The Journal of Supercomputing
https://doi.org/10.1007/s11227-019-02925-3

Toward a BLAS library truly portable across different accelerator types

Eduardo Rodriguez-Gutiez, et al. ***[full author details at the end of the article]***

Abstract

Scientific applications are some of the most computationally demanding software pieces. Their core is usually a set of linear algebra operations, which may represent a significant part of the overall run-time of the application. BLAS libraries aim to solve this problem by exposing a set of highly optimized, reusable routines. There are several implementations specifically tuned for different types of computing platforms, including coprocessors. Some examples include the one bundled with the Intel MKL library, which targets Intel CPUs or Xeon Phi coprocessors, or the cuBLAS library, which is specifically designed for NVIDIA GPUs. Nowadays, computing nodes in many supercomputing clusters include one or more different coprocessor types. To fully exploit these platforms might require programs that can adapt at run-time to the chosen device type, hardwiring in the program the code needed to use a different library for each device type that can be selected. This also forces the programmer to deal with different interface particularities and mechanisms to manage the memory transfers of the data structures used as parameters. This paper presents a unified, performance-oriented, and portable interface for BLAS. This interface has been integrated into a heterogeneous programming model (Controllers) which supports groups of CPU cores, Xeon Phi accelerators, or NVIDIA GPUs in a transparent way. The contribution of this paper includes: An abstraction layer to hide programming differences between diverse BLAS libraries; new types of kernel classes to support the context manipulation of different external BLAS libraries; a new kernel selection policy that considers both programmer kernels and different external libraries; a complete new Controller library interface for the whole collection of BLAS routines. This proposal enables the creation of BLAS-based portable codes that can execute on top of different types of accelerators by changing a single initialization parameter. Our software internally exploits different preexisting and widely known BLAS library implementations, such as cuBLAS, MAGMA, or the one found in Intel MKL. It transparently uses the most appropriate library for the selected device. Our experimental results show that our abstraction does not introduce significant performance penalties, while achieving the desired portability.

Keywords BLAS · Parallel programming · Scientific libraries · Heterogeneous programming · Accelerators · Coprocessors · GPU · Xeon Phi · MIC · CUDA

Published online: 10 June 2019

1 Introduction

The basic linear algebra subprograms (BLAS) specification [9] is widely known in the field of scientific computing [6, 26]. It contains the definition of three sets (called levels) of low-level mathematical routines. They perform vector-vector, matrix-vector, and matrix-matrix operations, respectively. There are several BLAS implementations specifically designed or tuned for different kinds of computing platforms. Some examples include cuBLAS [36], whose functions are optimized for CUDA-enabled GPUs, and the implementation included within the Intel Math Kernel Library (MKL) [22], which exploits the specific features of Intel platforms, such as Xeon CPUs, or Xeon Phi coprocessors.

Despite the potential performance gains of using hardware accelerators, there are several issues associated with their use. First, each coprocessor type has its own programming model, and architecture features which should be taken into account for programming them. Second, the code and data transfers to these devices are costly operations. For a given sequence of calls to BLAS routines, a good choice of the steps where the data has to be synchronized or transferred between host and devices becomes key. The mechanisms available to perform these transfers are not the same for all the different BLAS implementations. Even the same implementation may allow several options with different transparency and responsibility degrees for the programmer. Intel's Math Kernel Libraries (MKL) and cuBLAS/nvBLAS from NVIDIA provide different offloading mechanisms [46], including fully automatic, assisted by the compiler (Compiler-Assisted Offload, CAO), or relying on manually programmed transfers.

In addition, despite the efforts to standardize BLAS [6, 9] and create a uniform interface, in practice the different implementations present subtle differences that must be taken into account. For instance, the same parameters may have to be passed either by value or by reference to different libraries. Different names or constants should be used to indicate the same type of operations or data. Some libraries even have extra parameters which are specific to that implementation, such as a data structure to represent the library context in cuBLAS. Thus, writing a program that uses BLAS routines that target different kinds of computing devices needs the combined use of different libraries, different interfaces, and decisions in the same code.

In this work, we present a software abstraction layer on top of different BLAS implementations. The proposal is implemented in the Controller model [34], which allows the implementation of algorithms as programming units or kernels with a single name and interface, but several implementations. These implementations range from generic ones that use an abstraction which allows its execution on any platform, to those specifically optimized for a kind of platform, or even for a given architecture family. The Controller chooses the most adequate implementation of the invoked kernel based on the platform or device associated to it.

This paper provides the following contributions: (a) An abstraction layer to hide programming differences between diverse BLAS libraries; (b) a new system to define kernel classes that support the context and device manipulation of

different external BLAS libraries; (c) a new kernel selection policy that considers both programmer kernels and different external libraries; and (d) a complete new Controller library interface for the whole collection of BLAS routines. Our current implementation supports internally three different libraries: MKL, MAGMA, and cuBLAS.

This proposal allows the programming of portable software with sequences of calls to programmer-defined kernels and BLAS routines. It provides a uniform interface, allowing the selection of the computational device on which the code executes, by simply changing a single parameter that indicates the target device in the initialization of a Controller object. In this work, we focus on the use of BLAS libraries in a single device at the same time. The library chooses, among those available, the implementation that best suits the device type selected by the user. Internally, it calls preexisting BLAS libraries specifically optimized for the target platform, such as those provided by hardware vendors or third parties. Our proposal provides both code portability and performance. This approach, based on an extensible library, can also be used as run-time to generate portable code from more abstract programming models.

We present an experimental study to verify that the proposed abstraction introduces portability without significantly compromising performance. We use case studies representing different scenarios of using BLAS routines alone, in combination with other BLAS routines, or even with other non-BLAS kernels developed manually. They show that our approach does not impose artificial constraints on the use of BLAS routines in comparison with applications directly programmed with the MKL, cuBLAS, or MAGMA interfaces.

The rest of the article is structured as follows. The second section discusses some related work. The third section is a summary of the features of the background tools used in this work. In the fourth section, the solutions used in our proposal are detailed. The fifth section describes the experimental work carried out to validate this proposal. Finally, the conclusions and future work are discussed in the sixth and last section.

2 Related work

The idea of executing BLAS functions in parallel heterogeneous environments is not new. The specific hardware features of different architectures have been analyzed by various authors to produce libraries or routines targeting specific platforms, for example, IBM 3090 mainframes [29], FPGAs [42], ARM devices [3], and KNL Xeon Phi processors [28].

Heterogeneity has also been partially considered in distributed versions of BLAS. There is a BLAS library designed to exploit distributed parallel platforms called PBLAS, presented by Choi et al. [7]. A derived version, HeteroPBLAS [31], exploits HeteroMPI [27] to execute BLAS algorithms while exploiting distributed memory parallelism on clusters with heterogeneous CPUs. The use of this variant of MPI adds load balancing capabilities. However, HeteroPBLAS cannot take

advantage of the computing power of coprocessors attached to the computing nodes, since it just runs on CPU cores.

The portability of BLAS routines has also been considered. LAPACK [2] is a library of linear algebra routines of a higher level than BLAS. Internally, LAPACK routines are designed to perform a set of calls to a BLAS library. The MAGMA library [11, 44] is an implementation of the LAPACK interface designed to work using GPUs. MAGMA is built on top of two sets of BLAS routines, one designed and optimized to be executed on GPUs (MAGMABLAS) and another for CPUs (BLASF77). Another version, derived from the MAGMA code and thus sharing the same interface, is MAGMA MIC [10], which has been designed for Intel Xeon Phi coprocessors. Finally, clMAGMA [12] is another derivative of MAGMA, which has been rewritten using OpenCL.

OpenCL offers a generic interface to coordinate the execution of parallel code on different types of devices, including CPUs and accelerators. However, for maximum performance, the internal implementations of the BLAS kernel codes should be tuned differently for each type of device, taking into account its architectural features. Thus, the performance portability is not easily achievable. AMD is currently contributing to the development of clBLAS [24], a BLAS library written in OpenCL, which is part of the clMathLibraries. Another BLAS library, specifically designed to target the Radeon GPUs, is rocBLAS [8].

