Controllers: An abstraction to ease the use of hardware accelerators

Ana Moreton-Fernandez, Hector Ortega-Arranz, and Arturo Gonzalez-Escribano
Universidad de Valladolid, Spain

December 2016

Abstract

Nowadays the use of hardware accelerators, such as the Graphics Processing Units (GPUs) or XeonPhi coprocessors, is key to solve computationally costly problems that require High Performance Computing (HPC). However, programming solutions for an efficient deployment in this kind of devices is a very complex task that relies on the manual management of memory transfers and configuration parameters. The programmer has to carry out a deep study of the particular data needed to be computed at each moment, in different computing platforms, also considering architectural details.

We introduce the *Controller* concept as an abstract entity that allows the programmer to easily manage the communications and kernel launching details on hardware accelerators in a transparent way. This model also provides the possibility of defining and launching CPU kernels in multi-core processors with the same abstraction and methodology used for the accelerators. It internally combines different native programming models and technologies to exploit the potential of each kind of device. Additionally, the model also allows the programmer to simplify the proper selection of values for several configuration parameters that can be selected when a kernel is launched. This is done through a qualitative characterization process of the kernel code to be executed.

Finally, we present the implementation of the Controller model in a prototype library, together with its application in several case studies. Its use has led to reductions in the development and porting costs, with significantly low overheads in the execution times when compared to manually programmed and optimized solutions using directly CUDA and OpenMP.

1 Introduction

Currently, the systems used for High Performance Computing (HPC) include accelerator devices. This trend is noticeable from personal computers to classical supercomputers. Examples of these HPC heterogeneous environments can be seen in the first positions of the current top 500 list of supercomputers. When developing solutions to be deployed in heterogeneous systems, we can: (1) Use a single programming model responsible of managing the architectural and conceptual differences between the different computational devices, e.g. OpenACC; or (2) Use a combination of different programming models specific for each kind of computational devices, e.g. MPI together with CUDA, OpenCL, or OpenMP.

One of the main drawbacks of the first approach is the difficulty to represent non-completely regular programs, with non-trivial communications or synchronizations. Besides, the final generated code resulting from programming with this kind of models is not usually as optimized as the original code. For example, the current OpenACC compiler implementations do not offer a complete solution to the problem of automatically choose appropriate values for several kernel-launching parameters without any programmer guidance, as required by the standard. These parameters include the threadblock size and its multi-dimensional geometry, or the configuration of the sizes of the L1 cache vs. the shared-memory in modern GPUs. On the other hand, implementing solutions following the second approach requires a deeper knowledge of the different parallel programming models involved, using different synchronization and memory access strategies for different devices. Additionally, the programmer is the final responsible of properly managing the data transfer among the different memory spaces of the computational devices, at the most appropriate times, together with the choice of proper values for kernel-launching configuration parameters. However, this manual adjustment gives to the programmer the possibility to optimize the use of the particular resources of each specific device. Other approaches that try to create a conceptual bridge between these two approaches are domain specific, or are based in sophisticated compiler technology for the generation of coordination or kernel codes.

In this paper we present the concept of *Controller* as an abstract entity that allows the programmer to transparently manage the launching of series of tasks on accelerator devices, and/or multi-core CPU processors. It is a redefinition of the *Communicator* entity presented in [1], with new functionalities and a complete new internal design that leads to a very efficient implementation. The Controller uses the most appropriate programming models (CUDA, OpenMP, ...) to exploit the computational resources of the accelerators and hosts machines. The model has several important features: (1) A mechanism to define common kernels reusable across different types of devices, or specialized kernels for specific device kinds, including the possibility to consider a subset of CPU cores as a single independent device; (2) 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; (3) A transparent mechanism of memory management, including optimized communications of the data structures between the host and the corresponding images in the accelerators; (4) An abstraction for indexed data structures that unifies the data management in kernels for different kinds of devices (such as GPUs, or multi-threaded vector CPU cores).

This paper also describes a prototype, presented as a C/C++ library extension, that implements the Controller entity. The prototype is designed to exploit NVIDIA's GPUs using the CUDA parallel programming model, or subsets of CPU cores using OpenMP. We also present an experimental study based on several case studies, some of them obtained from the CUDA Toolkit Samples. The study shows how the use of the prototype implies a reduction of the programming effort needed to implement and port these applications, when it is compared with the original versions that directly use the specific native parallel programming models. Besides, we show that our implementation does not introduce significant performance overheads.

The structure of the article is as follows. In Sect. 2 we discuss some related work. Section 3 presents the Controller model. Section 4 explains the prototype library, its usage to develop a program and relevant implementation details. Section 5 describes the experimental study. Finally, Sect. 6 describes the conclusions of this work and problems that will be addressed in the future.

2 Related work

In this section we discuss some work related to our proposal. We cover some examples of different approaches that ease the integrated programming of different computational devices, including accelerators.

There are several works integrating, in the same tool, different parallel languages or models, or considering different device types. One of these models, widely known, is OpenCL. It provides the *Context* abstraction, defining a memory model in which associated data is shared or moved between the host and the device memories when needed. Although OpenCL is internally exploiting the vendor drivers and native programming tools, its abstractions have been proved to prevent obtaining the same efficiency as when using directly the vendor programming models, for several common situations [11]. Moreover, the implementations are not easily reachable with respect to the definition of the management policies of the internal queues, or the possibility of change them. Another drawback of this model is the manual management of kernel compilation at run-time, for different architectures in different contexts, that is desired to be generalized and simplified. The llCoMP tool [17] is a source-to-source compiler that translates C annotated code to MPI+OpenMP or CUDA. However, it does not support the joint use of CUDA with the other parallel models. SkelCL [18] enhances the OpenCL interface to allow the coordination of different GPUs on the same machine. The works [3, 12] introduce hybrid MPI+CUDA approaches to coordinate GPUs in the same or different host machines. Apart from their specific limitations, they do not have an abstract support to easily manage homogeneously units of different nature, including CPU cores, as in our approach.

There are other approaches that are more domain specific, but include small abstractions to ease the management between the accelerator and the host (e.g. MCUDA [19] or hiCUDA [8]). They do not consider guided optimizations, nor allow the programmer to control the load distribution or the devices coordination. Other approaches with similar limitations also consider CUDA, MPI, and OpenMP combinations (e.g. [21, 9]).

More general approaches propose complete integrated frameworks, such as OMPICUDA [13], StarPU [10], or the skeleton programming framework based on it, SkePU [4]. In general, they hide the coordination details to the programmer, to the point of constraining the potential optimizations that could be achieved manually. The selection of launching parameters like the threadblock size is tackled in SkePU, but using a trial-and-error methods, thus leaving no possibility to extrapolate the results to other kernel codes or architectures. A higher-level approach is to rely on compiler technology to transparently generate code for different kinds of devices (both for coordination and for kernels) from a single unified language. For example, C++14 is used in PACXX [7], a transformation system integrated into the LLVM compiler framework. It transform explicit parallel constructions that use the concept of kernel and launching is an abstract an elegant form. On the other hand, some of the solutions are dependent on features of this language, and they are not easily portable to other ones. The decisions about launching parameters, such as the threadblock choice, are still the programmer responsibility alone.

