

Communicators: An abstraction to ease the use of hardware accelerators

Alejandro Alonso-Mayo

Universidad de Valladolid
alejandro.alonso.mayo@alumnos.uva.es

Hector Ortega-Arranz

Dept. Informática
Universidad de Valladolid
hector@infor.uva.es

Arturo Gonzalez-Escribano

Dept. Informática
Universidad de Valladolid
arturo@infor.uva.es

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 in each moment at the different computing platforms considering architectural details.

We introduce the *communicator* concept as an abstract entity that allows the programmer to easily manage the communications and kernel launching details on hardware accelerators or multi-core devices in a transparent way. Furthermore, this model also gives the possibility to the programmer of launching CPU kernels in the multi-core processors with the same abstraction and methodology used for the accelerators. In this way, the burden of coding two different codes for managing the different computational devices is alleviated.

Additionally, this entity allows the programmer to simplify the proper selection of values for kernel-launching configuration parameters. This is done through a simple characterization process of the kernel code to be executed. A programming model involving the communicator entity is described in this article.

Finally, we also present a prototype library that implements the communicator model, together with its application in several study cases. Its use has led to reductions in the development costs with significantly low overheads in the execution times when compared to manually programmed and optimized solutions using CUDA and OpenMP directly.

Categories and Subject Descriptors D.3.2 Programming Languages [*Language Classifications*]: Concurrent, distributed, and parallel languages

General Terms Parallel programming, Software

Keywords Communicators, CUDA, GPUs, Kernel characterization, Memory transfers, Optimizations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm...\$15.00.
<http://dx.doi.org/10.1145/nnnnnnn.nnnnnn>

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 top 500 list of current supercomputers [15].

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 [1]; or (2) Use a combination of different programming models specific for each kind of computational devices, e.g. MPI [9] together with CUDA [8] or OpenCL [13], and OpenMP [3], as in the works [2, 7].

One of the main drawbacks of the first approach is the difficult to represent irregular programs with non-trivial communications or synchronizations. Besides, the final generated code resulting from programming with this kind of models is not usually optimized as the original code. Moreover, the possible optimizations a programmer can obtain, by selecting proper values for kernel-launching configuration parameters, are not considered in the OpenAcc standard.

On the other hand, implementing solutions following the second approach requires the programmer to have a deeper knowledge of the different parallel programming models involved. Additionally, she is the final responsible of properly managing the data transfer among the different memory spaces of the computational devices, at the most appropriate times, and to choose proper values for kernel-launching configuration parameters. However, this gives to the programmer the possibility to optimize the use of the particular resources of each specific device.

There are many libraries of specific functions for particular fields, or devices, that include small abstractions to ease the management between the accelerator and the host (e.g. MCUDA [14] or hiCUDA [6]), but without considering guided optimizations.

In this article is presented the concept of *communicator* 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. This entity uses the most appropriate programming models (CUDA, OpenMP, ...) to control the computational resources of the accelerators and hosts machines. The communicator model includes two features: (1) An optimization system to select proper values for kernel-launching configuration parameters, guided by simple code characterizations provided by the programmer; (2) a transparent mechanism of memory management, including optimized communications of the data structures between the host and the corresponding images in the accelerators; and (3) an abstraction for indexed data structures that allows to codify simple kernels that can be used for different kinds of devices

(such as GPUs, or multi-threaded vectorial CPU cores) with none or minimal changes.

A prototype that implements the concept of this entity is also described in this article. This prototype is designed for exploiting NVIDIA GPUs using the CUDA parallel programming model, or the multi-core CPUs using OpenMP. Together with the description of the prototype, it is presented an experimental study based on three study cases (matrix addition, matrix multiplication, and a simple Jacobi stencil program). The study shows how the use of the prototype implies a reduction of the programming effort needed when writing these applications, compared to creating them using the specific parallel programming models. Besides, the use of the prototype does not introduce significant overheads.

The structure of the article is as follows. Section 2 presents the communicator model. Section 3 explains the interface of our library, and its usage to develop a program. Section 4 poses the experimental setup which results are presented in Sect. 5. Finally, Sect. 6 describes the conclusions of this work and the directions that can be addressed in future iterations.

