Supporting the Xeon Phi Coprocessor in a Heterogeneous Programming Model

Ana Moreton-Fernandez(✉), Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano, and Diego R. Llanos

Departamento de Informática, Edif. Tecn. de la Información, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain
{ana,eduardo,arturo,diego}@infor.uva.es

Abstract. Supercomputers are becoming more heterogeneous. They are composed by several machines with different computation capabilities and different kinds and families of accelerators, such as GPUs or Intel Xeon Phi coprocessors. Programming these machines is a hard task, that requires a deep study of the architectural details, in order to exploit efficiently each computational unit.

In this paper, we present an extension of a GPU-CPU heterogeneous programming model, to include support for Intel Xeon Phi coprocessors. This contribution extends the previous model and its implementation, by taking advantage of both the GPU communication model and the CPU execution model of the original approach, to derive a new approach for the Xeon Phi. Our experimental results show that using our approach, the programming effort needed for changing the kind of target devices is highly reduced for several study cases. For example, using our model to program a Mandelbrot benchmark, the 97% of the application code is reused between a GPU implementation and a Xeon Phi implementation.

1 Introduction

Supporting computational accelerators such as GPUs or Xeon Phi coprocessors in current programming models is vital to exploit modern parallel platforms. Different kinds and families of accelerators are used in modern high-performance platforms, as we observe in the configuration of the TOP500 supercomputers [17]. However, programming solutions for an efficient deployment in accelerator devices in general is a very complex task [12], that relies on the manual management of memory transfers and execution configuration parameters. For each different computing device, the programmer has to carry out a deep study of the particular data needed to be computed at each moment, considering architectural details to exploit efficiently the specific execution system [1].

Many works address the problem of heterogeneous systems management (e.g. [3,4,15]) following two alternatives: automatically generating specific codes from sequential or higher-level programming abstractions, or using runtime libraries that make transparent the use of different device types. Using current heterogeneous code generators or compilers, the code should be recompiled

© Springer International Publishing AG 2017
F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 457–469, 2017.
DOI: 10.1007/978-3-319-64203-1_33

for each different execution platform in order to better exploit the performance capabilities of the system. One example is OpenAcc [19]. It provides a simple and abstract programming framework for accelerators. However, the code should be recompiled with their specific compilers for each different execution architecture in order to achieve a good performance.

As for libraries, some works focused on specific kind of applications, address the portability problem internally using several native programming models. For example, MAGMA library [5] provides a unified programming environment for heterogeneous systems using both CPUs and accelerators, such as GPUs or Intel Xeon Phi, for dense linear algebra algorithms. However, most heterogeneous libraries rely on the OpenCL abstraction. OpenCL [16] is a widespread programming framework to deal with heterogeneous devices. The OpenCL *context* abstraction allows the memory management of multiple devices of the same nature (using the same *platform* in OpenCL notation). The abstractions introduced by OpenCL have been proved to prevent the obtaining of the same efficiency as when using directly the vendor programming models, for several common situations [9]. Many state-of-the-art heterogeneous frameworks and libraries with a high level of abstraction [2,7,8,14,18,20], that rely on OpenCL as execution layer, typically inherit some of these problems. Additionally, during the last decade several high-performance libraries targeting specific accelerator devices or CPU architectures, have been developed using the vendor specific programming models, such as cuBLAS [13] or MKL. For making the most of such works, it is advisable the use of such native or vendor programming models and compilers, for each different kind of device.

