Multiple back-end support for the Armadillo linear algebra interface

Paolo Viviani[1,3], Marco Aldinucci[1], Massimo Torquati[2], and Roberto d'Ippolito[3]

[1]Computer Science Department, University of Torino, Italy
[2]Computer Science Department, University of Pisa, Italy
[3]Noesis Solutions NV, Leuven, Belgium

December 2nd 2016

Abstract

The Armadillo C++ library provides programmers with a high-level Matlab-like syntax for linear algebra. Its design aims at providing a good balance between speed and ease of use. It can be linked with different back-ends, i.e. different LAPACK-compliant libraries. In this work we present a novel run-time support of Armadillo, which gracefully extends mainstream implementation to enable dynamic back-end switching and multiple back-end support. The extension is specifically designed to not affect Armadillo class template prototypes, thus to be easily interoperable with future evolutions of the Armadillo library itself. The proposed software stack is then tested for functionality and performance against a kernel code extracted from an industrial application.

1 Introduction

Numerical linear algebra operations play a key role in a number of scientific and industrial applications, spanning from computational fluid dynamics to machine learning. Numerical operations need to be executed as fast as possible in order to significantly improve the overall performance of applications. In this regard, hybrid multicore-GPU platforms (and chips) can effectively support the computation density required by numerical routines even on off-the-shelf platforms. Notwithstanding, the performance tuning of parallel code for hybrid platforms is still complicated and requires skills that are beyond capabilities of applicative domain experts. Ideally, they require a platform-independent, textbook-like API supported by a run-time exhibiting high efficiency on wide range of problems and platforms. These requirements (see Sec. 2), are the major driving force of this work that is motivated by a specific industrial use case provided by Noesis Solutions NV.

The area of parallel techniques and libraries for linear algebra is mature, nevertheless, a review of existing solutions revealed a number of shortcomings present in the available tools that prevent their wide adoption by domain experts. The performance need crucially pushes library designers toward specialisation for algorithms, which can be pursued at design or configuration time or at the run-time. In this latter case, the API fatally put on weight (i.e. parameters) impairing usability. This work aims to bridge these shortcomings using widely-used libraries as building blocks (so-called back-ends).

In the long term, we advocate a self-optimising, high-level of abstraction library providing applications with dynamic and automatic selection of the best back-end library for each operation on a given platform. This requires 1) the capability to support multiple concurrent back-ends libraries (i.e. the *mechanisms*), and 2) the capability to schedule operations on the best subset of back-ends (i.e. the *policies*). This work focus on the mechanisms, whereas policies are beyond the scope of the present work.

The rest of this paper is organised as follows: Sec. 2 outlines the requirements that the presented software stack is supposed to fulfil, Sec. 3 presents a description of similar existing tools, Sec. 4 describes the main design principles adopted for building the proposed software stack, Sec. 5 reports the experimental evaluation on a (mock-up) of a real-world industrial use case, and finally Sec. 6 outlines some conclusions and some possible future works.

2 Use case and requirements

Noesis Solutions is a simulation innovation partner to manufacturers in automotive, aerospace and other engineering-intense industries. Specialised in simulation Process Integration and numerical Design Optimization (PIDO), its flagship software Optimus leverages Noesis' extensive experience in optimization and system integration methodologies to increase the efficiency of established engineering practices and processes. Noesis actively researches a number of new technologies in the field of process integration, capturing and improvement of engineering knowledge within multidisciplinary industrial processes, advanced methods for modelling and optimization of the behaviour of large engineering systems in the virtual prototype stage, parallelization of computational effort, and assessment of quality and robustness of the final product.

Optimus makes heavy use of linear algebra routines within its kernel code: a significant example is the calculation of so-called *Response Surface Models* (RSMs), which are used to perform interpolation and approximation of discrete datasets. One of the most time consuming operations required by the RSMs computation is the inversion of a rank-deficient matrix in order to solve a linear system: this can be achieved by calculating the *Moore-Penrose pseudoinverse* using the *Singular Value Decomposition* (SVD) [8], which represents the original use case for the library hereby introduced. As an extension of this use case we also considered the scenario involving the calculation of multiple response surfaces on slightly different datasets, that require a number of independent linear systems to be solved with the technique mentioned above.

Beside, this work is intended to be applied as widely as possible to the Optimus code, hence the software stack is expected to be general purpose enough and to fulfil performance and usability requirements that are discussed below.

