 A data structure storing matrices with periodic spacing between column initial entries.

```
structure matrix(T)
    data : T*
    ld : integer
    m : integer
    n : integer
end structure
```

2.4 Design choices in vector and matrix data structures

The data structure of algorithm 5 follows the BLAS interface very well, however, it has its own limitations. In particular, the size information for the data array is lost, and the address to which the data entry points is no longer considered as the initial address in the memory allocated for the vector data. Thus the vector data structure in algorithm 5 can longer be used to manage the memory allocation for the vector.

The problem is that in software engineering terms a vector data structures must perform 2 operations,

- manage the memory allocation for the vector, and
- provide a flexible method to access the data in the memory allocated by the vector.

Thus, it makes sense to split that data structures and have different structures perform each function.[1] The interface of BLAS remains quite flat, so all arguments are provided to BLAS functions using raw function arguments instead of data structure fields. It often helps though to think of vectors in terms of a data structure that provides flexible access, as the memory will often be managed externally. For instance, when scaling the rows of a matrix in **??**, the memory allocation for the vector is managed by the data structure for the matrix.

3 The BLAS library and its interface

Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing common basic linear algebra operations. An overview of the interface of BLAS is provided by the quick reference guide in appendix A. The interface presented is for Fortran programs. Programs written in C can use the FORTRAN interface directly, for simplicity however, equivalent definitions are available in the CBLAS interface for C programs.

There are multiple implementations of the BLAS interface, each optimized for some set of target platforms. The Netlib BLAS is the reference implementation that emphasizes code clarity and basic optimizations that work on all architectures. The definitions of the various functions are provided in the documentation and are simple enough that to be accessible to all after some familiarisation with the code base.[2] Netlib BLAS also provides C bindings for the FORTRAN implementation through a library implementing the CBLAS interface.

3.1 Naming conventions and data representation

The names of the functions used by BLAS can be unintuitive, however, the naming scheme is systematic. The operations of BLAS are grouped in 3 levels, according to their computational complexity.

Level 1: Functions of BLAS with computational complexity $O(n)$ (and some with complexity $O(1)$ as well). These include operations on one or more vectors, such as the vector norm and the dot product.

[1] This is the approach used in the ranges library of C++20.

[2] https://www.netlib.org/blas/#_blas_routines

	Storage type		
Algebraic properties	Standard (-)	Banded (B)	Packed (P)
General (G)	GE	GB	
Symmetric (S)	SY	SB	SP
Hermitian (H)	HE	HB	HP
Triangular (T)	TR	TB	TP

Table 1: Indicator of mathematical and storage matrix types

Level 2: Functions of BLAS with computational complexity $O(n^2)$. These include matrix-vector products.

Level 3: Functions of BLAS with computational complexity $O(n^3)$. These include matrix-matrix products.

BLAS was designed when versions of FORTRAN with limits in their identifier names were dominating numerical computing, so the names use characters economically. The first character of the functions name is a prefix that denotes the data type over which the function operates. The official BLAS interface as implemented by Netlib supports the following arithmetic types and numerical precisions:

- single precision real numbers (**S**) with 32-bits,
- double precision real numbers (**D**) with 64-bits,
- single precision complex numbers (**C**) with 64-bits, and
- double precision complex numbers (**Z**) with 128-bits.

The required algebraic type (real or complex) depends on the problem type. The precision type depends on the application type, for instance engineering simulations require double precision calculations, were as machine learning application such as classifiers based on Multilayer Perceptrons can work with single or even lower precision. There are proposals to support lower and mixed precision operations,[3] mainly to facilitate machine learning applications.

Following the data type specifier, a letter code specifies the type of the operands and the operation. Operands can be scalars, vectors, or matrices. Scalars and vectors have one storage form, however, matrices can be stored in multiple manners. The mathematical properties of matrices are exploited in the storage types to save on memory accesses and memory use. The matrix storage types are presented in table 1, and the storage patterns for the standard and packed typed are depicted in fig. 2. The banded type is similarly simple, but it requires larger matrices to demonstrate effectively. You can find the definitions of all storage patterns supported by BLAS in the BLAS forum standard [1].

The final part of the function names, is the operation type. The notation for the operation types is significantly more varied than the other name components.

- For *level 1* operations, there are few specific rules and the names are mostly mnemonic or phonetic approximations. For instance, xAXPY denotes the operation $x \leftarrow \alpha x + y$.
- For most *level 2* operations, the sequence **MV** is used denoting multiplication of matrix (M) with vector (V).
- For most *level 3* operations, the sequence **MM** is used denoting multiplication of matrix (M) with matrix (M).