In this paper, we extend a programming model for heterogeneous platforms that is not based on the use of OpenCL. The original proposal, named *Controller* [11], is presented as a library that internally exploits vendor-specific programming models available on the platform. It introduces an abstract entity to allow the transparent launching of series of tasks on a GPU or on a CPU. It exploits their native or vendor specific programming models, thus enabling the potential performance obtained by them. In this work we present an extension of this Controller heterogeneous programming model that includes the support for Intel Xeon Phi coprocessors, also known as Many Integrated Cores (MICs). The model is based on the mix of the communication model originally designed for GPUs in the Controller library with the execution model originally designed for groups of CPU cores. We develop a complete runtime execution system that includes methods for task launching, transparent data transfers between the MIC accelerator and the host, and a queue system to manage the kernel executions with a customized grain choice. It perfectly fits with the previous Controller library, thus standardizing and abstracting to the programmer the issues related to the programming of different kinds of accelerators. It provides a MIC runtime support for a heterogeneous programming model, that unifies the programming for heterogeneous systems composed by MIC coprocessors, GPUS, or CPUs, also obtaining the same performance than using their native programming models.

We also present an experimental study with four study cases. We show that our approach is highly flexible, with minimum programming effort for changing the kind of target devices. Moreover, the performance results show that our implementation does not introduce significant performance penalties compared with reference codes which use the native/vendor programming models directly.

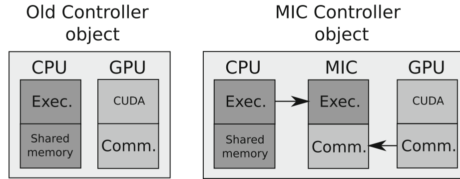

Fig. 1. Left: previous Controller model, only supporting CPU and GPU. Right: the MIC Controller proposed model, mixing features from the GPU-CPU submodels.

2 Approach to Support MIC Accelerators

The work presented in [11] proposed the Controller, a simple heterogeneous programming model to deal with the issues of hybrid computation in an abstract way to the programmer. This model defines an object able to transparently manage either a group of CPU cores or a GPU accelerator using internally native programming techniques (OpenMP and CUDA respectively).

For our new proposal, we distinguish two internal parts in the Controller, which provide support for each kind of computational device: Execution and communication management models (see left of Fig. 1). The abstract device formed by CPU-cores shares the memory space with the host. Thus, the controller object only has to provide an execution model that manages a task queue and that adapts the fine-grain computations used in the Controller model to a coarser granularity, more appropriate for CPU threads. On the other hand, for GPUs, the CUDA programming model already provides an execution system to enqueue and launch kernels with the same granularity level used in the Controller model. However, the communication operations across different memory spaces on host and GPUs required the implementation of a more sophisticated mechanism to integrate different policies and techniques in the Controller model.

In this work we understand as independent parts in the Controller abstraction both, the execution and the communication management models from different kind of devices. Thus, we propose their mixture to support a new type of accelerator such as the MIC coprocessors. In order to do that, we use: (1) In terms of execution, the Controller model for groups of CPU cores, that blends blocks of fine grained kernels into coarse CPU tasks, is appropriated for MIC coprocessors. (2) In terms of memory management, the abstract model for data communications needed for the MIC coprocessors is equivalent to the GPU communication model in the old Controller approach.

The application of this idea leads to a homogeneous programming model for heterogeneous systems including MIC coprocessors, where the issues related to the programming of different types of accelerators are transparent for the programmer. In this work we show the implementation of this idea in the Controller library, to support computational devices such as MIC coprocessors, GPUs, and groups of CPU-cores, without redesigning or changing the high-level programming model and interface.

3 Programming with the Controller Model

In this section we describe the concepts related to programming using the Controller model. It introduces the reader to Hitmap [6], the library used for the managing of data-structures, and the properties of the Controller model [11] abstract interface and its programming methodology.

Hitmap Library. Hitmap is the library used in the Controllers to provide a common interface for the data management inside the generic portable kernels. Hitmap defines the *HitTile* structure, an abstract entity for n-dimensional arrays and array tiles of arbitrary size. A *HitTile* structure is a handler to store array meta-data, along with the pointer to the actual memory space. There are only three functions of Hitmap needed to work with the Controller library. The function *hit_tileDomainAlloc* is used to declare the index domains of a tile array and allocate the data memory. The function *hit_tileFree* is used to free the data memory and clean the handler. The function *hit_tileElem* is used in the host or kernel codes to access the elements of a tile. It receives a tile name, a number of dimensions, and the indexes values of the desired element. The data are accessed in row major order in all cases, independently of the device implementation.

