High Productivity Multi-device Exploitation with the Heterogeneous Programming Library

Moisés Viñas[a], Basilio B. Fraguela[a], Diego Andrade[a], Ramón Doallo[a]

[a] *Universidade da Coruña, Grupo de Arquitectura de Computadores, A Coruña, Spain*

Abstract

Heterogeneous devices require much more work from programmers than traditional CPUs, particularly when there are several of them, as each one has its own memory space. Multi-device applications require to distribute kernel executions and, even worse, arrays portions that must be kept coherent among the different device memories and the host memory. In addition, when devices with different characteristics participate in a computation, optimally distributing the work among them is not trivial. In this paper we extend an existing framework for the programming of accelerators called Heterogeneous Programming Library (HPL) with three kinds of improvements that facilitate these tasks. The first two ones are the ability to define subarrays and subkernels, which distribute kernels on different devices. The last one is a convenient extension of the subkernel mechanism to distribute computations among heterogeneous devices seeking the best work balance among them. This last contribution includes two analytical models that have proved to automatically provide very good work distributions. Our experiments also show the large programmability advantages of our approach and the negligible overhead incurred.

Keywords: programmability, heterogeneity, parallelism, portability, libraries, load balancing, OpenCL

1. Introduction

The main drawbacks of heterogeneous systems are their higher programming complexity and the potential lack of portability, unless OpenCL is used. The added programming costs stem from the usual requirement to use extended languages and environments, together with the need to deal with the particular characteristics and limitations of these devices, which often require a tedious error-prone management. This latter problem is particularly strong in OpenCL, being the price to pay for its portability. Besides, the cost of these activities grows with the number of devices available. An additional difficulty that arises in applications that try to exploit all the resources available in heterogeneous environments is the optional distribution of work among the devices, given their very diverse features.

Email addresses: moises.vinas@udc.es (Moisés Viñas), basilio.fraguela@udc.es (Basilio B. Fraguela), diego.andrade@udc.es (Diego Andrade), doallo@udc.es (Ramón Doallo)

This situation has led to much research to improve the programmability of heterogeneous systems [9, 4, 5, 20, 24, 18, 32]. A recent proposal is a pure library-based environment called Heterogeneous Programming Library (HPL) [43]. It avoids requiring language extensions by providing a language embedded inside C++ in which the kernels to run in the heterogeneous devices can be written. HPL has maximum portability thanks to OpenCL being its backend, and it fully automates the management required by heterogeneous resources while introducing negligible overheads. Also, its adaptive runtime allows to even outperform in some cases native OpenCL codes [44].

This paper explores three mechanisms to reduce the programming effort of applications that exploit several heterogeneous devices using HPL as framework given its advantages. A first idea is the ability to use subarrays, that is, regions of arrays, that can be used in one or several devices without becoming separate data structures, so that it is always possible to keep a view of the underlying full array, while automatically keeping the consistency of the data. The second idea are mechanisms to split kernels for their execution in several devices, giving place to what we call subkernels. Our experiments show that these mechanisms can largely improve programmability, their overhead being minimal. The final contribution is an alternative to our initial subkernel proposal that is more flexible and allows to easily explore and choose the best workload distribution when a computation is accelerated using devices with different capabilities. This proposal, which includes analytical models capable of choosing optimal or near-optimal work distributions, not only helps programmability but also performance.

The rest of this paper is organized as follows. The next section explains the semantics and syntax of HPL. This is followed by the explanation of the new extensions in Section 3 and our evaluation in Section 4. Related work is reviewed in Section 5, with our conclusions and future work proposals closing the paper.

2. The Heterogeneous Programming Library

The Heterogeneous Programming Library (HPL) is publicly available under GPL license at `http://hpl.des.udc.es`. Its programming model follows that of OpenCL. The hardware model considers a host with some computational devices. Each device has a separate memory and processors that can only work on the device memory, so that data must be transferred in and out of it. Also, all the processors in a device run the same code in a SPMD fashion. We now explain the basics of HPL programming and how its multi-device programs look like.

2.1. Library Basics

HPL applications consist of a main program that runs in the host, and functions to run in the heterogeneous devices, called kernels. Each kernel is executed in parallel by a number of threads defined by a space of between one and three dimensions called global domain. These threads can be grouped in subsets of a size given by another space of the same number of dimensions as the global domain, called local domain. The threads in the same group enjoy barrier synchronizations and share a fast scratchpad memory unavailable to other threads or

```
1  void mxProduct(Array<float,2> c, Array<float,2> a, Array<float,2> b, Int p)
2  { Int i;
3
4    c[idx][idy] = 0.f;
5    for_(i = 0, i < p, i++)
6        c[idx][idy] += a[idx][i] * b[i][idy];
7  }
8  ...
9  Array<float,2> c(M, N), a(M, P), b(P, N);
10
11 eval(mxProduct)(c, a, b, P);
```

Figure 1: Naïve matrix product in HPL

the host. There is also a regular global memory, a constant memory that the device threads can only read, and a private memory that is exclusive of each thread.

HPL kernels are regular C++ functions written in a C-like embedded language provided by the library [43], whose details are out of the scope of this paper. The function arguments are the kernel inputs and outputs, and they must have type `Array<`*type, ndim [, memoryFlag]*`>`, which represents a *ndim*-dimensional array of elements of the C++ type *type*, or a scalar for *ndim=0*. The *memoryFlag* indicates the kind of device memory, and it defaults to `Global` (global device memory) when it is not specified in the host code or the kernel arguments. However, its default value for variables declared inside the kernel functions is `Private`, which locates them in memory private to each thread, or following OpenCL terminology, work-item. There are also convenient types to define scalars (e.g. `Float`, `Uint`, ...).

Figure 1 shows in lines 1-7 a naïve HPL kernel for the matrix product `c=a`×`b` in which thread (`idx`, `idy`) computes `c[idx][idy]`, where `idx` and `idy` are predefined variables provided by HPL that uniquely identify each thread within the global domain. The differences with a regular C function are the use of the predefined variables that identify the thread running the kernel, the HPL required data types, and the use of macro `for_` to represent a `for` loop. The associated host code, in lines 9-11, declares the kernel arguments and invokes it with syntax `eval(`*f*`)(`*args*`)` where *f* is the kernel function. Non-scalar arguments must be declared in the host code as `Array`s, while standard C/C++ scalars are directly supported. By default the dimensions of the global domain of the kernel correspond to the dimensions of the first argument, while the local domain is chosen by HPL, which suits our example. Method invocations between the `eval` and its arguments allow to specify these domains and the device in which the kernel must run. For example, `eval(f).device(GPU, 0).global(50).local(10)(a, b)` runs kernel `f` in the GPU number 0 on the arguments `a` and `b` using a global domain of 50 threads divided in groups of 10.

Since device memories are separated from that of the host, HPL builds buffers for the arrays that are not yet allocated in the target device and transfers those inputs that are

missing in the device before the kernel begins its execution. HPL analyzes kernels to learn which are their input and output arrays. It also tracks the accesses to the `Array`s in the host code to know whether they are read or written. These mechanisms allow HPL to learn where is the most up-to-date version of an array, and which arrays need to be transferred in each direction between the host and the device where each kernel is executed without any user intervention. A lazy copying policy that minimizes the number of transfers is applied, so that only when an access to data that is not available in a memory is requested, a transfer from the memory with the current version is performed. All these operations are totally transparent to users, who only need to focus on the application semantics.

2.2. Multi-device Applications

Once the host requests the execution of a kernel, it continues running its own code. Therefore the subsequent computations in the host, including kernel launches in other devices, proceed in parallel with the kernel computations in the device, which enables the development of multi-device applications. The host only waits for a kernel completion when its outputs are required for some other computation.

HPL facilitates the development of multi-device applications by providing a coherent view of `Array`s to the devices and the host, `Array`s being the unit of transfer, allocation and coherency. This is completely automated thanks to the mechanisms that allow the library to know when an `Array` is read or written, commented in Section 2.1.

While the programming style enabled by HPL for multi-device programs is much better than that of alternatives such as OpenCL, it has some limitations. The most important one is that since the `Array` is the unit of coherency, using the same `Array` in several kernel executions serializes them, even if each kernel operates on disjoint parts of its data, unless it is a read-only input to all these kernels. This way, the parallel execution of kernels that update different portions of an array in several devices requires defining a different `Array` per device, associated to the specific portion updated in that device. This policy also makes sense for read-only arrays when each device only needs to read a portion of them. The objective in this case is to minimize the data transferred, as the `Array` is also the unit of allocation and transfer.

Figure 2, which parallelizes `cm=am`$\times$`bm` among several GPUs splitting the work by rows, illustrates this programming style. For simplicity the code assumes that the number of rows `M` is a multiple of the number `ndevices` of GPUs, obtained in line 4 with a function provided by HPL. The underlying matrices are declared in line 1 as regular arrays. Since the whole matrix `bm` is used in each one of the parallel kernel executions, a single HPL `Array b` is declared in line 2 that contains it. It deserves to be mentioned that the constructor of an `Array` allows an optional pointer to the host memory where its data resides. If not provided, HPL allocates and deallocates the space for the data in the host as needed.