Many operations support optional arguments for each input to use its transpose, or in the case of complex arguments its conjugate transpose as well. There are also some routines that allow mixing complex and real numbers. The quick reference guide of BLAS, also included in appendix A, contains a list the available functions.

[3] https://icl.utk.edu/bblas/sc17/files/bblas-sc17-riedy.pdf

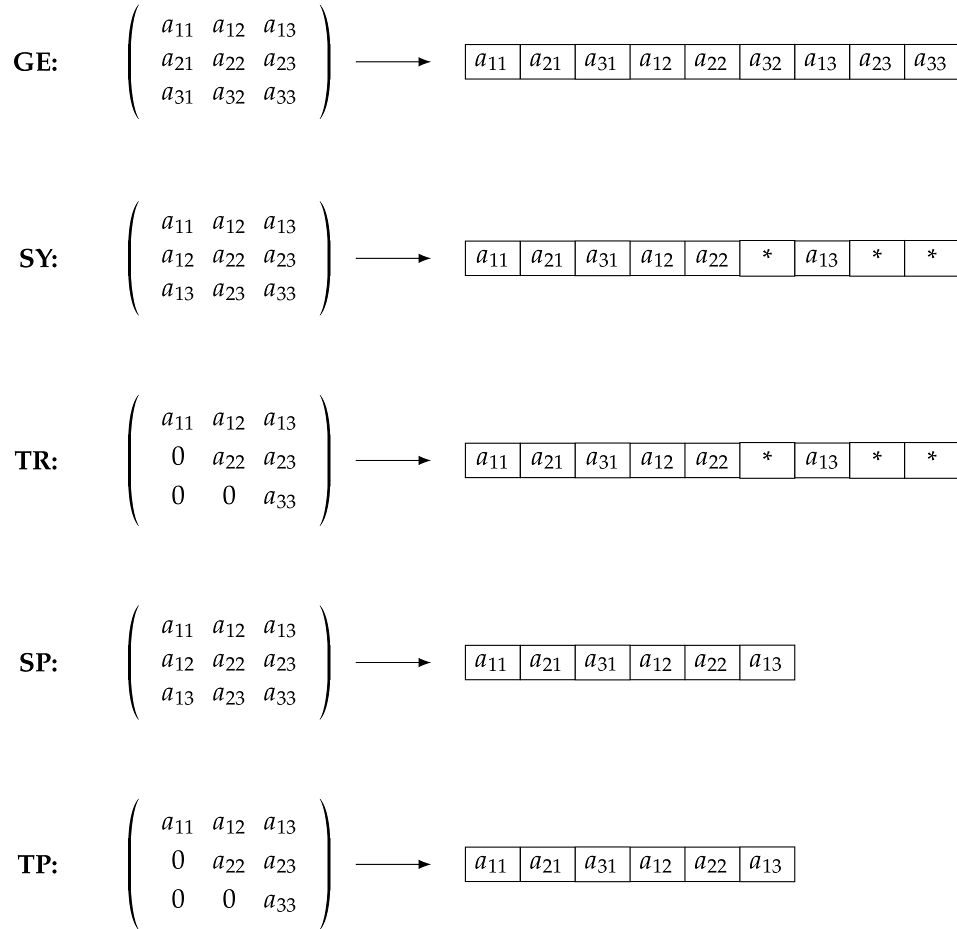


Figure 2: Demonstration of a few storage types supported by BLAS. In symmetric or triangular matrices stored in standard form, the lower triangular part is left with undefined content. This speedup computations as the undefined entries do not have to be stored. In packed formats, the undefined entries are dropped almost halving the required storage. Note that generic matrices cannot be stored in packed form.

3.2 Useful extensions of the official BLAS interface

There are some very useful operations that are not part of the official standard, but are implemented in some BLAS libraries like OpenBLAS and MKL. For instance, when evaluating a product which we know that it will result in a symmetric matrix, the `xGEMMT` operation multiplies two matrices, but only updates the upper or lower triangular part of the output. This is a more frequent operation than one may expect, as it appears in the calculation of energy (quadratic forms)! Consider for instance, the product

$$ABA^T \tag{15}$$

where B is symmetric and A is a compatible generic matrix. The product

$$C = AB \tag{16}$$

is not symmetric in general, but the whole result

$$CA^T \tag{17}$$

is always symmetric! As a result, only the evaluation of the upper or lower triangular part of the matrix is required almost halving the number of memory accesses required.