Controller Features. The Controller model provides a systematic programming methodology together with several important features: (1) A mechanism to define from common kernels reusable across different types of devices, to specialized kernels for specific device kinds; (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. These features makes the use of our proposal adaptable to the programmer knowledge. Thus, for non-experts users, it is possible to program a generic approach achieving good performance. On the other hand, a user with programming experience in the execution platform can take advantage of this knowledge and achieve better results.

Kernel Definitions. 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 may be a specific value for a given type of device indicating a specialized code. The original library supported 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 code to execute in the host which includes calls to specialized GPU or CPU libraries, such as cuBLAS or MKL routines.

```
1  /* Matrix addition: Generic kernel code for any type of device */
2  KERNEL(MatAdd, 3,
3         OUT, HitTile_float, C,  IN,  HitTile_float, A,  IN,  HitTile_float, B ){
4   int x = thread.x; int y = thread.y;
5   hit_tileElem( C, 2, x, y ) = hit_tileElem( A, 2, x, y ) +
6                                hit_tileElem( B, 2, x, y );
7  }
8  /* Host program using the Controller library */
9  int main(){
10  int SIZE = 10000;
11  /* Stage 1: Controller creation */
12  Cntrl comm;
13  CntrlCreate(&comm, CNTRL_GPU, 0);
14  /* Stage 2: Data structures creation and initialization */
15  HitTile_float A; HitTile_float B; HitTile_float C;
16  hit_tileDomainAlloc( &A, float, 2, SIZE, SIZE );
17  hit_tileDomainAlloc( &B, float, 2, SIZE, SIZE );
18  hit_tileDomainAlloc( &C, float, 2, SIZE, SIZE );
19  initMatrices(&A, &B, &C);
20  /* Stage 3: Data structures attachment */
21  CntrlAttach(&comm, &A); CntrlAttach(&comm, &B); CntrlAttach(&comm, &C);
22  /* Stage 4: Kernel launching */
23  Thread threadsSpace;
24  ThreadInit( threads, 2, SIZE, SIZE );
25  CntrlLaunch(comm, MatAdd, threadsSpace, 3, &A, &B, &C);
26  /* Stage 5: Data structures detachment */
27  CntrlDetach(&comm, &C);
28 }
```

Fig. 2. Kernel definition and configuration, and host program of a matrix addition using the Controller library.

We can see a kernel definition in lines 2–7 of Fig. 2. The kernel-definition primitive specifies 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.

Programming Methodology. Building a Controller host program follows simple development guidelines: (1) The Controller entity creation, associating to this object the computational device to be managed. A Controller entity should be created for each computational device that will be used for computation. (2) The attachment of the data structures to the Controller object. Data structures that will be accessed by a kernel should be previously attached to the Controller entity. (3) The launching of the computational kernels on the Controller object. (4) The detachment of the data structures.

Figure 2 shows a matrix addition implementation that performs the computation on a GPU using the Controller model. In the main program, first, a

Controller object is created, assigning a GPU to the object (lines 12 to 13). Data structures are created and initialized on the host (lines 15 to 19). After that, these data structures are attached to the previously created Controller (line 21). In the step 4, the program launches the kernel *MatAdd*. It uses a *Thread* object to specify the number and index space of the threads to be launched. In this example a thread is launched for each element of the matrix C (lines 23 to 25). Finally, the program detaches the matrix with the results (line 27).

In this paper, we propose a method to integrate the support of MIC coprocessors in this model, that allows the efficient execution of this program on the Xeon Phi, only by changing the `CNTRL_GPU` parameter by a new `CNTRL_XPHI` parameter on the line 13 of the code.