SYCL-BLAS [1] defines an encapsulated C++ template function for each BLAS routine. They are internally described in an abstract form. Several calls to BLAS can be analyzed, fused, or optimized by the compiler. The SYCL system is built on top of OpenCL to allow the generation of compiled versions for different architectures from the same single code. Thus, it cannot benefit from the performance provided by libraries fine-tuned by device vendors. The ArrayFire library [30, 48] presents an abstraction that exploits several BLAS implementations for different devices. However, it is restricted to predefined datatypes, a program can only work with one backend at the same time [13], and the arrays are not portable across devices which need different backends, compromising portability. On the other hand, mixing calls to BLAS routines with kernels manually optimized for a specific device is also restricted. The proposed abstractions introduce potential performance issues that compromise efficiency in some cases [38, 45]. Both SYCL-BLAS and ArrayFire lack support for Xeon Phi platforms using their native programming interfaces or by offloading Intel MKL functions. In this type of devices, these mechanisms perform better than using the OpenCL abstractions and libraries [4].

Armadillo [43] is a C++ linear algebra library based on templates and metaprogramming. It is built on top of BLAS and LAPACK libraries, supporting the use of MKL, OpenBLAS, or nvBLAS for GPUs. Armadillo has been extended by Viviani et al. [45] to allow run-time switching of the BLAS library used as a backend. However, their implementation relies on library routines with automatic data transfer between host and coprocessor. First, only a reduced set of routines available in the BLAS libraries are implemented with these automatic offloading mechanisms. Second, as we will show later, this automatic transfer feature often leads to unnecessary data transfers and poor data reusability, which impacts on performance for chained BLAS calls.

There are other solutions that build higher-level abstractions on top of BLAS libraries while considering heterogeneous platforms, such as [23] and [41]. These solutions do not expose the full BLAS interface to the programmer. The programmer cannot freely use several different BLAS routines in combination with other kernels. In some cases, such as [23], the system does not provide a single level of abstraction and the library programmer needs to deal with specific details of the device and data transfer management. In other cases, such as [41], the programs cannot transparently adapt the BLAS implementation to the target device at run-time. As they focus on GPUs, the generated codes, even those that use OpenCL, are not performance portable to other devices and coprocessor types such as Intel Xeon Phi.

In comparison with all the previously commented approaches, our proposal is designed to offer both code and performance portability, using an expandable set of third-party BLAS libraries. Our solution allows the development of a single high-level code, mixing different devices and backends, abstracting the differences between the available libraries. To obtain the best possible performance, it internally uses the appropriate techniques of the native/vendor programming models for data transfers or for interleaving calls to BLAS with user-defined kernels.

3 Background

In this section, we provide a summary of the features of the tools and libraries used as background for the development of this work.

3.1 Specialized versions of BLAS

BLAS, acronym for Basic Linear Algebra Subprograms, is the specification for a set of low-level linear algebra routines. BLAS supports four base datatypes for arrays. Every routine usually has four different versions, one for each datatype. The datatype is noted on the function name by using an `s` when the function is designed for single-precision floating point numbers (e.g., SGEMM); a `d` for double-precision floating point numbers (e.g., DGEMM); a `c` for complex numbers stored as two single-precision floating point numbers (e.g., SGEMM); and finally `z`, for complex numbers stored as double-precision floating point numbers (e.g., ZGEMM). An asterisk (*) is usually used instead of the corresponding datatype identifier when speaking generally about a routine.

Several routines are defined on each BLAS level. On level 1, we can find routines to perform scalar-vector or vector-vector operations. One instance is the *AXPY routine, which multiplies every element of a vector by a scalar. Another example is I*AMAX, which, given a vector, returns the index of the cell that stores the highest value. Level 2 includes routines for matrix-vector operations, such as *GEMV, which returns the result of multiplying a matrix by a vector. Finally, level 3 contains matrix-matrix operations, such as *GEMM, which obtains the result of multiplying two matrices.

On each device-specialized implementation of BLAS, we can find several offloading mechanisms to move data and execute code in coprocessors. For instance, MKL allows the *Automatic Offloading* (AO) [20] of specific functions to a Xeon Phi. This mechanism implies the automatic movement of the input data toward the coprocessor and the retrieval of the outputs back to the host. These data movements are performed transparently to the programmer. Even though the user has to explicitly enable this kind of mechanism, the decision of whether it is activated or not for a specific call to the MKL BLAS library depends exclusively on the implementation. The run-time takes into account the data sizes to realize whether it is worthwhile offloading the function call [21]. The offloading to the coprocessor is only executed when the library, taking into account data sizes, realizes that it is worth executing the BLAS function on the coprocessor, despite the overheads associated with the data transfers [21]. Otherwise, the routine executes on the host. This feature is only available for a reduced set of BLAS functions. Sequences of calls to those functions lead to multiple unnecessary data movements [5].

Another option is the use of the Intel Language Extensions for Offload (LEO). This consists of several annotations (pragmas) for the Intel compiler that allow the execution of explicit data communications between the host and the device whenever needed [35]. The interface of LEO is abstract and hides the low-level details and primitives for doing operations with the coprocessor, such as memory allocation, data transfers using device pointers. Invocations to one or several MKL functions inside LEO blocks are automatically executed in the coprocessor. This is known as the Compiler-Assisted Offload (CAO) mode. The LEO pragma specifies the data transfers or the reuse of arrays transferred by previous LEO pragmas. This is a more general solution than the previous one. All BLAS functions in MKL can be executed using this technique and the data can be reused between routine calls, thus avoiding unnecessary data movements. However, the decisions become the programmer's responsibility and the movements have to be synchronized with the start or end of a LEO block. This option allows a greater degree of control at the expense of a greater effort to determine and code the optimal movements. Another drawback is the lower portability of the resulting code.

NVIDIA provides cuBLAS, a library specifically tailored to run on CUDA capable devices. This vendor also provides an alternative library, named nvBLAS, with a solution similar to the MKL automatic offloading mechanism. It is completely compatible with the BLAS interface. Like AO in MKL, the run-time system may decide not to execute the function in the device, while the mechanism is also available only for a specific set of functions [37].

When the normal cuBLAS implementation is used, the memory transfers between host and GPU must be managed through the usual calls to the CUDA library. This is equivalent to the use of Intel LEO, but with a lower-level API. It gives more control, but requires more development effort and reduces portability.

MAGMA [11, 44] emerges as the evolution of the LAPACK library, optimized for GPUs. Several cuBLAS functions, such as GEMM, have incorporated some of the MAGMA BLAS optimizations, and therefore, MAGMA has replaced their own implementations by calls to cuBLAS [14, 15]. MAGMA functions use a hybrid automatic approach, breaking the LAPACK routines into tasks. These tasks

are dynamically scheduled to be executed on the GPU or on the host CPU. Subsequently, a MAGMA version, called MAGMA MIC [10], was designed for Xeon Phi coprocessors, based on offloading through the use of the Xeon Phi low-level driver functions (libSCIF and COI). For the programmer, and in terms of data transfers, the use of MAGMA or MAGMA MIC is equivalent to the use of Compiler-Assisted Offload.

There are several differences between MKL, cuBLAS, or MAGMA interfaces. The MAGMA BLAS interface for the latest version available online (2.2.0) is pretty similar to cuBLAS. However, the second version of the cuBLAS API requires a new *handle* parameter that is implementation specific. This is a data structure that stores information about the library context and should be created, passed as first parameter on each call to this library, and finally destroyed when no more BLAS routines are to be used.

Minor differences arise in the way of passing specific parameters in MKL, cuBLAS, MAGMA and MAGMA MIC. For instance, scalars alpha, beta, etc., have to be passed by reference (using pointers) on all cuBLAS calls, whereas they are passed by value on Intel MKL. Another problem lies in the different values that each implementation assign to the constants or enumerated datatypes used to define operations, such as `cublasOperation_t` or `CBLAS_TRANSPOSE`. CBLAS libraries (the C binding of the original BLAS) have an extra parameter to declare if the input matrix must be accessed in *row-major order* (common for programming languages like C) or *column-major order* (used by software coded in Fortran). MAGMA MIC is based on MAGMA. Thus, very minor differences appear between these two implementations, comparing with both MKL or cuBLAS.