Performance

The goal is to provide state-of-the-art performance for the largest possible catalogue of linear algebra operation. To achieve that, the framework is expected to exploit multicore CPUs and different GPU accelerators. It is fundamental to provide high performance on advanced linear algebra operations like linear system solving and matrix decompositions.

Usability

As stated above, the idea behind this work is to provide applicative domain experts with a simple, textbook-like, interface. In order to hide the complexity of the heterogeneous hardware to the developer, it is fundamental that the code requires no modifications when running on the CPU or on the GPU, while also being able to move the execution between the host and the device at run-time. Moreover, such interface should be easily incorporated into existing C++ code.

We will show that the proposed library is fully compliant to these requirements where similar tools fall short. The use case introduced above will be used to validate this approach by means of a production code mock-up that will be described in detail in Sec. 5.

3 Related Work

Given the large spectrum of applications for dense linear algebra, it is not surprising that there are several tools for different languages that allow to perform numerical linear algebra calculations. In this section we will discuss such tools and how they compare to the library presented in this work. Since our main focus is towards high performance computations, we will consider only those libraries that are clearly intended for high performance purposes, moreover the comparison will be made with other C/C++ libraries, since it is the language of choice for the use case envisioned by the authors, but the advantages of the approach presented here holds

also when considering libraries for other languages like NumPy [16], Theano [4] or Torch [6].

We identify two main ways to categorise dense linear algebra libraries: 1) by the kind of operations they implement, 2) by the level of abstraction that they provide. Table 1 gives an overview of several related tools classified according to these two criteria. For the first aspect, we mainly distinguish between *basic* and *advanced* linear algebra libraries for which we set as reference the the original BLAS library [7], and the LAPACK library [3], respectively. These two libraries are the de-facto standard for dense linear algebra and they present a number of implementations by different vendors which basically share the same API (Intel MKL, OpenBLAS [17]).

For the second aspect, i.e. the level of abstraction, all these libraries expose a very low-level interface that is not suitable to be used without a specific expertise. For this reason, a number of wrappers have been developed to provide programmers with a simplest API to such functionalities; the most popular are summarised in the bottom row of Table 1. These tools usually provide a much larger catalogue of functions than those we considered here as basic and advanced Linear Algebra, like sorting, slicing, cumulative summing on arrays and more [13], which are outside the scope of this work. Notice that, some of the libraries reported in Table 1 target only the CPU (BLAS, openBLAS, Intel MKL, Armadillo, Eigen, LAPACK), some others instead make use of CUDA/OpenCL GPUs (CuBLAS, Magma, cuSolver, CULA Dense, clBLAS); among them only ViennaCL, ArrayFire and Lama can use different back-end libraries to target both CPU and GPU.

The present work is aimed to fill two main gaps in the landscape of high-performance linear algebra libraries: apart from ArrayFire (for which this functionality has been published during the late phase of development of this work), none of the presented tools is able to switch the computing back-end at run-time, since they usually need a custom installation mapping to specific libraries or at least a recompilation of the user code; moreover other high-level tools claiming to be able to exploit GPU capabilities, including those mentioned above for other languages, usually refer only to BLAS functionalities, whereas the LAPACK tier is computed on the CPU only. Beside this, none of the mentioned tools (apart from ArrayFire), can guarantee that the code is totally portable from one architecture to another with no modifications of the application source code.

Among all libraries mentioned in Table 1, Armadillo, OpenBLAS, Magma and ArrayFire are particularly relevant for this work. For this we recap their main features.

3.1 Armadillo

Armadillo is a C++ template library for linear algebra with an high-level API, which is deliberately similar to Matlab [13]. It provides a large number of functions to manipulate custom objects representing vectors, matrices and cubes (namely 3rd-order tensors); among these functions it also provides a wrapper to an underlying BLAS and LAPACK implementation (hereafter we will refer to such implementation as the back-end). It employs a number of internal layers in order to translate simple function calls like

```
Solve(A,b); //Solves the linear system Ax=b
```

to more complex but equivalent syntax of LAPACK

```
dgesv( &n, &nrhs, a, &lda, ipiv, b, &ldb, &info );
```

Armadillo is designed to support whatever library providing an API compliant with BLAS and LAPACK, such as MKL or OpenBLAS. It is also able to perform a few of the operations included in the back-end with its own implementation, but they are not designed for high performance.

