

Supporting the Xeon Phi coprocessor in a Heterogeneous Programming Model

Ana Moreton-Fernandez, Eduardo Rodriguez-Gutierrez, Arturo Gonzalez-Escribano, and Diego R. Llanos

Departamento de Informática, Edif. Tecn. de la Información, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain.

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, for exploiting efficiently each computational unit.

In this paper, we present an extension of a heterogeneous programming model, in order to also support Intel Xeon Phi coprocessors. This contribution extends an existing heterogeneous programming library, by taking advantage of both the GPU communication model and the CPU execution model of the original library. Our experimental results show that using our approach, the programming effort needed for changing the target devices is highly reduced, by for example reusing the 97% of the code between a GPU implementation and a Xeon Phi implementation for the Mandelbrot benchmark.

1 Introduction and Background

Supporting computational accelerators such as GPUs or Xeon Phi coprocessors in the state-of-the-art programming models is vital to exploit current Supercomputers. They are composed by several machines with different computation capabilities and different kinds and families of accelerators, as we observe for example in the configuration of the TOP500 supercomputers [16].

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 to exploit efficiently such execution systems [2].

Many works address the problem of heterogeneous systems management (e.g. [5, 14, 4]) following two alternatives: generating specific codes, or using runtime libraries. Using current heterogeneous code generators, the code should be recompiled for each different execution platform in order to better exploit the performance capabilities of the system. As for libraries, some works, for specific kind of applications, address the portability problem using native programming

models. For example, MAGMA library [6] provides a unified programming environment for heterogeneous systems using both CPUs and accelerators, such as a GPU or Intel Xeon Phi, for dense linear algebra algorithms. However, most of heterogeneous libraries rely on the OpenCL abstraction. OpenCL [15] is a widespread programming framework to deal with heterogeneous devices. The OpenCL context abstraction allows the management of multiple devices of the same nature (using the same *platform* in OpenCL notation). The abstractions introduced by OpenCL have been proved that prevent the obtaining of the same efficiency as when using directly the vendor programming models, for several common situations [10]. As a result, many state-of-the-art heterogeneous frameworks and libraries of higher level of abstraction [9, 13, 8, 18, 3, 17], that rely on OpenCL as execution layer, typically inherit some of these problems.

Additionally, during the last decade several high-performance accelerator pre-defined libraries, such as cuBLAS [12] or MKL, have been developed using the vendor specific programming models. For making the most of such works, it is recommendable the use of such native or vendor specific programming models and compilers for each different kind of device.

In this paper, we extend a heterogeneous programming model that is not based on the use of OpenCL. The original proposal, named *Controller* [1], is a vendor-specific, compiler-agnostic library that 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, known also under the name of Many Integrated Cores (MICs). The model is based on the mix of the GPU communication model with the CPU execution model of the Controller library. We develop a complete runtime execution system that includes methods for task launching, data transfers between the MIC accelerator and the host, and a queue system to manage the kernel executions. It perfectly fits with the previous Controller library, thus standardizing and abstracting to the programmer the issues related to the different accelerator programming.

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 target devices. Moreover, the performance results show that our implementation does not introduce significant performance penalties compared with reference codes.

2 Proposal: MIC Controller model

The work presented in [1] proposed a simple heterogeneous programming model to deal with the hybrid-computation-related issues in an abstract way to the programmer. The model defines an object able to manage either a group of CPU cores or a GPU accelerator (see left of Fig. 1), using internally native programming techniques (OpenMP and CUDA respectively).

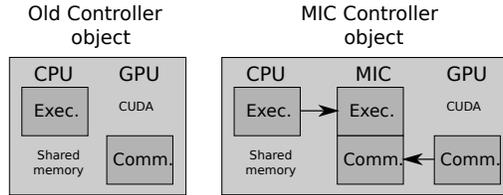


Fig. 1. Left: Previous controller model, only supporting CPU and GPU. Right: The MIC proposed model, mixing the GPU-CPU model features.

In our proposal, we distinguish two parts in the internal model, supporting each kind of computational device: Execution and communication models. As the CPU-core group devices share the memory space with the host, the controller object only has to provide an execution model. On the other hand, for GPUs, CUDA programming model provides a execution system to enqueue and launch kernels with the proper granularity level that is used in the Controller model. The GPU implementation of the execution model is straightforward, but it provides an internal mechanism to implement policies and techniques for dealing memory communication operations across different memory spaces on host and accelerators (see left of Fig. 1).

In this work we propose to integrate MIC coprocessors in the Controller model distinguishing two aspects: The execution model and the memory model. In terms of execution, the Controller model proposed for groups of CPU core, that blends blocks of fine grained kernels into coarse CPU tasks, is appropriated for Xeon Phi coprocessors. In terms of memory management, the abstract model for data communications needed for the MIC coprocessors is equivalent to the Controller model for the GPU. Thus, supporting new kind of accelerators, as the Xeon Phi, can be done by combining existing models for execution, and memory communications across different spaces, independently.