Since the parallel execution of kernels requires separate `Array`s for `cm`, an array of `ndevices` pointers to `Array`s is built in line 5. Then each `Array` of the appropriate size is created, associating its storage to the corresponding portion of matrix `cm` in line 9. The same approach is followed for matrix `am` in order to minimize the amount of data transferred. There is also the added benefit that this way we can continue using the original kernel that

```
1  float cm[M][N], am[M][P], bm[P][N];
2  Array<float,2> **c, **a, b(P, N, bm);
3
4  const int ndevices = getDeviceNumber(GPU);
5  c = new Array<float, 2> * [ndevices];
6  a = new Array<float, 2> * [ndevices];
7
8  for(i = 0; i < ndevices; i++) {
9    c[i] = new Array<float, 2>(M/ndevices, N, cm+i*(M/ndevices*N));
10   a[i] = new Array<float, 2>(M/ndevices, P, am+i*(M/ndevices*P));
11 }
12 ...
13 for(i=0; i< ndevices; i++)
14   eval(mxProduct).device(GPU, i)(*c[i], *a[i], b, P);
```

Figure 2: Matrix product on multiple GPUs using basic HPL features

uses the same row index in the `Arrays` `c` and `a`. Otherwise the kernel would require an additional argument to specify which is the row of the whole array `a` that is associated to the computation of the first row of the subarray of `c` computed by this kernel execution. Finally, the kernel executions in line 14 use the i-th `Arrays` of `c` and `a` for the run in the i-th GPU. We will use this code as running example to illustrate the new proposals presented in Sect. 3 because of its simplicity. A more advanced example based on a stencil code can be found in Appendix A.

3. Improving Multi-device Support

We propose three features to facilitate the exploitation of multiple devices, which are the usage of subarrays and two approaches to easily split kernels on multiple devices. The last one of these mechanisms also incorporates facilities to optimally use resources with different computing properties that participate in the computation of a kernel. The three proposals are now described in turn, followed by the description of unified memory exploitation, a general optimization that has been recently incorporated in HPL, and whose usage requires no effort from the user.

3.1. Subarrays

Making the `Array` the unit of allocation, transfer and consistency forces to build separate `Arrays` to enable parallel executions and to reduce the amount of data allocated and copied. Allowing the independent use of regions of an `Array`, while providing a coherent view of it, is much more convenient. We thus enabled the selection of regions of `Arrays` using `Ranges`, `Range`(a, b) corresponding to the inclusive range of integers $[a, b]$. HPL subarrays are not copies by value, but references to the data of the underlying `Array`, so that updating them changes the corresponding portion of their parent `Array`. Using this feature, the matrix

```
1 float cm[M][N], am[M][P], bm[P][N];
2 Array<float,2> c(M, N, cm), a(M, P, am), b(P, N, bm);
3
4 const int ndevices = getDeviceNumber(GPU);
5
6 for(i=0; i< ndevices; i++) {
7   Range rows(i * (M/ndevices), (i+1) * (M/ndevices) - 1);
8   eval(mxProduct).device(GPU, i)(c(rows, Range(0, N-1)), a(rows, Range(0, P-1)), b, P);
9 }
```

Figure 3: Matrix product on multiple GPUs using subarrays

product example using multiple GPUs can be written as Fig. 3 shows. The main `Array`s are defined using as storage the original C-style arrays defined in line 1, following an approach similar to that of Fig. 2, but they could have been defined on their own, as in Fig. 1. `Range`s are objects of their own, so for example line 7 builds the `Range rows`, which identifies the range of `M/ndevices` consecutive rows of the matrices `c` and `a` that are used in the `i`-th device. Line 8 uses this range together with appropriate ranges that span the `N` columns of `c` and the `P` columns of `a` to select the subarrays used in the execution in the `i`-th GPU.

Supporting subarrays, with the need to keep them consistent with their parent arrays, required deeply redesigning the HPL internal coherency mechanisms. The runtime keeps track of the relations between the different `Array`s that need to be kept consistent, enforcing waits for pending writes and performing the required transfers as necessary in a process that is transparent to the user. The management is highly optimized in several ways, as HPL caches the subarrays structures in order to avoid costly creation processes in each reuse, performs the minimum possible number of transfers, and whenever a subarray is used in a memory that already holds an `Array` that contains the new subarray, this new subarray does not allocate new memory, but just maps in the memory of the existing `Array`. A detailed explanation of the coherency and data movement strategy implemented can be found in Appendix B.

It is important to mention that subarrays can be also used to copy portions of an `Array` using the assignment operator. Assignments are allowed between `Array`s of the same size and from scalars to arbitrary `Array`s, which replicates the scalar in all the elements of the destination. Finally, subarrays also facilitate the implementation of algorithms in single device environments when they need to work on different portions of arrays in different kernel invocations (e.g. matrix factorizations).

3.2. Subkernels based on annotations

Launching kernels in multiple devices, with the associated iteration on the devices and selection of the portion of each array to process in each kernel, can be simplified in several ways. We propose here a simple mechanism to split a kernel in multiple executions, called subkernels, ensuring that each subkernel uses the appropriate arguments. Our approach

input: Number of devices requested for the kernel execution n
input: Vector of devices to use in the kernel execution $\overrightarrow{d} = (d_0, d_1, ..., d_{n-1})$
input: Number of arguments for the kernel k
input: Vector of arguments to the kernel $\overrightarrow{arg} = (arg_0, arg_1, ..., arg_{k-1})$

```
1  if n = 0 then
2  |  getAccelerators(d, n)
3  end
4  for i = 0 to n − 1 do
5  |  for j = 0 to k − 1 do
6  |  |  if arg_j is a scalar or a non-annotated array then
7  |  |  |  localArg_j = arg_j
8  |  |  else
9  |  |  |  localArg_j = getPartition(arg_j, n, i)
10 |  |  end
11 |  end
12 |  eval(d_i)(localArg)
13 end
```

Figure 4: Algorithm for an `eval` invocation based on annotations

consists in allowing a single `eval` invocation to split a kernel execution among an arbitrary number of devices, and annotating the arguments to specify how they must be partitioned among such devices.

A first modification made to enable this mechanism was to extend the `device` modifier of an `eval` to support as argument a vector of HPL device handles. This allows the user to specify among which devices the kernel must be split. The second and most critical modification required is the introduction of annotations in the kernel array arguments, whose purpose is to express how these arrays must be distributed among the devices. For example, the notation `PART1(a)` indicates that `Array a` must be evenly distributed among the devices involved in the kernel execution by its first (most significant) dimension, so that each chunk is a consecutive region of data. Namely, if the size of this dimension is n and the kernel must be split among D devices, the array will be partitioned along this dimension in $D-1$ chunks of $\lceil n/D \rceil$ elements that will be assigned in order to the first $D-1$ devices, while the last device will receive the last $n - \lceil n/D \rceil (D-1)$ elements of the dimension. Similar notations allow to partition the arrays by other dimensions. Our annotations also support an optional argument to express overlapping, so that `PART1(a,n)` extends each partition in `n` additional elements in each direction, which is convenient to parallelize stencil codes. Non annotated arrays are not partitioned, but replicated in each device. Also, non annotated arrays as well as arrays labeled with different annotations can be mixed in the same invocation. In this case, the non annotated arrays are replicated (i.e. used as a whole) for the subkernel run in each device, while for each partitioned array the i-th device gets the i-th chunk of the array according for the partitioning specified for it.

```
1 float cm[M][N], am[M][P], bm[P][N];
2 Array<float,2> c(M, N, cm), a(M, P, am), b(P, N, bm);
3
4 eval(mxProduct)(PART1(c), PART1(a), b, P);
```

Figure 5: Matrix product on multiple GPUs using annotations

The behavior just described is better illustrated in Fig. 4, which represents the implementation of an `eval` based on annotations. The pseudocode uses a vector notation to represent lists of arguments for conveniency. This strategy is used whenever at least one annotated array appears in the list of arguments to a kernel, even if the invocation used no `device` modifier. In this latter situation, managed in lines 1-3 of the pseudocode, the execution is distributed among all the accelerators in the system, which are retrieved through the function `getAccelerators`. The other function used in the algorithm, `getPartition(a,`n, i`)`, retrieves the i-th out of n subarrays in which the array `a` is partitioned according to the annotation applied to it.

When the global domain is not specified in an `eval` of this kind, the consistent behavior of adopting as global domain of each subkernel the one associated to its first argument is followed. Similarly, unspecified local domains are chosen by HPL. The modifiers that specify these domains have been extended allowing that any or all their arguments are vectors of integers, so that the i-th component indicates the domain for the execution of the subkernel in the i-th device.