4 Integrating MIC coprocessors in the Controller library

The original version of the Controller library supports the deployment of kernels on GPUs or virtual computational devices formed by groups of CPU-cores. In this section we present the support of the MIC devices in the Controller library. We implement the **MIC controller object** containing several functionalities, such as the identification, initialization and management of MIC devices, an adapted internal queue to manage the asynchronous kernel executions, and a method to lock accesses to the HitTile data structures on the host while they are managed in the device memory.

```
1  /* Internal attach function */
2  void attachToXPHI(CntrlXPHI* cntrl,
3                    HitTile *tile){
4   Lock(tile, cntrl);
5   int MIC= cntrl->MIC;
6   char *data = (char *)(*tile).data;
7   int numBytes = hit_tileSize(tile);
8   #pragma offload target(mic:MIC)    \
9            in( data:length(numBytes) \
10             alloc_if(1) free_if(0) )
11 }
```

```
1  /* Internal detach function */
2  void detachToXPHI(CntrlXPHI* cntrl,
3                    HitTile *tile){
4   int MIC= cntrl->MIC;
5   char *data = (char *)(*tile).data;
6   int numBytes = hit_tileSize(tile);
7   #pragma offload target(mic:MIC)   \
8      in( data:length(0)             \
9            alloc_if(0) free_if(0) ) \
10     out( data:length(numBytes)     \
11           alloc_if(0) free_if(1) )
12  Unlock(tile, cntrl);
13  }
```

Fig. 3. Excerpts of the Controller internal code that perform data transfers of a HitTile object. Left: from the host to a MIC coprocessor. Right: from a MIC coprocessor to the host.

4.1 Attaching and Detaching Data Structures on the MIC

In computational devices such as GPUs or MIC coprocessors, where their memory spaces are separated from the host memory space, the attachment/detachment operation also implies a data transfer.

We have implemented two internal functions to perform the data transfers to/from the MIC coprocessor, using the Intel Language Extensions for Offload (LEO). These functions are executed internally when the program invokes an attachment or a detachment operation respectively. Figure 3 shows a summarized version of the code of both functions.

On the left, we see the code used to attach a tile to a MIC controller object (represented in the figure by the `CntrlXPHI` type). In this function, first the attached tile is locked on the host. Second, the code extracts: (1) The MIC identifier assigned to the controller object (line 5); (2) The pointer to the actual data (line 6); and (3) The number of bytes to be transferred (line 7); After that, the function performs the actual data transfer from the host to the MIC, ensuring that there is allocated memory space in the target device (using `alloc_if(1)`), and that after this offloading the actual data will be maintained (using `free_if(0)`).

On the right, we show the code used to detach a tile whose data have been modified from a MIC controller object. As in the attachment, first the code extracts the information about the data transfer (lines 4 to 6). Second, the actual data transfer from the coprocessor to the host is specified using a pragma. For determining the pointer of the data previously transferred, the program uses the `in` modifier to make the data pointer available in the Xeon Phi, and sets the *length* to 0 to prevent any data from being copied (lines 8 to 9). Once the pointer is available on the MIC, the pragma also specifies the data transfer and the freeing of the MIC space memory (lines 10 to 11). Finally, the data structure is unlocked on the host.

```
1  /* Auxiliar macros for kernels with one parameter */
2  #define STRINGIFY(a) #a
3  #define XPHI_WRAPPER_PARAMS1(io1, type1, value1)                               \
4          type1 value1
5  #define XPHI_WRAPPER_VALUES1(io1, type1, value1)                               \
6          value1
7  #define XPHI_WRAPPER_CAST1(io1, type1, value1)                                 \
8          type1 value1_p = (type1)args[2];                                       \
9          HitTile value1_t = *(HitTile*)value1_p;                                \
10         char *data_tile1= (char *) (value1_t).data;
11 #define XPHI_OFFLOAD_PARAMS1(MIC, io1, type1, value1)                          \
12         offload target(mic:MIC) in(threads:length(3)) in(value1_t)             \
13                                 in(data_tile1:length(0) alloc_if(0) free_if(0))
14 #define XPHI_POINTERS1(io1, type1, value1)                                     \
15         HitTile value1 = value1_t;                                             \
16         value1.data = data_tile1;
```

Fig. 4. Auxiliary macros defined for a one parameter kernel.