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

3 Background: Controller model

In this section we describe the Controller model [1], the approach upon we build our MIC execution system. We also describe Hitmap [7], the library used for the data-structures managing.

```

1  /* Generic kernel codes for any 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 threads;
24     ThreadInit( threads, 2, SIZE, SIZE );
25     CntrlLaunch(comm, MatAdd, threads, 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.

3.1 Hitmap library

Hitmap is the library chosen to manage the partition and mapping of data structures. It is used in the Controller model to manage the data distribution across devices, and to provide a common interface to implement data management inside generic portable kernels.

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. There are only three functions of Hitmap needed to work with the Controller library. The function *hit_tileDomainAlloc* is used to declare and allocate the index domains of a tile array. 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 implementation.

3.2 Controller model

The Controller model provides a structured programming methodology together with several important features: (1) A mechanism to define common kernels

reusable across different types of devices, or 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.

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 a specific value for a specialized code for a given type of device. Currently, the library supports the specific declarations `KERNEL.GPU` for CUDA code targeting NVIDIA's GPUs, `KERNEL.CPU` for host machine code targeting sets of CPU cores, and `KERNEL.GPU.WRAPPER`, `KERNEL.CPU.WRAPPER`, for host machine code which includes calls to specialized GPU or CPU libraries, such as cuBLAS or MKL routines.

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. We see a kernel definition in lines 2-7 of Fig. 2.

Controller programming methodology: A Controller host program follows simple development guidelines:

- The Controller entity creation, assigning the computational device to manage by this controller object. A controller entity should be created for each computational device that will be used for computation.
- The attachment of the data structures to the controller object. Data structures that will be accessed by a kernel should be also previously attached to the controller entity.
- The launching of the computational kernels on the controller object.
- The detachment of the data structures.

Figure 2 shows a matrix addition Controller implementation that performs the computation on a GPU. 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 of 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 internal method in the programming model that allows also the efficient execution of this program on the Xeon Phi, only by changing the `CNTRL.GPU` modifier by `CNTRL.XPHI` on the line 13 of the code.

<pre> 1 /* Internal attach function */ 2 void attachToXPHI(CntrlXPHI* cntrl, 3 HitTile *tile){ 4 Lock(tile, cntrl); 5 int MIC= cntrl->MIC; 6 float *data = (float*)(*tile).data; 7 int numElems = hit_NumElem(tile); 8 #pragma offload target(mic:MIC) \ 9 in(data:length(numElems)) \ 10 alloc_if(1) \ 11 free_if(0) 12 13 } </pre>	<pre> 1 /* Internal detach function */ 2 void detachToXPHI(CntrlXPHI* cntrl, 3 HitTile *tile){ 4 int MIC= cntrl->MIC; 5 float *data = (float*)(*tile).data; 6 int numElems = hit_NumElem(tile); 7 #pragma offload target(mic:MIC) \ 8 in(data:length(0)) alloc_if(0) \ 9 free_if(0) \ 10 out(data:length(numElems)) \ 11 alloc_if(0) \ 12 free_if(1) 13 Unlock(tile, cntrl); 14 } </pre>
---	---

Fig. 3. Internal codes that perform data transfers. Left: Internal code to transfer the data of a HitTile object to a MIC coprocessor from a host. Right: Internal code to transfer the data of a HitTile object from a MIC coprocessor to a host.

4 Integrating Xeon Phi coprocessor in the Controller programming library

A previous version of the Controller library supports the deployment of kernels on GPUs or computational devices formed by groups of CPU-cores. In this section we present the implementation support of the MIC devices in the Controller library. We implement a **MIC controller object** containing several functionalities such as: an internal queue to manage the asynchronous task executions, a variable to store the MIC identifier that the controller object is managing, or a method to lock and unlock data structures.

4.1 Attaching and detaching data structures to MIC Controller object

Attaching a data structure to a Controller object implies to lock the data structure until that structure is detached. In computational devices such as GPUs or MIC coprocessors, where their memory spaces are separated to the host memory space, the attachment/detachment operation also implies a data transfer.

We have implemented in this extension library two internal functions to perform the data transfers to/from the MIC coprocessor, using the Intel Language Extensions for Offload (LEO). The functions are executed internally when an attachment or detachment operation is called respectively. Figure 3 shows 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. Secondly, 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 elements 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 from a MIC controller object. As in the attachment, first the code extracts the information about the data transfer (lines 4 to 6). Secondly, the actual data transfer from the coprocessor to the host is defined 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 defines the data transfer and the freeing of the MIC space memory (lines 10 to 12). Finally, the data structure is unlocked.

4.2 Kernel definitions

A kernel definition specifies the device that fits with its implementation by using the primitive `KERNEL_<type>`. We have developed a framework to support MIC kernel definitions in the Controller library. A MIC kernel definition is transformed in three functions through macros. We show the code of the three functions in Fig. 5.

Single-element function: The first function implements the kernel that the programmer has defined for computing each element. The function is named *kernel.xphi_##name*, where the *##name* element is the first parameter of the kernel definition. It is defined as a MIC function using the attribute *target(mic)*. The parameters are a set of indexes represented by a *Thread* object, that represent 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. It receives a variable number of parameters. The first one is the controller object, the second one the domain where computation should be performed and the rest are the data structures needed for computation. Lines 10 to 12 of Fig. 5 show how the information is extracted from the parameters (auxiliary macros for the transformations are defined in Fig. 4). The next 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 transferred data (line 15), and the parallel computation is performed on the specified domain (lines 16 to 28).

Task addition function: The third one is named *name##_xphi*. It is the internal implementation of a kernel launch. In its body, the function implements the task addition to the internal controller queue. The information needed for the addition is: The controller object, the pointer of the coarse-grain parallel

