Int J Parallel Prog (2019) 47:94–113
https://doi.org/10.1007/s10766-017-0542-x

Multi-device Controllers: A Library to Simplify Parallel Heterogeneous Programming

Ana Moreton-Fernandez[1] · **Arturo Gonzalez-Escribano**[1] · **Diego R. Llanos**[1]

Received: 19 May 2017 / Accepted: 2 December 2017 / Published online: 9 December 2017
© Springer Science+Business Media, LLC, part of Springer Nature 2017

Abstract Current HPC clusters are composed by several machines with different computation capabilities and different kinds and families of accelerators. Programming efficiently for these heterogeneous systems has become an important challenge. There are many proposals to simplify the programming and management of accelerator devices, and the hybrid programming, mixing accelerators and CPU cores. However, in many cases, portability compromises the efficiency on different devices, and there are details concerning the coordination of different types of devices that should still be tackled by the programmer. In this work, we introduce the *Multi-Controller*, an abstract entity implemented in a library that coordinates the management of heterogeneous devices, including accelerators with different capabilities and sets of CPU-cores. Our proposal improves state-of-the-art solutions, simplifying data partition, mapping and the transparent deployment of both, simple generic kernels portable across different device types, and specialized implementations defined and optimized using specific native or vendor programming models (such as CUDA for NVIDIA's GPUs, or OpenMP for CPU-cores). The run-time system automatically selects and deploys the most appropriate implementation of each kernel for each device, managing data movements and hiding the launch details. The results of an experimental study with five study cases indicates that our abstraction allows the development of flexible and highly efficient programs that adapt to the heterogeneous environment.

✉ Ana Moreton-Fernandez
ana@infor.uva.es

Arturo Gonzalez-Escribano
arturo@infor.uva.es

Diego R. Llanos
diego@infor.uva.es

[1] Departamento de Informática, Universidad de Valladolid, Valladolid, Spain

Keywords Accelerators · Hybrid computation · GPUs · Kernel characterization · Memory transfers

1 Introduction

Current HPC clusters are composed by several machines with different computation capabilities and accelerators, such as Graphics Processing Units (GPUs) or XeonPhi coprocessors [20]. It has been proved that the use of these heterogeneous systems improves performance in many applications [18]. However, programming efficiently for these heterogeneous systems has become an important challenge. The different computational units (CPUs, GPUs, XeonPhi, ..) that can form a cluster typically have different programming requirements and constraints to achieve the best performance. Thus, different programming models are used for each kind of device. For example, the CUDA programming model achieves the best performance in NVIDIA GPUs [8], and OpenMP has been shown to be an efficient programming model for multi-cores in shared-memory systems or XeonPhi coprocessors.

There are many proposals to simplify the programming and management of accelerator devices, and the hybrid programming mixing accelerators and CPU cores. However, in many cases, portability compromises the efficiency on different devices. Depending on the proposal, some details concerning the coordination of different types of devices are still tackled by the programmer, such as computation partition and balance, data mapping and locality, or data movement coordination across different memory hierarchies.

In this work, we introduce the *Multi-Controller (MCtrl)*, an abstract entity implemented in a library, that coordinates the management of heterogeneous devices, including accelerators with different capabilities and sets of CPU-cores. It helps the programmer to handle the computation partition, mapping, and transparent execution of complex tasks in such hybrid and heterogeneous environments, independently of the target devices selected at run-time. Our proposal allows the exploitation of simple generic kernels that can fit in any device; very specialized kernels, defined and optimized by the programmer for each architecture; and even wrappers to call third-party predefined libraries (such as e.g. cuBLAS [14]). This allows the exploitation of native or vendor specific programming models, in a highly efficient way. The most appropriate kernels for each target device are automatically selected by the entity during program execution.

Our work is developed on the concept of the *Controller* presented in [1,12]. While the Controller transparently manages data movements and the launching of a series of kernels on a given target device, the Multi-Controller coordinates several Controllers associated to different devices or groups of CPU-cores. It is implemented as an extensible library, and it can use the best programming models, tools and compilers for each potential device.

We present an experimental study with five case studies. We show that our approach is highly flexible, with minimum programming effort for changing the target devices. The results of a performance study, comparing our approach with optimized reference

codes, show that our implementation does not introduce significant performance penalties.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 introduces the libraries used to build our proposal. Section 4 explains the proposed model. Section 5 shows the experimental study, and finally, Sect. 6 sets out the conclusions and future work.

2 Related Work

In this section, we analyse different works proposed in the literature that target the simplification of parallel programming on heterogeneous systems, composed by different computational devices, including accelerators. These proposals avoid the need for a manual combination of the specific programming models for each computation device. They introduce unified programming models or tool abstractions to manage the architectural differences between computational units of different natures or capabilities.

A widespread programming framework to deal with heterogeneous devices is OpenCL [19]. The OpenCL context abstraction allows the management of multiple devices of the same nature (using the same *platform* in OpenCL notation). However, the coordination of devices of different natures, and the management of data sharing or partitioning, computation mapping, load balancing, and communication across them are tricky and should be manually solved and coded by the programmer. Moreover, the abstractions introduced by OpenCL derive, in many cases, in not reaching the best possible performance (see e.g. [8]). Many libraries of higher levels of abstraction, that rely on OpenCL as the execution layer, typically inherit some of these problems (see e.g. [19]). An interesting example of abstraction built on top of OpenCL is the Maat library [16]. It provides a unified context with an abstract view regardless of the number and nature of the devices, for GPU and CPU platforms. The main differences with our proposal are related to our choice of internally using native or vendor low-level programming models. We propose the possibility of declaring both unified and specialized/optimized kernels for different architectures, selected at run-time for each particular device. Our proposal allows the exploitation of features of the native programming models, or specialized third-party libraries optimized for the device. We introduce more flexibility to select the desired devices associated to a Multicontroller, applying such techniques as grouping CPU-cores as a single device, in order to exploit native multi-threaded parallelism inside a kernel. The performance results presented for the Maat library are only compared with other OpenCL implementations. The performance effects compared with the use of more specific programming models, such as combinations of CUDA for GPUs, and OpenMP for multicore CPUs, have not yet been stated.

More general approaches propose complete integrated frameworks that exploit lower-level specific programming models. Some examples include OMPICUDA [9], Cashemere [6], StarPU [7], or the skeleton programming framework based on it, SkePU [2]. PACXX [4] is a transformation system integrated into the LLVM compiler framework. It generates code for different kinds of devices, and it transforms

explicit parallel constructions that use the concept of kernel and launch in an abstract and elegant form. Scogland et al. [17] also propose a mixed approach by adapting OpenMP pragmas to heterogeneous systems. In general, all these approaches hide the coordination details to the programmer, to the point of constraining the potential optimizations that could be achieved manually. Techniques to select launch parameters, such as the threadblock size, are for example tackled in SkePU using trial-and-error, with no possibility of extrapolating the results to other kernel codes or architectures.