Our solution solves previous limitations. It integrates the coordination of computational devices of very different natures in the context of a compiler library, using a new approach based on simple abstractions. It hides the differences of execution models up to the point of allowing the development of generic kernels portable across devices. But at the same time, it allows the integration of native or vendor programming models, applying specific optimization techniques, and avoiding efficiency losses associated with other generic approaches.

Our approach can be also used as a run-time system to generalize the porting of programs across different type of accelerator devices. Several automatic code-generation compile-time tools [2, 5, 14] can derive, from sequential code with pragma annotations: (1) The data dependences among the kernel launches; (2) The needed data transfers between the host and the target devices, and; (3) The domains on which kernels should be executed for accelerators. Such tools have been used typically to generate code for only one kind of accelerator. For example, in [14], the OpenMP 4.0 *#pragma offload* feature is used to generate codes to execute on the XeonPhi accelerator. Using the same information extracted by this technologies, together with the abstractions and generic programming guidelines of our proposal, it is possible to generate a generic code valid for any kind of execution device. This approach will be studied in a future work.

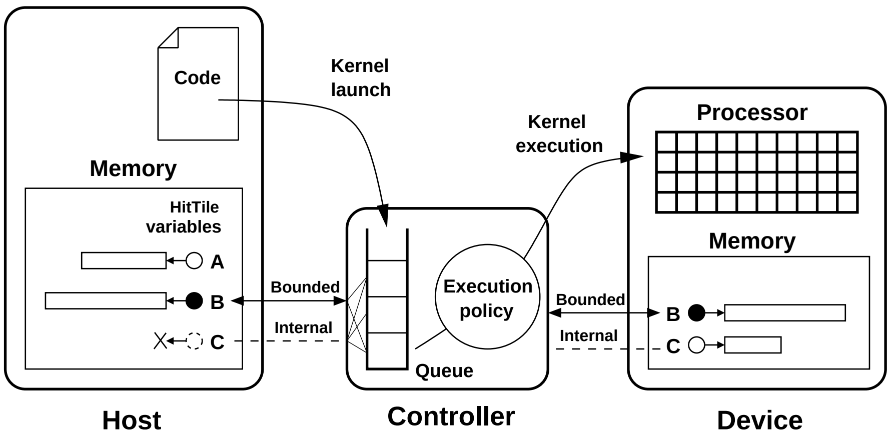

Figure 1: Diagram of the Controller model architecture. The kernel-launching requests can be enqueued. The Controller entity manages the execution of enqueued kernels, and for bounded variables, the data transfers between memory spaces. In the figure, the host variable A is not bounded to the Controller. Variable B is a bounded variable, with a duplicated image for the data in the device memory. Variable C is an internal variable of the Controller, defined in the host, but allocated only in the device.

3 Controller Model

The *Controller* model introduces a simplified way to program applications that can exploit heterogeneous computational platforms including accelerators or/and multi-core CPUs. Its architecture is represented in Fig. 1. The device Controllers coordinate the execution of series of kernels. These kernels are declared as functions, that are managed by the Controller entities. Controllers automatically manage the two main concepts used in a program that exploits accelerators:

- Kernel management, including the kernel launching and configuration. The Controller manages the deployment/execution of sequences of kernel functions in the computational device associated to the Controller. The Controllers can include policies to exploit concurrent kernel execution techniques, interleave computations with communications, or reorder the sequence of kernels. The kernel configuration is the selection of specific configuration parameters for the kernel launching, that can be associated to a particular kernel and computational platform.
- Data management, including the data transfers carried out across the memory hierarchies of the host and the accelerators and the abstraction used to access data elements independently of the target device, the threads indexes space, or the data layout.

3.1 Kernel management

Kernel launching is an operation that inserts in the Controller queue an order to execute a kernel, with given real parameters, when the associated device (accelerator or set of CPU cores) is available. This is done using a launching primitive (see Fig. 1). The Controller internals will ensure that the input data have been transferred to the device memory if needed, and that previous kernels have ended, before doing the real launch. The Controller execution policy could reorder the kernels in the queue to maximize the execution efficiency as far as the input/output dependencies across kernels are not violated. The main features and contributions of our model in the kernel management are described below:

3.1.1 Kernel definition and launching: Support of both generic and specialized codes

The model provides a generic method to declare kernels. We define a *kernel implementation* as a tuple: $(< name >, < deviceType >, < parameters\_list >, < code >)$, where *deviceType* is a symbol, or list of symbols (e.g. GPU, CPU), indicating the specific kind of device/s where the kernel can be deployed. Thus, we propose to allow the declaration of several implementations of the same kernel name, indicating the architecture/s for which this particular kernel is more suitable.

We also propose the possibility of defining kernel implementations for a *generic* device type. These kernels will be portable across different types of devices and architectures, and will be used by default when no specific kernel implementation is provided in the code for a given target device. As they are simply considered another kernel implementation, with a different device type symbol, they can be declared in the same program together with other implementations of the same kernel name. This allows the creation of libraries of kernel implementations, developing first a generic and portable one that will work in any platform, and introducing gradually more specific and optimized kernel implementations for specific or new target devices.

The code of the generic kernels implementation, in order to be portable across different device types, should comply in our model with a set of properties (see also Sect. 4.6 for a discussion about an example of a generic kernel used on both multi-core

CPU and GPU devices). Properties of a generic kernel: (1) The code is pure data-parallel, in the sense that it can be executed by many logical threads, and the programmer ensures that no data dependencies or race conditions can arise across them; (2) The code operates on indexed data structures using abstract 1, 2, or 3-dimensional thread indexes provided by the runtime system (t_x, t_y, t_z), that are internally adjusted for each device to access data in row-major order keeping maximum coalescing and/or vectorization properties in the specific device; (3) The code does not explicitly use resources, primitives, or synchronization mechanisms, that are specific of the programming model of a given device; (4) All accesses to data structures are done through an abstract interface provided by the runtime system, that is independent of the target device chosen (see Sect. 3.2.2 for our proposal to choose such an abstract interface).

A unified kernel launching function, at run-time, matches the architecture of the device to the best implementation provided at compile time for that architecture. In this way, it is possible to transparently use either generic portable kernels, optimized kernels programmed in the native models of a specific device, or even wrappers to call specialized libraries for specific device types. The Controller runtime system can choose the most appropriate one.

Finally, we propose to require in the parameters list of a kernel implementation, that the programmer reports the input/output *role* of each kernel parameter. This will be used by the Controller to detect which data should be transferred among different memory spaces, at each moment, depending on the sequence of kernels launched.

3.1.2 Characterisation of kernels for execution

Our model considers the characterization of the kernel code to automatically optimize launching parameters, such as the thread-block geometries. We propose to integrate the model of qualitative characteristics presented in [15, 20] in our Controller. To use this model, the programmer should examine the kernel code, and he should conceptually characterize it in classes according to three main criteria. We introduce an extension of the kernel implementation tuple with a new element to provide the classification of the kernel code. The Controller internally uses an associative table to implement the selection of launching parameters according to the rules proposed on the previously cited papers. More details about the criteria and examples are presented in Sect. 4.4.