2. Communicator Model

The communicator model introduces a simplified way to program applications that can exploit heterogeneous computational platforms including accelerators or/and multi-core CPUs. *Communicators* coordinate the execution of series of kernels. These kernels are specific functions, that are managed by the communicator entities. Communicators automatically manage the:

- Kernel launching, that is the deployment/execution of sequences of function kernels in the computational platform. The communicators can include policies to exploit concurrent kernel techniques, interleave computations with communications, or reorder the sequence of kernels.
- Configuration, that is the layout of certain computational platform variables and configuration parameters in order to obtain a particular execution behavior.
- Communications, that are the data transfers carried out among the memories of the host and the accelerators. Note that this offloading is only required when the model launches executions in accelerator devices.

2.1 Kernel launching

Kernel launching is an operation that enqueues in a communicator the order to execute a kernel, with given real parameters, when the associated device (accelerator or CPU cores) is available. This is done through a launching primitive. The communicator internally ensures that the input data has been transferred and previous kernels have ended before proceeding to do the real launch. The communicator 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.

2.2 Configuration of kernels for execution

Our model considers two main kinds of information that are needed at the communicator internals to choose proper kernel-launching parameters, and manage the device memory:

- (a) Kernel code characterization in terms of parameters related to global memory access patterns, computational load ratio, and data sharing across block ratio.
- (b) The input/output *role* of the kernel parameters. With this information the communicator can control which data transfers among different memories are needed.

2.3 Communications

Accelerators may have their own memory spaces, forcing to transfer the data to be processed, and the obtained results, between the memory of the host platform and the memory of the device accelerator. The manual management of this transferring is 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 kernel launching. A communicator is associated, in the moment of its creation, with a particular accelerator, and it transparently manages the variable images of the memory space of the device.

The communicator can decide when and how these transfers should be carried out, depending on their use inside their corresponding kernels and the kernels enqueued for launching. The model also allows the programmer to use the original names of the variables instead of defining their corresponding images in the accelerator device.

Depending on the features of the used variables inside these contexts we can distinguish two types: Binded variables and internal variables.

Binded variables

Binded variables are host variables that have an image in the memory space of the accelerator. The model allows to bind on a host variable to the communicator. Since then, it becomes binded, and its data should not be modified by the program at the host side until an unbinding operation is applied on it.

The first kernel requiring the use of a binded variable as an input (*IN role*) will force the communicator to transparently ensure that its data have been transferred. Applying an unbinding operation to a binded variable will force the transfer of its data from the accelerator to the host if it is an output variable (*OUT role*). The main program waits for the end of the kernels using the variable and transmission of the data.

Internal variables

Internal variables are variables whose scope is delimited to the code 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, the data are not going to be transferred to the host memory.

They are created in the communicator through an operation applied to the host variable. The programmer declares a data structure without allocating it in the host. This is done just to clone the type, size, and structure in the communicator memory space. Since this moment, the image of this variable could be used by the kernels launched in this communicator. To destroy an internal variable it is needed to apply another operation using the reference name of the host variable.

3. Communicator Library Interface

We have designed a library implementation of the communicator model. In our implementation, a kernel is a function coded for a computational device with particular input/output parameters. The communicator library interface defines primitives to:

```

1  /* Communicator creation. */
2  Communicator comGPU, comCPU;
3  CommCreate(&comGPU, COMM_GPU, 2);
4  CommCreate(&comCPU, COMM_CPU);
5
6  /* Tile declaration and allocation. */
7  HitTile_float matrixHost;
8  HitTile_float matrixInternal;
9  hit_tileDomainAlloc(&matrixHost, float, \
10                     2, rows, columns);
11 hit_tileDomain(&matrixInternal, float, \
12                2, rows, columns);
13
14 /* Tile initialization. */
15 CommAttach(&comGPU, &matrixHost);
16 CommInternalCreate(&comGPU, &matrixInternal);
17 CommLaunch(&comGPU, copyCell, t_config, \
18            2, matrixInternal, matrixHost);
19 CommLaunch(&comGPU, updateCell, t_config, \
20            2, matrixHost, matrixInternal);
21
22 /* User code
23  * (not modifying binded variables) */
24 ...
25
26 /* Unbinding and freeing resources. */
27 CommDetach(&comGPU, &matrixHost);
28 CommInternalDestroy(&comGPU, &matrixInternal);
29 CommDestroy(&comGPU);
30 CommDestroy(&comCPU);

```

Figure 1. Example of main code using the communicators primitives.