This approach allows to write our running example based on a matrix product distributed among the GPUs in the system as Fig. 5 shows. The annotations on the argument arrays `c` and `a` indicate that they must be partitioned by their first dimension (`PART1`), that is, by rows, among the participating devices. Since the array argument `b` is not annotated, it will be replicated in the devices used. Finally, scalars such as `P` are always just inputs to the kernels and they cannot be annotated. As we can see, this code distributes the data structures and the kernel execution in exactly the same way as Figs. 2 and 3, while being considerably simpler.

3.3. Subkernels based on execution plans

While the proposal in Sect. 3.2 is very convenient, it has some restrictions. The most important one is that when several argument arrays are partitioned, it necessarily uses the i-th subarray generated of each one of them in the i-th subkernel. Although this perfectly fits most kernels, some require more flexible patterns. Also, some kernels may require a more ad-hoc partitioning. This led us to design a more complex subkernel generation strategy. We first describe the general use of this strategy in Sect. 3.3.1. This is followed on an explanation of how HPL allows to use it to perform automatic load balancing between devices in Sect. 3.3.2.

```
1  void partitioner(FRunner& fr, Range rg[2],
2                   Array<float,2>& c, Array<float,2>& a,
3                   Array<float,2>& b, int P) {
4    fr( c(rg[0], rg[1]), a(rg[0], Range(0, P−1)), b(Range(0, P−1), rg[1]), P);
5  }
6
7  ExecutionPlan ep(partitioner);
8  ep.add(GPU, 0, 50);
9  ep.add(GPU, 1, 50);
10 eval(mxProduct).executionPlan(ep)(c, a, b, P);
```

Figure 6: Matrix product on multiple GPUs using an execution plan

3.3.1. Execution plans

This proposal is based on an object, which we call execution plan (`ExecutionPlan` class), that encapsulates the information needed to partition the work expressed by the global domain of a kernel. This information is the list of devices to use, the percentage of the work that each device will perform, and how to partition the arguments for each subkernel responsible for a portion of the computation. This last item is expressed by means of a user-provided function, which we call partitioner, that given a range of work and the kernel arguments, performs the subkernel invocation partitioning accordingly the arguments.

When an `eval` is provided with an execution plan, it uses this object to partition the global domain provided for the kernel in chunks that are proportional to the amount of work to be performed in each device. Our current implementation partitions the most significant dimension of the global domain of the kernel in chunks that are proportional to the ratios requested for the user. The reason for focusing the partitioning on this dimension is that the less significant dimensions are usually the ones with more locality and where coalesced accesses (in devices such as GPUs) are exploited. The resulting global domain partitions are also automatically rounded to ensure that they are multiples of the local domain if the user has specified one. Then, it invokes the partitioner with a functor object called `FRunner` that encapsulates the information needed to run a subkernel, an array of ranges that provide the portion of the global domain assigned to that subkernel in each dimension, and the kernel arguments. The partitioner must invoke the `FRunner` as a function whose arguments are the subarrays of the global argument arrays that are associated to the range(s) of the global domain computed by the execution plan.

Figure 6 implements our matrix product example using this approach. The `ExecutionPlan` is built in line 7 providing the partitioner defined in lines 1-5. A partitioner always has as first argument the `FRunner` that runs the kernel. The second argument is always an array of up to three ranges that describes the portion of the global domain associated to a subkernel, each `Range` being associated to one dimension of the problem. These two arguments are followed by the actual list of arguments of the kernel. While its body can contain more things, a partitioner only needs to invoke the `FRunner` with the arguments associated to the

portion of the global domain associated to the input ranges. Lines 8 and 9 add GPUs 0 and 1 to the `ExecutionPlan ep`, respectively, each one being assigned 50% of the work. Finally, the parallel multi-device kernel execution using `ep` is requested in line 10. As we can see, this strategy allows to easily and flexibly split the work between devices assigning arbitrary portions of work to each one of them while supporting any custom required subarrays.

3.3.2. Automatic load balancing

Execution plans can be either completely specified by the user, as seen above, or programmed to search for a distribution of work that balances the load among the devices that participate in the kernel execution. This second possibility is very appealing when heterogeneous devices with different communication latencies and bandwidths and/or computational capabilities can be used to execute a portion of the considered kernel. In order to use an execution plan in this second way, the user has to specify in its constructor the search algorithm to use and whether only the computation or also the transfer times of the kernel must be taken into account in the balancing. Also, there is no need to provide the ratios of work for each device that is added to the plan, as they will be automatically derived by our library. The search will be performed the first time that the execution plan is used in the execution of the kernel. Subsequent usages of the object will reuse the distribution found unless the user resets it.

Currently HPL execution plans provide three algorithms to search for the best work distribution. The simplest and most expensive one is the `EXHAUSTIVE` search, which tries all the legal combinations of distributions that differ in a given minimum step and chooses the best one. The default step variation is 5% of the global domain, but users can choose a different one. Just as in the other search algorithms, the library will only time the kernel execution or it will also perform and time the transfers for the inputs and the outputs to choose the best option depending on what the user specified in the construction of the execution plan. Notice that the two possibilities make sense in different scenarios, as sometimes the user may know that the data will have to be transferred for each execution of the kernel, while in other situations, such as iterative algorithms, the vast majority of the kernel executions do not require transfers. It also deserves to be mentioned that in order to avoid noise measurement problems, each distribution considered both in this scheme and in the ones described below is timed a number of times that the user can configure, the default being twice.

The other two possibilities are based on profiling and a simple model that relates the portion of the global domain assigned to a device with its runtime. Both models start with an execution in which each one of the $NDevices$ devices involved is assigned $(100/NDevices)\%$ of the global domain, and the times gathered are stored in a vector of times $\overrightarrow{t}$. Figure 7 shows the load balancing algorithm of the first model, called `SINGLE_STEP_MODEL` because it is not iterative. Starting from the measured times $\overrightarrow{t}$, this algorithm assigns to each device a ratio of work that is proportional to the speedup it achieved with respect to the slowest device in the set in this initial execution (line 3). The rationale is that this speedup is proportional to the amount of work that the corresponding device can receive in a balanced distribution of work, as this number should be proportional to its computing (and transfer, if included) performance. Lines 4 and 5 scale this initial ratio to a percentage making

1 **Algorithm** `balance(`$\overrightarrow{t}$`)`

input : Vector of times measured in each device $\overrightarrow{t} = (t_0, t_1, ..., t_n), n = NDevices - 1$

output: Vector of percentage of work to be performed in each device $\overrightarrow{w} = (w_0, w_1, ..., w_n), n = NDevices - 1$

2 $MaxTime = \max\{t_i,\ 0 \leq i < NDevices\}$

3 $u_i = MaxTime/t_i,\ 0 \leq i < NDevices$

4 $U = \sum_{i=0}^{Ndevices-1} u_i$

5 $w_i = u_i \times 100/U, 0 \leq i < NDevices$

6 $Adjust = 0$

7 **for** $i = 0$ **to** $NDevices - 1$ **do**

8 **if** $\frac{t_i \times w_i}{100/Ndevices} < Threshold$ **then**

9 $Adjust = Adjust + w_i$

10 $w_i = 0$

11 **end**

12 **end**

13 $w_i = w_i \times 100/(100 - Adjust), 0 \leq i < NDevices$

14 **return** $\overrightarrow{w}$

Figure 7: `SINGLE_STEP_MODEL` load balancing algorithm

sure the addition for all the devices covers the whole global domain. Lines 6-12 compute whether with this distribution the estimated runtime for some device falls below a measured threshold. If this is the case, the device receives no work (line 10), and its portion of the problem is added to *Adjust* (line 9), so that it is redistributed among the remaining devices in line 13. This last stage of the algorithm avoids sending very small amounts of work to devices that do not compensate the reduced fixed overhead associated to this process. Notice that since this model requires a single execution, and it is made at runtime on correct inputs, thus generating correct outputs, this model can make just as many kernel executions as an untuned code.