3.2 The original controller model

This work aims to provide a functional and performance portable BLAS library. Thus, as the foundation of our implementation, we select a programming model designed for heterogeneous computing, which allows the development of specialized versions of the same kernel for different architectures or programming platforms. Different levels of abstraction and granularity are supported, providing a lot of versatility and control to the system-software programmer. The selected programming model and framework is the Controller model [32–34]. In this section, we describe the main features that support its selection.

The Controller model for heterogeneous programming is implemented as a modular library, extensible and portable. It is programmed in C language, with an object-oriented approach in mind. It is usable and interoperable across both C and C++ languages. It introduces a new abstract entity, called *Controller*, which is associated with a device on creation. This device can be either a GPU, an Intel Xeon Phi (MIC) or a group of CPU cores that are managed transparently like an accelerator. The Controller entity includes mechanisms to: (1) attach or detach a data structure to/from its context; (2) create or destroy data structures which are only allocated in the device context; and (3) launch kernels (computing routines) that are executed on the device.

In order to achieve uniformity on the data structure management, the Controller model implementation relies on the Hitmap library [17]. This library allows the definition of indexed data structures, such as arrays or graphs. Data structures are instances of a structured type named *HitTile*, which contains metadata and pointers to the memory areas containing the data. Hitmap allows the declaration and manipulation of HitTiles with base types which can be either native or structured types declared by the programmer (see lines 2–5 in Fig. 1). The library includes a uniform interface to declare index spaces, the construction or destruction of HitTiles, and access to the data stored in a HitTile. The same interface is used to access data in both the host and kernel codes, with a portable row-major order view (see lines 16, 21, and 24 in Fig. 1).

In order to define a kernel, the Controller library uses a generic interface declaration where the kernel name, number and names of parameters, datatypes, and their Input/Output role are detailed. Kernel parameters in the Controller model are values of native or programmer-defined types or references to HitTiles of any base type. (see lines 9–13 in Fig. 1). A kernel with a unique name can be declared several times by using different platform identifiers. Thus, each declaration can be tuned for the specific architecture, even using pragmas, datatypes, or primitives which are specific to a programming model and the compiler that generates the code for it. For instance, it is possible to use CUDA syntax and primitives on kernels declared for GPUs in order to reserve and use shared memory, synchronize threads in a block, etc. There are situations where the code of a kernel is simple enough for it to be appropriate for any device. They can be simple prototype kernels or simple operations that do not require specific device optimizations. Some examples include the

```
1  /* Structured type for 2D points */
2  typedef struct {
3          int x, y;
4  } Point2D;
5  hit_tileNewType( Point2D );
6
7  /* Controller kernel: scopy */
8  /* Relocate data in 1D arrays with strides */
9  CAL_KERNEL_GENERIC( scopy, 5, IVAL, int, n,
10                                IN, HitTile_Point2D*, x,
11                                IVAL, int, incx,
12                                IO, HitTile_Point2D*, y,
13                                IVAL, int, incy )
14 {
15    int row = threadId.x;
16    hit(x, row*incx) = hit(y, row*incy);
17 }
18
19 ...
20    // Domain declaration and allocation
21    HitTile_Point2D vA = hitTile(Point2D, hitShape(size));
22
23    // Initialization of one cell in the host
24    hit( vA, 3 ) = { 10, 20 };
25 ...
```

Fig. 1 Example of types and kernel declaration, and HitTile manipulation, using the Controller model

initialization of a single matrix element, a dot product, the update of a matrix element with the values of the neighbor cells in a stencil computation, etc. To avoid the declaration of several kernels with the same code, one for each device, the Controller model also provides a GENERIC type of kernel. The programmer can use this to declare new kernels that include only neutral code that can be executed on any kind of platform (see an example in line 9 of Fig. 1). The Controller provides a thread-index identifier space similar to that of CUDA or OpenCL, except that it is defined in row-major order for all types of devices (CPUs, Xeon Phi, GPUs). In CPU and Xeon Phi devices, it makes the code easier to be vectorized. For GPU devices, the indexes are automatically adapted to column-major order to preserve the coalescing property when neighbor threads access neighbor data. This improves the portability of the kernel codes. The thread-space and kernel definitions are done in fine grain. The Controller is responsible for grouping the threads into blocks (for GPUs) or generating coarse-grained OpenMP tasks that internally iterate with loops through their local index space for an efficient execution.

The HitTile real parameters in a kernel launch should have been previously attached to, or created in, its context. The attachment of a data structure to a Controller context implies that the data in the host is transferred to the image created on the device context before it is used by a kernel. In the detachment operation, if the structure has been modified on the device (it has been used as output by any previously launched kernel), the updated data is copied back to the memory image in the host. Memory transfers to the device can be either *eager* or *lazy*. They are executed during the attachment operation or they are delayed until the data is needed by a launched kernel. The Controller is responsible for managing the queue of kernel execution requests and for handling the memory transfers transparently.

This system of kernel declarations allows the progressive development of a library of specialized versions of each kernel. The Controller selects the most appropriate version for the target architecture by prioritizing it over the generic ones.

The Controller model includes one more which is key for the development of kernel libraries. It includes the possibility of declaring a special type of kernel whose code is executed in the host, even if the target device is an accelerator. This allows the creation of kernels which execute in the host, but include calls to specialized libraries suited for a specific type of device. However, this type of kernel does not allow the generation, storage, and retrieval of the context information and device configuration needed by some BLAS library implementations.

4 Solution proposal

In this section, we describe our proposal to achieve a portable and efficient BLAS library. In this work, we revise and extend the Controller model to implement a multidevice/multilibrary backend. The purpose is to transparently support the different BLAS versions that can be used for different types of devices with a single unified interface. The new proposal is based on modifying the following features of the original Controller model: (a) change the original library kernel structure to support library contexts and its device configuration; (b) introduce a new extended

Table 1 Kernel classes for the Controller model, indicating which classes can be matched with each Controller type

Kernel class	Controller type	Content/library
GENERIC	*Any*	Generic code for any device
CPU	CPU	Optimized native CPU code
libCPU	*CPU*	*Calls to Intel MKL*
CUDA	CUDA	Optimized CUDA kernel code
libGPU	*CUDA*	Calls to *cuBLAS*
libMAGPU	*CUDA*	Calls to *MAGMA*
XPHI	Xeon Phi	Optimized Xeon Phi kernel code
libXPhi	*Xeon Phi*	Calls to *Intel MKL*
libMAXPhi	*Xeon Phi*	Calls to *MAGMA MIC*

The new introduced or modified classes are highlighted in italics

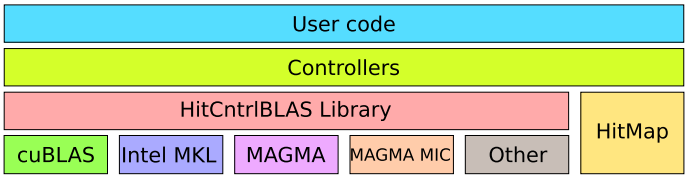

Fig. 2 Abstraction layers and tools used in the proposed solution

classification of kernels to support different tagged library implementations for the same routine; (c) design a more sophisticated policy module for kernel selection that also allows the prioritization of different libraries for each device type. These new features provide a flexible run-time priority mechanism to select the most appropriate library kernel version for the target architecture. Then, we can build a library of Controller kernels that works as an abstraction layer on top of any BLAS library (see Fig. 2), providing a full, unified and portable library interface for BLAS.

Library kernel structure modification For each type of device, the original Controller model supported only one specialized kernel and one non-specific library kernel with the same name. The original non-specific library kernels structure did not support context information or accessing the device as required by some BLAS implementations. Thus, the original library kernel structure is redesigned to allow both the manipulation of the device configuration and the generation, storage, and retrieval of library context information when needed. With the new changes, the library kernels can access the device directly. The Controller internal structure is updated to include an extension to store generic context information and device configuration for libraries. The structure of the general Xeon Phi kernels is also adapted in the new Controller version to support the interoperability of BLAS routines with manually developed kernels, avoiding performance degradation.