- A) Create of communicators,
- B) Declare the kernels,
- C) Characterize the kernels,
- C) Declare data structures for transparent memory management of different devices, and
- D) Launch the kernels.

3.1 Creation of communicators

A communicator is associated to a particular computational platform (accelerator or CPU-cores set) at the moment of its creation, in order to use the specific collection of functions related to the assigned device. This is done through the `CommCreate` function. This primitive has two main parameters: The name of the communicator variable, and the associated computing device. For heterogeneous systems hosting more than one accelerator of a kind it is needed the identification number. Lines 3 and 4 of Fig. 1 show examples of creating communicators associated to the third GPU of the system, and associated to the full set of CPU cores, respectively. After using it, the programmer can free its resources with the `CommDestroy` function (see Fig. 1, lines 29 and 30).

3.2 Declaration and configuration of kernels

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 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` and `KERNEL_CPU` for NVIDIA GPUs and sets of CPU cores.

```

1  /* Structured type definition:
2   * HitTile_float */
3  hit_tileNewType( float );
4
5
6  /* Kernel characterizations */
7
8  KERNEL_CHAR(copyCell,1,def,def,def)
9  KERNEL_CHAR(updateCell,1,full,low,high)
10
11
12 /* Kernel codes */
13
14 KERNEL(copyCell, 2, \
15         OUT, HitTile_float*, var_dst, \
16         IN, HitTile_float*, var_src){
17
18         /* code of copyCell kernel */
19 }
20
21 KERNEL(updateCell, 2, \
22         OUT, HitTile_float*, var_dst, \
23         IN, HitTile_float*, var_src){
24
25         /* code of updateCell kernel */
26 }

```

Figure 2. Examples of kernel characterizations (`KERNEL_CHAR` primitives), and role assignment of the kernel parameters (`KERNEL` primitives).

This primitive declares in brackets the number of parameters the kernel needs and a tuple of information for each parameter with:

- Its corresponding role:
 - IN: for input parameters,
 - OUT: for output parameters, and
 - IO: for input and output parameters;
- The type of the variable; and
- The name of the variable.

This configuration allows the communicators to determine if it is necessary to carry out data transfers with the main memory of the host, for the case of accelerator devices with separated memory spaces.

The primitive is followed by a structured block with the kernel code. Thus, it resembles a C function header. Lines 14 and 21 of Fig. 2 show some examples of the use of this primitive.

3.3 Kernel characterization

The kernel characterization is a process that obtains parameterized values for special code features following a particular model. The prototype library uses the model presented in the works of [11, 16]. This characterization allows the communicator to automatically decide which are proper values for the GPU configuration parameters (grid, threadblock and L1 cache memory sizes), and CPU threads granularity, in order to improve performance.

The primitive `KERNEL_CHAR`, taken from [12], is used to provide to the communicator the characterization of the kernels following the cited model. The corresponding parameters of the primitive are the following:

- a) Kernel name,
- b) Number of dimensions of the threads set for the kernel. The values can be 1, 2, or 3.
- c) Memory access pattern: Denotes the way the threads will access to memory. The values can be full (-coalesced), medium (-coalesced), or scatter.
- d) Computational load ratio: Number of arithmetic/logic operation vs. number of global memory accesses. The values can be high, medium, or low.
- e) Data sharing across blocks: ratio of data sharing compared to the number of memory accesses per thread. The values can be high, medium, or low.

For the case that the programmer is not able to perform the characterization of code parameters (c, d, and e), we have added the possibility to use the token *def*, that will assign a default configuration. The particular values for the default configuration, taken from the default configuration of [12], (256 threads per block, augmented L1 cache size for NVIDIA GPUs) are expected to properly work for the general case. Lines 8 and 9 of Fig. 2 show some examples of using this primitive, with the default configuration and with a particular kernel characterization, respectively.