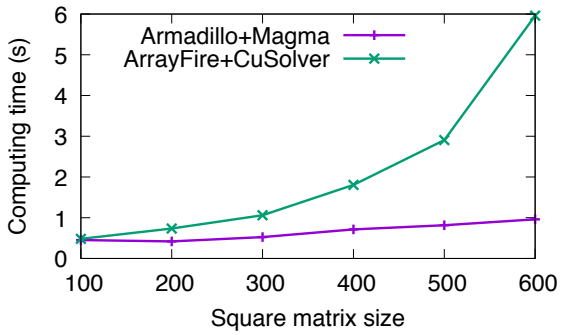


Figure 1: Comparison between Armadillo (with Magma back-end) and ArrayFire (with CuSolver back-end) for computing a Singular Value Decomposition. Experiments are run on a 2×Xeon E5-2660v0@2.20GHz + Nvidia Tesla M2090 platform.

Table 1: Classification of Linear Algebra libraries

	Basic Linear Algebra	Advanced Linear Algebra
Low-level interface	BLAS[7], OpenBLAS[17], Intel MKL, clBLAS, Nvidia cuBLAS	LAPACK[3], Plasma[1], cuSolver, Magma[15], OpenBLAS[17], Intel MKL, CULA Dense[10]
High-level interface	Armadillo[13], Eigen[9], Lama[11], viennaCL[14], ArrayFire[18]	Armadillo[13], Eigen[9], Lama[11], viennaCL[14], ArrayFire[18]

3.2 OpenBLAS

OpenBLAS is an open-source implementation of BLAS which, given the benchmarks provided by the authors, can be compared to the best-in-class proprietary libraries like Intel MKL [17]. The standard distribution of OpenBLAS also provides LAPACK functions, some of which are further optimised by the authors. The interface is compatible with the standard distribution of BLAS/LAPACK.

3.3 Magma

The development of Magma is aimed to replace LAPACK on heterogeneous architectures, with the typical Multicore+GPU platform as a paradigmatic example [1, 15]. The Magma library employs Directed Acyclic Graphs (DAGs) in order to dispatch the different tasks related to a given computation to different cores/devices, taking data dependencies into account and aiming for the best exploitation of the available hardware.

The motivation that drove our interest to Magma is twofold: it does not require the user to take care of data transfer between the host and the device and its API is only marginally different from the standard LAPACK interface. These two features make Magma a good candidate for a *drop-in* replacement of LAPACK on heterogeneous platforms.

3.4 ArrayFire

As we stated before, the two main gaps in the scenario of linear algebra tools can in principle be filled by ArrayFire [18], but a number of reasons led this work to a different solution: fist of all, ArrayFire provides different back-ends (CPU, CUDA and OpenCL), but it did not support run-time switching between back-ends until the late stage of development of this work, preventing it to fulfil the main goal of this work.
Moreover, it shows very poor performance when executing a key use-case related operation like the Singular Value Decomposition: Fig. 1 shows the relative performance of the ArrayFire CUDA back-end (which uses Nvidia CuSolver) and the presented library that relies on Magma as a computing back-end for CUDA.

4 Components and architecture

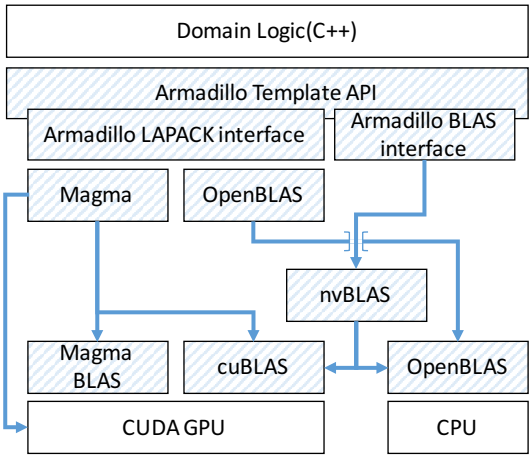


Figure 2: Proposed software stack. Blue arrows show the route of function calls.

As discussed in the previous section, none of the available tools completely fit the requirements, therefore we propose a new software architecture sketched in Fig. 2. The software stack proposed attempts to reuse as much as possible existing libraries as building blocks. This specifically targets one of the declared objectives, i.e. to minimise the code to be maintained. The API exposed to the user is the Armadillo API which is particularly

simple and user-friendly. The software stack, uses a number of the libreries discussed in Section 3, inter-alia Magma and OpenBLAS.