4.2 New Kernel Definitions

A kernel definition specifies the device that fits with the contained code by declaring it using the primitive `KERNEL_<type>`. We extent the Controller framework to support also MIC kernel definitions. A MIC kernel definition is rewritten

```
1  /* Macro of the kernel definition */
2  #define KERNEL_XPHI(name, nparams, params...)                                   \
3    /* Single-element function declaration */                                     \
4    static void __attribute__((target(mic)))                                      \
5        kernel_xphi_##name(Thread threadId, XPHI_WRAPPER_PARAMS##nparams(params)); \
6                                                                                  \
7    /* Parallel coarse-grained function */                                        \
8    static inline void wrapper_xphi_##name(void** args){                          \
9      int MIC=cntrl->MIC;                                                         \
10     CntrlXPHI* cntrl = (CntrlXPHI*) args[0];                                    \
11     Thread* threads = (Thread*)args[1];                                         \
12     XPHI_WRAPPER_CAST##nparams(params);                                         \
13     _Pragma( STRINGIFY(XPHI_OFFLOAD_PARAMS##nparams(MIC, params)) )             \
14     {                                                                           \
15      XPHI_POINTERS##nparams(params);                                            \
16     _Pragma("omp parallel"){                                                    \
17      int i,j,k;                                                                 \
18      Thread threadId;                                                           \
19        _Pragma("omp for private(i,j,k)")                                        \
20        for(i=0; i<=threads->x; i++){                                            \
21          for(j=0; j<=threads->y; j++){                                          \
22            for(k=0; k<=threads->z; k++){                                        \
23              threadId.x = i;                                                    \
24              threadId.y = j;                                                    \
25              threadId.z = k;                                                    \
26              kernel_xphi_##name(threadId, XPHI_WRAPPER_VALUES##nparams(params)); \
27        } } }                                                                    \
28      }}                                                                         \
29                                                                                 \
30   /* Task addition function */                                                  \
31   void name##_xphi(CntrlXPHI* cntrl, Thread thread,                             \
32                    XPHI_WRAPPER_PARAMS##nparams(params)){                       \
33       CntrlXPHIAddTask(cntrl, wrapper_xphi_##name, thread, nparams,             \
34                        XPHI_WRAPPER_VALUES##nparams(params));                   \
35   }                                                                             \
36   /* Single-element function definition */                                     \
37   static void __attribute__((target(mic)))                                      \
38       kernel_xphi_##name(Thread threadId, XPHI_WRAPPER_PARAMS##nparams(params))  \
```

Fig. 5. Functions internally generated by the MIC kernel definition: (1) Function to be executed by each fine-grain virtual thread: `kernel_xphi_##name`; (2) Function that executes a dequeued kernel, grouping virtual threads in coarse-grained OpenMP threads: `wrapper_xphi_##name`; (3) Function to enqueue a kernel-launching request: `name##_xphi`.

as three functions using macro functions. We show examples of the code of the three resulting functions in Fig. 5.

Fine-Grain Virtual Thread Function: The first function implements the kernel code that the programmer defined to execute for one index element of the fine-grain virtual threads space. In most array operations, it is used to compute one data element. The function is named `kernel_xphi_##name`, where *##name* is the kernel name, taken from the first parameter of the kernel definition primitive. It is defined as a MIC function using the attribute `target(mic)`. The parameters are a multi-dimensional index represented by a `Thread` object, that represents a point in the execution domain, and the actual kernel parameters. In Fig. 5, lines 4 to 5 show the function declaration and lines 37 to 38 the function definition.