3.4 Data structures

Regarding the data structures that the model handles, we have decided to use the *HitTile* structure, an abstract entity for arrays and tiles. This structure is defined in Hitmap [5], an efficient library for hierarchical tiling and mapping of arrays. The *HitTiles*, in the Hitmap library, are data structures similar to n-dimensional arrays that allow an advanced binding and partitioning, to create subtiles from another tile, to clone tiles, etc. When creating a *HitTile* variable, we can specify its domain by a selection of a subspace of array indexes, also known as *shape* (see Appendix A for a quick overview of the Hitmap library).

Hitmap also allows to perform communications between computing machines in an easy way through an abstract interface that internally uses a message passing paradigm (exploiting MPI). The access to the elements of these data structures is done using Hitmap access functions. They are designed for maximum efficiency, and are used transparently in the host, or in the accelerators, independently of the internal data layout [5]. Thus, it provides an homogeneous interface to manage data structures in both, host and accelerator device kernels.

Binding variables

In the proposed interface, the function *CommAttach* binds a tile defined in the code of the host with a communicator. After this binding the host should not manipulate the tile until it is unlinked by the function *CommDetach*.

The purpose of avoiding the host to modify the tile is to delegate the responsibility of keeping the data coherence between the host variable and its corresponding image on the accelerator to the communicator. In this way, the programmer can write programs without being the responsible of the transfers. For the particular case of communicators associated to CPU processors there is no need of transferring these data because they are already there. The communicator associated to the CPU processor is able to notice this fact and avoids to duplicate the data.

Both the attach and detach functions have two parameters: The communicator, and the pointer to the tile variable to be binded/unbinded. Lines 15 and 27 of Fig. 1 show examples of the creation of a link between a variable and a communicator, and the corresponding unbinding, respectively.

Creating internal variables

The function *CommInternalCreate* creates an internal device variable and associates it to a specific communicator. On the other hand, the function *CommInternalDestroy* is used to free the memory space in the assigned computational device.

Both functions receive two parameters: The communicator, and the pointer of a defined tile variable. For the case of communicators associated to accelerators, this kind of variables does not need to have allocated memory in the host side. The communicator will be the responsible of reserving the space in the memory of the corresponding computing device. Lines 16 and 28 of Fig. 1 show examples of the association of an internal variable with a communicator, and the corresponding liberation, respectively.

3.5 Kernel launching

The function *CommLaunch* is used to launch a kernel to the computational device of a communicator. The launched kernel will be enqueued, and eventually executed with the corresponding configuration derived from the information provided by the primitives previously explained in Sect. 3.2.

The function has the following parameters:

- The communicator,
- the name of the kernel,
- the index space of the thread set,
- the number of parameters required by the kernel, and
- the real parameters for the kernel execution.

These parameters should be variables associated to a communicator: Internal or binded. Lines 17 and 19 of Fig. 1 show examples of the launch of two different kernels.

4. Experimental Setup

This section describes the experiments we have carried out to check the functionality and evaluate potential performance issues introduced by the communicator prototype through the evaluation of the implemented prototype. We also evaluate the development effort of using the communicator prototype when compared to directly using common native programming models (CUDA or OpenMP).

4.1 Target architectures

The host machine used for the experimental evaluation is an Intel(R) Xeon E5-2620@2.1GHz, with 24 CPU cores and a global memory of 32GB DDR3 running a Centos 7 OS (Chimera). The hosted accelerator device is a NVIDIA GeForce GTX Titan Black, with a Kepler GK110B architecture. The experiments have been carried out using GCC 4.8.3, with the O3 flag, and the CUDA toolkit 7.5, with the corresponding architectural flag *arch=sm_35*. We use also a pure shared-memory machine (Heracles), a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, and 64 cores in total.

4.2 Case studies

We have programmed solutions for the following three problems: Matrix addition (MA), Matrix multiplication (MM), and the Jacobi PDE solver (JS). Figure 3 shows the implementation of the kernels used for the implemented solutions and Fig. 4 shows their characterization.