Unlike previous approaches, our library proposes a flexible tool that allows programming with generic and portable kernels and, at the same time, the integration of higher levels of optimization using the native or vendor provided programming models, tools or libraries, for higher efficiency and performance. The flexibility in terms of programmer control of the device selection and coordination at run-time also improves previous techniques.

3 Background

The Multi-Controller library we present in this paper is built upon two previous tools. The first, named Hitmap [3], is a library to manage the partition and mapping of data structures. It is used in our model to manage data distribution across devices and to provide a common interface to implement data management inside generic portable kernels. The second, named Controller [1,12], is a library that defines an abstract entity to transparently manage data movements and kernel launching for a single device. Our proposal combines them with a new layer of abstraction to coordinate the use of several devices of different architectures or natures. This section introduces the reader to both previous tools, before describing the new abstraction proposed.

3.1 Hitmap Library

Hitmap is a library designed for hierarchical tiling and mapping of dense and sparse arrays, or graphs. Hitmap is based on a distributed SPMD programming model, using abstractions to declare data structures with a global view. It automates the partition, mapping and communication of hierarchies of tiles, while still delivering good performance [3].

An object *HitShape* represents a subspace of domain indexes. For dense arrays, it is defined as an n-dimensional rectangular parallelotope. The limits on each dimension are represented with a *HitSig* object, containing the range limits. Each *HitSig* object is a tuple of three integer numbers $S = (b, e, s)$ (begin, end and stride), representing the indexes in one of the axes of the domain. Begin and stride members of a Signature represent the coefficients of a linear function $f_S(x) = sx + b$.

Hitmap defines the *HitTile* structure, an abstract entity for n-dimensional arrays and tiles. A *HitTile* structure is a handler to store array meta-data, along with the pointer to the actual memory space of the data. A *HitTile* maps actual data elements to the index subspace defined by a shape. There are only four functions of Hitmap needed to work with our Multi-Controller library. First, *hit_tileDomain* and *hit_tileDomainAlloc* are used to declare the index domains of a tile array. The second one also allocates the

memory for the data. The function *hit_tileFree* is used to free the data memory and clean the handler. The function *hit_tileElem* is used in host or kernels code to access the elements of a tile. It receives a tile name, a number of dimensions and the index values of the desired element. The data are accessed in row major order in all cases, independently of the implementation.

The Hitmap library includes many other functionalities. For example, the management of hierarchical subselections of parts of the tiles and the transparent management of distributed-arrays, with abstract partition and communication functionalities that internally use a message-passing paradigm (exploiting MPI). Hitmap has been extended with a much smaller handler (*HitKTile*), with the minimum information needed for data access to multidimensional arrays through a data pointer, which is useful for transparently transferring the data structures to accelerators in a more efficient form.

3.2 Controllers Library

The second abstraction upon which we have built our proposal is implemented in the Controllers library. It introduces an abstract entity that allows the transparent launching of series of tasks on a single accelerator device, also considering a group of CPU cores as a many-core device. The Controller model presents several important features: (1) A mechanism to define common kernels reusable across different types of devices, or specialized kernels for specific device types; (2) A transparent mechanism of memory management, including optimized communications of the data structures between the host and the corresponding images in the accelerators; (3) An optimization system to select proper values for kernel-launching configuration parameters (such as the threadblock geometry), guided by simple qualitative code characterization provided by the programmer.

In our proposal, we use the Controller entity to manage each device inside the new Multi-Controller that coordinates them. In the Controller library, a kernel is declared by using the primitive `KERNEL_<type>`. Where `type` may be empty to indicate a kernel usable on any kind of device, or a specific value for a specialized code for a given type of device. This is useful when different optimizations on the kernel code are required for different devices. Currently, the library supports the specific declarations `KERNEL_GPU` for CUDA code targeting NVIDIA's GPUs, `KERNEL_CPU` for host machine code targeting sets of CPU cores, and `KERNEL_GPU_WRAPPER`, `KERNEL_CPU_WRAPPER`, for host machine code which includes calls to specialized GPU or CPU libraries, such as cuBLAS or MKL routines. Moreover, the work presented in [11] extends the Controller library to support the Xeon Phi co-processor.

The kernel-definition primitives declare, in brackets, the number of parameters of the kernel, with a tuple of information for each parameter. The parameter information includes its type, name and input/output role:

- IN: for input HitTile parameters, whose elements are only read.
- OUT: for output HitTile parameters, whose elements are only written.
- IO: for input and output HitTile parameters, whose elements can be both read and written.

```
1  /* Kernel characterizations */
2  KERNEL_CHAR(MatAdd,1,full,low,low)
3
4  /* Generic kernel codes for any device */
5  KERNEL(MatAdd, 3, OUT, HitTile_float, C,  IN,  HitTile_float, A,
6                    IN,  HitTile_float, B ) {
7       int x = thread.x;
8       int y = thread.y;
9       hit_tileElem( C, 2, x, y ) =
10          hit_tileElem( A, 2, x, y ) + hit_tileElem( B, 2, x, y );
11 }
```

Fig. 1 Kernel definition and configuration of a matrix addition using the Controller library

– INVAL: for input parameters of any type passed by value.

The programmer can also provide a kernel characterization, in terms of code features, that helps to automatically determine proper kernel launching parameters. In the current prototype, the CPU thread granularity is determined by a simple regular blocking policy, which does not require a specific kernel characterization. For GPU kernels, the library integrates the model presented in [15,21]. This model allows the determination of configuration parameters (grid, threadblock and L1 cache memory sizes) for NVIDIA's GPUs. The primitive `KERNEL_CHAR` receives the kernel name, the number of dimensions of the thread space (1, 2, or 3) and descriptive values for the characterization model. These values are a qualitative description of the characteristics of the kernel code provided by the programmer. They are related to: (a) the coalescing property of the global memory access patterns (full, medium, scatter); (b) the ratio of arithmetic/logic operations per global memory access (high, medium, low); and (c) the ratio of data sharing accesses in a block per global memory access (high, medium, low).

Figure 1 shows an example of the kernel for a matrix addition computation. We see a kernel characterization in line 2 and a kernel definition in lines 5 and 6.