The second model is iterative, so it can adjust its distribution with more precision in situations where the workload is not necessarily directly proportional to the ratio of the global domain assigned to each device. This model, called `ITERATIVE_MODEL`, starts applying the `SINGLE_STEP_MODEL` and measures the runtimes of the distribution it chooses. Then, as long as the runtime for the slowest device is δ% or more longer than the runtime in the fastest device, it recomputes the percentage of the global domain assigned to each device using the algorithm in Fig. 8 and makes a new execution to measure the new times. The value of δ defaults to 5, but it can be chosen by the user. The algorithm estimates which would have been the runtime for each device in an execution in which all of them received the same amount of work based on the actual distribution of work made and the time measured, and then applies the `SINGLE_STEP_MODEL` to these times. After the second iteration the algorithm applies a relaxation mechanism (lines 4-6). Its purpose is to avoid oscillations in

1 **Algorithm** `adaptiveBalance`$(\overrightarrow{t}, \overrightarrow{v}, iter, \overrightarrow{w_{prev}})$

input : Vector of times measured in each device $\overrightarrow{t} = (t_0, t_1, ..., t_n), n = NDevices - 1$

input : Vector of percentage of work that was performed in each device $\overrightarrow{v} = (v_0, v_1, ..., v_n), n = NDevices - 1$

input : Number of iteration $iter$

input : Vector of percentages of work computed in the previous iteration or zeros in the first iteration $\overrightarrow{w_{prev}} = (w_{prev_0}, w_{prev_1}, ..., w_{prev_n}), n = NDevices - 1$

output: Adapted vector of percentage of work to be performed in each device $\overrightarrow{w} = (w_0, w_1, ..., w_n), n = NDevices - 1$

2 $t'_i = \frac{t_i}{v_i} \times \frac{100}{NDevices}, 0 \leq i < NDevices$

3 $\overrightarrow{w} =$ `balance(`$\overrightarrow{t'}$`)`

4 **if** $iter > 2$ **then**

5 $\quad \overrightarrow{w} = \overrightarrow{w_{prev}} + \frac{\overrightarrow{w} - \overrightarrow{w_{prev}}}{iter - 1}$

6 **end**

7 **return** $\overrightarrow{w}$

Figure 8: `ITERATIVE_MODEL` adaptive load balancing algorithm

the algorithm that could delay it or even incur in an infinite loop, something that indeed happened in our experiments before this mechanism was added to the algorithm. When the relaxation is applied, instead of suggesting as optimal distribution the newly computed one, the algorithm rather moves the existing distribution $\overrightarrow{w_{prev}}$ in the direction of the new prediction $\overrightarrow{w}$, the length of the movement being shorter in each successive iteration thanks to the division by $iter - 1$.

A problem that appears when users let the library choose the work distribution is that they lose the information on which portion of each array has been loaded and updated in each device. This never endangers the correctness of the HPL applications because whenever the host code or a kernel invocation tries to use an `Array` or a portion of it, our library knows where is the most up-to-date version of this data. Nevertheless performance will suffer if these automatic mechanisms introduce unnecessary transfers. HPL provides two mechanisms to avoid these problems. First, an `ExecutionPlan` can be built as a copy of an existing one so that it retains the same devices and work distribution, only changing the partitioner. Second, these objects provide methods to retrieve the distribution performed so that the user can know which portion of each array was used in each device and act accordingly.

3.4. Unified memory exploitation

Some devices have a memory that is separated from that of the host, while others work on the same physical memory. This is the case of the CPU when it is used under OpenCL, or the integrated GPUs that ship with many current CPU models. When the memory of a device is unified with that of the host transfers between both memories can be avoided or optimized. The exploitation of this property in OpenCL requires programmers to follow a series of steps that are different from the usual ones, an example being working based

Table 1: Benchmarks characteristics.

Benchmark	Kernels	Data	Baseline	
		exchanges	SLOCS	PE (Ks)
EP	1 u	-	325	1612
FT	3 u + 7 r	all to all	1656	35219
MG	6 u + 37 r	stencil	3076	106666
MMRow	1 u	-	220	1082
Summa	1 r	broadcast	298	1867
ShaWa	3 r	stencil	572	3430

on mapping and unmapping of OpenCL buffers instead of usual read and write operations. This way, performing this optimization further adds to the complexity of OpenCL programs.

HPL automatically detects when the device used for a kernel execution has its memory unified with that of the host, and internally applies the suitable OpenCL protocol to minimize the communication cost between this device and the host, making the optimization totally transparent and effortless to users.

4. Evaluation

Since the HPL backend is OpenCL, this is the standard tool with which it is fairer to compare our library. The C++ OpenCL API has been chosen for the baseline, as this is the language in which HPL, and thus its benchmarks, have been developed, so that both approaches enjoy the same language.

Table 1 describes the benchmarks used in the experiments in terms of their name, the number of kernels used in unique (u) and repetitive (r) invocations (i.e. inside some loop), the most common pattern of communication between subtasks when they are split among several devices and the source lines of code (SLOCs) and Halstead's programming effort [17] (PE, expressed in thousands) of the host-side implementation of the baseline. This latter metric is an estimation of the cost of the development of a code based on a reasoned formula that takes into account the number of unique operands, unique operators, total operands and total operators found in the code. For this, the metric regards as operands the constants and identifiers, while the symbols or combinations of symbols that affect the value or ordering of operands constitute the operators. We think that the programming effort is a fairer measurement of the productivity than SLOCs, since lines of code can largely vary in length and complexity. Our programmability analysis only focuses on the host side of applications because kernels are not affected by the proposals made in this paper, their length and complexity being besides very similar in OpenCL and HPL. Our baselines encapsulate the lengthy initialization of OpenCL in reusable routines that are placed in a library out the host code, which just calls these routines. As a result they are not measured in the baseline, which corresponds to the minimum amount of code that is required for these applications when using the OpenCL host C++ API.

Table 2: Programmability improvement for several strategies.

Benchmark	HPL-md		HPL-sub		HPL-ann		HPL-exp	
	ΔSLOC	ΔPE	ΔSLOC	ΔPE	ΔSLOC	ΔPE	ΔSLOC	ΔPE
EP	16.9	36.7	15.1	32.6	17.5	39.0	16.0	33.1
FT	18.8	37.4	15.7	31.3	25.9	42.1	16.7	25.9
MG	24.3	30.7	23.4	26.1	25.7	31.8	21.1	24.4
MMRow	18.2	29.0	18.6	32.3	29.1	51.9	20.5	40.2
Summa	25.2	37.7	39.9	61.8	-	-	37.3	56.4
ShaWa	31.0	43.3	40.2	52.2	56.3	76.7	50.0	58.8

EP, FT and MG codes come from the SNU NPB suite [37], an optimized OpenCL implementation of the NAS Parallel Benchmarks. MMRow is the matrix multiplication distributed by rows that has been used as running example in the preceding sections, but using an optimized kernel instead of the naïve kernel in Fig. 1. Namely, for the sake of reproducibility we have used the matrix product kernel taken from the Nvidia OpenCL SDK. Summa code implements the Summa algorithm for matrix multiplication [42], which divides the matrices in tiles and interleaves stages of local multiplication in each device with stages of communications consisting of broadcasts across columns and across rows of the two input arrays. Finally ShaWa is a shallow water simulator [45] whose main computational pattern is a stencil. This way, it is parallelized using ghost regions that replicate a portion of the data in another processor and must be refreshed in each iteration.

4.1. Programmability

Table 2 shows the percentual reduction in SLOCs and programming effort with respect to the baseline when the applications are developed with basic multi-device HPL (md), subarrays (sub), subkernels based on annotating the distribution of the arguments (ann) and subkernels based on execution plans (exp). These three last techniques were explained on Sections 3.1, 3.2 and 3.3, respectively. Notice that the higher the reductions, the better the programmability. Annotations can only be used when the i-th regions obtained in the partitioning of the arrays are always used together, that is, the first subkernel uses the first subarray of all the inputs, the second subkernel the second subarray, etc., which is not the case in Summa. It also deserves to be mentioned that since stencils require using a portion of the array assigned to a device for communications, which is not straightforward in HPL-md, and we had two benchmarks (MG and ShaWa) based on stencils, we wrote their HPL-md versions in different ways. Namely, our MG HPL-md version copies the whole arrays to the host, then exchanges the neighbor rows in the host memory, and finally copies the arrays back to the accelerators in the next iteration. Although this is much more inefficient than only transferring the required neighbor rows of the arrays, Section 4.2 will show that this version can outperform the native optimized version in some environments. Our ShaWa HPL-md code is written in the most possible efficient way without using subarrays. Namely

it uses auxiliary arrays of a single row that (1) receive their data from the main arrays by means of copy kernels executed in the accelerators, (2) are exchanged between the devices to copy this information and (3) provide their data to the destination array also by means of a copy kernel run in the accelerator.

Programming effort reductions are always stronger than SLOC reductions because this indicator takes into account the complexity of each line, the OpenCL API often having many parameters. Since we argue that this more complex metric is fairer than SLOCs, this is good news.

The techniques introduced in Section 3 provide better programmability than HPL-md in all the tests except the HPL-sub and HPL-exp implementations of the NPB applications. In the case of HPL-sub the main reason is that the arguments of the kernels of these applications have different sizes, which does not allow to reuse `Ranges` in their indexing. Nevertheless subarrays positively impact MMRow, and largely improve upon HPL-md in Summa and ShaWa. Their programmability metrics improvements over the baseline are in fact between 21% and 64% larger in relative terms that those of HPL-md for these benchmarks. In the case of MG another important reason why HPL-sub and HPL-exp do not offer better programmability than HPL-md is the very simple strategy used by this latter implementation, which copies the whole arrays to memory, while all the other versions only exchange one row.