Matrix addition

The Matrix addition consists on the sum of two different matrices, that storing the result in a third one: $C = A + B$. The computation

```

1  /***** Matrix addition, same kernel for CPU and GPU *****/
2  KERNEL(k_ma, 3, IN, HitTile_int*, a, IN, HitTile_int*, b, OUT, HitTile_int*, c){
3      for(int k=0;k<100;k++){
4          hit_tileElemAt(*a, 2, thread.x,thread.y) += hit_tileElemAt(*a, 2, thread.x,thread.y) + \
5              hit_tileElemAt(*b, 2, thread.x,thread.y);
6      }
7  }
8
9  /***** Matrix multiplication, same kernels for CPU and GPU *****/
10
11 KERNEL_GPU(k_mm, 3, IN, HitTile_int*, a, IN, HitTile_int*, b, OUT, HitTile_int*, c){
12     for(int k = 0; k < hit_tileDimCard(*c, 0); k++){
13         hit_tileElemAt(*c, 2, thread.x, thread.y) += hit_tileElemAt(*a, 2, thread.x, k) * \
14             hit_tileElemAt(*b, 2, k, thread.y);
15     } }
16
17 /***** Jacobi PDE solver, same kernels for CPU and GPU *****/
18 KERNEL(k_copy, 2, OUT, HitTile_float*, dst, IN, HitTile_float*, src){
19     hit_tileElemAt(*dst, 2, thread.x, thread.y) = hit_tileElemAt(*src, 2, thread.x, thread.y);
20 }
21
22 KERNEL(k_update, 2, OUT, HitTile_float*, dst, IN, HitTile_float*, src){
23     hit_tileElemAt(*dst,2,thread.x+1,thread.y+1) = (hit_tileElemAt(*src,2,thread.x+1,thread.y) + \
24         hit_tileElemAt(*src,2,thread.x+1,thread.y+2) + \
25         hit_tileElemAt(*src,2,thread.x,thread.y+1) + \
26         hit_tileElemAt(*src,2,thread.x+2,thread.y+1)) / 4;
27 }

```

Figure 3. Communicator kernel implementation of the solutions for the three case studies.

of each cell does not imply any kind of dependencies with the computation of another one.

The GPU solution to the problem involves just one kernel with a bidimensional grid and bidimensional threadblocks. Depending of the size of the grid and the matrices, each block of threads computes the result values of several blocks of the matrix iteratively, following the implementation of the examples presented by the CUDA community in the programming guide [10]. On each iteration the accesses to global memory are fully coalesced. The CPU solution is similar to the GPU version.

Matrix multiplication

The Matrix multiplication consists on the product of two different 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.

The coprocessors solution to this problem involves just one kernel with a bidimensional grid and bidimensional threadblocks. Each GPU thread $t_{i,j}$ is responsible of computing the product operation ($\sum_{k=0}^{n-1} A[i][k] * B[k][j]$) in of one position of the C matrix. Our program does not exploit any optimization related to the use of the shared memory to enforce a mixed coalesced and not coalesced global memory access patterns (to matrix A, and B respectively). The CPU solution is similar to the GPU version.

Jacobi PDE solver

This program computes the stability point of a Partial Differential Equation (PDE). In this case the Poisson equation to compute the Heat Transfer on a 2-dimensional surface. It uses an iterative Jacobi method to solve the equation system generated when representing the space as a 2D grid of points with the same distances and boundary conditions. On each iteration each cell of a matrix is updated depending on the values of its four neighbors (horizontal and vertical). When all the cell values are computed then a new iteration is started until a finishing criterion is fulfilled. Note that an

```

1  /* Characterization for MA kernel */
2  KERNEL_CHAR(k_ma, 2, full, low, low)
3
4  /* Characterization for MM kernel */
5  KERNEL_CHAR(k_mm, 2, medium, medium, high)
6
7  /* Characterization for JS kernels */
8  KERNEL_CHAR(k_copy, 2, full, low, low)
9  KERNEL_CHAR(k_update, 2, medium, medium, medium)

```

Figure 4. Characterizations of the kernels implemented in the solutions of the three case studies.

auxiliary matrix is required to store the new value of a cell without race conditions with the neighbor cells. The pointers of the original matrix and the auxiliary one are rotated on each iteration. Figure 5 shows the pseudo code of the sequential variant of this method.