4 Multiple-device Controller (MCtrl) Library

The Multiple-Device Controller (MCtrl) library provides a simplified way to program applications, targeting heterogeneous systems with different kinds of computational units. In this paper, we define the computational unit/target device as an accelerator (GPU, Xeon Phi, etc.) or a group of CPU-cores considered as a single independent device. The goal of this library is: (1) to automatize the data partition and data transfer between the host and multiple target devices, as well as (2) to transparently coordinate the division and execution of the computation among different computational units, independently of the kind of target device exploited (GPUs, group of CPU-cores, etc).

The library has an object-oriented design, despite the fact that it is mainly developed in C language. The classes are implemented as C structures with associated functions. The Multi-Controller model architecture is presented in Fig. 2.

The Multi-Controller object provides functions to manage:

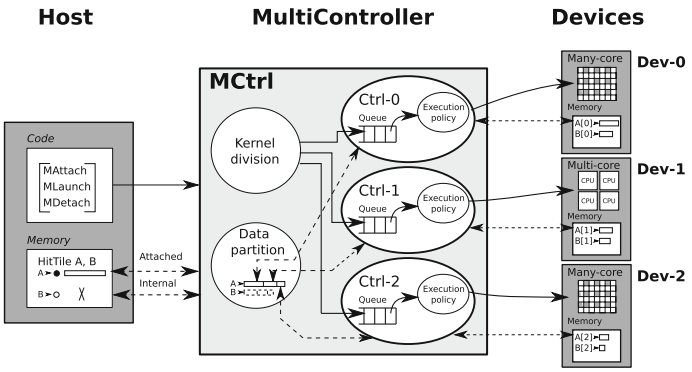

Fig. 2 Diagram of the Multiple-Device Controller library (MCtrl)

– *Multi-device coordination* The Multi-Controller is associated with a set of different devices at construction. It internally creates Controller objects to interact with each device. The Multi-Controller provides an abstract interface that enables it to be managed as a single computational device, independently of the internally associated devices.
– *Data structures* The Multi-Controller abstraction creates a unified memory context for all of the associated devices, where internal data structures can be created, or data structures from the main host thread can be attached. Data structures can be replicated or partitioned and distributed across the devices as the programmer requires. In the current prototype, a simple static partitioning method has been included. It divides the structures into as many irregular parts as the number of devices selected in the Multi-Controller construction. The size of each part is calculated proportionally from a list of weights. Data movements across different device memory hierarchies are transparently managed by the internal Controller objects associated to each device.
– *Kernel definition and launching* The Multi-Controller model integrates the Controller's idea of multi-version kernel definition. Thus, kernel launching in a Multi-Controller simply uses a kernel name. The internal Controller selects, at run-time, the most appropriate kernel version or implementation for the associated device, from those provided by the programmer. The Multi-Controller internally divides the computation associated to a kernel launching among the different devices. The kernel execution on a device is performed asynchronously with respect to the execution of the kernels in the rest of the devices. Synchronizations are required only by data requests on the main host thread.

4.1 Multi-controller Construction

A Multi-Controller object is constructed to manage a specific collection of devices. Its construction functionality receives an ordered list of device specifications. Each device specification is used to internally create a Controller object associated to the computational resource. In the current prototype we support device specifications that

include: (1) NVIDIA's GPUs, specifying their CUDA device number, and (2) Groups of main host CPU-cores, specifying a range of core identifiers, according to their internal numbering in the CPU information provided by the operating system.

The Multi-Controller internally creates a queue to temporarily store the requests for kernel launchings, before dividing the computation and mapping it to the queues of the internal device Controllers. The synchronization and coordination operations of each Controller are executed on its own task, which makes the use of a host thread asynchronous only when activity is needed, for minimal interference with other host threads. In the current prototype, the internal device Controllers are implemented using OpenMP tasks.

4.2 Data Structures and Domains

One of the objectives of the Multi-Controller library is to provide a homogeneous interface to work with data structures in different device types, preserving the coalescing or vectorization properties of the code due to the data accesses order. The previous Controller library uses Hitmap to provide such an interface.

For Multi-Controllers, we propose to exploit and extent Hitmap functionalities to provide a transparent abstraction for the partition, subselection, and mapping of parts of the data structures to the different devices associated to a Multi-Controller. The Multi-Controller model proposes a single memory context for the whole set of associated heterogeneous devices. Data structures from the main host thread can be attached to the Multi-Controller context, and they should not be manipulated on the main host thread until they are detached from the Multi-Controller. The Multi-Controller can decide when the real data movements should be done, synchronously or asynchronously, to the actual devices, depending on the kernels enqueued for execution and their data dependencies.

To avoid redundant data movements across memory hierarchies, the model provides the programmer with a flexible attachment functionality. The current proposal is focused on applications where: (1) No data transfers between devices are needed across several kernel executions; and (2) any part of the computation needs, either a whole data structure or a subset of the data structure that does not overlap with other subparts. Thus, data structures should be assigned as a whole to all the devices, or partitioned in non-overlapping parts, assigning one for each device.

To support this model, we have extended the HitTile objects in the Hitmap library with the capability of dividing itself into several sub-selections and storing the information about the partition inside the object. Internally, when a HitTile is attached to a Multi-Controller, it performs the following steps:

1. First, it checks that the HitTile is not already attached to any Multi-Controller. If that particular HitTile object is already attached, the program raises an error, as a second attachment could lead to race conditions due to the concurrent execution of kernels in different Multi-Controllers.
2. If it is an attachment without the partition option, the whole space of indexes of the data structure are mapped to each device. If it is an attachment with the partition option activated, the Multi-Controller divides the index space of the HitTile data

structure into a number of parts equal to the number of devices defined in the Multi-Controller. The partition policies introduced in Hitmap are responsible for dividing the data structure with no-overlapped domains. The partition size corresponding to each device is proportional to the weights provided in an array of floating point numbers, one for each device. More sophisticated partition policies can be easily added in the future, thanks to the modular plug-in system in the Hitmap library. The information about the mapping is stored in the HitTile object for further reference.

3. Finally, the Multi-Controller creates a HitTile structure for the space or sub-space of indexes mapped for each device, and proceeds to carry out the data transfers to the assigned device when needed. Transfers are not needed for groups of CPU-cores of the host, or accelerators that can share the host memory space. The transfer policies inside the Multi-Controller can take decisions about when and how to make the transfers. For example, the current Multi-Controller prototype implements both immediate and lazy transfers. The implementation of asynchronous transfers is currently an on-going work.

When the data structure is detached, the Multi-Controller object ensures the consistency of the whole data structure in the main host thread. This may imply data transfers from some or all of the associated devices. The information stored in the objects about the index space mapped to each device is used for the transfers and eliminated at the end of the detachment procedure. The semantic of this operation makes it synchronous. The main host thread should be blocked until its state is consolidated.