Kernel classes To transparently support several alternative BLAS libraries for the same device type, we have extended the kernel classes. The Controller library has an enumerated type for the names of the kernel classes that can be used by the programmer. It is straightforward to introduce new class names and to use

them to define kernels. We propose to use a name defining a new type of kernel for each vendor-provided or third-party scientific library, on each device (see Table 1). With these new classes, we can define kernels with the same name and interface, whose code invokes functions of the desired library and device type.

Selection policy The second part is to modify the internal method of the Controller that selects the version of a given kernel, which is executed for a given target device. Supporting several library implementations of the same function for the same device implies a decision about which version is prioritized at launching time. It is also possible that some functions are defined in one library implementation, but not in others. We define a selection policy module inside the Controller that works with an order or priority list of kernel classes for each type of device. In the current implementation, a fixed policy is hardcoded into the Controller entity code. This policy defines, for each kernel name used in a launch operation, which implementation is selected. It chooses one of the versions available from among those declared in the source file or through included header files. If no implementation is available at compile time, a compilation error is raised. The current run-time policy prioritizes the implementation that uses the specific vendor library (libCPU, libXPhi, or libGPU). If this version is not present at compile time, it uses the kernels that use the MAGMA library version (libMAXPhi, or libMAGGPU). If it has still not been found, it looks for a kernel provided by the programmer (the original kernel classes CPU, GPU, XPhi). Finally, if no other implementation is found, it executes the non-optimized kernel, designed for any kind of device (the original GENERIC kernel class). This kernel selection stage does not fail at run-time. At least one eligible kernel implementation should be provided at compile time in order to generate the executable.

Multi-library Finally, to provide the common interface for each BLAS function name, we develop a collection of kernels, which are wrappers to call the actual functions of the corresponding library. This intermediate layer is a library named HitCntrlBLAS (see Fig. 2). In this library, each BLAS function is a Controller kernel with the same name and five different implementations. There is one implementation for each new kernel class. Each one represents a combination of device and library type (MKL for CPUs, MKL for Xeon Phi, MAGMA MIC for Xeon Phi, cuBLAS for GPUs and MAGMA for GPUs). Lines 2–10 in Fig. 3 show an example of kernel implementation for the *scopy* function of BLAS, targeting Xeon Phi, calling the MKL implementation.

The proposed interface for each function is common and uniform. Inside each implementation, the details needed to perform the proper call to the corresponding library are managed. The Controller internal fields have been extended to support any contextual data needed by a specific library. For example, the libGPU kernel implementations manage internally the handle needed for cuBLAS calls. See lines 14–21 in Fig. 3. This handle is created and destroyed with the Controller if it is associated with a GPU during the creation operation. Other differences regarding the APIs of the libraries used are solved within each kernel implementation.

Data transfer management has been previously solved inside the Controller library. It uses either CUDA or CAO, depending on the type of device associated

```
1  /* CAL_BLAS_XPHI.cpp */
2  // Declaration of the MKL Xeon Phi-targeted implementation
3  CAL_KERNEL_LIBXPHI( scopy, 5, IVAL, int, n,
4                                IN, HitTile_float*, x,
5                                IVAL, int, incx,
6                                IO, HitTile_float*, y,
7                                IVAL, int, incy)
8  {
9     cblas_scopy(n, x.data, incx, y.data, incy);
10 }
11
12 /* CAL_BLAS_LIBGPU.cu */
13 // Declaration of the LIBGPU-targeted implementation
14 CAL_KERNEL_LIBGPU( scopy, 5, IVAL, int, n,
15                              IN, HitTile_float*, x,
16                              IVAL, int, incx,
17                              IO, HitTile_float*, y,
18                              IVAL, int, incy )
19 {
20    cublasScopy(comm->handleCUBLAS, n, x.data, incx, y.data, incy);
21 }
```

Fig. 3 Example of the declaration of two different implementations of the *scopy* BLAS kernel, in the HitCntrlBLAS library: *libGPU* for GPU-CUDA devices using cuBLAS, and *libXPhi* for Intel MIC Xeon Phi using MKL

to the Controller. The HitTile structures from the Hitmap library also provide a uniform interface for data structure management before or after the third-party library call.

The result is a system that allows the programmer to develop a single main program, using the Controller entity to launch generic BLAS kernels with a portable and uniform interface. The programmer can select the available libraries or change the selected devices with a single parameter value on the Controller object construction. An example for a study case is presented in the following section.

Main code example Figure 4 presents a snippet of the main coordination code of a Controller program to compute a sequence of matrix-matrix multiplications. It uses the modified Controller model and the HitCntrlBLAS library previously discussed. In lines 2–9, the Controller system is initialized and the data structures needed in the host are declared and initialized using HitTiles. Line 12 shows the construction of a Controller associated to the first GPU on the system. In lines 15 and 16, we show how an internal variable, with memory only allocated in the device, is created for the *C* matrix. The HitTile is declared as a tile with no memory allocated in the host. This tile structure is used to declare the array dimensions, to store the pointer to the device data when it is created in the Controller (line 16), and to track the HitTile use in the kernel launching functions. In line 19, the two input matrices are attached to the Controller context, implying data transfers to the device. In line 22, an object representing the index space of logical threads is constructed. This space is internally transformed into a grid of groups (for GPUs), or a collection of coarse-grained tasks (for OpenMP threads in CPUs or Xeon Phi). It is used in the kernel launch operations found in lines 26–31. These lines represent the main code of the algorithm, launching kernels of the HitCntrlBLAS library. Line 35 detaches a matrix

```
1  int main (int argc, char* argv[]) {
2    CAL_CntrlInit( CAL_MANAGER_BASIC );
3
4    // Index domain declaration and allocation for each matrix/vector
5    HitTile_float matrixA = hitTile(float, hitShape((rows), (columns)));
6    HitTile_float matrixB = hitTile(float, hitShape((rows), (columns)));
7
8    // Matrix initialization in the host (example of modifying one cell)
9    hit( matrixA, 3, 5 ) = ... ;
10
11   // Controller creation
12   CALCntrl ctrl = CAL_CntrlCreate( CAL_CNTRL_GPU, CAL_DEVICE_0 );
13
14   // Creation of structure only allocated in the accelerator
15   HitTile_float matrixC = hitTileNoMem(float,hitShape((rows),(columns));
16   CAL_CntrlInternal(&ctrl, matrixC);
17
18   // Attaching matrices to the Controller context
19   CAL_CntrlAttach(&ctrl, matrixA); CAL_CntrlAttach(&ctrl, matrixB);
20
21   // Domain of logical threads
22   CALThread threads = CAL_ThreadInit( rows, columns );
23
24   // Kernel launching
25   for (int times = 0; times < final_power; times++) {
26     CAL_CntrlLaunch(ctrl, sgemm, threads, 13, &CAL_NO_TRANSPOSE,
27                     &CAL_NO_TRANSPOSE, rows, columns, columns,
28                     1.0, &matrixA, columns, &matrixB, columns,
29                     0.0, &matrixC, rows);
30     CAL_CntrlLaunch(ctrl, scopy, threads, 5,
31                     rows*columns, &matrixC, 1, &matrixA, 1);
32   }
33
34   // Bring results to host and free device memory
35   CAL_CntrlDetach(&ctrl, matrixA);
36   // Free the rest of device and host memory
37   CAL_CntrlDetach(&ctrl, matrixB);
38   CAL_CntrlDestroyInternal(&ctrl, matrixC);
39   CAL_CntrlDestroy(&ctrl);
40   hit_tileFree(matrixA); hit_tileFree(matrixB);
41 }
```

Fig. 4 Code snippet of the main program for a sequence of matrix-matrix multiplications, using Controllers and HitCntrlBLAS libraries. In this case, the Controller is attached to the first GPU found in the host

used as an output parameter during the launch operations. Thus, it implies a transfer of data from the device to the host. Lines 37–40 detach, destroy, and free the rest of the data structures in both the device and the host.

5 Experimental study

This section presents an experimental study with two objectives. The first is to show the portability of the approach. The second is to show that this portability does not compromise efficiency. We test that the proposed abstractions do not impose a relevant performance overhead when comparing with the same programs developed with the original BLAS libraries and manually tuning the memory transfers.