Parallel Coarse-Grained Function: The second one (`wrapper_xphi_##name`) performs the offloaded coarse-grained parallel computation in the MIC device. It receives a variable number of parameters. The first one is the controller object, the second one the domain of fine-grain thread indexes to compute and the rest are the data structures corresponding to the real parameters. Lines 10 to 12 of Fig. 5 show how the information is extracted from the parameters (auxiliary macros for the transformations were defined in Fig. 4). The rest of the body of the function defines the offload region. The offload pragma transfers the data-structure handlers, the domain represented by a `Thread` object, and the pointer to the actual data for each HitTile. As in the detachment operation, in order to determine the data previously transferred, the offload pragma uses the `in` modifier to make the data pointer available in the Xeon Phi, and sets the *length* to 0 to prevent any data from being copied (see line 13 of Fig. 4). Inside the offload region, the HitTile handlers update their data pointer to the actual offloaded data (line 15). After that, the parallel computation is performed on the specified domain (lines 16 to 28), grouping virtual thread indexes in actual coarse-grained threads, by using an OpenMP parallel loop.

Kernel Launch Request: The third one is named `name##_xphi`. It is the internal implementation of a kernel launch for a MIC. In its body, the function implements the enqueuing of the kernel execution request in the Controller object. The information needed is: The controller object, the pointer to the coarse-grained parallel computation function, and its real parameters (the index space where the application will be executed, the number of kernel parameters, and the actual kernel parameters). See lines 31 to 35 of Fig. 5.

4.3 Queue Management and Kernel Launching

As opposite as the CUDA programming model, the offloading MIC coprocessor programming model does not provide a queue system to manage asynchronous kernel launchings. We have developed a queue system for the asynchronous execution of several kernel launches on the MIC coprocessor, currently using a FIFO policy in our prototype. When a MIC controller object is created, an asynchronous `OpenMP task` is launched. This task uses OpenMP locks to block until there are kernel-launching requests in the queue. Then, it dequeues the request and dispatches/executes it. The execution of a task on the MIC is carried out by simply executing the already offloaded parallel `wrapper_xphi_##name` generated function, specified in the request structure, that contains pointers to the function and parameters. The Controller destructor enqueues a special request that notifies to the OpenMP queue-controlling task that it should release the Controller resources and finish.

5 Experimental Study

We perform an experimental study to evaluate the potential advantages and constraints of the integration of the MIC coprocessor in the original Controller

library. The section includes: (1) A description of the considered study cases, (2) a performance study of our proposal, and (3) a development effort comparison between programming using the new Controller extension and using device vendor programming models.

5.1 Study Cases

We select four benchmarks to test our approach and implementation.

Matrix Addition. It implements a sum of two matrices, storing the result in a third one: $C = A + B$. For the Controller version, we use the same generic kernel implementation tested in previous works for CPU-cores and GPUs, without any modification.

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 original program, obtained from the CUDA Toolkit Samples, independently applies the formula to the input values of an array, calculating and storing their results. Again, the Controller version uses the same generic kernel definition for both GPUs, and MICs accelerators.

Matrix Multiplication. It computes the product of two matrices, storing the result in a third one: $C = A * B$. The read patterns on A and B matrices should be adapted to exploit coalescence and shared memory in GPUs, and to properly exploit caches and vectorization on MICs. These features lead to different optimizations in both types of accelerators. Thus, the Controller version declares different specialized and optimized kernels for each kind of device.

Mandelbrot Algorithm. The Mandelbrot algorithm is used to compute fractal geometric images. The Controller version uses a single generic kernel definition for both GPUs and MICs accelerators.

Table 1. Performance results (seconds) comparing LEO reference codes with Controller codes for different input sizes (left/right). Experiments executed on a Intel Xeon E5-2620 v2 @2.1 GHz, 32 Gb DDR3 main memory, and with the Xeon Phi Knights Corner 3120A coprocessor. Compiler used: ICC 17.0.0 version with the flags -*O3*, and -*openmp*.