Coprocessors solution consist on two kernels. The first is the *kernel_update*, that is the responsible of computing the new values and store them in the auxiliary matrix. The second is a *kernel_copy*, that is the responsible of copying the new data from the auxiliary matrix to the original one of the last iteration if the number of iterations is odd.

4.3 Measures: Development effort and performance overhead

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 metrics: CO-COMO lines of code, number of tokens, and McCabe cyclomatic complexity. The first two ones measure the volume of code that the programmer should develop. The third one measures the rational

```

1 // Iteration loop
2 for(int i = 0; i < iteraciones; i++){
3
4 // UPDATE each matrix cell
5 // with the mean of its 4 neighbors
6 for(int j = 1; j < MatrixSide-1; j++){
7   for(int k = 1; k < MatrixSide-1; k++){
8
9       matrix[j][k] = (matrixAUX[j][k-1] +
10                      + matrixAUX[j][k+1] +
11                      + matrixAUX[j-1][k] +
12                      + matrixAUX[j+1][k])/
13                      / 4;
14   }
15 }
16
17 // SWAP of the matrices pointers
18 *matrixTMP = *matrixAUX;
19 *matrixAUX = *matrix;
20 *matrix = *matrixTMP;
21 }

```

Figure 5. Pseudo code for the sequential solution of the Jacobi iterative method used to solve the PDEs.

effort needed to program it in terms of code divergences and potential casuistry that should be considered to develop, test and debug.

We measure these metrics in: the baseline versions, using the specific programming models (OpenMP for the CPU variant, and CUDA for the GPU variant), and the versions using our communicator library interface. For a fairly comparison both approaches use the same kernels with the same values for the configuration parameters, that are injected manually for the baseline versions, and through the primitives for the communicator ones.

The second part of our experimental study measures the impact of using our communicator prototype in terms of performance. We compare the total execution times when launching the baseline and the communicator prototype versions of the three different study cases with different sizes of the input set. 10×10 , 100×100 , $1\,000 \times 1\,000$, and $10\,000 \times 10\,000$. We test the CPU versions using just 1 thread, and 24 threads (maximum number of CPU cores of the machine). On each case, the final execution time is taken as the average of 10 repetitions. The number of iterations executed for the Jacobi PDE solver is 100.

5. Experimental Results

This section shows and describes the results obtained from our experimental evaluations measuring the development effort metrics and the performance overhead for the study cases.

5.1 Development effort metrics

Table 1 shows the measures of the metrics for the three study cases. We can appreciate that the values of all the metrics for the GPU versions are significantly reduced. On the other hand, for CPU versions when we compare the communicator version with direct OpenMP implementation derived from the sequential code only is the cyclomatic complexity is reduced. However, considering the CPU cores as another coprocessor device, as it can be done in our model, simplifies the development effort of porting one solution to the other device type. Column (a) of Table 2 shows the number of tokens which should be modified to port the CUDA GPU program to the OpenMP program, whereas Column (b) of the same table shows the number of tokens that changes between our communicator based programs for GPU and CPU cores. The portability across devices in our solution is extremely reduced.

Study Case	Version	Lines of Code	#Tokens	Cyclomatic Complexity
Matrix add.	CUDA	68	747	5
	Comm. GPU	43	389	3
	OpenMP	32	251	4
Matrix mult.	Comm. CPU	42	387	3
	CUDA	70	778	6
	Comm. GPU	43	391	4
Jacobi solver	OpenMP	35	269	5
	Comm. CPU	45	405	5
	CUDA	85	882	17
Jacobi solver	Comm. GPU	61	617	13
	OpenMP	61	554	17
	Comm. CPU	59	615	13

Table 1. Measures of the developing effort metrics for the three study cases. Metric values of communicator version (*Comm.*) lower than baseline version are highlighted.