HPL-exp is the next technique in terms of easiness for the programmer, as it achieves an average 26.9% SLOCs reduction with respect to the baseline, compared to the 22.4% of HPL-md and the 25.5% of HPL-sub. Similarly, it reduces the programming effort by a noticeable 39.8%, above the 35.8% of HPL-md and the 39.4% of HPL-sub. HPL-exp main programmability advantages are that execution plans avoid loops and the computations of most of the ranges required to split the work. On the other hand, this strategy requires defining the execution plan and the related partitioner. Also, the kernels with arguments of different sizes make sometimes insufficient the predefined `Ranges` provided by the execution plans. So in these cases the user must define new Ranges, like in the HPL-sub case. This is the reason why HPL-exp does not offer better programmability than HPL-md in the NPB. In the case of MG, this is also motivated by the simplicity of the HPL-md implementation. The biggest asset of HPL-exp with respect to the other options is that it is the only one that allows to exploit the automatic load balancing features described in Section 3.3.2, which are evaluated in Section 4.2.2.

Finally, HPL-ann systematically improves upon HPL-md and the other alternatives presented in this work. This happens even in the simplest benchmarks, where it is more complicated, as the baseline is all the host code, including host computations and data initialization. This way, for example annotations remarkably achieve up to 60% larger SLOCs reduction over the baseline than HPL-md in MMRow. The largest improvement takes place in this application and ShaWa, where HPL-ann almost doubles the programming effort reduction of HPL-md over OpenCL. On average, in the five benchmarks where it can be used, HPL-ann reduces the SLOCS and the programming effort by 30.9% and 48.3% with respect to the baseline, respectively, making it the default option when it is applicable and no automatic load balancing is needed. Overall these results largely justify the interest of our

Table 3: Execution time of the baselines (in seconds).

Accelerator	EP	FT	MG	MMROW	SUMMA	SHAWA
Fermi	1.49	33.81	14.86	5.74	39.33	973.20
K20	1.43	33.47	17.81	3.55	20.90	400.00
Xeon Phi	2.15	9.65	15.73	10.29	64.93	1819.20

proposals.

4.2. Performance

The performance evaluation uses three platforms, two based on GPUs and another one on Xeon Phis. The first one, called Fermi, has a 6-core Intel Xeon X5650 (2.67 GHz), 12 GB memory and two Nvidia M2050 GPUs with 3 GB and 448 cores at 1.55 GHz each. The second platform, called K20, has two 8-core Intel E5-2660 (2.2 GHz), 64 GB memory and two Nvidia K20m GPUs with 5 GB and 2496 cores at 0.705 GHz each. Both systems used the Nvidia OpenCL driver version 325.15. The Xeon Phi platform, called Phi for short, has the same host CPU and main memory as the K20 system, the accelerators being two Intel Xeon Phi 5110P with 60 cores at 1.056 GHz and 8GB with the Intel OpenCL driver version 4.5.0.8. The compiler was g++ 4.7.2 with optimization level `O3`. The problem sizes used in the experiments are C, B and B for EP, FT and MG, respectively. MMRow and Summa multiply double precision matrices of 8000×8000 elements, while ShaWa processes a 2000×2000 mesh.

The same version of the programs were used for all the tests. This means that since GPUs and Xeon Phis have a quite different architecture, we had to choose for which architecture to optimize the kernels. Since two of our platforms were based on GPUs and only one on Xeon Phis, we used kernels optimized for GPUs. This is reflected in Table 3, which shows the runtime of the baseline OpenCL applications in the three platforms when using the two accelerators available. It is interesting that despite this fact the Xeon Phi obtained the best performance for the most memory intensive benchmark, showing the interest of the two kinds of accelerators for different problems.

We now first compare the performance of the different HPL notations with the native OpenCL baseline and then evaluate our algorithms for work distribution.

4.2.1. Comparison with the OpenCL baseline

Figures 9 to 11 show the speedup of the HPL versions with respect to the OpenCL baseline in executions using the 2 accelerators in the Fermi, K20 and Phi systems, respectively. Notice that there is no data for Summa using annotations because this benchmark cannot be written using this strategy. The kernels of the HPL versions mimic those of the OpenCL baselines, so that they exhibit exactly the same behavior and performance, which has already been proved in all the previous publications about HPL, such as [43]. While HPL does not try to optimize the kernels provided by the user, it has an adaptive runtime that chooses

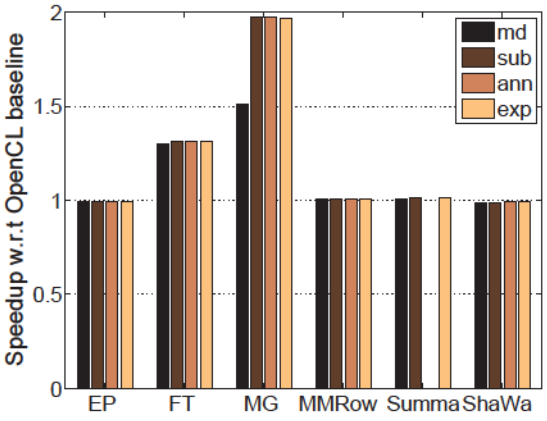

Figure 9: Performance in the Fermi system using both GPUs

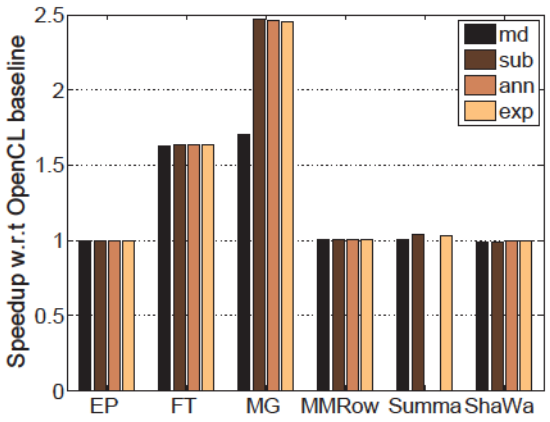

Figure 10: Performance in the K20 system using both GPUs

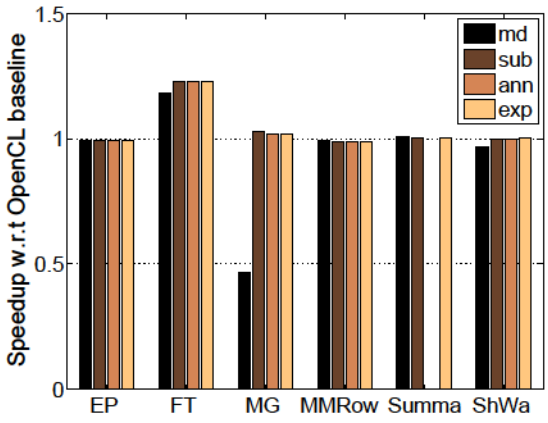

Figure 11: Performance in the Phi system using both Xeon Phis

the best strategy to exchange data between different devices [44], the possibilities being (a) the use of two transfers in sequence using as temporary buffer the host memory, or (b) the built-in `clEnqueueCopyBuffer` OpenCL routine. This allows HPL to noticeably outperform manually optimized codes in multi-device environments when those codes do not rely on the best strategy for the accelerator considered. This is the case for the FT and MG multi-device implementations from [37], which use a strategy for data exchange between devices that is well-suited for the Xeon Phi but not for the GPUs. As a result, HPL strongly outperforms them in the Fermi and K20. It is particularly interesting that despite following an inefficient strategy for the data exchanges that copies the whole arrays to the host memory, our MG HPL-md is much faster than the OpenCL baseline (51% in Fermi and 70% in K20) thanks to the selection of the best transfer method. The speedup of HPL MG grows to 98% and 146% in Fermi and K20, respectively, in the versions written using the proposals in this

paper, which largely facilitate the implementation of stencils. The situation is different for the Xeon Phi because the data exchange strategy of the OpenCL baseline is the best one. Despite this fact, our HPL-md version for FT still provides a 18.3% speedup, with the versions enabled by the novelties presented in this paper reaching a 22.8% speedup. As for MG, the optimized MG HPL versions get a performance similar to the baseline, while the naïve HPL-md version has a considerable lower performance. ShaWa HPL-md is slightly slower than the baseline (1% in Fermi, 1.7% in K20 and 3.3% in the Xeon Phi) because the lack of subarrays complicates the refresh of the shadow region of one row in each GPU, requiring additional buffers and copies. The slowdown is reduced to between 0.2% and 0.6% when HPL incorporates the novelties described in this paper, allowing to use a simple assignment to a subarray for these updates.

All in all, the programmability improvements proposed have from a neutral to a very positive impact on performance while allowing a much more natural way of expressing the algorithms at hand.

4.2.2. Automatic load balancing

The main interest of the execution plan approach lies in its ability to automatically optimize the distribution of work among different devices. We have measured the performance of the automatic load balancing provided by execution plans in the most time consuming kernel of each one of our benchmarks. Summa was excluded of the experiment because it is based on the assumption that each parallel task operates on a sub-matrix of the same size. While the kernels of EP, MMRow and ShaWa have a high arithmetic intensity, those of FT and MG have a high ratio of memory accesses per computation. They also have different patterns, as ShaWa and MG follow a stencil pattern, MMRow operates on a tiled way on its matrices, FT computes a complex FFT using a scratch array and EP makes most of its computation in private and local memory. This way, our tests rely on kernels with very different nature.