Code	Mat. Add.	Black-Scholes	Mat. Mult.	Mandel-brot
Size	5000^2	10^6	4096^2	4000^2
LEO Code	1.67	0.60	2.59	6.49
Ctrl. Code	1.43	0.74	2.88	6.86

Code	Mat. Add.	Black-Scholes	Mat. Mult.	Mandel-brot
Size	20000^2	$5 * 10^7$	8192^2	20000^2
LEO Code	24.99	5.49	19.87	148.47
Ctrl. Code	24.65	5.01	19.27	147.36

Table 2. Measurements of development effort metrics for the codes of the study cases. Left: comparison of number of code lines, code tokens, and cyclomatic complexity between the Controller version and the version using native programming models. Right: 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 MIC versions using the native models, or the Controller library.

Case study	Version	Lines of Code	#Tokens	Cyclomatic Complexity
Matrix addition	LEO	26	210	3
	Ctrl.MIC	35	317	1
Black Scholes	LEO	80	525	6
	Ctrl.MIC	89	693	5
Matrix mult.	LEO	23	217	4
	Ctrl.MIC	37	337	3
Mandelbrot	LEO	32	319	5
	Ctrl.MIC	46	488	4

Case study	CUDA → LEO	Ctrl.GPUs → Ctrl.MICs
Matrix Addition	Common 13% Delete 30% Change 57%	Common 92% Delete 0% Change 8%
Black-Scholes	Common 53% Delete 25% Change 22%	Common 69% Delete 21% Change 10%
Matrix multiplication	Common 8% Delete 43% Change 48%	Common 49% Delete 3% Change 47%
Mandelbrot	Common 32% Delete 61% Change 7%	Common 97% Delete 0% Change 3%

5.2 Performance Study

In this section we show how low is the performance overhead produced by the implementation of our proposed MIC library extension. Table 1 shows the total times spent (including computation and data transfers) by the four benchmarks with two different problem sizes. Codes have been implemented with our proposal, and directly with the Intel Language Extensions for Offload (LEO) and OpenMP. A similar comparison for groups of CPU-cores and GPUs were presented in [11]. Both studies indicate only a small constant penalty performance due to the management of the queue system, that is only noticeable in the results for the smaller problem sizes presented on the left of Table 1. For bigger problem sizes, some performance gain is obtained due to Hitmap optimizations in the internal management of the data structures. In general terms, the performance obtained by using our approach is similar to the native programming models.

5.3 Development Effort Measures

This section includes two development effort comparisons. First, between the proposed Controller implementation and the reference codes (using LEO and OpenMP for MIC, and CUDA for GPUs). Secondly, comparing measures of the code changes needed to port a GPU implementation to a MIC implementation, using the Controller or the native programming models.

The results of the first comparison are presented on the left of Table 2. We measure three classical development effort metrics: Number of lines of code; Number of tokens, and McCabe's cyclomatic complexity [10]. The measured codes include kernel definitions, kernel characterizations, the coordination host code, and data structures management. We observe that the use of the Controller library implies less cyclomatic complexity, but more number of lines and tokens.

However, the goal of the library is to provide an homogeneous interface to deal with any kind of accelerator. For this reason, we also compare the effort needed for transforming GPU codes in order to port them to a MIC device. See results on the right of Table 2. We analyze the percentage of words of each implementation that are common and can be reused, should be deleted, or should be changed. The largest changes are on the matrix multiplication benchmark, because of the implementation of different optimized kernels for each device. For the other benchmarks, we see that using our proposal the programming effort needed to change the target computational device is extremely low. These measures show the level of abstraction and standardization achieved by our proposal.