Study Case	CUDA → OpenMP	Comm. GPUs → Comm. CPUs
Matrix addition	130	10
Matrix multiplication	152	75
Jacobi solver	284	17

(a) (b)

Table 2. Comparison of developing effort in terms of number of tokens when porting from native programming languages (a), and when porting from our communicator based GPU implementation to the communicator based CPU version. Communicator porting results lower than native porting ones are highlighted.

5.2 Performance overhead

Figures 6 and 7 show the execution times of the three study cases in a shared-memory system. The communicator should perform insertions in the memory map when variables are binded, and searching in the memory map when it launches a kernel. Even so, we observe that the codes that use *Communicators* do not introduce noticeable overhead in any study case.

6. Conclusions and Future Work

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

It provides mechanisms: (1) to associate communicators to devices, (2) to define portable kernels that can be reused across different types of devices, (3) to select proper values for launching parameters on different devices through the characterization of the kernels, and (4) to automatically deal with different memory spaces of the host and the devices when needed. This model homogenizes the kernel programming and 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 computational cost 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, other techniques that exploit the modern features of some accelerators, such as asynchronous communi-

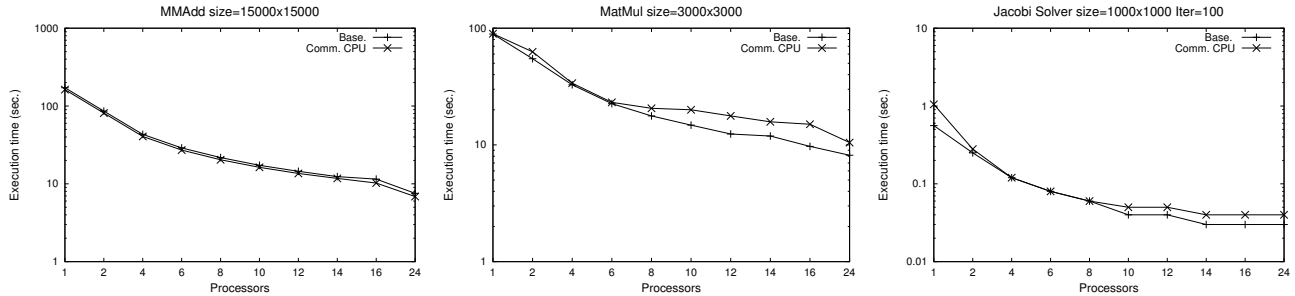


Figure 6. Execution times (seconds) of the baseline (Base.) and *communicator* (Comm.) versions of the three study cases in Heracles.

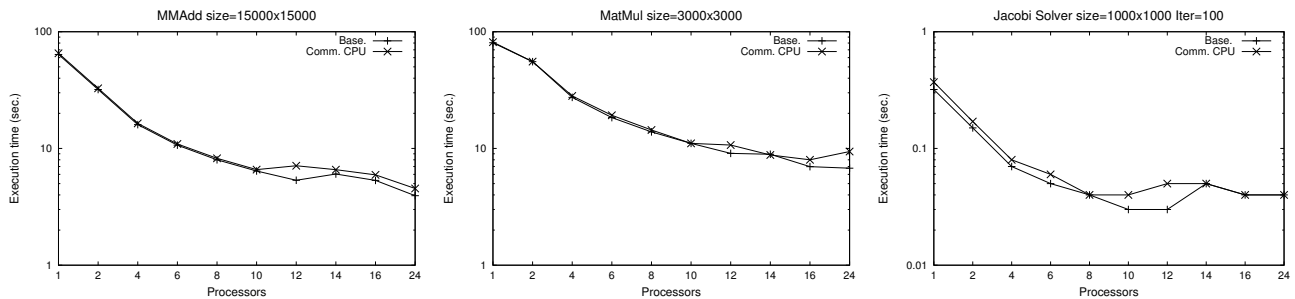


Figure 7. Execution times (seconds) of the baseline (Base.) and *communicator* (Comm.) versions of the three study cases in Chimera.

cations, communication-computation overlapping, efficiency improvements of our prototype implementation, and kernel launching policies.