3.2 Data management

Accelerators may have their own memory spaces, forcing to transfer the input data of the kernels, and the obtained results, between the memory of the host platform and the memory of the device accelerator. The manual management of these data movements can be cumbersome and error-prone. Moreover, it is difficult to predict in advance when asynchronous data transfers are possible, or when data should stay in the accelerator device memory, as this depends on the exact sequence of kernels launching. Our model abstracts from the programmer all these issues. The main features/contributions of our model on data management are the following:

3.2.1 Data transfers between the host and the accelerators

In our model, a Controller is associated, in the moment of its creation, with a particular accelerator or subset of CPU cores, and it transparently manages in the accelerator memory space the images of the host memory data structures. The Controller can decide when and how the transfers should be carried out, depending on the data structures used in the corresponding kernels, and the sequence of kernels enqueued for launching. The programmer can use the original names of the host variables in the kernel transparently. Depending on the role of the variables (named data-structures) used as real parameters in kernel launches, we can distinguish two types: Bounded variables and internal variables.

Bounded variables

Bounded variables are host variables that have an image in the memory space of the accelerator (see Fig. 1). The model defines an operation to bind a host variable to the Controller. Once a variable is bounded, its data should not be modified by the program at the host side until an unbinding operation is applied.

The first kernel requiring the use of a bounded variable as an input will force the Controller to transparently ensure that its data have been already transferred. Applying an unbinding operation to a bounded variable will force the transfer of its data from the accelerator to the host, if it has been used as output by any kernel. The main program waits until the end of the kernels using that variable, and the end of the data transmission to the host if needed.

Internal variables

Internal variables are variables whose scope is delimited to the kernels executed in the accelerator. They are only handled inside the memory space of the device, and they will not have allocation in the host memory space. Thus, they never imply a data transfer (see Fig. 1). In the particular case of a device representing a subset of CPU cores, the memory should be transparently allocated and managed in the host device.

The model defines an operation to create an internal variable in the Controller. A data structure is declared without allocating memory to it in the host. The name of this data structure is used in the creation operation to clone the type, size, and internal structure in the Controller memory space. Since that moment, the name can be used as real parameter in kernel launching as a reference of the internal variable. To destroy an internal variable it is needed to apply another operation using the reference name of the host variable.

3.2.2 Uniform data accesses

One of the key features that a programming model for heterogeneous systems should provide is the ability to manage uniformly the data structures on a program. As previously commented when discussing generic kernel definitions, the data accesses on the body of the kernel should be independent of the target device chosen. We propose the use of abstract thread indexes and data-accessing methods in the kernel codes. They are devised in order to design the codes to work efficiently when accessing elements in row-major order, independently of the device. We propose the use of the data-handler abstraction for arrays introduced by Hitmap [6], a library for hierarchically distributed arrays. Efficient implementations have been provided for different devices, such as CPUs, or GPUs. See more details about the Hitmap functionalities used in the implementation of our model in Sect. 4.1.

4 The Controllers library

We have developed an implementation of the Controllers model. It is designed as a library written in C99 code. Thus, it can be used to develop C/C++ programs, independently of the chosen compiler. The library defines functions, but also relies on preprocessor macros for code rewriting. Although this allows an efficient interface implementation in a compiler agnostic way, the programmer should take care to use the interface in the expected way to avoid problems derived from some common pitfalls of macro substitutions (unexpected type checking issues, misnesting due to incorrect code injection in the macro parameters, etc.).

The current implementation supports NVIDIA's GPU devices using CUDA internally, and subsets of CPU codes using OpenMP internally. In our implementation, a kernel is a function coded for any, or for a specific computational device with particular input/output parameters. The Controllers library interface defines primitives to: Create Controllers; Declare and characterize kernels; Manage host data structures that can be bounded or created as internal variables in the Controllers, and transparently accessed inside the kernels; and Launch the kernels.

4.1 Data structures and Hitmap

Regarding the data structures that the model handles, we have decided to integrate our implementation with Hitmap [6], an efficient library for hierarchical tiling and mapping of arrays. 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. The circles for the variables A, B, and C in Fig. 1 represent the handlers. The *HitTile* structure should be specialized for each array base type at the beginning of the program.

There are only four functions of Hitmap needed to work with the Controllers implementation. First, *hit_tileDomain* and *hit_tileDomainAlloc* are used to declare the index domains of a tile array, also allocating the memory for the data in the second case. 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 indexes values of the desired element. This function provides an homogeneous interface to manage data structures in both, host and accelerator device kernels. The data are stored and accessed in row major order in all cases.

Hitmap library includes many other functionalities. They include management of hierarchical subselections of parts of the tiles, and transparent management of distributed-arrays, with abstract partition and communication functionalities that internally use a message-passing paradigm (exploiting MPI). This will allow in the future the transparent integration of the Controllers in distributed multi-node clusters.

Most of the meta-data in the *HitTile* structure are only needed for advanced functionalities, such as distributed-array partitions and communications in the host code. In our new implementation we define a new much smaller handler (*HitKTile*) with the minimum information needed for data accesses to multidimensional arrays through a data pointer. The kernel launching interface will transparently transform the handlers to this new type, substituting the data pointer with its equivalent in the device memory space. The data accesses inside the kernels uses this internal pointer along with a minimum number of arithmetic operations, exposing the expressions to the native compiler to open the possibility of further optimizations. The result obtains as good performance as the classical array accesses.

In Hitmap, the implementation of different kinds of tiles hides to the programmer the details of the data access for different internal data layouts. The Hitmap library already integrates sparse domains and sparse data structures into the HitTile abstraction. The future transformation of these handler structures, to implement efficient and portable kernels, should follow a similar approach as the one used for dense arrays.

4.2 Controllers and variables management

Initialization and destruction

A Controller is associated to a particular computational platform (accelerator or CPU-cores set) at the moment of its creation. This is done through the `CtrlCreate` function. This primitive has two main parameters: The name of the Controller variable, and the identification of the associated computing device. The programmer can free the resources with the `CtrlDestroy` function.

The Controllers associated to CPU cores internally use OpenMP. To allow the launching of CPU kernels asynchronously, as in the case of GPUs, our implementation uses OpenMP tasks. A master thread executes the host code, and one OpenMP task is activated for each core associated to a CPU Controller. The Controller initializes on its creation an internal array of structures with one element per assigned core task, to control their activities and their synchronization.

Binding variables

The function `CtrlAttach` binds a tile defined in the code of the host with a Controller. The function `CtrlDetach` unbinds it. If the memory space of the device is not the same as the host, the attach operation allocates memory for the data in the device space. After the binding, the host should not manipulate the tile data until it is unbounded. During the time the variable is bounded, operations to copy the data to and from the associated device can happen at any time, as an internal decision of the Controller. For the particular case of Controllers associated to CPU cores, there is no data duplication. The kernels may be modifying the host variable data at any time.

In the current implementation we have integrated part of the variables management in the Hitmap tile handlers. The handler stores the pointer to the device memory, a reference to the Controller is bounded to, and flags to indicate the clean/dirty state of the data. This avoids the duplication of bindings, attempts to unbind variables from the wrong Controller, etc. The flags help the Controller to choose the proper moments to synchronize the data between the host and the device memory image.