Inherited from the Controller library, the Multi-Controller model also allows the attachment of HitTile structures that have a defined index space, but no memory allocated in the main host thread. This creates partitioned internal memory buffers (replicated or partitioned) inside the space of the devices, which are transparently treated inside the kernel functions as any other data structure. The detachment of these structures simply frees the corresponding subparts and internal resources in the devices.

In the case of partitioned data structures, the actual parameters used in the execution invocations of the Multi-Controller are substituted by the HitTiles created by subselecting the mapped portion of the index spaces for each device. Inside the kernel functions, they are transparently used as normal, whole HitTile data-structures. The kernel launching interface will transparently transform the HitTile handlers for the real parameter handlers to an internal HitKTile type, substituting the data pointer with its equivalent in the device memory space when needed. The data access primitives used inside the kernel code are transparently rewritten to use the pointer contained in these objects, along with a minimum number of arithmetic operations, to access the data. The resulting code exposes the arithmetic expressions to the native compiler in order to open the possibility of further optimizations. The result obtains as good performance as direct array accesses in static codes.

4.3 Kernel Launching

The Multi-Controller model proposes a unified space of indexes for logical threads across the whole set of associated heterogeneous devices. One instance of the kernel

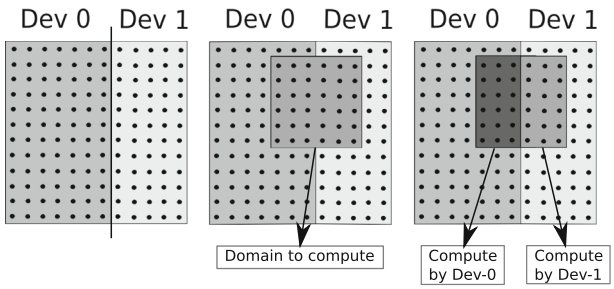

Fig. 3 Calculating the domains to compute for each device

function is executed by each logical thread. This model directly fits with the *thread grid* abstraction in current GPU programming models, such as CUDA or OpenCL. In the case of groups of CPU-cores, the internal Controller objects are responsible for executing the kernel invocations of a grid of many logical threads inside a limited set of coarse threads (e.g. one OpenMP thread per core) for efficiency.

The Multi-Controller kernel launching function receives, as parameters, the name of the kernel, the real parameters for the kernel (whole data structures attached to the Multi-controller, or single typed values), and a definition of the index space for the logical threads.

Internally, the Multi-Controller performs, at runtime, the intersection between the indexes subset defined by grid, or domain of threads, specified by the programmer, and the data structure domains or sub-selections performed by the attachment procedure. The result points to the domain of logical threads where each device should perform the computation. With this method, the computation is transparently divided as a function of the data partitions previously performed. Figure 3 shows a graphical representation of an example of this procedure. Stage 1 of Fig. 3 shows an example of the partition of a data-structure domain. Stage 2 overlaps the domain (grid of threads) where computation is required. Stage 3 shows the domains of logical threads that should be executed on each device.

With the information about how the computation is divided, the Multi-Controller object deploys the kernel launches on the internal Controller objects with a non-empty sub-space of threads mapped. Thus, the global computation launching is subdivided into sub-kernels that are enqueued for execution in their corresponding device queues.

The information provided by the programmer in the characterization primitives of the kernel declarations is internally used to determine the launch parameters for the appropriate devices, such as thread-block geometries.

4.4 Programming Methodology and Example

In this section, we discuss, using an example, how a program is developed using our proposal. The proposed methodology derives in clearly structured programs, using simple development guidelines. Figure 4 shows the typical stages of an application programmed using the Multi-Controller library. After the creation of the Multi-Controller

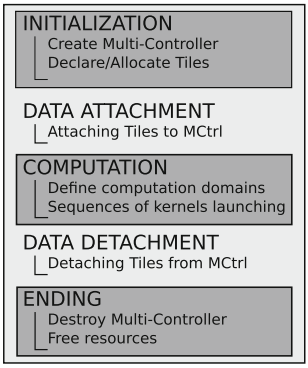

Fig. 4 Typical programming stages using the MCtrl library. The data-structures attachment/detachment, and kernel launching stages, can be repeated and interleaved as desired

object, data structures declared in the main host thread can be attached to the object. Computations are started defining the thread space and invocating kernel launchings in the Multi-Controller object. The detachments consolidate the state of the whole data structures in the main host thread.

Figure 5 presents two codes of a matrix addition programmed using our proposal. On the left of Fig. 5, a group of ten CPU-cores and a GPU are exploited, assigning 10 and 90% of the computation to each, respectively. On the other hand, in the code on the right, two GPUs are exploited for computation. In this case, each GPU performs 50% of the computation. Both codes follow the basic structure of the programming stages presented in Fig. 4.

First, the programmer creates a Multi-Controller object. The creation of this controller includes the definition of the different kinds of devices controlled by this object, as well as a parameter specifying their computation features. The parameter, when the device is a group of CPU-cores, corresponds to the range of CPU-cores used. For a GPU device, this parameter indicates the GPU identifier (see lines 7 to 9).

The second step consists of the data structures creation and allocation. HitTile objects are created and allocated using a HitShape object that represents a domain. To do that, we have applied the function `hit_tileDomainAlloc(...)` (see lines 16 to 24). HitTiles are attached to the Multi-Controller using the `MCtrlAttach(..)` function (see lines 29 to 34). For the attachment, an array of floats is needed, indicating the weights used to divide the data structure among the different target devices. The Multi-Controller is responsible for the actual data attachment on the final device where the computation will be performed.

After that, the computation domain (index domain for logical threads) is defined by a HitShape object (see line 38). Typically, the kernel execution is performed for each element of a matrix, as in all the cases studied in this paper (see Sect. 5.1). In this cases, the computation domain and the data structure domain are equal.