Armadillo answers to the need of a textbook-like API for linear algebra operations, this feature cuts down the development time and the learning curve for the application domain expert who needs to implement a complex business logic. Moreover, its large user-base and active development, represent a valuable aspect when comparing Armadillo to other similar tools. Armadillo relies on external libraries for implementing LAPACK and BLAS operations, for which it exposes an interface that can be easily isolated in the code. This work focused on modifying such high-level interface so that Armadillo is able to select which LAPACK implementation will be used to perform the operations defined by the user within the front-end. In a nutshell, we affected the way Armadillo translates the function names used internally to the conventional LAPACK naming. The overall impact of such extension is limited to three files and a few hundred lines of code, allowing easy portability to future versions of the library.

The choice of using a standard BLAS/LAPACK distribution for the CPU back-end is not particularly critical since there are no other features than performance that can guide the selection. Intel MKL is commonly regarded as the reference implementation for what concern computational speed on Intel CPUs, while OpenBLAS comes very close without the burden of a commercial license, eventually becoming the library of choice for this work. It is worth to remark that, for what concerns this back-end, it is trivial to replace OpenBLAS with MKL or any other library implementation. This can be done simply linking the selected library at compile time when building the new software stack.

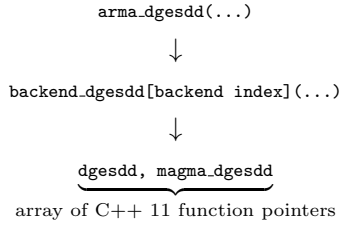
For what concern the heterogeneous back-end, Magma presents a distinctive aspect that makes it standing out: it does not require that the input matrix is already on the GPU when calling a function, i.e. it automatically manage data transfer. Such feature is fundamental in order to provide code portability between CPU and GPU, since all the host/device memory management operations are hidden behind a LAPACK-like interface. Furthermore, Magma does not require the user to know what is going on behind the scenes between the host and the device. The fact that Magma is Open Source software make it a better choice when compared to libraries that in principle would allow the same approach, like for example CULA Dense [10], which is a commercial tool targeted to custom and cluster installations.

4.1 Support for run-time back-end switching

Armadillo makes use of a number of internal layers in order to map the high-level user call to the LAPACK syntax for a given operation. The last step of this translation process points to functions declared as

```
arma_<LAPACK name>( ... )
```

these names corresponds 1-to-1 to the naming convention of the underlying LAPACK implementation (e.g. capitalised, w/o trailing underscore). This mapping has been modified so that it is 1-to-N and allows multiple back-ends to coexist.
From the implementation point of view, the `arma_<>` names has been mapped to multiple C++ 11 function pointers, so that the final translation process can be resumed in the following way (let us consider `dgesdd` as target operation):



With this structure it is possible to dynamically select either the OpenBLAS back-end (`dgesdd`) or the Magma back-end (`magma_dgesdd`) by setting the array subscript to the right value. However, since the default back-end is managed by way of global variable (in thread-local storage), a tiny extension of the Armadillo API is needed to manage the different back-ends. Listing 1 shows the typical usage of the extended armadillo library. The usage of a global variable to percolate information related to a function call (i.e. Armadillo operation) has been carefully weighted during the design stage. Using an additional function parameter to pass this information down to the back-end call were also possible, but this requires to rewrite a

```
// Check if supported CUDA driver and device are present
arma::arma_magma_init();
// Set the Magma back-end and device at run-time
arma::arma_set_backend(1);
arma::arma_set_device(0);
// Domain logic ...
// Finalises Magma back-end for a clean exit
arma::magma_finalize();
```

Listing 1: Typical usage of the extended Armadillo library.

large part of Armadillo run-time support and this is against design principles declared in Sec. 2. The presented solution does not affect Armadillo API and requires a very localised patch to Armadillo implementation, which can be easily propagated to the next Armadillo releases. Notice that, the presented solution should be looked in the perspective of future work where the selection of target back-end will be moved to a fully internal self-optimising mechanism, thus no longer exposed to the programmer. Also, this approach makes it easy adding more LAPACK back-ends, e.g. Plasma [1] and clMagma [5], by simply extending the array to more elements (i.e. function pointers).