We performed the experiments using two configurations. In the first one, HPL was asked to automatically distribute the work among the CPU and one GPU in the K20 system. In the second configuration HPL had to split the work among the CPU and the two K20 GPUs. The OpenCL driver used for the CPU was the version 1.2.0.8 from Intel. Also, in both cases the two algorithms proposed in Section 3.3.2 were tried. We also sought for the optimal distribution using its exhaustive search feature using steps of 1% of the workload. Also, in all the experiments the CPU benefited from the automatic unified memory support provided by HPL (see Section 3.4).

In all of the five benchmarks, the two automatic distribution algorithms based on analytical models provided the optimum distribution identified by the exhaustive search in the CPU+single GPU scenario. Figure 12 shows the relative performance of the distributions found by the analytical models with respect to the best one found by exhaustive search when using the CPU in conjunction with the two K20 GPUs. Both models found again the best point for the FT and MG kernels, and they were just 0.4% slower than the optimal distribution in MMRow. In ShaWa they chose a distribution that is only 2% different from

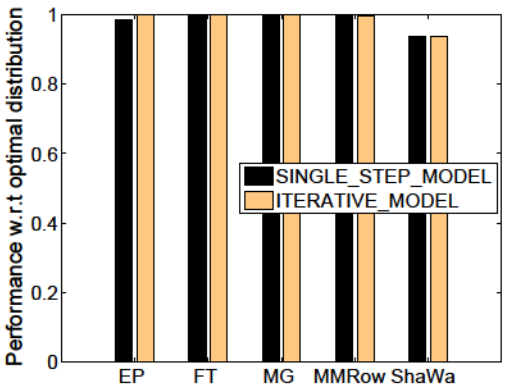

Figure 12: Relative performance of the distributions found by the analytical models with respect to the best one found when using one CPU and two K20 GPUs.

Table 4: Slowdown of the worst distribution with respect to the best one in the two configurations tested.

Environment	EP	FT	MG	MMRow	ShaWa
CPU + GPU	2.76	17.73	51.61	3.98	6.68
CPU + 2 GPU	4.22	17.73	51.61	7.75	10.80

the optimal one (namely, it assigned to the CPU 8% of the total work, while the optimum portion was 6%), but since this is a problem extraordinarily well suited for GPUs [45], this distribution was 6.3% slower than the optimum one, which is still a very good value. Finally, in EP the `ITERATIVE_MODEL` showed that it can better pinpoint the best distribution than the `SINGLE_STEP_MODEL`, as it found the optimum distribution, while the simpler model found a distribution just 1.4% slower. Table 4 shows how many times slower is the worst distribution with respect to the best one for each kernel and configuration considering all the distributions tried by the exhaustive search using steps of 1% of the workload. This table further helps to assess the quality of our balancing algorithms, as we can see that the algorithms achieve optimal or near-optimal distributions in environments in which the worst distribution is between 2.76 and 51.61 times, or in percentages, between 176% and 5061%, slower than the best one.

Regarding the search cost, we used the default HPL configuration that runs each test twice to reduce the measurement noise. This way, since the `SINGLE_STEP_MODEL` requires evaluating a single distribution, while the `ITERATIVE_MODEL` always converged in two, or very seldomly, three iterations, the optimization processes based on analytical models only required between 2 and 6 executions. It also deserves to be mentioned that the non-first one executions of the `ITERATIVE_MODEL` start from a point near the optimal one, making them often much faster than the initial excution. This way, in practice the `ITERATIVE_MODEL` only required 48% more time than the `SINGLE_STEP_MODEL`.

The `ITERATIVE_MODEL` is particularly interesting in the presence of applications in which

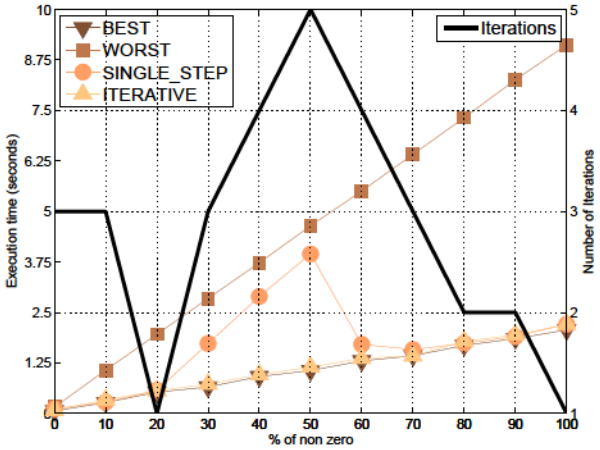

Figure 13: Performance of the distributions found by the analytical models with respect to the best one found (left axis) and number of iterations required for convergence by the `ITERATIVE_MODEL` (right axis) when distributing an unbalanced load between the CPU and a GPU in the K20 system.

the workload is not uniformly distributed across the problem domain. Since the benchmarks considered in the previous experiments are balanced, we designed a simple unbalanced benchmark to compare the behavior for the `SINGLE_STEP_MODEL` and the `ITERATIVE_MODEL` in this kind of problems. Our benchmark considers a single-precision floating-point matrix `a` and it performs a single flop where `a(i,j)` is zero and 1001 flops, without additional memory accesses, where it is a non-zero. We then run this benchmark using input matrices in which 0%, 10%, 20%, ..., 100% of the entries where non-zeros, concentrating all the zeros in the top rows of the matrix and the non-zeros in the bottom rows. Figure 13 shows the results of this experiment when distributing by rows a 4096×4096 input matrix among the CPU and a GPU in the K20 system. The figure represents for each percentage of non-zeros the runtime achieved by the best and the worst possible distributions found using the exhaustive search, as well as by the distributions computed by the `SINGLE_STEP_MODEL` and the `ITERATIVE_MODEL`. We can see that the `ITERATIVE_MODEL` always finds a near-optimal distribution of the work, while the `SINGLE_STEP_MODEL` finds distributions that can be up to four times slower. It is therefore very interesting to use the `ITERATIVE_MODEL` when we know or suspect that the workload may be unbalanced. Regarding the number of iterations required by the `ITERATIVE_MODEL` to converge, it is always between one and a maximum of 5, the average being just 2.8.

Overall, we find these results to be very satisfactory, not only because of the quality of the distributions found and the very reasonable cost of our models, but also because of the simplicity of the API involved.

5. Related work

Most efforts to facilitate the use of multiple heterogeneous devices mainly try to avoid communication APIs in clusters. These proposals provide a programming model in which

a sequential program can allocate buffers and submit tasks to the devices that exist in a cluster. While some of these works [23, 34, 40] are based on CUDA, which restricts their portability, many [3, 10, 14, 19] rely on OpenCL. Most of these latter proposals closely follow the OpenCL API and concepts with some extensions, and thus require a much lower level management than HPL. Exceptions that abstract some details are [3, 14]. Nevertheless, the Many GPUs Package [3] involves compiler directives that must indicate the inputs and outputs of each task and specify synchronization points, or a library that in addition to these specifications explicitly uses contexts and buffers. It also includes a gather-scatter API which, unlike HPL, requires to scatter and gather the data in a single task which is the only one that can work with the chunks. Similarly, libWater [14] relies on explicit kernel creation processes, buffers associated to devices that are explicitly read and written, and synchronizations based on OpenCL-like events, supporting neither subbuffers nor automated kernel partitioning. HPL is currently restricted to the exploitation of the devices in a single node, but it offers a much higher level view. This way n-dimensional arrays rather than buffers in a given memory or device are the objects that users manipulate, being able to work even on subregions of these arrays, and leaving the synchronizations, buffer allocations, data transfers and consistency management to the HPL runtime.

A particularly original proposal is PARRAY [7], a language to represent arrays of data and threads using a very flexible notation together with a novel Single-Program-Multiple-Codeblock programming style. Unlike HPL arrays, PARRAY data arrays are entities located in a specific physical memory and they must be managed (created, updated and deallocated) by hand.

Our work is also related to the task superscalar paradigm, because HPL synchronizations and scheduling are automatically defined by the task data dependencies. Nevertheless, the existing proposals to apply this paradigm to heterogeneous computing [8, 2] require users to explicitly annotate the tasks inputs and outputs, contrary to the fully automated extraction of the dependencies of HPL. They also suffer from long boilerplate codes to use OpenCL and lack mechanisms to split a kernel in parallel subtasks with a single command. In addition [8] requires a special compiler and does not provide convenient array classes with mechanisms to define and operate on subarrays. Partitioning arrays using predefined distributions is allowed by [2], although unlike HPL, it does not allow selecting arbitrary subarrays. A task superscalar project that, like ours, automatically extracts the data dependencies of the parallel tasks is [13], but it only supports regular CPUs.