The source code of both the library and the case study programs, together with the experimental data obtained during the performance evaluation, is available upon request to the authors.

5.1 Case study: problems description

We study three different problems that present different challenges for a heterogeneous programming model, showing different features of the proposal and covering different scenarios found in a wide range of application classes. The first is a loop of calls to the same BLAS routine, a *GEMM matrix-matrix multiplication, which is a key kernel in multiple scientific applications and a classic test-bed case for high-performance programming systems. The main kernel of this program presents a computational load that increases quickly with the input-data size, quickly diminishing the effect of the data transfers. It allows the efficiency of the kernel launching and execution procedures to be tested easily. The second implements a classical GEMVER algorithm using a sequence of calls to different BLAS libraries. This program uses kernels with a low computational load. It allows the testing of the interface and parameter passing across BLAS routines, as well as the efficiency of the device and data transfer management for a wide range of input-data sizes. The last one is a program that applies the Sobel filter to an image, mixing kernels developed manually by the programmer with calls to BLAS routines. It tests the interface across the different programming abstractions, library and manually developed kernels. It also presents a better balance than the previous cases between data transfer and computation times.

Matrix multiplication sequence The first computes $C = A \times B^n$. It multiplies a square matrix A by another square matrix B several times [47]. We choose an implementation based on a loop that computes the recursive expression: $C_k = C_{k-1} \times B : k \in [1, n]$, where $C_0 = A$. Several BLAS libraries, such as Intel MKL, do not allow the use of the same variable as input and output on the same call. Therefore, the result of multiplying $C_k \times B$ cannot be stored back in C in the same function call. This forces the programmer to use a third matrix, allocated only on the device memory, to store the partial result. We use a second kernel on each iteration to copy the result values from the temporary matrix to the first matrix operand. This second kernel has no computational load, just memory accesses. Our implementation has two different parameters to play with, the number of times that the matrix B is multiplied and the matrix sizes. Before the main iteration, both initial matrices A and B are copied to the device memory.

**GEMVER* Although this operator has been introduced as a single function in the extensions proposed for the BLAS 2.5 standard [6], it has not yet been implemented in the current versions of MKL, cuBLAS, or MAGMA. This algorithm takes as input a matrix A, six vectors u_1, u_2, v_1, v_2, y, z and two scalars, α and β. It executes the following operations with them:

$$\hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T$$
$$x \leftarrow \beta \hat{A}^T y + z$$
$$w \leftarrow \alpha \hat{A} x$$

where $\hat{A}$ and x are internal variables (matrix and vector, respectively) and w is the output vector [19]. The version of this algorithm used to create the GEMVER reference program is included in Polybench [40]. This algorithm is suitable for several interesting optimizations. Polybench contains two code versions, one programmed directly and another using BLAS function calls.

Sobel operator The Sobel operator [16] is a well-known image processing algorithm for edge detection. It applies two *stencil operators* to a grayscale input image. Each operator obtains the *derivatives* in the x and y directions, denoted as G_x and G_y, respectively. We use 3×3 sized Sobel stencil matrices, called *kernels* or *filters* in image processing literature. The use of separable filters may result in some performance improvement [39]. Finally, the gradient magnitude is computed by squaring each cell value on G_x and G_y, adding the two resulting matrices and computing the square root of each cell.

This case shows how our solution allows the programming of codes that mix generic BLAS library calls with kernels directly developed by the programmer. These kernels can include code and optimizations which are specific to a lower-level programming model such as CUDA. The two filter operators are implemented with the original kernel classes in the Controller implementation. The cell-wise squaring operation can be implemented as the Hadamard [18] product of a matrix, which can in turn be implemented by using the *SBMV BLAS operation with a band matrix [25]. However, this function is not implemented in MAGMA and MAGMA MIC. In order to obtain comparable results across the different libraries, we integrate this operation into the columns convolution kernel instead. The sum of the two resulting matrices is done using the BLAS function *AXPY. Finally, the cell-wise square root of the resulting matrix is implemented again using the original kernel classes.

5.2 Implementation details

In this section, we describe how the Controller and reference versions of each problem have been implemented.

Reference implementations In order to check potential performance overheads introduced by our abstractions, we need reference implementations of the three problems considered. The first reference version of each problem is developed using C with calls to MKL, targeting the Xeon Phi platform. They include specific Intel compiler pragmas to execute the BLAS routines through Compiler Assisted Offload, optimizing the data transfers. The second and third reference versions target NVIDIA's GPUs. They use C++, CUDA, and either cuBLAS or MAGMA to generate the GPU programs. It has been verified in all cases that the BLAS functions are truly executed on their respective coprocessors, while also checking that the compiler or run-time system does not take the decision of running any part of the function on the host.

Controller-based implementations The first stage of implementation is the development of the HitCntrlBLAS library. cuBLAS and Intel MKL kernel versions are coded in different source files, in order to compile each of them with their specific compilers. Header files are created for each *sublibrary*. The second stage is the

implementation of the case studies. We program only one Controller-based main code for each problem. The experiments with each library or device can be done modifying only the value of the parameter that specifies the device in the Controller construction.

5.3 Performance results

We conduct experiments to analyze the potential overhead introduced by the Controller abstractions in our current implementation. We execute the different versions of the codes in two different machines, with four different coprocessors. The study considers two families of Xeon Phi coprocessors (KNC: 3120A Knights Corner, and KNL: 7220A Knights Landing), and two different architectures of NVIDIA GPUs (Maxwell: Titan X, and Volta: Tesla V100). The details of the machines, coprocessors, operating system, development tools, and versions of the BLAS libraries are summarized in Table 2. The programs are executed using the *numactl* command to set the affinity of the program threads to the NUMA node which is associated with the PCI bus where the coprocessor is connected. This greatly reduces the stochastic performance behavior of the data transfers from/to the coprocessor on different executions.

Figures 5 and 6 show the execution times obtained for each experiment configuration and target platform. The y-axis represents the execution time in logarithmic scale. The x-axis represents the input-data size n, where n is the cardinality of each dimension of the square input matrices. The sizes range from very small (100×100) with almost negligible computational cost comparing to the data transfers, to very large sizes (17000×17000), up to the point of exhausting the device memory in some cases. The values represent the mean of ten executions for each experiment.

Figures 5 and 6 show that the Controller programs present similar performance behavior as their equivalent reference versions in all cases, with the lines

Table 2 Hardware and software features of the machines used in the experiments

Feature	Chimera	Manticore
Operating system	CentOS 7.4.1708	CentOS 7.5.1804
Processor	Xeon E5-2620v3	Xeon Platinum 8160
Clock speeed	2.40GHz	2.10GHz
Main memory	32 GB DDR3	256 GB DDR4 ECC
Cache memory	15 MB	33 MB
Coprocessor		
Intel	Xeon Phi 3120A	–
Intel	Xeon Phi 7220A	–
NVIDIA	GTX Titan X	Tesla V100 PCIe
Tools		
Intel	XE 2017.5.061	XE 2019.0.117
NVIDIA	CUDA 9.0.102	CUDA 10.0.130
GCC	4.8.5	4.8.5
MAGMA	2.2.0	2.5.0