Creating internal variables

The function `CtrlInternalCreate` creates an internal variable on the device memory space. On the other hand, the function `CtrlInternalDestroy` is used to free the memory space in the assigned computational device. For the case of Controllers associated to accelerators, this kind of variables does not need to have allocated memory in the host side. A tile initialized with a domain information is enough. Even if it would have allocated memory, there is no synchronization of data between the host and the device image created by this functionality.

4.3 Declaration and configuration of kernels

A kernel is declared by using the primitive `KERNEL_<type>`. Where `type` may be empty to indicate a generic 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 primitives `KERNEL_GPU` for CUDA code targeting NVIDIA's GPUs, `KERNEL_CPU` for host machine code targeting sets of CPU cores, and `KERNEL_GPU_WRAPPER` for host machine code which includes calls to specialized GPU libraries, like for example cuBLAS routines. The Controller ensures that both kinds of GPU kernels are executed with exclusive control of the associated device. This allows to automatically coordinate the launch of sequences of classical kernels and calls to GPU libraries in the same device with our model.

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, with elements can be both read and written.
- INVAL: for input parameters of any type passed by value.

For the case of accelerator devices, with separated memory spaces, this configuration allows the Controllers to determine if it is necessary to carry out data transfers with the main memory of the host when kernels are launched. The primitive is followed by a structured block with the kernel code. Thus, it resembles a C function header.

The Controller CPU implementation contains the loops to execute the kernel code for each element of the threads index space, that is internally assigned to a specific OpenMP thread. Both CPU and GPU kernels code use a predefined *thread* structure with three integer elements *x,y,z* indicating the 3-dimensional indexes of the corresponding fine-grain thread. In order to ease the kernel reuse across different device kinds, these indexes are adapted to have the same row-major meaning in both types of

```
/* CUDA kernel launch */
/* Parameters:
  name: Kernel name
  threads: CtrlThreads, index space limits
  arch: GPU architecture
  params: real parameters of the launch
  kchar: characterization of the kernel
*/
dim3 block = CALBlockModel(name,kchar,arch);
dim3 grid = CALGridDivUp(threads,block);
wrapper_gpu_##name<<<grid, block>>>
                     (threads, params);
...
__global__ void wrapper_gpu_##name( ... ) {
   CALThread threadId = { threads.dims,
     threadIdx.y + blockDim.y * blockIdx.y,
     threadIdx.x + blockDim.x * blockIdx.x,
     threadIdx.z + blockDim.z * blockIdx.z };
   ...
   kernel_gpu_##name( threadId, params );
}
```

```
/* OpenMP index space partition and kernel launch */
/* Parameters:
  name: Kernel name
  n_th: OpenMP thread identifier
  numCores: number of CPU-cores associated to
            the current controller
  threads: CtrlThreads structure with limits of the
           threads index space
  params: real parameters of the kernel launch
*/
int stripSize =(int) ceil( threads->x/(numCores) );
int begin= n_th*(stripSize);
int end= MIN(first + (stripSize) - 1,(threads->x)-1);
for(i=begin; i<=end; i++)
   for(j=0; j<threads->y; j++){
         threadId.x = i;
         threadId.y = j;
         threadId.z = 0;
         kernel_cpu_##name(threadId, params);
      }
```

Figure 2: Excerpts of the Controller library code generated for kernel deployment/launching on a CUDA capable GPU device (left), and a group of CPU-cores (right). In the CUDA version the block geometry is selected using the kernel characterization provided by the programmer. We also show the part of the wrapper function launched, that reverses the thread indexes before calling the actual kernel code. For the case of the CPU-cores, each OpenMP thread (assigned to a core) executes this code. In this example, the 2-dimensional index space is divided into blocks by rows without balancing the remaining. The code executes the loops that call the kernel code for each index-element mapped to the thread. It simulates the many-thread approach used for GPU programming using coarser-grain OpenMP tasks.

kernels. Also, the space of valid indexes that can be defined in the kernel launching is independent of the threadblock sizes in GPUs. Actual threads with identifiers outside the chosen launching index space will be clipped transparently by the Controller launching system. For specialized GPU kernels, the kernel code may use CUDA primitives mixed with the Hitmap data accesses that use the new portable index system.

4.4 Kernel characterization

The kernel characterization is a programmer hint to help the system to automatically determine proper kernel launching parameters in terms of special code features and platform architecture information. The CPU threads granularity in our prototype is determined by a simple regular blocking policy, that does not require a specific kernel characterization.

For GPU kernels, our current prototype library integrates the model presented in [15, 20]. This model allows to determine configuration parameters (grid, threadblock and L1 cache memory sizes), for NVIDIA's GPUs. The primitive `KERNEL_CHAR`, taken from [16], is used to provide to the Controller the characterization of the kernels. The primitive 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 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). For the default case, when the programmer cannot provide a proper characterization for all the parameters, a *def* keyword can be used instead of one of the given values, and the model provides typical threadblock values for each CUDA architecture, that work well in a general case, maximizing occupancy if possible, etc. For a more detailed description of this qualitative descriptors with tentative quantitative ranges, see the works [15, 20, 16].

Our implementation extends this characterization with the possibility to specify a fixed size for the threadblock. This is useful for kernels that rely on specific block sizes or geometries to manipulate shared memory (for example the matrix multiplication code in the CUDA Toolkit Samples).

4.5 Kernel launching

The function `CtrlLaunch` is used to launch a kernel, with given real parameters, to the computational device associated to a Controller. The launched kernel will be enqueued, and eventually executed with the corresponding configuration derived from the information provided by the characterization primitives. Currently, the prototype only supports a First-Coming-First-Service policy for kernels execution. The launching function has the following parameters: (a) The Controller; (b) The name of the

```
1  /* HitTile specialization
2   * Creates new type HitTile_float */
3  hit_tileNewType( float );
4
5  /* Kernel characterizations */
6  KERNEL_CHAR(copyCell,1,def,def,def)
7  KERNEL_CHAR(updateCell,1,full,low,high)
8
9  /* Generic kernel codes for any device */
10 KERNEL(copyCell, 2,
11   OUT, HitTile_float, dst,
12   IN,  HitTile_float, src) {
13
14   hit_tileElem( dst, 2, thread.x, thread.y ) =
15      hit_tileElem( src, 2, thread.x, thread.y );
16 }
17
18 KERNEL(updateCell, 2,
19   OUT, HitTile_float, dst,
20   IN,  HitTile_float, src) {
21
22   int row = thread.x + 1;
23   int col = thread.y + 1;
24   hit_tileElem( dst, 2, row, col ) =
25     ( hit_tileElem( src, 2, row+1, col   )
26     + hit_tileElem( src, 2, row-1, col   )
27     + hit_tileElem( src, 2, row,   col+1 )
28     + hit_tileElem( src, 2, row,   col-1 )
29     ) / 4;
30 }
```

```
1  /* Tile declaration and allocation (host) */
2  HitTile_float M, Mcopy;
3  hit_tileDomainAlloc(&M, float, 2, rows, columns);
4  hit_tileDomain(&Mcopy, float, 2, rows, columns);
5
6  /* Tile initialization */
7  for (int i=0; i<rows; i++)
8    for (int j=0; j<rows; j++)
9      hit_tileElem( M, 2, i, j ) = ...
10
11 /* Controller creation */
12 Controller ctrlGPU, ctrlCPU;
13 CtrlCreate(&ctrlGPU, COMM_GPU, 2);
14 CtrlCreate(&ctrlCPU, COMM_CPU, 4, 7);
15
16 /* Tile binding and creation (device) */
17 CtrlAttach(&ctrlGPU, &M);
18 CtrlInternalCreate(&ctrlGPU, &Mcopy);
19
20 /* Kernel launching */
21 domain1 = CtrlThreads( 2, rows, columns );
22 domain2 = CtrlThreads( 2, rows-2, columns-2 );
23 for (iter=0; iter< num_iterations; iter++) {
24   CtrlLaunch(&ctrlGPU, copyCell, domain1,
25             2, Mcopy, M);
26   CtrlLaunch(&ctrlGPU, updateCell, domain2,
27             2, M, Mcopy);
28 }
29
30 /* Unbinding and freeing resources */
31 CtrlDetach(&ctrlGPU, &M);
32 CtrlInternalDestroy(&ctrlGPU,&Mcopy);
33 CtrlDestroy(&ctrlGPU); CtrlDestroy(&ctrlCPU);
34 hit_tileFree(&M); hit_tileFree(MCopy);
```