3.3 Design choices for the functional interface of BLAS

The BLAS specification describes a systematically designed interface that exposes to the user a set of pure functions. All the functions operate on basic data types or their arrays, and do not allocate memory or any other resource. This is a deliberate design choice, as it allows the user to design their structures in a manner best fitting their problem.

For example, imagine that we are using the data structures in algorithm 5 to store vectors. Then, to perform the operation

$$y \leftarrow \alpha x + y \tag{18}$$

the vectors x and y are passed to the level 1 function `xAXPY` with the call

$$\text{xAXPY}(N, \alpha, \text{x.data}, \text{x.incr}, \text{y.data}, \text{y.incr}) \tag{19}$$

where we should ensure that

$$N = \text{x.n} = \text{y.n}. \tag{20}$$

Depending on the type of the vector, we should chose from `SAXPY`, `DAXPY`, `CAXPY`, and `ZAXPY`.

The user is responsible to ensure that all memory accesses are within bounds in BLAS. For instance, in relation (19), the calling function should ensure that the relation in relation (20) holds and that

$$\begin{cases} \text{x.n} \cdot \text{x.incr} < \text{size allocated for x.data} \\ \text{y.n} \cdot \text{y.incr} < \text{size allocated for y.data} \end{cases} \tag{21}$$

so that `xAXPY` does not read or write data outside the memory allocated for the vector arguments.

Furthermore, in many BLAS functions the input arrays can be overwritten with the results for efficiency. Let's consider a pseudocode definition of `xAXPY` in algorithm 7. This function allows us to scale a vector x by a scalar α in place, similar to algorithm 4, by calling

$$\text{xAXPY}(N = \text{x.n}, \text{alpha} = \alpha, \text{x} = \text{x.data}, \text{incx} = \text{x.incr}, \text{y} = \text{x.data}, \text{incy} = \text{x.incr}) \tag{22}$$

which overwrites the data in x. This is not just an artifact of the specific implementation of `xAXPY`, but the official specification for `xAXPY`. All relevant functions in BLAS can overwrite their inputs in a similar manner.

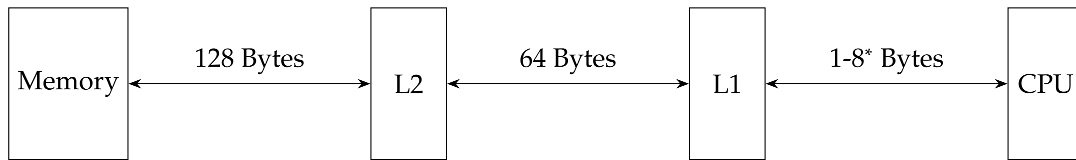
Algorithm 7 Pseudocode for `xAXPY(N, alpha, x, incx, y, incy)`.

```
function xAXPY(N, alpha, x, incx, y, incy)
    for i = 0 : N-1 do
        y[i*incy] ← y[i*incy] + alpha*x[i*incx]
    end for
end function
```

4 The effects of caches in BLAS function performance

In modern hardware, it is often beneficial to access the memory in chunks of fixed size. Vectorized SMID instructions allow the unrolling of loops to apply an operations over multiple inputs at once. Even if the vectorized operation is slower than individual operations, the data are loaded in the registers in one instruction, instead of multiple small loads, increasing the overall throughput. An architecture typical of modern cache systems in depicted in fig. 3. Typically architectures implement instruction to load whole cache lines from the L1 data cache, like the

SIMD instruction of AVX-512 that operate on 64 byte vectors[4].



*up to 64 for some special SIMD instructions sets such as AVX-512

Figure 3: This figure depicts a typical cache architecture of a modern CPU with some simplifications. For instance, the L1 cache is almost always split into an instruction and a data cache, and there is usually a L3 cache as well. However, the basic principles of operation of all caches is that access to any element within a cache line will cause the fetch of the whole line from further down the cache hierarchy.

However, due to the specifics of the cache implementation, cache line start at specific memory addresses. If the data is not stored within the boundaries of the cache lines, the SIMD load operations cannot be applied and loop unrolling is deactivated. For instance in fig. 4 the green array is aligned exactly with the cache lines and can be fetched in one SIMD operation. The red array however, even if it is the same size is not aligned requiring at least two operations!