The kernel launching is performed in lines 41–45. The parameters of the `MCtrlLaunch` function are: (a) The Multi-Controller object; (b) the domain that defines the computation space index; (c) the name of the kernel; (d) the number of parameters required by the kernel; and (e) the real parameters for the kernel execution. It deploys the kernel executions on all the computational devices associated to

```
// Multi-Device Controller (MCtrl)
// creation
CALMCtrl cntrlMult;
// Two devices: a 10 CPU-core group,
//              and a GPU
CAL_MCtrlCreate2(cntrlMult,
                 CAL_CNTRL_CPU,
                 RANGE(0,9),
                 CAL_CNTRL_GPU, 0 );

// Specifying the weights
// corresponding to each device
float percents[2] = {10, 90};

// Define whole data structures
HitTile_float A, B, C;
HitShape domain;
domain =hit_shapeStd2(SIZE, SIZE);
hit_tileDomainAlloc(A, 2, float,
                    domain);
hit_tileDomainAlloc(B, 2, float,
                    domain);
hit_tileDomainAlloc(C, 2, float,
                    domain);

// Attach the data structures to
// a MDC, determining the weights
// for each device
CAL_MCtrlAttach(A, cntrlMult,
                percents);
CAL_MCtrlAttach(B, cntrlMult,
                percents);
CAL_MCtrlAttach(C, cntrlMult,
                percents);

// Determine the threads to launch.
HitShape threads;
threads = hit_shapeStd2(SIZE, SIZE);

// Perform the commputation
CAL_MCtrlLaunch(cntrlMult, threads,
                MatAdd, 3,
                hit_CM(&C),
                hit_CM(&A),
                hit_CM(&B) );

// Copy result from MDC memory
// to host memory
CAL_MCtrlDetach(A, cntrlMult);
CAL_MCtrlDetach(B, cntrlMult);
CAL_MCtrlDetach(C, cntrlMult);

// Destroy MCtrl
CAL_MCtrlDestroy2(cntrlMult);

//Free CHitTiles
hit_Free(A);
hit_Free(B);
hit_Free(C);
```

```
// Multi-Device Controller (MCtrl)
// creation
CALMCtrl cntrlMult;
// Two devices: two GPUs

CAL_MCtrlCreate2(cntrlMult,
                 CAL_CNTRL_GPU, 0,

                 CAL_CNTRL_GPU, 1 );

// Specifying the weights
// corresponding to each device
float percents[2] = {50, 50};

// Define whole data structures
HitTile_float A, B, C;
HitShape domain;
domain =hit_shapeStd2(SIZE, SIZE);
hit_tileDomainAlloc(A, 2, float,
                    domain);
hit_tileDomainAlloc(B, 2, float,
                    domain);
hit_tileDomainAlloc(C, 2, float,
                    domain);

// Attach the data structures to
// a MDC, determining the weights
// for each device
CAL_MCtrlAttach(A, cntrlMult,
                percents);
CAL_MCtrlAttach(B, cntrlMult,
                percents);
CAL_MCtrlAttach(C, cntrlMult,
                percents);

// Determine the threads to launch.
HitShape threads;
threads = hit_shapeStd2(SIZE, SIZE);

// Perform the commputation
CAL_MCtrlLaunch(cntrlMult, threads,
                MatAdd, 3,
                hit_CM(&C),
                hit_CM(&A),
                hit_CM(&B) );

// Copy result from MDC memory
// to host memory
CAL_MCtrlDetach(A, cntrlMult);
CAL_MCtrlDetach(B, cntrlMult);
CAL_MCtrlDetach(C, cntrlMult);

// Destroy MCtrl
CAL_MCtrlDestroy2(cntrlMult);

//Free CHitTiles
hit_Free(A);
hit_Free(B);
hit_Free(C);
```

Fig. 5 Matrix addition example programmed using our approach: Exploiting a group of 10 CPU-cores and a GPU for the computation (left); and exploiting two GPUs for the computation (right)

the Multi-Controller. It internally enqueues, in each device Controller, a copy of the kernel launching, adapting the thread index domain with the goal of each target device will perform only its corresponding part of the computation.

Finally, once the computation has finished, HitTiles are detached from the Multi-Controller, the Multi-Controller is destroyed, and the data structures are free (see lines 49–59).

5 Experimental Study

This section presents an experimental study to show how this approach simplifies the programming effort to adapt programs to different sets of heterogeneous devices, and the efficiency obtained by our prototype implementation of the Multi-Controller library. First, we present several benchmarks used in the study, discussing their features. Second, we present some development effort measures. Finally, we provide a comparison of performance measures obtained by programs which use device vendor or native programming models, and programs developed with the Multi-Controller library.

5.1 Study Cases

We have used four common benchmarks as base-lines for five case studies to test our proposal.

5.1.1 Matrix Addition

The Matrix addition consists of the sum of two different matrices, storing the result in a third one: $C = A + B$. The computation of each cell does not imply any kind of dependency with the computation of another one. The solution developed using our proposal involves just one kernel with a bidimensional grid and bidimensional threadblocks. Depending on the size of the grid and the matrices, each block of threads computes the resulting values of several blocks of the matrix iteratively, following the example implementation presented to the CUDA community in the programming guide [13]. The accesses to global memory are fully coalesced. This benchmark requires a large quantity of data transfers to the accelerators used, while the computation load is really low. Using our model, the CPU solution for this problem is similar to the GPU version. Only one generic kernel should be defined by the programmer.

5.1.2 Matrix Multiplication

The Matrix multiplication computes the product of two different square matrices, storing the result in a third one: $C = A * B$. The computation of each cell of the resulting matrix is not dependent on any other computation.

A direct simple solution to this problem involves one generic kernel, using a bidimensional grid of threads, for both CPU and GPU. Each thread $t_{i,j}$ is responsible for computing the dot product operation ($\sum_{k=0}^{n-1} A[i][k] * B[k][j]$), storing the result in the (i, j) position of the C matrix. Nevertheless, different logical threads use elements of A or B that are also read by other logical threads. Thus, data can be reused and

shared across the computation of several cells. Moreover, the read patterns on A and B matrices should be studied and adapted to exploit coalescence in GPUs, and properly exploit caches and vectorization on CPUs. This leads to interesting optimizations in both GPU and CPU devices. Thus, in our model, we can declare different specialized kernels for each kind of device.

The optimized GPU implementation in the CUDA Toolkit Samples exploits the shared-memory for better performance. The threads on each threadblock can use shared-memory to collectively load a square block of A and B matrices in a coalesced way. Then, they can efficiently perform a block matrix multiplication using the elements on the shared-memory. Several iterative stages should be applied to compute all the matrix block multiplications needed at each threads block. Threads need to use block synchronizations on the global memory read operations, using a specific CUDA code. This code, due to the way it uses the shared memory and aligns the read operations, forces the use of a specific square threadblock size (32×32). We have simply modified the CUDA Toolkit Samples code to use the abstract Multi-Controller thread indexes and the HitTile structures in the data accesses.

The current CPU kernel version is the generic, simple implementation of the dot product of a row of A and a column of B to compute the result for a single output element. Further optimizations based on loop reordering and tiling for better cache usage can be automatically applied by the native C compiler.