Figure 3: Example of the kernel characterization and definition (left) and main code (right), for a stencil program implementing an iterative Jacobi PDE solver for the Poisson's heat diffusion equation. The kernels are usable for both CPU and GPU Controllers.

kernel; (c) The index space of the thread set; (d) The number of parameters required by the kernel; and (e) The real parameters for the kernel execution.

The dimensions of the threads index space are specified using a *CtrlThreads* structure that stores up to three integer values representing the cardinality on each dimension. It is equivalent to the *dim3* type in CUDA, but it is used for both, GPU or CPU kernels independently. The variable parameters should be tile variables associated to the Controller, internal or bounded, or any value of the proper type for the INVAL parameters.

Internally, the execution of a GPU kernel implies: (1) the creation of the small handlers using the original tile handlers information; (2) The use of the characterization model to select the grid and threadblock geometries; and (3) the execution of a wrapper function. This wrapper reorders the thread indexes to be used as in the CPU kernels code to access data elements in row-major order, and ensures that threads outside of the required index space return immediately before executing the kernel code (see an excerpt of this code on the left of Fig. 2). For CPUs, the kernel execution internally implies the partition of the index space in coarse blocks. Figure 2 (right) shows an example of a simplified code that performs this partition. The kernel launching (line 18) calls to an inlined function that is generated when this kernel implementation for CPU-cores is declared. A final global synchronization is needed to preserve the launching semantic before proceeding to launch another kernel.

4.6 Programming example

In this section we present a practical application of the library concepts described in the previous section by using an example. Figure 3 shows the implementation of a Stencil 2-D application. On the left of the figure, first, we declare at the beginning of the program the specialized Hitmap array types to be used in the program (see line 3 in Fig. 3 left).

Kernel characterization: Two kernels are characterized at lines 6 and 7 of Fig. 3 (left). The first kernel characterization uses the default configuration, while the second one looks for a particular configuration for the *updateCell* kernel. This last characterization is based on the code properties. According to the classifications presented in [15, 20], the global accesses of the program suggest an almost fully coalesced memory access pattern. The ratio of arithmetic operations per data element is low because less arithmetic operations are done compared to read operations. The ratio of data shared between threads of the same block is high, because most neighbor values are reused across their computations.

Kernel definition: The kernel definitions start in lines 10 and 18. The kernel definitions include the name of the kernel, the

```
/* Recurrence equation kernel */
KERNEL_CHAR(kRecurrence, 2, full, high, low)

/* Black Scholes kernel */
KERNEL_CHAR(kBlackScholes, 1, full, medium, low)

/* Stencil Jacobi kernels */
KERNEL_CHAR(kCopy, 2, full, low, low)
KERNEL_CHAR(kUpdate, 2, medium, medium, medium)

/* GPU matrix multiplication kernel */
KERNEL_CHAR(kMatrixMult, 2, fixed-square-32)
```

```
/* Kernel wrapper for cuBLAS matrix mult. */
KERNEL_GPU_WRAPPER( HitTile_float A,
                    HitTile_float B,
                    HitTile_float C ) {
  const float alpha = 1.0f;
  const float beta  = 0.0f;
  cublasHandle_t handle;

  cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
       B.dimx, A.dimx, A.dimy, &alpha,
       hit_ktileRawData(B), B.dimx,
       hit_ktileRawData(A), A.dimx, &beta,
       hit_ktileRawData(C), B.dimx );
}
```

Figure 4: Characterization of the generic or GPU specialized kernels for the case studies (left). Example of kernel wrapper to execute a specialized GPU library function (right).

number of parameters, a specification for each parameter of its input/output mode and type, and the parameter name. Both are generic kernel codes valid for any kind of device as the *KERNEL* primitive points out. The code in the kernel bodies is executed in parallel on the device by as many threads as we choose in the main program. Another kernel definition example can be seen on Fig. 4 (right). In this case, it is a specific kernel code for GPUs that calls a function of a specialized library for NVIDIA's GPUs.

Data management: All the accesses to the variables in the generic kernel body (either reads or writes) must be done through the Hitmap functions. For example, line 14 of Fig. 3 (left) shows the copy of an element of the matrix *src* on the matrix *dst*. It uses the *hit_tileElem* function to access the data element corresponding to the indexes assigned to the logical thread that executes the kernel. We can see another example on line 24 of Fig. 3 (left). The *dst* matrix element corresponding to the thread indexes is updated using the values of the neighbor cells in the matrix *src*.

We show on the right of Fig. 3 the code of the main function that coordinates the application execution.

Data structures: Data structures are created and initialized in the first part of the main program (see lines 1 to 9 in Fig. 3 right). The *M* matrix is allocated on the host, where it is initialized. Thus, we use the *hit_tileDomainAlloc* function to create the data structure. However, the *Mcopy* matrix is an internal variable, and does not need to have allocation in the host memory space. Only a virtual representation is created, without assigning actual memory, by using the *hit_tileDomain* function.

Controller entity: Lines 13 and 14 of Fig. 3 (right) show examples of the Controller creation. One associated to the third GPU of the system, and another one associated to the subset of CPU cores with indexes in the range 4 to 7. Once the Controller is created, host data structures are binded to the target devices, and internal data structures are created in the target devices through the Controller (see lines 17 and 18 of Fig. 3 (right)). **Kernel launching:** With the data structures in the target device, we can launch the kernels. The kernel would be executed by as many threads as a *CtrlThread* object specifies. We see in lines 21 and 22 how two *CtrlThread* objects are created. The first one includes the domain of the whole data structure, and the second one the whole data structure without the borders. After that, in lines 24 and 26 the two kernels are launched using their different thread-index domains, by using the different *CtrlThread* objects. After the desired number of iterations of the kernels, the control of the result data structure is transferred again to the host by unbinding it, and the Controller is destroyed (see lines 31 to 33) .

To summarize, we observe that the final program presents an organized sequence of programming phases, that leads to a clear structure. Easy generic guidelines to program with this library can be deduced from this simple example.