Multiple devices can also be exploited using compiler directives like OpenMP and OpenACC. However, this alternative suffers from many limitations and differences with respect to our proposal. First, none of these families of directives reaches the degree of portability of HPL. For example, while in this paper we have used GPUs and Xeon Phis under HPL, as of today [30] no OpenACC compiler supports Xeon Phis and there has only been experimental support of OpenMP for GPUs that has not been publicly released [26]. Nevertheless, the most important problem of compiler directives is that they suffer from lack of a clear performance model, reduced user capability to control the result, and strong dependence on the compiler quality. These problems are even more important in accelerators, whose performance is very sensitive to implementation decisions and where users have made an

specific investment to reach higher performance. This way, in [12] out of the four transformations tested that would allow a loop nest to efficiently use a GPU, the compilers would only apply one of them. Similarly, out of three common parallelizable idioms the compilers were only successful with one, the other ones resulting in a sequential execution. They also compared the quality of the code generated by the compilers for 15 Rodinia benchmarks [6] with hand-tuned CUDA versions. While in 6 benchmarks the compiler reached 85% of the performance of the CUDA version, 8 of them did not reach even 50%. The differences are also usually large when full applications are considered. This way, while in [30] the compiler directives were between 10% and 30% slower than the manually developed kernels, and OpenMP clearly outperformed them in the Xeon Phi (although in this case the authors stress that the experiments suggest that there was a performance problem caused by some issue with the architecture or software), [31], [16] and [36] report around 1.5, 3 and 3.5 times more speedup using CUDA than OpenACC in their applications, respectively. The problems of lack of control are well exemplified by [35], which reports a slowdown in their OpenACC version of one benchmark of over 26x that they were not able to correct. HPL does not suffer these problems, as it allows users to totally control their kernels.

Regarding the subject of this paper, contrary to HPL, the current families of directives for accelerators do not automatically maintain coherent the copies of the same data on different memories. On the contrary, they make programmers responsible for the tracking of the state of these copies and the insertion of directives to maintain the coherency. Similarly, they cannot automatically deallocate buffers, of course keeping safe their content, whenever the devices run out of memory. Rather, they just follow the reference counts manipulated by means of the compiler directives inserted by the programmers, making them responsible for the memory management. Also, none of them provides mechanisms for the convenient distribution of arrays and kernels among different devices such as the annotations proposed in this paper, and they totally lack automatic load balancing features like the ones enabled by the use of executions plans. Also, contrary to HPL, these directives require the subarrays to be contiguous blocks of memory (see Sect. 2.7.1 in [32] and Sect 2.15.5.1 in [33]). The only exception to this rule are dynamic multidimensional C arrays based on arrays of pointers under OpenACC, which are rarely used in HPC because of the lower performance they lead to. Besides, in this case each pointer is managed separately, further reducing the potential performance. Finally, in the case of OpenACC, [46] shows that the lack of certain directives makes the exploitation of multiple accelerators much more complex.

A related work is BBMM [35], an automatic multi-GPU memory manager restricted to loop nests with affine bounds and affine array access functions based on compile and runtime components. HPL does not suffer from these limitations and others implied by the required compile-time analysis such as the existence of conditionals whose outcome can only be known at runtime. In fact, several of the benchmarks used in our evaluation do not fulfill the conditions required by BBMM. Also, the tool generates both the host and the kernel code, thus suffering from the limitations of current compiler technology discussed above.

Skeleton libraries [9, 29, 1, 39] are another approach to exploit multiple heterogeneous devices with reduced programming effort. HPL has a much wider scope of application than these tools, as they can only express computations whose structure conforms to one of their

skeletons.

It is also important to remember that none of these approaches enjoys the run-time code generation possibilities enabled by the HPL embedded language, which have been shown to allow the generation of high performance kernels [11].

The fact that our proposal can automatically find a suitable distribution of work among multiple heterogeneous devices is another difference with the preceding works, and it relates it to the works on automatic work distribution. For example, [27] is restricted to CPUs and Nvidia GPUs, only considers a CPU and a GPU, and is based on an offline training, being thus less adaptive and general than our dynamic system, which relies on runtime information and supports any arbitrary combination of devices. A work that shares the first two limitations but uses dynamic measurements is [28]. OpenCL is the base for [15][21], but they rely on offline static models whose construction requires extensive training runs that need to be repeated when there changes in the platform. Another problem of these approaches is that their decisions are based on static code features and straightforward runtime features, thus kernels whose behavior can strongly vary depending on the contents of the input data, either as a whole or in different work-items, are not well suited for them. Finally, [38] is based on profiling and only considers one CPU and one GPU. Although it elaborates on how to extend the proposal to one CPU and multiple identical GPUs, the idea is only tested with a single kernel. Our approach has been designed in a completely general way and it has been successfully validated using kernels with very different nature in a multi-GPU plus CPU environment. Also, contrary to the preceding work we know of, HPL provides a extremely simple and usable API to find and exploit the optimal distribution among any number of devices.

6. Conclusions

One of the biggest problems for the exploitation of heterogeneity is the associated programming complexity, which grows when several devices are in use. This paper describes and evaluates several mechanisms to facilitate the exploitation of multiple devices in a node using a purely library-based approach called HPL that relies on OpenCL to provide portability. The first alternative consists in allowing the use of subarrays as kernel arguments, as well as source and destination of array assignments, which is required even for the implementation of some algorithms in single-device environments, although this is not explored in this paper. The second one consists in defining portions of a kernel to run in parallel in different devices, which can be achieved using a high-level notation that automatically partitions or replicates the kernel arguments in the devices. A third contribution is an execution plan in which the user provides a partitioning function that based on regions precomputed by our library selects the appropriate subarray of each argument to be used in the kernel execution performed in each device. A very relevant part of this last approach are analytical models that automatically determine the best partitioning based on run-time profiling.

The resulting schemes are highly flexible, enjoy task superscalar execution with automatic synchronization and can reduce up to 76.7% the programming effort with respect to streamlined OpenCL baselines. The overheads of our implementation are negligible, while

the absolute performance can be in fact much larger than that of OpenCL thanks to the HPL adaptive runtime, which achieves in our tests up to a 146% speedup with respect to manually developed OpenCL codes. The quality of the work distributions chosen by our execution plans is also outstanding, as they were optimal in most of the experiments, experiencing a maximum slowdown of 6.3% with respect to the best distribution found using an exhaustive search. This way we think that HPL is a very promising approach for the exploitation of heterogeneous systems and it largely benefits from the contributions described in this paper.

Our future work involves adding new mechanisms to HPL in order to further improve programmability, for example integrating more object-oriented APIs in its kernels. We also plan to extend it with notations and a runtime focused on kernel tuning, so that users can easily express optimization possibilities for their codes that the library can explore for them. Finally, further optimizations such as stream processing could be incorporated in the library.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (Projects TIN2013-42148-P and TIN2016-75845-P), by the Galician Government (consolidation program of competitive reference groups GRC2013/055), and the EU under the COST Program Action IC1305, Network for Sustainable Ultrascale Computing (NESUS).

Appendix A. Example with communication between devices

This Section illustrates the programming style based on separate arrays described in Sect. 2.2 and those enabled by the proposals in Sect. 3 using a more complex example than the matrix product. We have chosen for this purpose a stencil code, namely the well-known Game Of Life, whose multi-array version is shown in Fig. A.14. The application simulates an ecosystem through a number of time steps in which the role of input and output array is swapped between two arrays in each iteration. The computation takes the form of a stencil in which the new value of each cell in the next iteration depends on the value of its neighbors in the current iteration. As a result, the computations in distributed memories, among which the arrays are distributed by rows, require a copy of the neighboring area whose update has been assigned to other devices in order to compute the new value of the cells located in the borders of the arrays. This copy is called ghost region and it needs to be updated in each iteration. The most primitive implementation, based on separate arrays, is shown in Fig. A.14. Here the user is responsible for manually calculating the limits in which the original arrays will have to be partitioned, which is the role of the `obtain_sizes` function in line 1, and accordingly allocating the independent arrays required in the loop in line 27. The approach of building the HPL `Arrays` using pointers to the underlying storage of the existing C++ arrays is unsafe here because the `Arrays` would overlap because of the ghost regions, thus potentially overwriting in the borders right data with wrong data when being read from the accelerators. For this reason in this case it is safer to copy in and out

```
1  void obtain_sizes(std::vector<int>& start, std::vector<int>& end, int N, int ndev)
2  {/*Computes the starting and ending row of each subarray needed*/}
3
4  void exec_and_sync(std::vector<Array<int,2>*>& input_a,
5                     std::vector<Array<int,2>*> &output_a, std::vector<Device>& devices) {
6    Array<int,2> tmp_up(1, M), tmp_down(1, M);
7
8    for(int i = 0; i < devices.size(); i++)
9      eval(life).device(devices[i])(*input_a[i], *output_a[i]);
10
11   for(int j = 1; j < devices.size(); j++) {
12     const int nrows_lower = output_a[j-1]->getDimension(0);
13
14     eval(copy_kernel).device(devices[j-1])(*output_a[j-1], tmp_up, (nrows_lower-2)*M, (nrows_lower
           -1)*M, 0, M);
15     eval(copy_kernel).device(devices[j])(*output_a[j], tmp_down, M, 2*M, 0, M);
16     eval(copy_kernel).device(devices[j-1])(tmp_down, *output_a[j-1], 0, M, (nrows_lower-1)*M,
           nrows_lower*M);
17     eval(copy_kernel).device(devices[j])(tmp_up, *output_a[j], 0, M, 0, M);
18   }
19 }
20
21 std::vector<Array<int,2>*> w_in, w_out;
22 std::vector<int> s_s, s_e;
23 std::vector<Devices> devices;
24
25 obtain_subarray_sizes(s_s, s_e, N, devices.size());
26
27 for(i = 0; i < devices.size(); i++) {
28   w_in.push_back(new Array<int,2>(s_e[i]-s_s[i], M));
29   w_out.push_back(new Array<int,2>(s_e[i]-s_s[i], M));
30 }
31 /* copy data from the input array to each Array in w_in (not shown) */
32 for(time_step = 0; time_step < NUM_TIME_STEPS; time_step++) {
33   exec_and_sync(w_in, w_out, devices);
34   std::swap(w_in, w_out);
35 }
36 /* copy data from each Array in w_in to the destination array (not shown) */
```