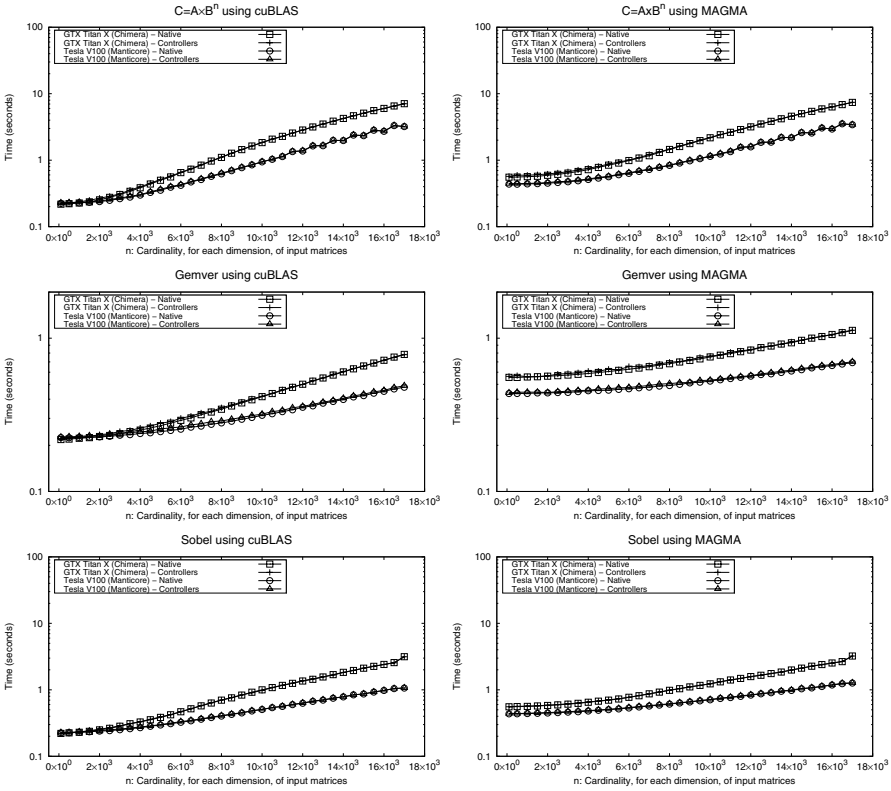

Fig. 5 Experimental results for the three case study problems ($C = A \times B^n$, GEMVER, and Sobel), executed on NVIDIA GPUs, using cuBLAS (left column) and MAGMA (right column). Each plot includes the execution times of the reference versions doing direct calls to the chosen BLAS library and the *Controller* version internally using the same library

representing the results of a Controller program and its equivalent reference version mainly overlapped. This happens consistently across the different scenarios generated by the selected examples and input-data sizes, in terms of load balance or computational vs. device management costs.

Several interesting effects related to the behavior of the different library implementations and platforms can be observed. As commented at the beginning of Sect. 5.1, for small input-data sizes, all problems present the same scenario, where the computational load of the kernels is really small and the device management and data transfer between host and device are the bottleneck. The left column in Fig. 5 presents the results using the native cuBLAS library for the two GPUs. In these plots, for the smallest input-data sizes, the performance of both the Titan X and the Tesla V100 are similar. However, when using the MAGMA library for GPUs (right column of the same figure), we observe that the times of device management and data transfers is higher than when using cuBLAS, and less efficient for the older Titan X than for the Tesla V100. Due to its low computational cost

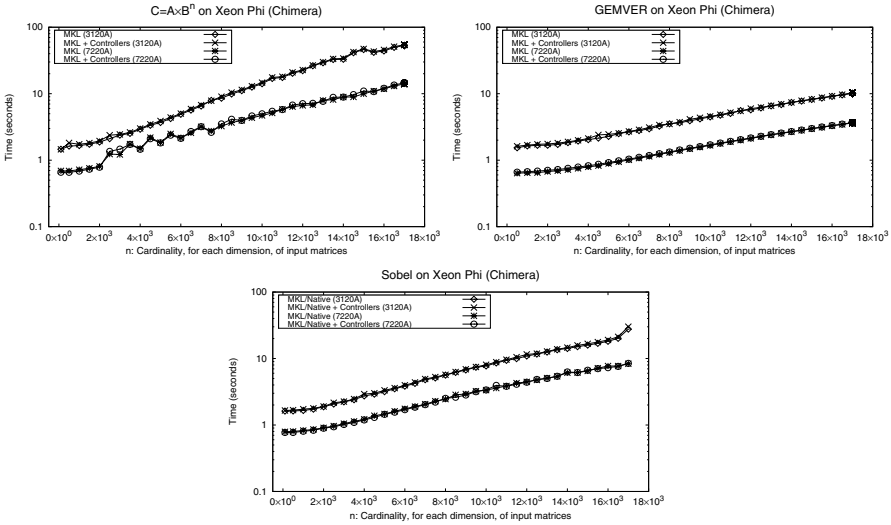

Fig. 6 Experimental results for the three case study problems ($C = A \times B^n$, GEMVER, and Sobel), executed on the Xeon Phi platforms using the BLAS implementation of the MKL library. Each plot includes the execution times of the reference versions doing direct calls to the MKL library and the *Controller* version internally using the same library

kernels, the GEMVER problem is dominated by the device management times, even for the biggest input-data sizes (see middle row of the figure). Thus, the cuBLAS versions consistently present a better performance than the equivalent MAGMA versions for all the range of input-data sizes tested. Nevertheless, as the computational load grows slightly with the data sizes, the execution times of the Tesla V100 and the Titan X consequently diverge due to the higher computational power of the Tesla V100. In the case of the Matrix Multiplication Sequence (the two plots in the top row of Fig. 5), and the Sobel program (the two plots in the bottom row of the same figure), the kernel computational costs grow much faster with the input-data sizes than in GEMVER. The execution times of the kernels quickly grow and dominate the overall performance. Thus, when the data sizes grow, we observe that the cuBLAS and MAGMA versions converge to the same performance values (faster in the matrix multiplication sequence, as expected), confirming that the MAGMA overheads are originated in the device and data transfer management. Figure 6 presents the results for the Xeon Phi programs using the MKL BLAS library. The more modern KNL architecture consistently performs better in both data transfers and kernel execution. It presents a performance improvement of approximately 2.5× comparing with KNC.

To better analyze the potential overheads introduced by the Controller abstractions, we analyze the ratio between the execution time of each Controller program and the execution time of the equivalent reference version (the one that uses the same external library in the same platform). Figure 7 presents examples of the ratio results for some representative combinations of case study problem, platform, and library implementation. Other combinations present similar results.

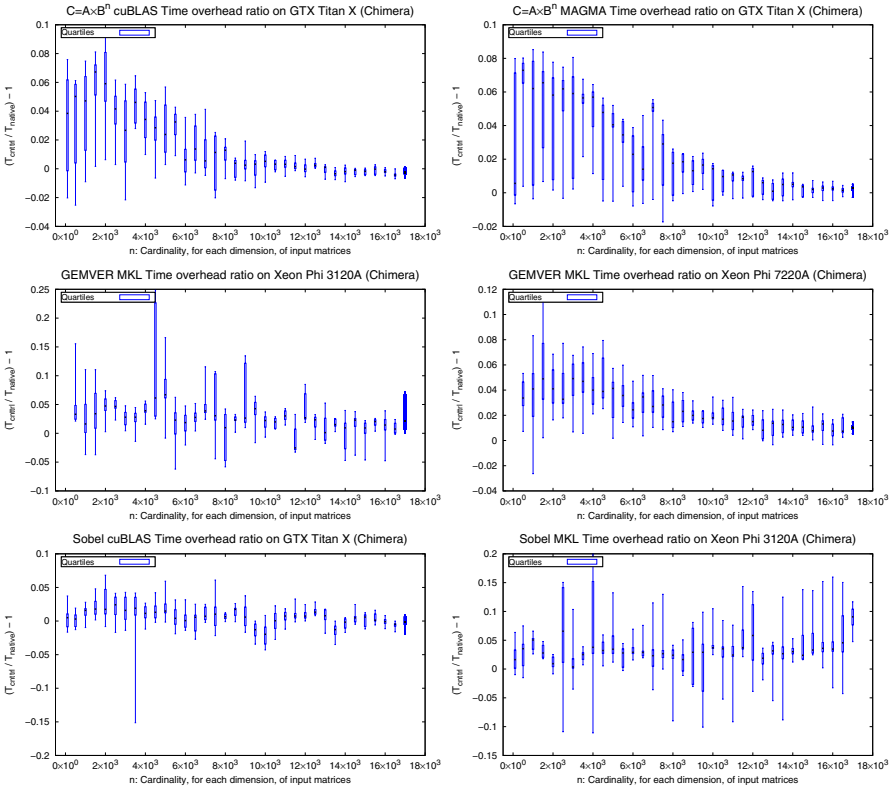

Fig. 7 Controller overhead: Ratio of execution time between the Controller and reference versions for some representative combinations of application and lower-level libraries