5 Experimental study

This section describes the experiments we have carried out to check the functionality, and to evaluate the potential performance issues, introduced in the Controller prototype implementation. We also evaluate the development effort of using the Controllers prototype when compared to directly using common native programming models (CUDA or OpenMP).

5.1 Case studies

As case studies we have selected the following programs. All of them work with floating point numbers. From the CUDA Toolkit Samples, we have selected the *Black-Scholes* program, and two *Matrix Multiplication* examples, one using GPU shared-memory, and another directly calling the optimized *CUBlas* routine. We also use the stencil program that is shown in Fig. 3. Finally, we have selected a code to independently apply a trivial recurrence equation to elements in a matrix. A quick description and the motivation to choose these examples follows. The characterizations of the kernels using the proposed model, are shown in the left of Fig. 4. The parameters where chosen following the guidance presented in [15, 20].

Case study	Version	Lines of Code	#Tokens	Cyclomatic Complexity
Recurrence equation	CUDA	44	404	5
	Ctrl.GPU	33	315	3
	OpenMP	27	243	4
	Ctrl.CPU	36	339	3
Black Scholes	CUDA	148	903	8
	Ctrl.GPU	106	704	7
	OpenMP	81	539	7
	Ctrl.CPU	90	707	6
Jacobi solver	CUDA	43	445	8
	Ctrl.GPU	40	371	4
	OpenMP	33	310	6
	Ctrl.CPU	47	456	5
Matrix mult.	CUDA	71	614	5
	Ctrl.GPU	47	429	4
	OpenMP	22	235	4
	Ctrl.CPU	40	389	4

Case study	CUDA $\rightarrow$ OpenMP	Ctrl.GPUs $\rightarrow$ Ctrl.CPUs
Recurrence equation	Common 14% Delete 29% Change 57%	Common 95% Delete 4% Change 1%
Black-Scholes	Common 56% Delete 32% Change 11%	Common 72% Delete 17% Change 11%
Jacobi solver	Common 13% Delete 42% Change 44%	Common 91% Delete 0% Change 9%
Matrix multiplication	Common 8% Delete 4% Change 88%	Common 49% Delete 3% Change 48%

Table 1: Measurements of the development effort metrics for the codes of the case studies (left). Comparison in terms of the percentage of words that are common and can be reused, should be deleted, or should be changed, when porting codes between GPU and CPU versions using the native models, or the Controllers model (right).

Recurrence equation

This code uses two input matrices A, B, and writes a result matrix C that is originally initialized with zeros. The kernel computes the first 500 terms of a trivial recurrence equation that involves only two single addition operations: $C_x(i,j) = C_{x-1}(i,j) + A(i,j) + B(i,j)$, providing as a result the last term computed for each position. There is only one kernel launching for the whole program. It is an embarrassing parallel code, where each element is computed independently. The code involves a high number of independent and repetitive coalesced reads and writes to global memory on each thread. This artificial code is specifically designed to test the efficiency of the data accesses through the new small tile handlers inside the kernel codes. A common kernel is used for both CPU and GPU Controllers.

Black-Scholes

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 in the CUDA Toolkit Sample independently applies the formula to a chosen number of input values stores in an array, calculating and storing their results. Thus, it is also 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 result stored in registers or temporal variables. The kernel is called 512 times consecutively. This program is adequate to measure the cost of multiple kernel calls of more sophisticated arithmetic computation, with a very low number of global memory accesses per thread. A common kernel is used for both CPU and GPU Controllers.

Stencil computation

This program computes the stability point of a Partial Differential Equation (PDE), in this case the Poisson's equation the heat diffusion. It uses a Jacobi iterative method on a 2-dimensional discretized space, represented as a matrix. The program implementation is shown in Fig. 3. It is an 4-point stencil program that executes a fixed number of time iterations. On each iteration it independently computes a new value for each cell in the matrix, using the information of the four neighbor cells.

This kernel presents a similar number of arithmetic operations per thread as global memory accesses. The read operations are not completely coalesced due to the neighbor accesses, and the positions read by each thread are overlapped with the neighbor threads. A neighbor synchronization is needed at the end of each iteration, before a new simulation step starts. This kernel measures the effects of non-completely coalesced global memory accesses.

These stencil simulation programs are usually optimized to swap the two data structures (the one read, and the one write) after each iteration. We have intentionally skipped this optimization, implementing another kernel that simply copies the new generated data from the written variable to the original one. It shows the ability of the characterization model to provide different launching parameters for different kernels in the same program. The program is based on a repetitive invocation of two kernels with very low load per thread. Thus, it tests the efficiency of the implementation of the kernel launching procedures. Again, the same kernel is defined for both CPU and GPU platforms.

Case study		Recurrence	Black-Scholes	Jacobi	Matrix Mult.
Number of iterations		500	512	200	-
CPU Data size		$15\,000 \times 15\,000$	1 000 000	$10\,000 \times 10\,000$	$3\,000 \times 3\,000$
GPU Data Size	S	$10\,000 \times 10\,000$	1 000 000	$1\,000 \times 1\,000$	$3\,008 \times 3\,008$
	M	$15\,000 \times 15\,000$	10 000 000	$5\,000 \times 5\,000$	$5\,024 \times 5\,024$
	L	$20\,000 \times 20\,000$	50 000 000	$10\,000 \times 10\,000$	$10\,016 \times 10\,016$
	XL	$25\,000 \times 25\,000$	100 000 000	$20\,000 \times 20\,000$	$15\,008 \times 15\,008$

Table 2: Input data sizes and number of iterations selected for each case study in the performance experimental study.

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 another computation. Nevertheless, different cells used elements of A or B that are also read by other cell computations. Thus, data can be reused and shared across the computation of each cell. Moreover the read patterns on A and B matrices should be studied to exploit coalescence in GPUs, and properly exploit caches in CPUs. These lead to interesting optimizations in both GPU and CPU devices. Thus, we use different specialized kernels for each kind of devices.

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 of 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, the 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 specific CUDA code. This code, due to the way it uses the shared memory and it aligns the read operations, forces the use of a specific square threadblock size (32×32). It is an example of the utility of an extension to the characterization model introduced in our implementation.

For the GPU Controllers we have a second implementation. It shows the functionality of our kernel-wrapper facility, that allows to implement and launch as a kernel a call to the optimized cuBLAS routine for matrix multiplication (see the right of Fig. 4). The wrapper kernels do not need characterization, as they contain only host code, and the library routines include the code that take the decisions about launching parameters.

5.2 Development effort and code complexity

The first part of our experimental study evaluates how the use of our proposed model affects the development effort when compared with using the native programming models for the two types of devices considered in the current version of the prototype: CUDA for GPUs, and OpenMP for multi-core CPUs.

We measure three classical development effort and code complexity metrics: COCOMO lines of code, number of tokens, and McCabe's cyclomatic complexity. The first two ones measure the volume of code that the programmer should develop. The third one measures the rational effort needed to program it in terms of code divergences and potential casuistry that should be considered to develop, test, and debug. The metrics are applied to the part of the code that includes the kernel, the functions invoked by them, and the host coordination code. We ignore input data initialization, error or results checking, performance instrumentation, and writing messages to the standard output. Thus, the considered host code includes the declaration, creation and initialization of Controller, data containers, and structures for parallelism coordination, the memory transfer between host and devices, and the kernel launching operations.