5.1.3 Black–Sholes

The Black–Scholes formula is based on a mathematical model of a financial market. The result estimates the price of European-style options. The program, obtained from the CUDA Toolkit Samples, independently applies the formula to a chosen number of input values stored in an array, calculating and storing their results. Thus, it is an embarrassing parallel program with perfectly coalesced accesses on a GPU. Each thread does only one read and one write operation to global memory. It applies several floating point operations, calculating intermediate results and storing them in registers or temporal variables. We have explored two case studies using this benchmark: A simple execution of the kernel (BlackScholes) and a program that iteratively launches a sequence of 2048 executions of this kernel for the same array (BlackScholes_2028).

As in the matrix addition benchmark, the data transfers are not negligible compared with the computation time. In our model, the same generic kernel definition is used for both CPU and GPU.

5.1.4 Mandelbrot Algorithm

The *escape time algorithm* is the simplest algorithm for generating a representation of fractal geometric images of the Mandelbrot set. This example presents an irregular workload per thread. Nevertheless, the computation space can be divided between two GPUs, giving the same amount of work to each GPU and thus producing good results. This example will clearly show the balancing effect of partitioning the workload among several GPUs. The application does not need any data transfer to start the computation, only to return the results. The parameters chosen for the experiments are: An image

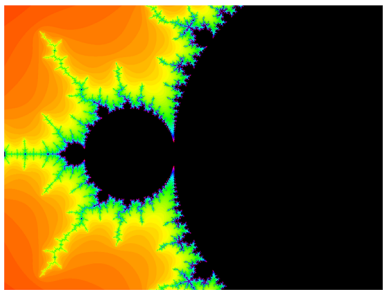

Fig. 6 An image of the Mandelbrot set with the limits x_{min}: −1.4748333, x_{max}: −0.9748333, y_{min}: −0.1791667, y_{max}: 0.1958333

size of 2048 × 2048 and a limit of 60,000 iterations per pixel. A visual representation of the chosen area is shown in Fig. 6. The GPU threadblock size used is 32 × 32 for all the experiments with this benchmark. In our implementation, the same generic kernel definition is used for both GPUs and CPUs.

5.2 Development Effort

In this section we compare, in terms of development effort, the use of our library with the most common native programming models for NVIDIAS's GPUs and multi-core CPUs, which are CUDA and OpenMP respectively. For this comparison, we use three classical development effort metrics: COCOMO lines of code, McCabe's cyclomatic complexity [10], and the Halstead development effort [5]. McCabe's cyclomatic complexity measure is a quantitative measure of the number of linearly independent paths through a program's source code, and the Halstead development effort metric is also a quantitative number based on the number of operators and operands in the source code. Low cyclomatic complexity and Halstead development effort results imply codes simpler to develop and debug. These metrics are typically used in the assessment of software design complexity. The metrics are applied to the parts of code that include: kernel definitions, kernel characterizations, the coordination code in the main host thread with the Multi-Controller management, and data structure management. We ignore code devoted to error or results checking, performance instrumentation and writing messages to the standard output.

Table 1 shows the different measures for the different codes evaluated. The results show that our library implies less development effort for the programmer than using CUDA for all the study cases. On the other hand, although the OpenMP programming model needs a smaller volume of lines of code, the cyclomatic complexity of our proposal is less because our abstraction hides some run-time decisions and checks.

Transforming a CUDA program into an OpenMP version, or the opposite, is not a trivial task. Remember Fig. 5, where we show two codes, using the Multi-Controller abstraction, that perform a matrix addition using different target device combinations. When we compare both codes, we observe that the effort required by the programmer to change the program in order to exploit 1 GPU + 10 CPU-cores, or two GPU devices, only involves 4 lines of code. We can see these four lines highlighted on both codes in the code.

Table 1 Development effort measures for the four benchmarks when they are programmed using Cuda, OpenMP, and the proposed Multi-Controller library

Benchmark	Code	Lines of code	Cyclomatic complexity	Halstead measure
Matrix addition	Cuda	72	7	202,361
	OpenMP	49	11	99,783
	MCtrl	61	5	103,528
Matrix mult.	Cuda	142	5	409,862
	OpenMP	45	9	81,136
	MCtrl	97	6	201,242
Black–Scholes	Cuda	211	7	742,735
	OpenMP	134	8	389,556
	MCtrl	163	6	486,956
Mandel	Cuda	49	3	159,481
	OpenMP	36	5	66,610
	MCtrl	55	3	124,212

This simplification makes all the differences on data transfers, kernel launchings, and data managements for different kind of computational devices such as accelerators or groups of CPU-cores transparent to the programmer.

5.3 Performance Results

In this section, we present performance results: (1) comparing our proposal with pure CUDA reference programs, and (2) evaluating the impact of using different computational units for the five case studies selected in our model. The goal of this study is to determine the potential performance penalty introduced by using our approach, as well as the performance gain obtained when exploiting a combination of heterogeneous devices with different computational capabilities.

The experiments have been executed on a host machine named *Hydra*, with two CPUs Intel Xeon E5-2609 v3 @1.90 GHz, 64 Gb DDR3 main memory, and two GPUs: an NVIDIA's GeForce Titan Z (named GPU-0) and a Titan Black X (named GPU-1). We exploit the two GPU devices, and multiple CPU-cores organized in a single virtual device in our model. For this test, we have decided to avoid performance effects derived from oversuscription or hyperthreading. As our Multi-Controller library uses one host thread for each device to be controlled, the number of CPU-cores we use to compute and execute kernels is 10.

The programs have been compiled using the CUDA Toolkit 8.0, and GCC 4.8.3. We have used the flags *-O3* and *-fopenmp* to exploit parallelism when using a group of CPU-cores as a computational unit. We have executed all the experiments ten times, registering the lowest total execution times. We have also measured separately the times spent in copying data to and from the target devices, and the computation time of the kernels. We include the time spent by our queue system inside the computation times.

We have tested three kinds of code:

- A native CUDA implementation of the different benchmarks tested. We present measures obtained in one or both GPUs in our target system, depending on the study. See the right-most and/or the left-most columns in the charts discussed below. **Cuda_Ref**: Measures in NVIDIA's GeForce Titan Black Z. **Cuda_Ref-2**: Measures in NVIDIA's GeForce Titan Black X.
- **CPU + GPU**: This code, programmed using the MCtrl library, executes the programmed application on two devices, a group of 10 CPU-cores, and an NVIDIA's GeForce Titan Black Z. Different mappings have been tested, determined by the percentage of data and computation assigned to each device.
- **GPU + GPU**: This code, programmed using the MCtrl library, executes the application on the two GPUs available in the target system. Again, different mappings have been tested, determined by the percentage of data and computation assigned to each device.