The figure presents box plots, showing for each input size the minimum, maximum, and median overheads, as well as the quartiles distribution. The two plots in the first row of Fig. 7 present the ratios for the matrix multiplication sequence problem in the Titan X GPU using cuBLAS and MAGMA. The median overhead is less than 8% in all cases, and for big input-data sizes it is lower than 1%. There is much more dispersion for small data sizes, where the execution times are really low. In these cases, the stochastic behavior of data transfers, accumulated with the overheads introduced by the queue management, parameter conversion, and launch procedures on the Controller model, are more noticeable. The two plots in the second row of Fig. 7 present the ratios for the GEMVER problem in the KNC and KNL Xeon Phi architectures, using MKL. The median of the overhead is between 0.01 and 7%. The GEMVER program has very low total execution times. They are dominated by the device and data transfer management, which are more unpredictable when the host program uses several OpenMP threads as in the Controller mode. Thus, the overheads in the KNC architecture have a high stochastic behavior, even for big input-data sizes. In the more modern KNL architecture, data transfer times are more consistent, leading to lower overheads and

less stochastic behavior for big input-data sizes (see right plot in the second row of the figure). The two plots in the third row of Fig. 7 present the ratios for the Sobel program, comparing the ratio results in the Titan X GPU using cuBLAS (left plot in the third row of the figure), with the ratio results in the KNC Xeon Phi using MKL (right plot in the third row of the figure). This problem presents a better balance between computational load and device and data transfer management costs across the range of input-data sizes. Thus, the median overhead is small in both plots (less than 1% for any input-data size). The Xeon Phi results present more dispersion than those of the Titan X, due to more unpredictable data transfer times and due to higher stochastic effects introduced by the interaction of the internal Controller thread with the Xeon Phi run-time. It is noticeable that, in the case of the Titan X GPU, some median values of the overhead are even less than 0%. The variability of the execution times introduced by the stochastic effects in the data transfers is sometimes higher than the very small overhead introduced by the Controller abstractions and implementation.

Taking into account the full collection of experiments, and after eliminating outliers, we can remark the following observations. The maximum overheads observed for the Xeon Phi architectures are found for the KNC platform and small input-data sizes in both the matrix multiplication sequence and the GEMVER problems. The values of the overhead are up to 14–15%, except for a specific size in the GEMVER problem where the overhead is up to 25%. Nevertheless, the median is less than 7% for all the Xeon Phi experiments, with most median values between 3 and 4%. In the case of the GPUs, the maximum overheads observed are also found for small input-data sizes in the matrix multiplication sequence problem, with values of up to 6–7%. The median is less than 5% for all the GPU experiments, with most median values between 1 and 2%.

6 Conclusion

In this work, we present a solution to build a BLAS library that is portable across different heterogeneous devices and hardware accelerators. The solution is built on top of the Controller model, a programming system for heterogeneous devices that transparently manages data transfers and kernel launching on different types of devices. We propose and introduce in the Controller model: (1) an abstraction layer to hide programming differences between diverse BLAS libraries; (2) a new system to define kernel classes to support the context and device manipulation of different external BLAS libraries; (3) a new kernel selection policy that considers both programmer kernels and different external libraries to select the most appropriate kernel for each type of device; and (4) a library of Controller kernels, named HitCntrlBLAS, that implements a unified interface for the whole set of BLAS routines, with the same abstraction used for kernels directly developed by the programmer. This proposal implements a multidevice and multilibrary backend. The programmer develops a single code that can be executed on different devices, exploiting different specifically tuned third-party libraries, by changing a single initialization parameter of the Controller object.

We present a case study showing that this solution allows the development of portable codes, mixing user-defined kernels with calls to BLAS functions. The results of the experimental study show that our abstractions lead to very small overheads that decrease for significant computation loads.

Future work includes the implementation of more flexible selection policies, the use of the proposed system to develop more complex real applications, and studying the use of the library in programs that exploit multiple devices at the same time.

Acknowledgements This research was supported by an FPI Grant (Formación de Personal Investigador) from the Spanish Ministry of Science and Innovation (MCINN) to E.R-G. It has been partially funded by the Spanish Ministerio de Economía, Industria y Competitividad and by the ERDF program of the European Union: PCAS Project (TIN2017-88614-R), CAPAP-H6 (TIN2016-81840-REDT), and Junta de Castilla y Leon—FEDER Grant VA082P17 (PROPHET Project). We used the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF). CETA-CIEMAT belongs to CIEMAT and the Government of Spain. Part of this work has been performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in particular, the author gratefully acknowledges the support of Dr. Christophe Dubach, the School of Informatics of the University of Edinburgh, and the computer resources and technical support provided by EPCC. The authors want to thank Dr. Ingo Wald for giving us the possibility of getting a KNL coprocessor.

References

1. Aliaga JI, Reyes R, Goli M (2017) SYCL-BLAS: leveraging expression trees for linear algebra. In: Proceedings of the 5th international workshop on OpenCL, IWOCL 2017. ACM, pp 32:1–32:5. https://doi.org/10.1145/3078155.3078189
2. Anderson E, Bai Z, Bischof C, Blackford L, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK users' guide, 3 edn. Software, Environments and Tools. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719604
3. Arm Ltd: Arm Performance Libraries. http://url.ie/13z15
4. Banaś K, Krużel F (2014) OpenCL performance portability for Xeon Phi coprocessor and NVIDIA GPUs: a case study of finite element numerical integration. In: Lopes L, Zilinskas J, Costan A, Cascella RG, Kecskemeti G, Jeannot E, Cannataro M, Ricci L, Benkner S, Petit S, Scarano V, Gracia J, Hunold S, Scott SL, Lankes S, Lengauer C, Carretero J, Breitbart J, Alexander M (eds.) Euro-Par 2014: parallel processing workshops, Lecture notes in computer science. Springer, Berlin, pp 158–169
5. Barker J, Bowden J (2013) Manycore parallelism through OpenMP. In: OpenMP in the era of low power devices and accelerators, Lecture notes in computer science, vol 8122. Springer, Berlin, pp 45–57. https://doi.org/10.1007/978-3-642-40698-0_4
6. Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2002) An updated set of basic linear algebra subprograms (BLAS). ACM Trans Math Softw 28(2):135–151. https://doi.org/10.1145/567806.567807
7. Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, Whaley RC (1995) A proposal for a set of parallel basic linear algebra subprograms. In: Applied parallel computing computations in physics, chemistry and engineering science, Lecture notes in computer science, vol 1041. Springer, Berlin, pp 107–114. https://doi.org/10.1007/3-540-60902-4_13
8. Dong T, Knox K, Chapman A, Tanner D, Liu J, Hao H (2017) rocBLAS: next generation BLAS implementation for ROCm platform. https://github.com/ROCmSoftwarePlatform/rocBLAS. Original-date: 2015-10-08T18:48:02Z
9. Dongarra J (2002) Preface: basic linear algebra subprograms technical (blast) forum standard. Int J High Perform Comput Appl 16(2):115–115. https://doi.org/10.1177/10943420020160020101