Table 1 (left) shows the measurements for these metrics in the baseline versions, using OpenMP for the CPU variant, and CUDA for the GPU variant, and the versions using our Controllers library interface. The results indicate that programming with Controllers for GPUs generates a significantly lower volume of code, and a reduced cyclomatic complexity, indicating a clearly lower development effort than using CUDA. A closer look at the codes indicates that most of the reduction is found in the host part of the codes, as expected. On the other hand, for this kind of data parallel codes, the comparisons with OpenMP codes show that using the Controllers interface reduces a little the cyclomatic complexity, but increases the code volume.

The main advantage of considering the CPU cores as an accelerator device in the Controller model is found in the portability of code between GPUs and CPUs. Table 1 (right) shows the percentage of words that can be reused, should be deleted, or should be changed to port from a CUDA program to the equivalent OpenMP version; and the same measurements when porting from a Controller version for GPUs to the equivalent Controller version for CPUs. The results clearly indicate that the portability of the Controller versions across different device types is really high in the three first cases, and still significant for the matrix multiplication program, that includes different specialized kernels. In this case the coordination code using Controllers is still the same, deriving in a significantly lesser number of words to change than when porting between CUDA and OpenMP.

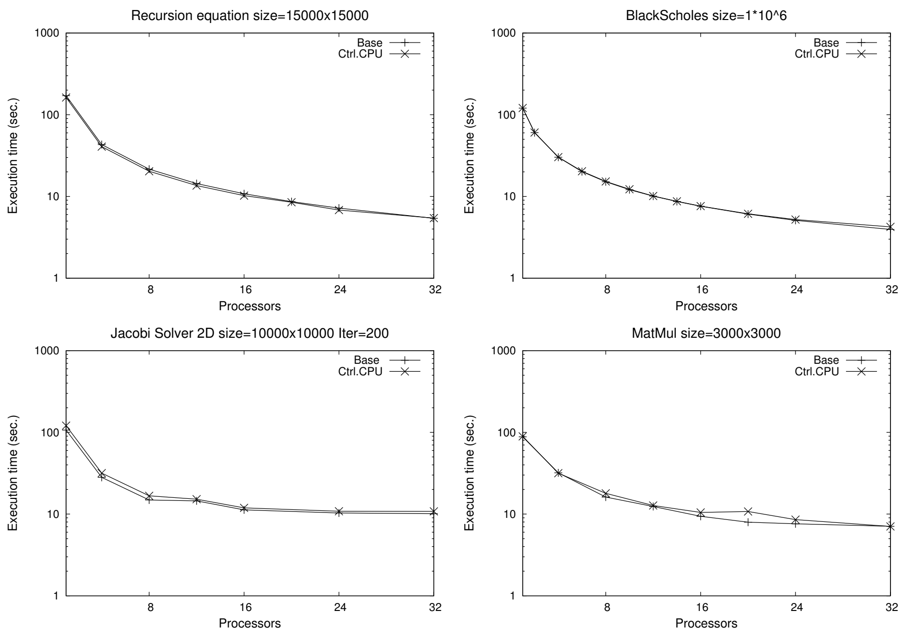

Figure 5: Execution times (seconds) in Heracles machine of the baseline (Base) and the Controller versions for CPU device (Ctrl.CPU) with a variable number of cores.

5.3 Performance study

The second part of our experimental study measures the impact of using our Controllers prototype in terms of performance. We have run the GPU implementations in an NVIDIA's GeForce Titan Black X, with CUDA capability 5.2, installed on a host machine named *Hydra*, with two CPUs Intel Xeon E5-2609 v3 @1.90GHz, and 64Gb DDR3 main memory. To test the CPU implementation we have used a shared-memory machine with a higher number of cores. It is named *Heracles*, and it is a Dell PowerEdge R815 server, with 2 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores per processor, and a total of 32 cores. The operating system in both machines is a Linux Centos 7 OS. The programs have been compiled using the CUDA Toolkit 7.5, and GCC 4.8.3. We have used the flags *-arch=compute_52*, *-O3*, and the *-mtune* choice appropriate for each host machine.

We present comparisons of the execution times skipping input data initialization, results checking, and control messages writing. The instrumented code includes Controllers creation and variables binding/unbinding operations (which may imply data transfers). In the same way, CUDA data copies between the host and the device memories are also included. In the case of GPU programs we also measure the execution time accumulated by the kernels launching alone, without memory transfer operations. Due to instabilities on the execution times of the host codes and data transfers, the results always show the mean of the execution time obtained on 5 repetitions of each test.

For the CPU versions we have selected input sizes with enough load to achieve scalability in our test machine with up to 32 threads. The exact input size parameters chosen are presented in Table 2. The total execution times of the baseline programs and the versions based on Controllers are presented in Fig. 5, for different number of active threads/cores. The results indicate that our implementation of the Controller abstraction for CPUs does not implies significant overheads for the scalability ranges tested.

For a fair comparison, both GPU approaches (CUDA and Controller based) use the same values for the configuration of launching parameters. In the case of the Black-Scholes program, the threadblock size value predicted by the characterization model is the same found in the original CUDA code. In the matrix multiplication case, it is fixed in both cases to 32×32. For the other two case studies, the values predicted by the characterization model are injected manually in the baseline versions. These values have shown to produce at least 96% of the best performance obtained with other threadblock geometries. We compare the performance obtained when launching the baseline and the Controllers versions with different input data sizes, selected to produce from very low to significant execution times (from tenths of a second, to more than four seconds). The exact input size parameters are presented in Table 2, with a class name (S, M, L, XL) for easier reference in the following discussion.

The results for GPU programs are presented in Table 3. We skip to present times for the matrix multiplication using the cuBLAS library. The memory transfers and launching operations are enclosed into the library routine, that is called by both the CUDA and the wrapper virtual kernel in the Controller version. Thus, the performance is the same except for stochastic behaviors that affect both codes equally.

The execution times of the kernels show a little overhead when using the Controller interface, due to an extra internal stage for thread index conversion and threads space clipping. Nevertheless, the overhead is always less than 1% of the kernels mean execution time. The recurrence equation kernel, specifically designed to test the efficiency of the data accesses through the small

		Recurrence		BlackScholes		Jacobi		Matrix Mult.	
Size	Measure	CUDA	Ctrl.GPU	CUDA	Ctrl.GPU	CUDA	Ctrl.GPU	CUDA	Ctrl.GPU
	Host	0.2580	0.2559	0.0084	0.0047	0.0033	0.0023	0.0357	0.0274
S	Kernels	0.5592	0.5723	0.0409	0.0454	0.0151	0.0167	0.0663	0.0666
	Total	0.8172	0.8283	0.0493	0.0500	0.0184	0.0191	0.1020	0.0941
	Host	0.5720	0.5721	0.0617	0.0654	0.0672	0.0676	0.0495	0.0416
M	Kernels	1.2753	1.2752	0.3862	0.3951	0.3203	0.3285	0.2878	0.2983
	Total	1.8473	1.8469	0.4479	0.4604	0.3875	0.3960	0.3373	0.3399
	Host	1.0249	1.0236	0.2955	0.2899	0.1167	0.1168	0.4977	0.4944
L	Kernels	2.2502	2.2502	1.9166	1.9221	1.2277	1.2306	2.1012	2.0925
	Total	3.2750	3.2739	2.2121	2.2120	1.3444	1.3474	2.5989	2.5869
	Host	1.5909	1.5864	0.5954	0.5695	1.1901	1.1915	0.4191	0.5748
XL	Kernels	3.4668	3.4669	3.8311	3.8652	4.8493	4.8574	7.0198	7.0203
	Total	5.0577	5.0529	4.4265	4.4346	6.0394	6.0490	7.4389	7.5441

Table 3: Execution time (seconds) for the case studies versions using CUDA, or Controllers for GPUs, with different input sizes. Host time measures the data communications, coordination code, and launching operations. Kernels time measures the actual kernel execution times. Total times are the addition of both times for easy comparison.

kernel tile handlers, shows the minimum overhead of the four cases.