Figure 7 shows the performance results obtained by our proposal when the data, and thus the computation, is divided among the group of CPU-cores and a GPU, the NVIDIA's GeForce Titan Black Z. In the applications where data transfers dominate the total time (Matrix Addition and Black–Scholes benchmarks), we can achieve a better performance by giving part of the computation to the group of CPU-cores. Despite the computational power of the GPU accelerator, the computation division improves performance by reducing the time spent in data transfers to/from the GPU.

When the computational load is high, such as in the matrix multiplication, and Black–Scholes_2048, the best performance is obtained by doing the whole computation in the GPU. This is typical for this kind of programs that really suit the GPU computational model. For the Mandelbrot benchmark, we observe that the partition of the computation obtains marginally better performance results than the CUDA reference codes. This particular behavior is because of the irregular workload of this benchmark.

Figure 8 shows the performance results obtained when we divide the computation between the two GPUs. When the computation load is really low and there are multiple kernel launchings, the time spent in the queue managements (remember that these times are taken into account in the computation time) can be noticeable, such in the BlackScholes_2048 case (where it takes approximately 0.04 s).

In our first prototype of the library, data transfers for several GPUs are still sequentialized. Thus, in those applications where data transfers lead the execution time, the best performance is obtained when the application is executed only in the most powerful GPU. However, when the computation time is much higher than the communication times, a division of the computation among the GPUs, proportional to their relative computation power for this problem, improves the performance. For example, in our study, without taking into account the data transfers, the kernel execution times are reduced by 34, 25 and 41%, as compared to the best CUDA reference codes which use a single device, for the matrix multiplication, Black–Scholes_2048 and Mandelbrot benchmarks, respectively. This behavior is shown in the experimentation results.

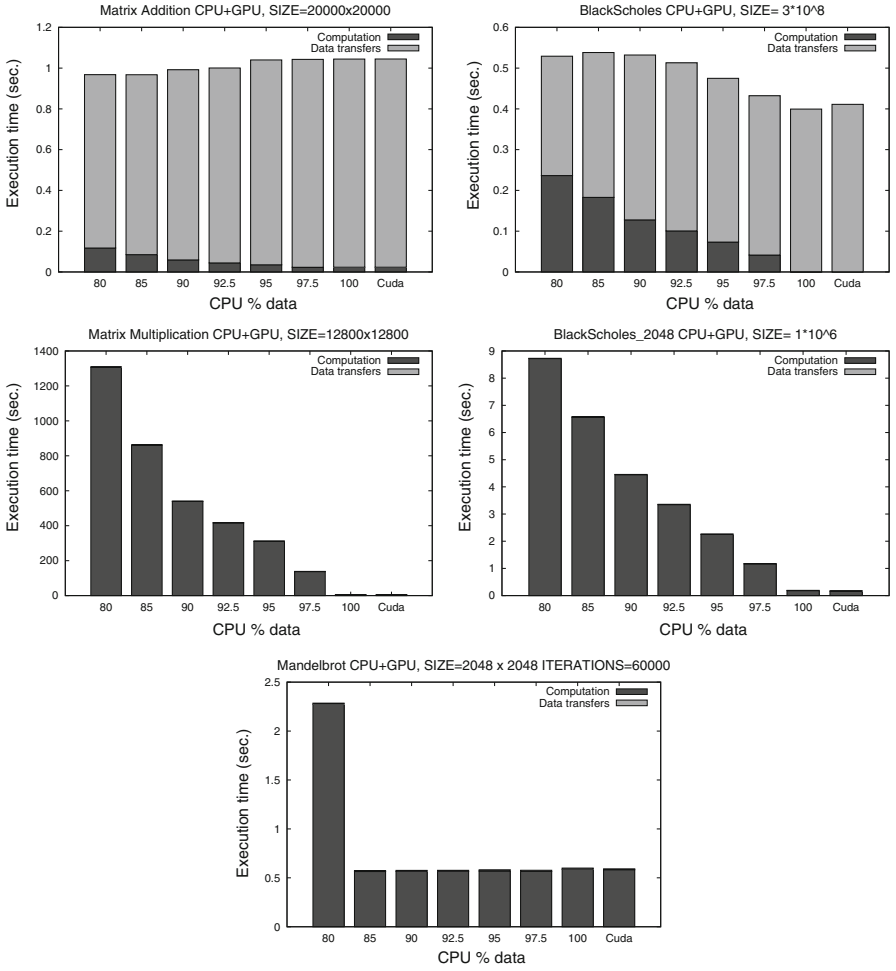

Fig. 7 Performance results (in seconds) for experiments on Hydra using a group of 10 CPU-cores and a GPU. The right-most columns show the result of the reference CUDA programs run on the same GPU

6 Conclusion

In this paper, we present the Multi-Controller (MCtrl), an abstract entity implemented in a library, that coordinates the management of heterogeneous devices, including accelerators with different capabilities and sets of CPU-cores. This entity offers a global view of the computation, transparently managing the coordination, data partition, mapping, and execution of whole computations on their associated devices. Our solution allows the use of simple generic kernels (portable across different device types), or specialized implementations defined and optimized using specific native or vendor programming models (such as CUDA for NVIDIA's GPUs, or OpenMP for CPU-cores). The run-time system automatically selects and deploys the most