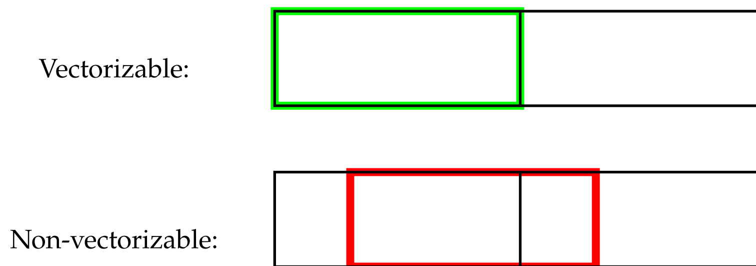


Figure 4: The green array is aligned exactly with the cache lines and can be fetched in one SIMD operation. The red array is not aligned and thus requires at least 2 operations.

Due to the performance limitation in accessing the memory, it is thus beneficial to place the matrix columns (or rows) so that they align with the cache lines. As a result, the *leading dimension* of the array is used to pad the columns to align with the cache lines.

5 Software libraries and BLAS

Compilation of source code does not generate executables directly, but object files in Executable and Linkable Format (ELF). There are multiple methods to compose executables from object files in ELF format, each method developed to meet specific needs. For instance, dynamic loading of dynamic libraries allows the implementation of interpreted environments within a single process.

Most executables are composed using libraries. The compilation of a single source file, generates an object file in ELF format. Multiple object files and libraries can be linked into new libraries, or can be linked into executables. There are 2 kinds of libraries, static and dynamic libraries.

Static libraries: Multiple object files strung together; a table of content can be added to accelerate

[4] https://www.intel.com/content/www/us/en/developer/articles/technical/data-alignment-to-assist-vectorization.html

symbol (functions and variables) lookup. They are using during linking to resolve all symbols in the generated executable.

Dynamic libraries: Similar to static libraries, but compiled in a way that allows them to be loaded at runtime, as the need arises. They are used in during loading or dynamically during program execution to resolve external symbols.

Furthermore, dynamic libraries can be loaded either,

- during dynamic linking at the beginning of the execution, or
- dynamically during program execution using the linker (`ld`) API.

A good introduction about how the need for the various methods of composing ELF executables, and the details of the linking and loading process is provided in [2].

BLAS is a numerical library, so it exports only functions. Even though the static libraries can be generated for the Netlib distribution of BLAS, typically BLAS libraries are linked dynamically as they are often used by many components of a program.

5.1 Linking with a dynamic BLAS library

To link a simple program with BLAS, first compile the program using the headers of BLAS. In this case we need the C header, `cblas.h`, since we are using the C bindings provided by BLAS.

```
> /usr/bin/cc -isystem ${HOME}/.local/netlib/include \
  -c example_execute.c \
  -o example_execute.o
```

This command will compile the library. The tutorial library also uses a local utility library which is compiled with the following command.

```
> /usr/bin/cc -o utils.o -c utils.c
```

The object file of the lilbrary is strung into a static library with the archive tool, `ar`, and a content table is generated with the index generation tool, `ranlib`.

```
> /usr/bin/ar qc libutils.a utils.o && /usr/bin/ranlib libutils.a
```

Finally, the executable object file, the local utility library, and the BLAS libraries are linked to create the final executable.

```
> /usr/bin/cc \
  -Wl,--no-as-needed -Wl,-rpath,${HOME}/.local/netlib/lib \
    example_execute.o \
    ${HOME}/.local/netlib/lib/libcblas.so \
    ${HOME}/.local/netlib/lib/libblas.so
    libutils.a \
  -o example_execute
```

The linking command is usually quite involved, especially in larger projects. In this case a static library, `libutils.a`, and two dynami libraries from BLAS, `libcblas.so` and `libblas.so`, are linked with the executable. The library `libblas.so` is linked because it is required by `libsblas.so`. The dependencies are scanned from left to right, so an object must appear *before* the objects it needs. Linux distributions provide various tools for inspecting libraries and executables.