6 Conclusions

In this paper we propose an extension to support the Intel Xeon Phi (MIC) coprocessors in a CPU-GPU homogeneous programming model for heterogeneous systems, that is implemented as a compiler agnostic library. To provide support for MIC coprocessors, our approach reuses and mixes the internal execution features for CPU-cores, and the internal memory and communication management features of the original GPU model. We have completely integrated the support for a MIC coprocessor in the library, without adding any constraint to the programming model. The experimental study shows the high flexibility of our approach, that implies a minimum programming effort for changing the execution target devices, without significatively penalizing the performance. Future work includes the integration of scientific libraries, such as MKL, as kernels in the Controller implementation, and an evaluation with applications of other domains.

Acknowledgments. This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H6 (TIN2016-81840-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

References

1. Contassot-Vivier, S., Vialle, S.: Algorithmic scheme for hybrid computing with CPU, Xeon-Phi/MIC and GPU devices on a single machine. Parallel Comput.: Road Exascale **27**, 25–34 (2016)
2. Deepika, H., Mangala, N., Babu, S.C.: Automatic program generation for heterogeneous architectures. In: 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 102–109. IEEE (2016)
3. Diogo, M., Grelck, C.: Towards heterogeneous computing without heterogeneous programming. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 279–294. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40447-4_18
4. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: a hybrid multi-core parallel programming environment. In: Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), vol. 28 (2007)

5. Dongarra, J., Gates, M., Haidar, A., Jia, Y., Kabir, K., Luszczek, P., Tomov, S.: HPC programming on Intel many-integrated-core hardware with Magma port to Xeon Phi. Sci. Program. **2015**(9), 1–11 (2015)
6. 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)
7. Grasso, I., Pellegrini, S., Cosenza, B., Fahringer, T.: A uniform approach for programming distributed heterogeneous computing systems. J. Parallel Distrib. Comput. **74**(12), 3228–3239 (2014)
8. Hijma, P., Jacobs, C.J., van Nieuwpoort, R.V., Bal, H.E.: Cashmere: heterogeneous many-core computing. In: 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 135–145. IEEE (2015)
9. Karimi, K., Dickson, N.G., Hamze, F.: A performance comparison of CUDA and OpenCL. arXiv preprint (2010). arXiv:1005.2581
10. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)
11. Moreton-Fernandez, A., Ortega-Arranz, H., Gonzalez-Escribano, A.: Controllers: an abstraction to ease the use of hardware accelerators. Int. J. High Perform. Comput. Appl. (2017). http://dx.doi.org/10.1177/1094342017702962
12. NESUS, Network for Sustainable Ultrascale Computing (Cost Action IC1305): A roadmap for research in sustainable ultrascale systems, October 2016
13. NVIDIA Corporation: Cublas library. NVIDIA Corporation, Santa Clara, California, vol. 15, no. 27 (2008)
14. 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)
15. Riebler, H., Vaz, G., Plessl, C., Trainiti, E.M., Durelli, G.C., Del Sozzo, E., Santambrogio, M.D., Bolchini, C.: Using just-in-time code generation for transparent resource management in heterogeneous systems. In: 2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI), pp. 1–5. IEEE (2016)
16. 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)
17. TOP500.org: Top500 supercomputing sites, January 2017. http://www.top500.org/
18. Viñas, M., Fraguela, B.B., Andrade, D., Doallo, R.: Towards a high level approach for the programming of heterogeneous clusters. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW), pp. 106–114. IEEE (2016)
19. Wienke, S., Springer, P., Terboven, C., Mey, D.: OpenACC—first experiences with real-world applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32820-6_85
20. Wu, S., Dong, X., Chen, H., Dang, B.: OCLS: a simplified high-level abstraction based framework for heterogeneous systems. In: Park, J., Yi, G., Jeong, Y.S., Shen, H. (eds.) Advances in Parallel and Distributed Computing and Ubiquitous Services. LNEE, vol. 368, pp. 57–65. Springer, Singapore (2016). doi:10.1007/978-981-10-0068-3_7