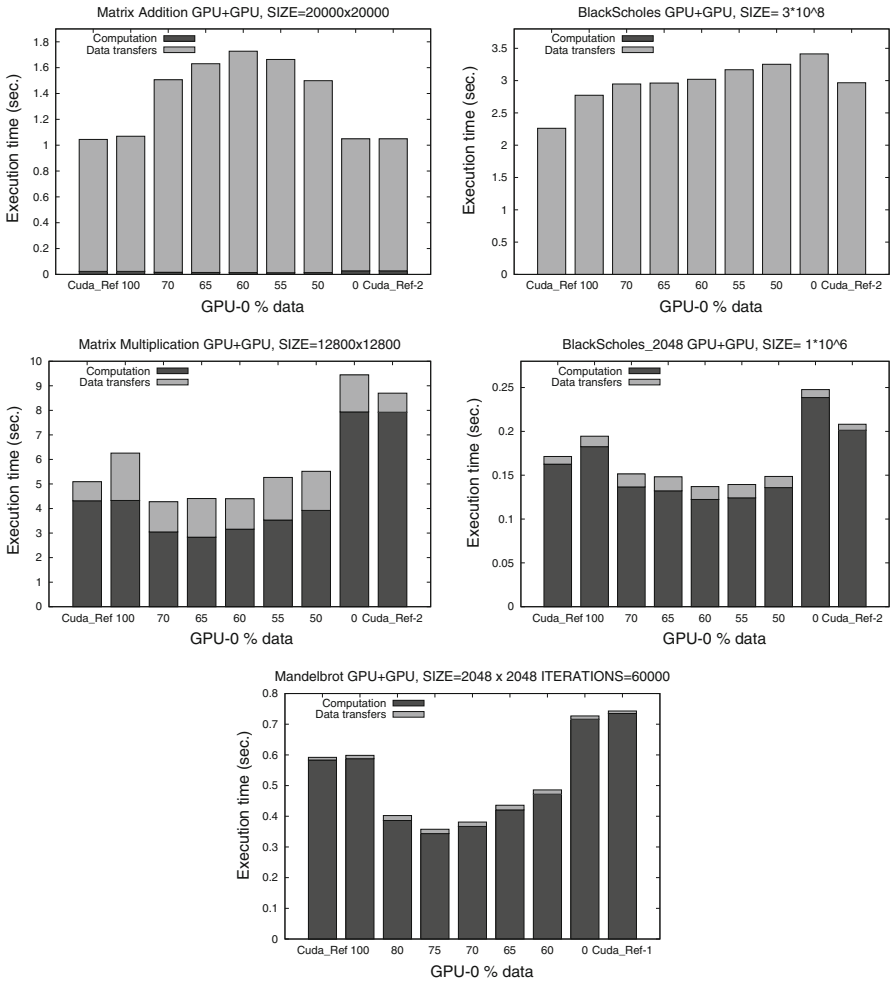

Fig. 8 Performance results (in seconds) for experiments on Hydra using two different GPUs. The left- and right-most columns show the results of the reference CUDA programs run on each of the two GPUs considered in the study

appropriate implementation of each kernel for each device, managing the data movements and hiding the launch details. Results of an experimental study with five study cases indicate that our abstraction allows the development of flexible and highly efficient programs that adapt to the heterogeneous environment. On-going and future work includes the study of the effect of more sophisticated techniques for data movement which overlap communication and computation, and a new experimental study including irregular benchmarks whose data should be divided using more complex mechanisms.

Acknowledgements This research has been partially supported by MICINN (Spain), the ERDF program of the European Union and Junta de Castilla y Leon: HomProg-HetSys Project (TIN2014-58876-P),

CAPAP-H6 (TIN2016-81840-REDT), COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS), and FEDER Grant VA082P17 (PROPHET Project).

References

1. Alonso-Mayo, A., Ortega-Arranz, H., Gonzalez-Escribano, A.: Communicators: an abstraction to ease the use of accelerators. In: HLPGPU'2016 (2016)
2. Dastgeer, U., Enmyren, J., Kessler, C.W.: Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In: Proceedings of IWMSE'11, pp. 25–32. ACM, New York (2011)
3. Gonzalez-Escribano, A., Torres, Y., Fresno, J., Llanos, D.R.: An extensible system for multilevel automatic data partition and mapping. IEEE Trans. Parallel Distrib. Syst. **25**(5), 1145–1154 (2014)
4. Haidl, M., Gorlatch, S.: PACXX: Towards a unified programming model for programming accelerators using C++14. In: Proceedings of LLVM-HPC'14. IEEE (2014)
5. Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York (1977)
6. Hijma, P., Jacobs, C.J., van Nieuwpoort, R.V., Bal, H.E.: Cashmere: Heterogeneous many-core computing. In: 2015 IEEE International and Parallel and Distributed Processing Symposium (IPDPS), pp. 135–145. IEEE (2015)
7. Hugo, A.E., Guermouche, A., Wacrenier, P.A., Namyst, R.: Composing multiple StarPU applications over heterogeneous machines: a supervised approach. In: Proceedings of IPDPSW'13 PhD Forum, pp. 1050–1059. IEEE, Washington (2013)
8. Karimi, K., Dickson, N.G., Hamze, F.: A performance comparison of cuda and opencl. arXiv preprint arXiv:1005.2581 (2010)
9. Liang, T., Li, H., Chiu, J.: Enabling mixed OpenMP/MPI programming on hybrid CPU/GPU computing architecture. In: Proceedings of IPDPSW'12, PhD Forum, pp. 2369–2377. IEEE, Washington (2012). https://doi.org/10.1109/IPDPSW.2012.294
10. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)
11. Moreton-Fernandez, A., Rodriguez-Gutiez, E., Gonzalez-Escribano, A., Llanos, D.R.: Supporting the xeon phi coprocessor in a heterogeneous programming model. In: European Conference on Parallel Processing, pp. 457–469. Springer, Cham (2017)
12. MoretonFernandez, A., OrtegaArranz, H., GonzalezEscribano, A.: Controllers: an abstraction to ease the use of hardware accelerators. Int. J. High Perform. Comput. Appl. (2017). https://doi.org/10.1177/1094342017702962
13. NVIDIA: NVIDIA CUDA C Programming Guide 7.5 (2015). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Last visit: November 16th (2015)
14. Nvidia, C.: Cublas Library, vol. 15, p. 27. NVIDIA Corporation, Santa Clara (2008)
15. Ortega-Arranz, H., Torres, Y., Gonzalez-Escribano, A., Llanos, D.R.: Optimizing an APSP implementation for NVIDIA GPUs using kernel characterization criteria. J. Supercomput. **70**(2), 786–798 (2014). https://doi.org/10.1007/s11227-014-1212-z
16. Pérez, B., Bosque, J.L., Beivide, R.: Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In: Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, pp. 42–51. ACM (2016)
17. Scogland, T.R., Rountree, B., Feng, W.C., de Supinski, B.R.: Heterogeneous task scheduling for accelerated openmp. In: 2012 IEEE 26th International and Parallel & Distributed Processing Symposium (IPDPS), pp. 144–155. IEEE (2012)
18. Shen, J., Varbanescu, A.L., Lu, Y., Zou, P., Sips, H.: Workload partitioning for accelerating applications on heterogeneous platforms. IEEE Trans. Parallel Distrib. Syst. **27**(9), 2766–2780 (2016)
19. Stone, J.E., Gohara, D., Shi, G.: Opencl: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(1–3), 66–73 (2010)
20. TOP500.org: Top500 supercomputing sites. WWW (2017). On http://www.top500.org/
21. Torres, Y., Gonzalez-Escribano, A., Llanos, D.R.: uBench: exposing the impact of CUDA block geometry in terms of performance. J. Supercomput. **65**(3), 1150–1163 (2013). https://doi.org/10.1007/s11227-013-0921-z