```

1  /* Auxiliary 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     float *data_tile1=(float*) (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 kernel parameter.

```

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 Xeon Phi kernel definition : 1) Function to apply to each element: *kernel_xphi_##name*; 2) Function that executes an enqueued kernel in parallel: *wrapper_xphi_##name*; 3) Function to add a task to the Controller queue: *name##_xphi*.

computational function, and its execution 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 Execution model: 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 FIFO queue system for the asynchronous execution of several kernel launches on the MIC coprocessor.

When a MIC controller object is created, an asynchronous *omp task* is launched. The program of this *omp task* is checking the possible new task queue additions. When there is a task in the queue, the controller dispatches/executes it, and restarts the checking again. The checking is implemented using *omp locks* avoiding thus active waits. The execution of a task on the MIC is carried out simply by the execution of the already offloaded parallel *wrapper_xphi_##name* generated function, that is associated with the task that is being executed. The function pointer, and its execution parameters are determined in the kernel launching.

The last task added is the controller destruction. It stops the checking, finishes the *omp task* and, destroys the controller.

5 Experimental study

We perform an experimental study to evaluate the potential advantages and constraints of the integration of the MIC coprocessor in a homogeneous library for CPU-GPU heterogeneous systems. The section consists of: (1) a description of the considered study cases, (2) a performance study of our proposal, and (3) a comparison of the development effort needed between programming using the new library extension or using device vendor programming models.

5.1 Study cases

We select four benchmarks to test the extension proposed in this work.

Matrix addition The Matrix addition consists of the sum of two different matrices, storing the result in a third one: $C = A + B$. Our MIC implementation for this problem is similar to the GPU version. Only one generic kernel is defined by the programmer.

Black-Sholes The Black-Scholes formula is based on a mathematical model of a financial market. The result estimates the price of European-style options. The program, obtained from the CUDA Toolkit Samples, independently applies the formula to the input values of an array, calculating and storing their results. In our implementation, the same generic kernel definition is used for both GPUs and MICs accelerators.

Code	Mat. Add.	Black-Scholes	Mat. Mult.	Mandelbrot	Code	Mat. Add.	Black-Scholes	Mat. Mult.	Mandelbrot
Size	5000 ²	10 ⁶	4096 ²	4000 ²	Size	20000 ²	5 * 10 ⁷	8192 ²	20000 ²
LEO Code	1.67	0.60	2.59	6.49	LEO Code	24.99	5.49	19.87	148.47
Ctrl. Code	1.43	0.74	2.88	6.86	Ctrl. Code	24.65	5.01	19.27	147.36

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

Case study	Version	Lines of Code	#Tokens	Cyclomatic Complexity	Case study	CUDA → Offloading	Ctrl.GPUs → Ctrl.MICs
Matrix addition	LEO	26	210	3	Matrix Addition	Common 13%	Common 92%
	Ctrl.MIC	35	317	1		Delete 30%	Delete 0%
Black Scholes	LEO	80	525	6	Black-Scholes	Change 57%	Change 8%
	Ctrl.MIC	89	693	5		Common 53%	Common 69%
Matrix mult.	LEO	23	217	4	Matrix multiplication	Delete 25%	Delete 21%
	Ctrl.MIC	37	337	3		Change 22%	Change 10%
Mandelbrot	LEO	32	319	5	Mandelbrot	Common 8%	Common 49%
	Ctrl.MIC	46	488	4		Delete 43%	Delete 3%
						Change 48%	Change 47%
						Common 32%	Common 97%
						Delete 61%	Delete 0%
						Change 7%	Change 3%

Table 2. Left: Measurements of the development effort metrics for the codes of the study cases. 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.

Matrix multiplication The Matrix multiplication computes the product of two different 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 in GPUs, and properly exploit caches and vectorization on MICs. These features lead to interesting optimizations in both accelerators. Our implementation declares different specialized and optimized kernels for each kind of device.

Mandelbrot algorithm The Mandelbrot algorithm is used to compute, create, and display fractal geometric images. In our implementation, the same generic kernel definition is used for both GPUs and MICs accelerators.

5.2 Performance study

In this section we show the low performance overhead produced by the MIC library extension implementation of our proposal, using the four study cases previously described.