BLAS support

The main focus of this work is directed towards the LAPACK class of functions, but a good implementation of the BLAS layer is a fundamental building block of a comprehensive linear algebra library. We separately consider the BLAS calls made within a higher level LAPACK function and the direct BLAS calls made from Armadillo. In the former case, each LAPACK back-end refers to a specific implementation that is internally defined, whereas the latter case is more complex. A BLAS call performed by Armadillo uses a standard BLAS syntax, which would normally map to OpenBLAS, in this case we leverage the nvBLAS library [12] provided by NVidia in order to offload the operation to a GPU when available. Figure 2 shows the relationship between the different libraries included in the framework, including the BLAS layer.

A more sophisticated approach would require the direct usage of the cuBLAS-XT API [12] within the Armadillo BLAS interface, avoiding the need of linking an external library and giving more control to the framework on where to perform a given BLAS operation; indeed such further implementation is planned as a future work.

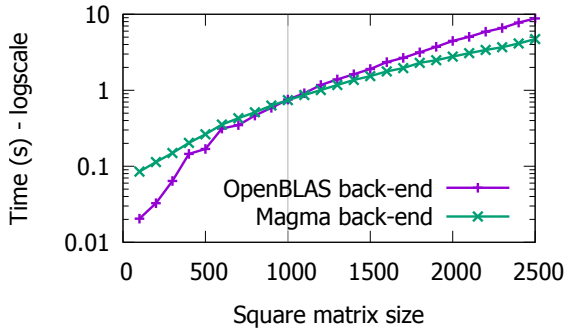
4.2 Multiple concurrent back-end support

The proposed use case involves the possibility to perform a number of identical LAPACK operations working on independent data issued from within a loop. This is a natural setting for further exploiting (either data or stream) parallelism on loop iterations. This scenario requires to further ensure correctness when concurrently running multiple independent back-ends within the same process, called *compartmentalisation*. In this sense, there is the need to carefully study the congruence with compartmentalisation of original Armadillo code (property 1) and the presented back-end switching mechanism (property 2).

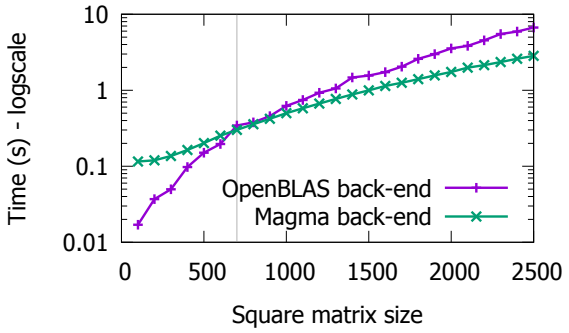
Concerning compartmentalisation, we observed that both Magma (from v2.0) and OpenBLAS can preserve correctness when concurrent calls are made within the same process, but this requires to disable their internal multithreading parallelism. This can be achieved by calling specific functions we exposed within our framework.
The same does not hold for nvBLAS, which is then excluded when testing this framework in a multithreaded fashion, hence direct BLAS calls here are handled by OpenBLAS only. This issue should be addressed with careful integration of the above mentioned cuBLAS-XT. As we shall see in Sec. 5, in the tested cases, the concurrent exploitation of multiple single threaded back-ends is faster than sequential execution of multithreaded back-ends.

For the property 1, we noticed that Armadillo is congruent to compartmentalisation since does not break correctness of the BLAS/LAPACK libraries. The correctness of the property 2 derives from the previous considerations and by implementation design: we perform a *privatisation* of the global variable used to switch between different back-ends by means of thread-local storage. It is worth noting that, since only *concurrently* executing operation needs to be compartmentalised, the run-time structure can be easily moved from *one thread per call* proposed here, to a master-worker structure over a workpool of threads. In the future work, we planned to use the master thread to implement self-optimising *policies*.

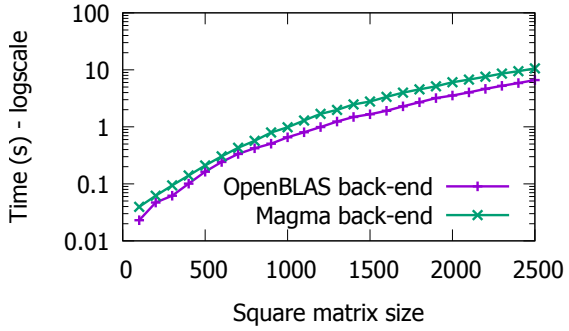
In this work we implemented the *one thread per call* policy by way of the C++11 language features `std::async` and `std::future`, which can guaran-



(a) 2×Xeon E5-2660v0@2.20GHz + Nvidia Tesla M2090.