10. Dongarra J, Gates M, Haidar A, Jia Y, Kabir K, Luszczek P, Tomov S (2015) HPC programming on intel many-integrated-core hardware with MAGMA port to Xeon Phi. Sci Program 2015:e502593. https://doi.org/10.1155/2015/502593
11. Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Tomov S, Yamazaki I (2014) Accelerating numerical dense linear algebra calculations with GPUs. In: Numerical computations with GPUs. Springer, Cham, pp 3–28. https://doi.org/10.1007/978-3-319-06548-9_1
12. Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra J (2012) From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing 38(8):391–407. https://doi.org/10.1016/j.parco.2011.10.002. http://www.sciencedirect.com/science/article/pii/S0167819111001335
13. Garigipati P, Brehler M (2017) Unified backend. https://goo.gl/fJ3JCj
14. Gates M (2012) MAGMA Forum: sgemm confusion. https://goo.gl/hKKSfL
15. Gates M (2016) MAGMA forum: performance issue. http://goo.gl/HdXapr
16. Gonzalez RC, Woods RE (2007) Digital image processing, 3rd edn. Pearson, Upper Saddle River
17. Gonzalez-Escribano A, Torres Y, Fresno J, Llanos D (2014) An extensible system for multilevel automatic data partition and mapping. IEEE Trans Parallel Distrib Syst 25(5):1145–1154. https://doi.org/10.1109/TPDS.2013.83
18. Horn RA, Johnson CR (1991) The hadamard product. Topics in matrix analysis. Cambridge University Press, Cambridge, pp 298–381. https://doi.org/10.1017/CBO97805118403710.006
19. Howell, G W, Demmel J W, Fulton C T, Hammarling S, Marmol K (2008) Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Trans Math Softw. **34**(3), 1–33 . https://doi.org/10.1145/1356052.1356055. http://portal.acm.org/citation.cfm?doid=1356052.1356055
20. Intel Corporation (2011) Using intel®MKL automatic offload on intel®Xeon Phi™ coprocessors. https://goo.gl/1kq8GB
21. Intel Corporation (2013). Intel®MKL automatic offload enabled functions for Intel Xeon Phi coprocessorshttps://goo.gl/9jV7PY
22. Intel Corporation (2017) Intel®Math kernel library (Intel® MKL). https://goo.gl/6tuzEi
23. Khaleghzadeh H, Zhong Z, Reddy R, Lastovetsky A (2018) Out-of-core implementation for accelerator kernels on heterogeneous clouds. J Supercomput 74(2):551–568. https://doi.org/10.1007/s11227-017-2141-4
24. Knox K, Liu J, Tanner D, Yalamanchili P, Kellner C, Perkins H, Dong T, Lehmann G, Nugteren C, Coquelle B (2017) clBLAS: a software library containing BLAS functions written in OpenCL. https://github.com/clMathLibraries/clBLAS. Original-date: 2013-08-13T15:05:53Z
25. Kovalev M, Kroeker M, Köhler M, Aoshima T (2017) Hadamard product?. Issue #1083 .xianyi/OpenBLAS (2017). https://goo.gl/veigLc
26. Kurzak J, Bader DA, Dongarra J (2010) Scientific computing with multicore and accelerators, 1st edn. CRC Press, Boca Raton
27. Lastovetsky A, Reddy R (2006) HeteroMPI: towards a message-passing library for heterogeneous networks of computers. J Parallel Distrib Comput 66(2):197–220. https://doi.org/10.1016/j.jpdc.2005.08.002. http://www.sciencedirect.com/science/article/pii/S0743731505002042
28. Lim R, Lee Y, Kim R, Choi J, Lee M (2018) Auto-tuning GEMM kernels on the Intel KNL and Intel Skylake-SP processors. J. Supercomput. https://doi.org/10.1007/s11227-018-2702-1
29. Ling P (1993) A set of high-performance level 3 BLAS structured and tuned for the IBM 3090 VF and implemented in Fortran 77. J Supercomput 7(3):323–355. https://doi.org/10.1007/BF01206242
30. Malcolm J, Yalamanchili P, McClanahan C, Venugopalakrishnan V, Patel K, Melonakos J (2012) ArrayFire: a GPU acceleration platform. In: Modeling and simulation for defense systems and applications VII, vol 8403. International Society for Optics and Photonics, p 84030A. https://doi.org/10.1117/12.921122. https://www.spiedigitallibrary.org/conference-proceedings-of-spie/8403/84030A/ArrayFire-a-GPU-acceleration-platform/10.1117/12.921122.short
31. Manumachu RR, Lastovetsky A, Alonso P (2008) Heterogeneous PBLAS: optimization of PBLAS for heterogeneous computational clusters. In: 2008 International symposium on parallel and distributed computing. IEEE Computer Society, Krakow, pp 73–80. https://doi.org/10.1109/ISPDC.2008.9. http://ieeexplore.ieee.org/document/4724232/
32. Moreton-Fernandez A, Gonzalez-Escribano A, Llanos DR (2017) Multi-device controllers: a library to simplify parallel heterogeneous programming. Int J Parallel Program. https://doi.org/10.1007/s10766-017-0542-x
33. Moreton-Fernandez A, Rodriguez-Gutiez E, Gonzalez-Escribano A, Llanos DR (2017) Supporting the Xeon Phi coprocessor in a heterogeneous programming model. In: Euro-Par 2017: parallel

processing, Lecture notes in computer science, vol 10417. Springer, Cham, pp 457–469. https://doi.org/10.1007/978-3-319-64203-1_33
34. Moreton-Fernandez A, Ortega-Arranz H, Gonzalez-Escribano A (2017) Controllers: an abstraction to ease the use of hardware accelerators. Int J High Perform Comput Appl, p 109434201770296 . https://doi.org/10.1177/1094342017702962
35. Newburn CJ, Dmitriev S, Narayanaswamy R, Wiegert J, Murty R, Chinchilla F, Deodhar R, McGuire R (2013) Offload compiler runtime for the intel ®xeon phi coprocessor. In: 2013 IEEE international symposium on parallel distributed processing, workshops and Phd forum, pp 1213–1225. https://doi.org/10.1109/IPDPSW.2013.251
36. NVIDIA Corporation (2017) cuBLAS library: user guide. https://goo.gl/Ryg2gp
37. NVIDIA Corporation (2017) NVBLAS. https://goo.gl/GHdLhm
38. Perrot G, Domas S, Couturier R (2016) An optimized GPU-based 2D convolution implementation. Concurr Comput: Pract Exp 28(16):4291–4304. https://doi.org/10.1002/cpe.3752
39. Podlozhnyuk V (2007) Image convolution with CUDA. Technical report, NVIDIA Corporation. http://goo.gl/n5oa5p
40. Pouchet LN PolyBench/C (2015) The polyhedral benchmark suite. https://goo.gl/NhNR6n
41. Rasch A, Bigge J, Wrodarczyk M, Schulze R, Gorlatch S (2019) dOCAL: high-level distributed programming with OpenCL and CUDA. J Supercomput. https://doi.org/10.1007/s11227-019-02829-2
42. Rousseaux S, Hubaux D, Guisset P, Legat JD (2007) A high performance FPGA-based accelerator for BLAS library implementation. In: Proceedings of reconfigurable systems summer institute (RSSI'07). Urbana. http://rssi.ncsa.illinois.edu/2007/proceedings/papers/rssi07_02_paper.pdf
43. Sanderson C, Curtin R (2016) Armadillo: a template-based C++ library for linear algebra. J Open Source Softw 1: 26. https://doi.org/10.21105/joss.00026. http://joss.theoj.org/papers/10.21105/joss.00026
44. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36(5):232–240. https://doi.org/10.1016/j.parco.2009.12.005. http://www.sciencedirect.com/science/article/pii/S0167819109001276
45. Viviani P, Aldinucci M, Torquati M, d'lppolito R (2017) Multiple back-end support for the armadillo linear algebra interface. In: Proceedings of the symposium on applied computing, SAC '17. ACM, pp 1566–1573. https://doi.org/10.1145/3019612.3019743
46. Wang E, Zhang Q, Shen B, Zhang G, Lu X, Wu Q, Wang Y (2014) High-performance computing on the Intel®Xeon Phi ™: how to fully exploit mic architectures. Springer, Berlin. https://www.springer.com/gp/book/9783319064857
47. Wende F, Klemm M, Steinke T, Reinefeld A (2015) Concurrent kernel offloading. In: High performance parallelism pearls, vol 1, 1 edn. Elsevier, pp 201–223. https://doi.org/10.1016/B978-0-12-802118-7.00012-1. https://linkinghub.elsevier.com/retrieve/pii/B9780128021187000121
48. Yalamanchili P, Arshad U, Mohammed Z, Garigipati P, Entschev P, Kloppenborg B, Malcolm J, Melonakos J (2015) ArrayFire—a high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire

Affiliations

Eduardo Rodriguez-Gutiez[1] · Ana Moreton-Fernandez[1] · Arturo Gonzalez-Escribano[1] · Diego R. Llanos[1]

✉ Arturo Gonzalez-Escribano
arturo@infor.uva.es

Eduardo Rodriguez-Gutiez
eduardo@infor.uva.es

Ana Moreton-Fernandez
ana@infor.uva.es

Diego R. Llanos
diego@infor.uva.es

[1] Dpto. de Informática, Universidad de Valladolid, Paseo de Belén, 15, 47011 Valladolid, Spain