Table 1 shows the times spent (including computation and data transfers) for the four benchmarks with two different problem sizes. Codes have been implemented directly by the Intel Language Extensions for Offload (LEO), using OpenMP internally, and by our proposal. Results for groups of CPU-cores and

GPUs were presented in [1], indicating a penalty performance up to 0.4 seconds. We can observe on Tab. 1 that the execution time overhead derived from using our MIC proposal is also up to 0.4 seconds, that is the time spent by the queue system. For bigger problem sizes, some performance gain is obtained compared with the reference codes due to Hitmap optimizations in the internal management of the data structures. But, 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 between the proposed implementation of the MIC Controller library and references codes. For the Xeon Phi reference codes we use LEO, using OpenMP internally, and CUDA for the GPU reference codes.

In the first comparison, found on the left of Tab. 2, we apply three classical development effort metrics: the number of lines of code, the number of tokens, and the McCabe's cyclomatic complexity [11]. The metrics are applied to the parts of code that include: kernel definitions, kernel characterizations, the coordination host code in the main program, and data structures management. We observe that the use of the Controller library implies less cyclomatic complexity, but more number of lines and tokens in the implementation.

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 on the right table of Tab. 2 the effort needed to transform a code in order to be executed in a different device. 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 demonstrate the abstraction and standardization capacity of our proposed library implementation.

6 Conclusions

In this paper we propose an extension to support the Intel Xeon Phi coprocessors in a homogeneous programming model for CPU-GPU heterogeneous systems, that is implemented as an agnostic library. 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 penalizing the performance.

Acknowledgment

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. Alonso-Mayo, A., Ortega-Arranz, H., Gonzalez-Escribano, A.: Communicators: An abstraction to ease the use of accelerators. In: High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (Jan 2016)
2. Contassot-Vivier, S., Vialle, S.: Algorithmic scheme for hybrid computing with cpu, xeon-phi/mic and gpu devices on a single machine. *Parallel Computing: On the Road to Exascale* 27, 25 (2016)
3. Deepika, H., Mangala, N., Babu, S.C.: Automatic program generation for heterogeneous architectures. In: *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*. pp. 102–109. IEEE (2016)
4. Diogo, M., Grelck, C.: Towards heterogeneous computing without heterogeneous programming. In: *International Symposium on Trends in Functional Programming*. pp. 279–294. Springer (2012)
5. 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)
6. 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. *Scientific Programming* 2015, 9 (2015)
7. Gonzalez-Escribano, A., Torres, Y., Fresno, J., Llanos, D.R.: An extensible system for multilevel automatic data partition and mapping. *IEEE Transactions on Parallel and Distributed Systems* 25(5), 1145–1154 (2014)
8. Grasso, I., Pellegrini, S., Cosenza, B., Fahringer, T.: A uniform approach for programming distributed heterogeneous computing systems. *Journal of parallel and distributed computing* 74(12), 3228–3239 (2014)
9. Hijma, P., Jacobs, C.J., van Nieuwpoort, R.V., Bal, H.E.: Cashmere: Heterogeneous many-core computing. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. pp. 135–145. IEEE (2015)
10. Karimi, K., Dickson, N.G., Hamze, F.: A Performance Comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581* (2010)
11. McCabe, T.J.: A complexity measure. *Software Engineering, IEEE Transactions on* (4), 308–320 (1976)
12. Nvidia, C.: *Cublas library*. NVIDIA Corporation, Santa Clara, California 15, 27 (2008)
13. 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)
14. 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: *Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2016 IEEE 2nd International Forum on*. pp. 1–5. IEEE (2016)

15. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12(1-3), 66–73 (2010)
16. TOP500.org: Top500 supercomputing sites (Jan 2017), on <http://www.top500.org/>
17. Viñas, M., Fraguera, B.B., Andrade, D., Doallo, R.: Towards a high level approach for the programming of heterogeneous clusters. In: *Parallel Processing Workshops (ICPPW), 2016 45th International Conference on*. pp. 106–114. IEEE (2016)
18. Wu, S., Dong, X., Chen, H., Dang, B.: Ocls: A simplified high-level abstraction based framework for heterogeneous systems. In: *Advances in Parallel and Distributed Computing and Ubiquitous Services*. pp. 57–65. Springer (2016)