(b) Xeon E5-2650v2@2.60GHz + Nvidia Tesla K40.



(c) Xeon E5-1650v3@3.50GHz + Nvidia Quadro K620, TCC mode enabled.

Figure 3: Solve $Ax = b$ with `dgesdd`. Performance comparison between OpenBLAS and Magma back-ends.

tee that each asynchronous call lives in a different thread. The next section presents a scenario where the usage of multiple concurrent back-ends brings a significant performance improvement.

5 Experimental validation

The experimental evaluation is aimed to assess that 1) a single back-end is not always better in performance than the others; 2) multiple back-ends may squeeze more horsepower than a single back-end on the same heterogeneous platform. In this latter respect, the scalability of the approach is also an interesting aspect. A comprehensive performance comparison among different back-end libraries is not really needed to fix OpenBLAS and Magma as low-level libraries to target CPU and GPU, respectively. They are widely considered state-of-the-art library under performance viewpoint and are compliant to the requirements stated in Sec. 2. For the sake of exemplification, in Fig. 1 Magma and NVidia's CuSolver are compared on SVD, which is the most expressed operation in Noesis' use case.

5.1 Dynamic back-end support: evaluation

Listing 2 shows a relevant benchmark for what concern black-box models interpolation within Optimus, since it represents more than the 99% of computing time for two of the response surface models included in Optimus. Calculations are carried out using double precision values. Timing considers

```
1  arma::arma_set_backend(backend);
2  // fill coefficients and constant terms to random
       doubles
3  arma::mat A = arma::randu<arma::mat>(n,n);
4  arma::vec b = arma::randu<arma::vec>(n);
5  // declare result vector
6  arma::vec results = arma::zeros<arma::vec>(n);
7  // solve Ax=b with pseudoinverse
8  results.col(i) = arma::pinv(A) * b.col(i);
```

Listing 2: Mock-up of the industrial use-case code: it represents a section of a single-output interpolating model.

```
1 arma::vec lin_solve(const arma::mat &A, const arma::vec &
      b, const int backend=0, const int device=0) {
2   arma::arma_set_backend(backend);
3   arma::arma_set_device(device);
4   arma::vec x = arma::pinv(A) * b;
5   return x;
6 }
7
8 int main(int argc, char *argv[]) {
9   // fill coefficient matrix with random doubles
10  arma::mat A = arma::randu<arma::mat>(n,n);
11  // Declare constant terms and results array of vectors
12  arma::mat b = arma::randu<arma::mat>(n,nDatasets);
13  std::vector<std::future<arma::vec>> results(nDatasets);
14  // solve Ax=b nDatasets times, for different b vectors
15  for (int i = 0; i < nDatasets; i++){
16    // Naive round robin sets the back-end and the device
17    // A is slightly modified each iteration
18    // ...
19    // Solve Ax=b asynchronously
20    results[i] = std::async( std::launch::async, lin_solve
          , A, b.col(i), backend, device );
21  }
22  // gather results
23  for(int i = 0; i < nDatasets; i++) results[i].get();
24 }
```

Listing 3: Mock-up of the industrial use-case code: represents the calculation of multiple interpolating model. Each loop iteration involves an independent computation, hence parallel asynchronous calls are employed.

only line 8 of Listing 2, the LAPACK back-end is used to perform `arma::pinv(A)`, which is based on the divide-and-conquer SVD (`dgesdd`). Notice that line 2 selects at run-time which back-end should be used to perform the calculation, OpenBLAS or Magma. Figures 3a, 3b and 3c compare computation time for the two supported back-ends. The vertical line indicates the break-even point for the Magma back-end.