`readelf`: Displays information about ELF files. For instance the dynamic option, `-dynamic` or `-d`, displays information about how the system will look for dynamic libraries during loading.

```
> readelf --dynamic example_execute
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libcblas.so.3]
 0x0000000000000001 (NEEDED)             Shared library: [libblas.so.3]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000001d (RUNPATH)            Library runpath: [/home/gkaf/.local/netlib/lib]
```

According to the listing, the libraries `libcblas.so` and `libblas.so` are needed, and the linker will look into the directory `/home/gkaf/.local/netlib/lib` before looking into the standard system locations.

ldd: Loads the executable libraries as loader would do during runtime.

```
> ldd example_execute
        linux-vdso.so.1 (0x00007ffc6275a000)
        libcblas.so.3 => /home/gkaf/.local/netlib/lib/libcblas.so.3 (0x00007f055a131000)
        libblas.so.3 => /home/gkaf/.local/netlib/lib/libblas.so.3 (0x00007f055a08c000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0559e8b000)
        libgfortran.so.5 => /lib/x86_64-linux-gnu/libgfortran.so.5 (0x00007f0559a00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f055a16e000)
        libquadmath.so.0 => /lib/x86_64-linux-gnu/libquadmath.so.0 (0x00007f0559e44000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f0559d63000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f0559d43000)
```

5.2 Linking with ELF generated from other compilers

In general, ELF files generated from different compilers, such as `gcc/gfortran`, `clang`, and the various Intel compilers, can be mixed provided that the ELF files have a compatible Application Binary Interface (ABI) [3], [4]. The ABI determines details such as

- basic properties of the processor instruction set,
- size, layouts, and alignment of basic data types,
- the calling convention, which controls how arguments are passed and return to and from functions, and
- how systems calls are made (i.e. memory allocation, file access, and so on).

In the case of BLAS, we could compile BLAS with `gcc` and our executable with `clang` and still be able to link the two. More importantly, the ABI compatibility alows linking with the library generated by `gfortran` for BLAS, as the library is written in FORTRAN. The Netlib library provides an interface for convenience, but for other libraries require calling the FORTRAN library directly. This is easier with language interoperability tools such the ISO C binding and Chasm that provided various datatype definitions, so that users do not have to select themselves data types which are compatible between languages.

6 Selecting a BLAS library

The Netlib reference implementation of BLAS, is as the name suggest, a generically optimized version of the BLAS library which balances clarity with performance. There are implementations that are targeted to specific hardware or implement more code optimizations in general to achieve a better performance. Such implementations should be preferred over the reference Netlib implementation when they are available[5]. Distributions that are worth trying are ATLAS and the more modern OpenBLAS, both distributed under BSD style licenses.

[5] https://www.netlib.org/blas/#_optimized_blas_library

References

[1] J. J. Dongarra, *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard* (International journal of high performance computing applications). Aug. 12, 2001. [Online]. Available: https://www.netlib.org/blas/blast-forum/blas-report.pdf.

[2] D. M. Beazley, B. D. Ward, and I. R. Cooke, "The inside store on shared libraries and dynamic loading," *Computing in Science & Engineering*, vol. 3, no. 5, pp. 90–97, Sep. 2001. [Online]. Available: https://www.dabeaz.com/papers/CiSE/c5090.pdf.

[3] TIS Committee, *Tool Interface Standard (TIS): Executable and Linking Format (ELF) Specification*, Version 1.2. May 1995. [Online]. Available: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[4] *System v: Application binary interface*, 4.1, Linux Foundation, Mar. 18, 1997. [Online]. Available: https://refspecs.linuxfoundation.org/elf/gabi41.pdf.

Appendices

A The interface of BLAS: quick reference guide

Level 1 BLAS

dim scalar vector vector scalars 5-element array		prefixes
`SUBROUTINE xROTG ( A, B, C, S )`	Generate plane rotation	S, D
`SUBROUTINE xROTMG( D1, D2, A, B, PARAM )`	Generate modified plane rotation	S, D
`SUBROUTINE xROT ( N, X, INCX, Y, INCY, C, S )`	Apply plane rotation	S, D
`SUBROUTINE xROTM ( N, X, INCX, Y, INCY, PARAM )`	Apply modified plane rotation	S, D
`SUBROUTINE xSWAP ( N, X, INCX, Y, INCY )`	$x \leftrightarrow y$	S, D, C, Z
`SUBROUTINE xSCAL ( N, ALPHA, X, INCX )`	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
`SUBROUTINE xCOPY ( N, X, INCX, Y, INCY )`	$y \leftarrow x$	S, D, C, Z
`SUBROUTINE xAXPY ( N, ALPHA, X, INCX, Y, INCY )`	$y \leftarrow \alpha x + y$	S, D, C, Z
`FUNCTION xDOT ( N, X, INCX, Y, INCY )`	$dot \leftarrow x^T y$	S, D, DS
`FUNCTION xDOTU ( N, X, INCX, Y, INCY )`	$dot \leftarrow x^T y$	C, Z
`FUNCTION xDOTC ( N, X, INCX, Y, INCY )`	$dot \leftarrow x^H y$	C, Z
`FUNCTION xxDOT ( N, X, INCX, Y, INCY )`	$dot \leftarrow \alpha + x^T y$	SDS
`FUNCTION xNRM2 ( N, X, INCX )`	$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
`FUNCTION xASUM ( N, X, INCX )`	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
`FUNCTION IxAMAX( N, X, INCX )`	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