A. Hitmap Library and other definitions

Hitmap [5] is a library for the management and run-time mapping of hierarchical tiling arrays. It is based on an SPMD model, and the message-passing paradigm. Hitmap has three main functionality modules: (a) Domain and tile management; (b) Mapping modules; and (c) Communication patterns. Hitmap defines objects to declare and manipulate multidimensional index domains and different types of indexed data structures [4]. Hitmap defines a plugin system to include new mapping modules: Virtual topology constructors and mapping functions named *Layouts*. The modules generate objects that can be queried at run-time to obtain information about the result of the mapping. Finally, it contains functionalities to build reusable communication patterns for tiles or subtiles across virtual processes. These functions internally use the MPI standard, exploiting efficient techniques like derived data types and asynchronous communications.

A.1 Definitions and notation

In this work, we focus on arrays with regular dense and strided domains. Their index *Domain* is a subspace of Z^n . Rectangular n -dimensional parallelootope domains, dense or strided, can be represented by a tuple of n *Signatures*. A Signature is a triplet of integer numbers $S\langle b, e, s \rangle$ (meaning *begin*, *end*, and *stride*). The set of indexes expressed by a signature is $S\langle b, e, s \rangle = \{b \leq i \leq e : (i - b) \bmod s = 0\}$.

A data structure or *Tile* ($T : D \rightarrow \text{type}$) is an object that associates data elements of a given *type* to index elements of a domain. *Array tiles* associate one data element to each domain element. The domain of a tile is denoted as $D(T)$. Hierarchical tiling is a technique that allows new Tile structures to be hierarchically declared with a subset of the domain of another Tile. The subtile maps the

index elements of its subdomain to the same data elements of the root tile $r(T)$ (the ancestor of the subtiling chain). The *Selection* function, $s : T \times D^* \rightarrow T$, is used to declare a new subtile: $s(t, d) = t' : r(t) = r(t'), D(t') = D(t) \cap d$.

A.2 Interface

Hitmap provides, among others, with the following basic functionalities for dense data structures management.

HitSig

- `HitSig hit_sig(begin, end, stride):`
Creates a structure of type *Signature*.

HitShape

- `HitShape hit_shape(dims, sig [, sig] ...):`
Creates a structure of type *Shape* to represent a domain.

Tile

- `void hit_tileNewType(newType):`
Defines a new type of tile structure with the indicated data type *newType*.
- `void hit_tileDomainShape(tile, tile_dataType, shape):`
Creates a tile using a domain defined by a *Shape* structure.
- `void hit_tileAlloc(tile):`
Reserves memory for the tile.
- `void hit_tileFree(tile):`
Frees the memory of the tile.
- `type hit_tileElemAt(tile, dims, ...):`
Access to a particular element of a tile.
- `void hit_tileSelect(newTile, oldTile, shape):`
Creates a subtile.

Acknowledgments

This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H5 network (TIN2014-53522-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

References

- [1] OpenACC Consortium. OpenACC: Directives for Accelerators. WWW, 2011–2015. on <http://www.openacc-standard.org/>.
- [2] 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. .
- [3] O. Consortium. Openmp 4.0 specifications. WWW, July 2013. on <http://openmp.org/wp/openmp-specifications/>.
- [4] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos. Blending extensibility and performance in dense and sparse parallel data management. *IEEE Trans. Parallel Distrib. Syst.*, 25(10):2509–2519, 2014.
- [5] 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.
- [6] 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. ISBN 978-1-60558-517-8.
- [7] N. Karunadasa and D. Ranasinghe. Accelerating high performance applications with CUDA and MPI. In *ICHS'2009*, pages 331–336, dec 2009. .
- [8] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. ISBN 0123814723, 9780123814722.
- [9] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1. Technical report, University of Tennessee, 2015. URL www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.
- [10] NVIDIA. NVIDIA CUDA C Programming Guide 7.5, 2015. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Last visit: November 16th, 2015.
- [11] 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. ISSN 0920-8542. .
- [12] 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. ISSN 0885-7458. .
- [13] J. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, may 2010. ISSN 1521-9615. .
- [14] 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. ISBN 978-3-540-89739-2.
- [15] TOP500.org. Top500 supercomputing sites. WWW, Nov 2014. on <http://www.top500.org/>.
- [16] 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. ISSN 0920-8542. .