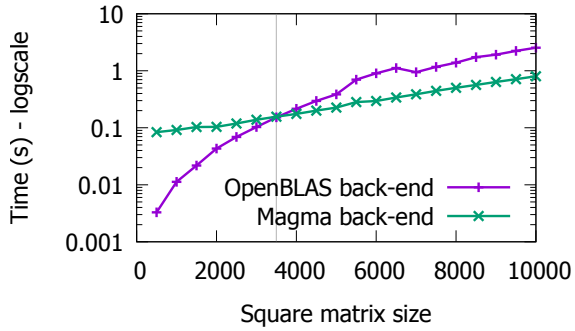


Figure 4: Compute Cholesky decomposition of symmetric positive definite square matrix using `dpotrf`. Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz + Nvidia Tesla K40

Presented results show that there is no best back-end for all conditions: Figures 3a and 3b show how the relative performance of the two back-ends depend on both the size of the matrix to be computed and the specific hardware configuration. The threshold that identifies the break-even point for Magma and OpenBLAS moves accordingly. Figure 3c presents the extreme case where the GPU is very low-end when compared to the CPU. For this case, the hybrid Magma back-end is disadvantageous for any size of the matrix.

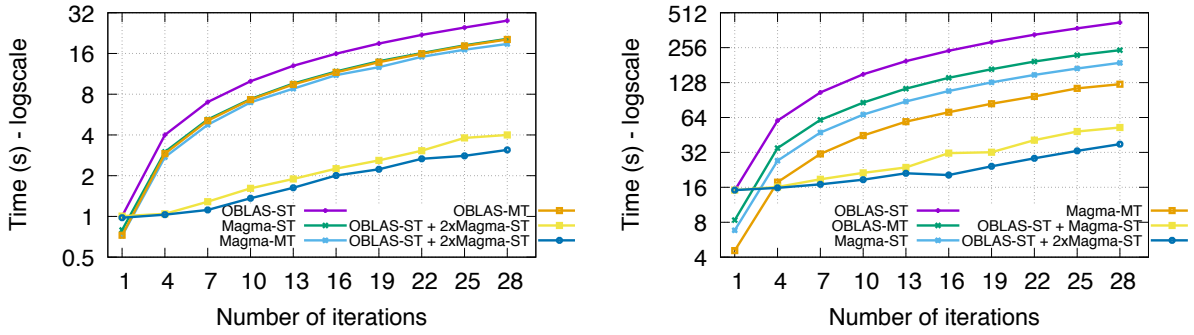
As further test case for this work, we present the results achieved for the Cholesky decomposition of a symmetric positive definite matrix. Such matrix factorization is used as a building block for another Response Surface Modelling technique, the scope of which is strictly related to the main use case. As figure 4 shows, the results achieved follow the same pattern as the previous case, but the break-even threshold for Magma is much different due to the different kernels involved in the computation.

5.2 Concurrent back-end support: evaluation

While in Listing 2 the calculation of the Moore-Penrose Pseudoinverse is carried out only once, in a real-life applications we possibly would like to calculate an interpolating model more than once on slightly different datasets (i.e. cross-validation tasks). In this sense, it is worth to probe the efficiency of implementing an addition level of parallelism in the code. Listing 3 presents a variation of the previous use case, where the matrix A and the constant terms are different for each iteration of the for loop; in this code we implemented the technique introduced in Sec. 4.2 in order to perform each loop iteration on a different back-end/device.

The specific test environment was equipped with 2 identical CUDA devices, hence we actually identified three computing back-ends: OpenBLAS, Magma on device 0 and Magma on device 1. To feed the different devices, we dispatched each iteration of the loop by using a naive round robin strategy, eventually obtaining a number of concurrent activities equal to the number of iterations. In this context, it is not possible to speed up the calculation of a single iteration of the loop, but we can measure the weak scaling of this implementation as the overall performance when computing multiple models with more working elements.

Even if we associate a specific device to the Magma back-end, the ratio of the whole opera-



(a) 1000x1000 matrix.

(b) 2500x2500 matrix.

Figure 5: Computing multiple interpolating model (see Listing 3) with different and multiple back-ends: OpenBlas (OBLAS) and Magma are tested in both MultiThreading (-MT) and SingleThreaded (-ST) versions. Experiments are run on a 2×Xeon E5-2660v0@2.20GHz + 2×Nvidia Tesla M2090 platform.

tion that is actually performed on the GPU depends on the internal implementation of Magma, in this sense the degree of concurrency that can be achieved it is not bound to the number of devices and back-ends, but to the use of the resources that is made by the implementation of the Magma library for the specific operation. Modelling such effects by means of a theoretical study can be an interesting subject for a future work. In this case, for correctness reasons, each back-end works in single threaded mode; for this, and given the previous considerations, there is still room for saturating the computational resources beyond using the three back-ends concurrently, indeed figure 5a shows the clear advantage of such approach compared to the sequential computation with the different back-ends in terms of weak scaling.