As expected, the execution times of the host codes, in both CUDA and Controller versions, are smaller than the corresponding kernels execution, specially for large input sizes (L and XL), that present significant kernel execution times. These times spent on host code coordination, and memory transfers, are less stable than the kernels execution times. Sometimes, the Controller versions show even better results than the equivalent host code in the CUDA versions. The same memory transfers in the CUDA codes are internally executed in the Controller versions. Memory transfers have the less predictable times, with a higher variance. The rest of the Controller operations are in general light loaded. The critical one is the kernel launching, that involves several checks and handler manipulations. Nevertheless, the overhead of the launching operation has been measured to be less than 1×10^{-5} seconds for the GPU implementation, in the machine used for this study. Thus, the accumulated overhead of many launching operations in the Black-Scholes, or the Jacobi programs, is not significant even for the smaller input sizes tested (class S). This overhead could only be clearly noticeable when executing long sequences of kernels with really low computational load, which are not in general appropriate for parallelism exploitation.

6 Conclusions

In this paper we propose the Controller model, a parallel programming model that simplifies the coding of applications for heterogeneous systems. It is based on an abstract entity, the Controller, that manages the launching of kernel sequences on accelerators, or sets of CPU cores.

It provides mechanisms to: (1) Associate Controllers to devices; (2) Define portable kernels that can be reused across different types of devices; (3) Define specialized kernels for the same program on different device types; (4) Automatically select proper values for launching parameters on different devices through a characterization of the kernels provided by the programmer, and; (5) Automatically deal with different memory spaces of the host and the devices when needed. This model unifies the kernel programming and data structures management, bringing closer the accelerator and multi-threaded programming, taking into account the architectural differences of the accelerator platforms to obtain good performance. Our experimental study shows the advantages of using this approach in terms of development effort metrics, and the efficiency of our prototype implementation for several case studies representing different computational costs, and global memory accesses scenarios.

Our future work includes the extension of the prototype in order to support the management of other types of accelerators, such as XeonPhi coprocessors; further CPU kernel optimizations; and other techniques that exploit the modern features of modern accelerators, such as asynchronous communications, communication-computation overlapping, and different kernel launching policies. The use of Hitmap partition and communication functionalities will be explored to use the Controllers in higher scale distributed multi-node environments.

Acknowledgements

This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

References

[1] A. Alonso-Mayo, H. Ortega-Arranz, and A. Gonzalez-Escribano. Communicators: An abstraction to ease the use of accelerators. In *HLPGPU'2016*, ene 2016.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10. ACM, 2008.

[3] Q. K. Chen and J. K. Zhang. A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA. In *ICISE'2009*, pages 86 –89, dec. 2009.

[4] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning SkePU: A Multi-backend Skeleton Programming Framework for multi-GPU Systems. In *Proc. IWMSE'11*, pages 25–32, New York, NY, USA, 2011. ACM.

[5] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 375–386. IEEE, 2013.

[6] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos. An extensible system for multilevel automatic data partition and mapping. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1145–1154, 2014.

[7] M. Haidl and S. Gorlatch. PACXX: Towards a Unified Programming Model for Programming Accelerators using C++14. In *Proc. LLVM-HPC'14*. IEEE, 2014.

[8] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In D. R. Kaeli and M. Leeser, editors, *GPGPU*, volume 383, pages 52–61. ACM, 2009.

[9] M. Howison, E. Bethel, and H. Childs. Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):17–29, 2012.

[10] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach. In *Proc. IPDPSW'13 PhD Forum*, pages 1050–1059, Washington, D.C., USA, 2013. IEEE.

[11] K. Karimi, N. G. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.

[12] N. Karunadasa and D. Ranasinghe. Accelerating high performance applications with CUDA and MPI. In *ICIIS'2009*, pages 331–336, dec 2009.

[13] T. Liang, H. Li, and J. Chiu. Enabling Mixed OpenMP/MPI Programming on Hybrid CPU/GPU Computing Architecture. In *Proc. IPDPSW'12, PhD Forum*, pages 2369–2377, Washington, D.C., USA, 2012. IEEE.

[14] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire. Offload compiler runtime for the intel® xeon phi coprocessor. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1213–1225. IEEE, 2013.

[15] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. Optimizing an APSP implementation for NVIDIA GPUs using kernel characterization criteria. *The Journal of Supercomputing*, 70(2):786–798, 2014.

[16] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. TuCCompi: A Multi-layer Model for Distributed Heterogeneous Computing with Tuning Capabilities. *International Journal of Parallel Programming*, 43(5):939–960, 2015.

[17] R. Reyes and F. de Sande. Optimization strategies in different CUDA architectures using llCoMP. *Microprocess. Microsyst.*, 36(2):78–87, Mar 2012.

[18] M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 7979 of *LNCS*, pages 258–272. Springer Berlin Heidelberg, Berlin, Germany, 2013.

[19] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In J. N. Amaral, editor, *LCPC'2008*, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. uBench: exposing the impact of CUDA block geometry in terms of performance. *The Journal of Supercomputing*, 65(3):1150–1163, 2013.

[21] C. Yang, C. Huang, and C. Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182(1):266–269, 2011.

Short bios of the authors

Ana Moreton-Fernandez received her MS in Electronics and her MS in Research in Information and Communication Technologies from the University of Valladolid, Spain, in 2013 and 2014, respectively. Currently, Ms. Moreton-Fernandez is a Ph.D. candidate at the University of Valladolid. Her research interests include parallel programming languages and models for heterogeneous systems.

Hector Ortega-Arranz received his M.Sc. in Computer Science and his M.Sc. in Research in Information and Communication Technologies from the University of Valladolid, Spain, in 2010 and 2011, respectively. He has been part of the Trasgo research group where he has developed the work for his Ph.D. Thesis, graduating in 2015. Nowadays he is part of a bio-tech research company where he applies the learned knowledge to reveal microbiological communities in crop soils using DNA sequencing. His research interests include algorithms, shortest-path problem approaches, parallel and distributed computing, GPGPU computing, and their application to any bio-field.

Arturo Gonzalez-Escribano received his Ph.D. in Computer Science from the the University of Valladolid, Spain, in 2003. He is associate professor at the University of Valladolid since 2008. He has participate and lead several funded research projects, technology transfers to enterprises, and a national research network. His research interests include high performance computing, parallel programming models and languages, compilers, run-time technology, and heterogeneous and embedded systems.