options dim b-width scalar matrix vector scalar vector		
`xGEMV ( TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
`xGBMV ( TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
`xHEMV ( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	C, Z
`xHBMV ( UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	C, Z
`xHPMV ( UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	C, Z
`xSYMV ( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	S, D
`xSBMV ( UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	S, D
`xSPMV ( UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	S, D
`xTRMV ( UPLO, TRANS, DIAG, N, A, LDA, X, INCX )`	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
`xTBMV ( UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX )`	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
`xTPMV ( UPLO, TRANS, DIAG, N, AP, X, INCX )`	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
`xTRSV ( UPLO, TRANS, DIAG, N, A, LDA, X, INCX )`	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
`xTBSV ( UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX )`	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
`xTPSV ( UPLO, TRANS, DIAG, N, AP, X, INCX )`	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
options dim scalar vector vector matrix		
`xGER ( M, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
`xGERU ( M, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
`xGERC ( M, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
`xHER ( UPLO, N, ALPHA, X, INCX, A, LDA )`	$A \leftarrow \alpha xx^H + A$	C, Z
`xHPR ( UPLO, N, ALPHA, X, INCX, AP )`	$A \leftarrow \alpha xx^H + A$	C, Z
`xHER2 ( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
`xHPR2 ( UPLO, N, ALPHA, X, INCX, Y, INCY, AP )`	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
`xSYR ( UPLO, N, ALPHA, X, INCX, A, LDA )`	$A \leftarrow \alpha xx^T + A$	S, D
`xSPR ( UPLO, N, ALPHA, X, INCX, AP )`	$A \leftarrow \alpha xx^T + A$	S, D
`xSYR2 ( UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
`xSPR2 ( UPLO, N, ALPHA, X, INCX, Y, INCY, AP )`	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Level 3 BLAS

options dim scalar matrix matrix scalar matrix		
`xGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC )`	$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
`xSYMM ( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC )`	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
`xHEMM ( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC )`	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
`xSYRK ( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC )`	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
`xHERK ( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC )`	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
`xSYR2K( UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC )`	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$	S, D, C, Z
`xHER2K( UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC )`	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$	C, Z
`xTRMM ( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB )`	$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
`xTRSM ( SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB )`	$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

2

Meaning of prefixes

S - REAL	C - COMPLEX
D - DOUBLE PRECISION	Z - COMPLEX*16 (this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS
In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS
Matrix types:

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Sym. Band	SP - Sum. Packed
HE - HErmitian	HB - Herm. Band	HP - Herm. Packed
TR - TRiangular	TB - Triang. Band	TP - Triang. Packed

Level 2 and Level 3 BLAS Options
Dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRANx	= '**N**o transpose', '**T**ranspose', '**C**onjugate transpose' (X, X^T, X^H)
UPLO	= '**U**pper triangular', '**L**ower triangular'
DIAG	= '**N**on-unit triangular', '**U**nit triangular'
SIDE	= '**L**eft', '**R**ight' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANSx = 'T' and TRANSx = 'C' have the same meaning.
For Hermitian matrices, TRANSx = 'T' is not allowed.
For complex symmetric matrices, TRANSx = 'H' is not allowed.

References

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Math. Soft.* 5 (1979) 308-325

J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* 14,1 (1988) 1-32

J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* (1989)

Obtaining the Software via `netlib@ornl.gov`

To receive a copy of the single-precision software, type in a mail message:

```
send sblas from blas
send sblas2 from blas
send sblas3 from blas
```

To receive a copy of the double-precision software, type in a mail message:

```
send dblas from blas
send dblas2 from blas
send dblas3 from blas
```

To receive a copy of the complex single-precision software, type in a mail message:

```
send cblas from blas
send cblas2 from blas
send cblas3 from blas
```

To receive a copy of the complex double-precision software, type in a mail message:

```
send zblas from blas
send zblas2 from blas
send zblas3 from blas
```

Send comments and questions to `lapack@cs.utk.edu` .

Basic Linear Algebra Subprograms

A Quick Reference Guide

University of Tennessee
Oak Ridge National Laboratory
Numerical Algorithms Group Ltd.

May 11, 1997