6 Conclusions

In this work we presented the design of a novel extension of Armadillo, a high-level linear algebra library, which is able to support multiple back-ends with both their dynamic switching and their parallel execution. This extension happen by way of a very careful design aiming to maximise the likely to interoperate with the (independent) evolution of the Armadillo library and LAPACK-compliant back-end libraries. Specifically, the design of the software stack is compliant with the requirements presented in Sec. 2. The presented prototype, which is under extensive industrial testing within Noesis laboratories, it makes it possible to both 1) achieve a significant performance improvement with respect to previous CPU implementation and 2) to significantly reduce the porting time of the domain logic onto heterogenous platforms.

Experimental evaluation demonstrated that, for the presented use case, there is no such thing like a single best back-end. The relative speed of each back-end depends on both the hardware configuration and the matrix size. For this, dynamic back-end switching is a truly enabling feature for the optimisation of this class of computations. This will require to implement a effective data-dependent scheduling policy, which is a topic for future works. Thanks to compartmentalisation, multiple instances of one or more back-ends can be used in parallel with greater efficiency than those provided by multithreaded versions of all of them. Still, the weak scaling is currently moderate, especially when targeting multiples GPUs, which is again a topic for future investigations.

6.1 Future development

The presented implementation leaves room for a number of improvements that is worth investigating. From the viewpoint of the industrial application, the main concern is to extend the support for multiple hardware architecture, namely to support GPUs from different vendors; this is expected to be achieved via the use of clMagma [5], but its early development stage prevented us from making a working implementation. Whether a LAPACK replacement supporting OpenCL would be avail-

able in the future, the integration in the presented library would be straightforward.

From performance viewpoint a number of issues are still to be refined: 1) testing other state-of-the-art implementations of LAPACK, such as PLASMA [1]; 2) study how to guess the most effective back-end for a given ⟨*hardware configuration, operation, problem size*⟩ in order to define effective scheduling policies; 3) study optimal back-end parallelism degree, i.e. the number of the back-end instances to be run at the same time to saturate computing resources without polluting the memory hierarchy. This two latter points turn out into the need of designing a high-level mechanism to offload asynchronous calls onto a workpool of back-ends with a programmable scheduling policy, as for example implemented in the Fastflow framework [2]. Also, it is possible to rethink the way direct BLAS calls are handled by Armadillo. In this respect the implementation of cuBLAS-XT [12] directly within the Armadillo BLAS interface can be useful. Moreover, it can be evisioned a back-end system also in this case, with the implementation of an OpenCL BLAS library.

Acknowledgment

This work has been partially supported by the EU H2020 Rephrase project (no. 644235), the ITEA2 project 12002 MACH, the EU FP7 REPARA project (no. 609666), and the NVidia "GPU research center" programme at University of Torino.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, 2009.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating code on multi-cores with fastflow. In *Proceedings of 17th International Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181. Springer, 2011.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Python for Scientific Computing Conference (SciPy)*, 2010.

[5] C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clMAGMA: high performance dense linear algebra with OpenCL. In *IWOCL*, pages 1:1–1:9. ACM, 2014.

[6] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[8] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[9] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. Accessed: 2016-11-25.

[10] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. In *Modeling and Simulation for Defense Systems and Applications V*. SPIE-Intl Soc Optical Eng, 2010.

[11] J. Kraus, M. Förster, T. Brandes, and T. Soddemann. Using lama for efficient amg on hybrid clusters. *Computer Science - Research and Development*, 28(2):211–220, 2012.

[12] NVIDIA Corporation. CUDA toolkit documentation. http://docs.nvidia.com/cuda/eula/index.html. Accessed: 2016-11-25.

[13] C. Sanderson and R. Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1:26, 2016.

[14] P. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin. Towards performance-portable, scalable, and convenient linear algebra. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013. USENIX.

[15] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232 – 240, 2010. Parallel Matrix Algorithms and Applications.

[16] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.

[17] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, New York, NY, USA, 2013. ACM.

[18] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos. ArrayFire - A high performance software library for parallel computing with an easy-to-use API, 2015.