Figure A.14: Game of Life example written with separated arrays and using multiple GPUs

the data from them, which is not shown and would happen in the lines 31 and 36, where associated comments have been placed. The main loop of the application, in line 32, runs the kernel and performs the exchange of neighboring rows in function `exec_and_sync`, and then swaps the vectors of `Arrays` so that the current output will be the next input and

```
1  void obtain_sizes(std::vector<int>& start, std::vector<int>& end, int N, int ndev)
2  {/*Computes the starting and ending row of each subarray needed*/}
3
4  void exec_and_sync(Array<int,2> &input_a, Array<int,2> &output_a,
5                     std::vector<int> &s_s, std::vector<int> &s_e, std::vector<Device>& devices) {
6    Range all_columns(0, M-1);
7
8    for(int i = 0; i < devices.size(); i++)
9          eval(life).device(devices[i])(input_a(Range(s_s[i], s_e[i]), all_columns),
10                                       output_a(Range(s_s[i], s_e[i]), all_columns));
11
12   for(int j = 1; j < devices.size(); j++) {
13     Array<int,2>& lower = output_a.getSubarray(j-1);
14     Array<int,2>& upper = output_a.getSubarray(j);
15     const int nrows_lower = lower.getDimension(0);
16
17     lower(Range(nrows_lower-1), all_columns) = upper(Range(1), all_columns);
18     upper(Range(0), all_columns) = lower(Range(nrows_lower-2), all_columns);
19   }
20 }
21
22 Array<int,2> w_in(N,M), w_out(N,M);
23 std::vector<int> s_s, s_e;
24 std::vector<Devices> devices;
25
26 obtain_sizes(s_s, s_e, N, devices.size());
27 /* main loop elided */
```

Figure A.15: Game of Life example written with subarrays and using multiple GPUs

vice versa. The function consists basically of two loops in which the first one launches a parallel kernel execution in each device and the second one performs the exchanges of rows between each pair of devices. Without the availability of subarrays, the most efficient way to make this exchange is to run kernels in the devices to copy the appropriate subregions of the output arrays into temporary arrays (`tmp_up` and `tmp_down`) and then invoke again these kernels in the other devices, this time in order to copy out the data from the temporaries to the ghost region. Notice how the automatic coherency provided by HPL for arrays makes it unnecessary to specify the movement of data between devices. For example, after writing `tmp_up` in the device `j-1`, using it as input in the kernel run in device `j` automatically copies the `Array` to that device so that the kernel has a coherent view of it.

Our Game Of Life example is written using subarrays in Fig. A.15. Here the user is still responsible for computing the region of the global array that each subarray has to cover, thus `obtain_sizes` and the related containers for the associated indices are still required. However,a single input and a single output `Array` are needed. The main loop (not shown)

```
1  void exec_and_sync(Array<int,2> &input_a, Array<int,2> &output_a, std::vector<Device> &
       devices) {
2    Range all_columns(0,M-1);
3
4    eval(life).device(devices)(PART1(input_a,1), PART1(output_a,1));
5
6    for(int j = 1; j < devices.size(); j++) {
7      Array<int,2>& lower = output_a.getSubarray(j-1);
8      Array<int,2>& upper = output_a.getSubarray(j);
9      int nrows_lower = lower.getDimension(0);
10
11     lower(Range(nrows_lower-1), all_columns) = upper(Range(1), all_columns);
12     upper(Range(0), all_columns) = lower(Range(nrows_lower-2), all_columns);
13   }
14 }
15
16 Array<int,2> w_in(N,M), w_out(N,M);
17 std::vector<Devices> devices;
```

Figure A.16: Game of Life example written with annotations and using multiple GPUs

is identical, with the exception of the change of the arguments of `exec_and_sync`, which uses the vectors of integers `s_s` and `s_e` with the limits for the indexing of the arrays in order to correctly build the required subarrays. Notice that these vectors were also needed in Fig. A.14 to know the sizes of the different `Arrays` and their mapping with respect to the underlying global arrays. The structure of the function is the same as in Fig. A.14, but now subarrays are selected within the only two arrays, and copies of data can be naturally performed by means of assignments between (sub)arrays, giving place to a cleaner notation. The fact that these subarrays overlap does not give place to incoherencies in the parent arrays `due to` two reasons. First, when a new subarray is generated, it is internally stored as a separate entity in a container held by its parent array so that it can be reused in the future. Second, when two subarrays overlap in their borders, HPL identifies the overlapped areas as ghost regions, automatically identifying which is the area actually owned by each subarray and which is the ghost region. This way, when the user requests to obtain a coherent view of the array, each subarray only updates its own region in the parent. The ghost regions found in the subarrays used in each device are allowed to be inconsistent, being the only case in which manual updates are required to maintain coherency. The reason for this policy is that if HPL automatically strictly maintained the coherency of the ghost regions, it would have to execute in sequence the kernels in the first loop, as each one of them writes to array positions replicated in another device. The `Array` API provides a method `getSubarray` that allows to retrieve the i-th subarray created within an array from its internal container. Our example benefits from this API in the second loop of `exec_and_sync`, which avoids repeating the somewhat more verbose indexings performed in the first loop.

```
1  void partitioner(FRunner& fr, Range rg[2], Array<int,2>& input_a, Array<int,2>& output_a) {
2    fr(input_a(Range(max(0,rg[0].origin−1), min(N−1, rg[0].end+1)), rg[1]),
3        output_a(Range(max(0,rg[0].origin−1), min(N−1, rg[0].end+1)), rg[1]));
4  }
5
6  void exec_and_sync(Array<int,2> &input_a, Array<int,2> &output_a, ExecutionPlan& ep) {
7    Range all_columns(0,M−1);
8
9    eval(life).executionPlan(ep)(input_a, output_a);
10
11   for(int j = 1; j < ep.getNumDevicesInUse(); j++) {
12     Array<int,2>& lower = output_a.getSubarray(j−1);
13     Array<int,2>& upper = output_a.getSubarray(j);
14     int nrows_lower = lower.getDimension(0);
15
16     lower(Range(nrows_lower−1), all_columns) = upper(Range(1), all_columns);
17     upper(Range(0), all_columns) = lower(Range(nrows_lower−2), all_columns);
18   }
19 }
20
21 Array<int,2> w_in(N,M), w_out(N,M);
22 std::vector<Devices> devices;
23 ExecutionPlan ep(partitioner, devices);
```

Figure A.17: Game of Life example written with execution plan and using multiple GPUs

As we can see in Fig. A.16, annotations largely simplify the Game Of Life example. It is no longer needed to compute the beginning and the end of each subarray, and manual indexing is only required for the update of the ghost regions. This latter part can also benefit from the `getSubarray` API because the partitioning generates its associated subarrays in the order in which the devices are stored in the vector provided to the `eval` that triggers the distributed kernel execution.

The version of Game Of Life based on execution plans is shown in Fig. A.17. The exchange of rows is almost identical to the one of the examples based on subarrays and annotations in Figs. A.15 and A.16, respectively. The only difference is that since the execution plan contains all the information on the execution of the kernel, `exec_and_sync` does not need to receive the vector of devices in use as a separate argument, and the number of devices, required to know the number of exchanges needed, is obtained from the execution plan. The kernel execution is automatically distributed in line 9 between the devices provided to the plan using the partitioner in lines 1-4. The example also illustrates in line 23 another convenient way to build an execution plan. This alternative splits the execution between the devices specified in the input vector of devices giving the same weight to each of them. The complexity of this version is apparently very similar to the one of the code in Fig. A.15, but

this is misleading because now the function `obtain_sizes`, whose content was elided, is no longer needed. More importantly, the use of an execution plan makes it trivial to perform load balancing using any of the algorithms provided by HPL.

Appendix B. Coherence and data movement

This appendix describes the internal mechanisms used by HPL to keep a coherent view of the arrays it manages. Our approach provides a single view of each array to the programmer, hiding the fact that each array can have several underlying copies, due to the distributed nature of the accelerators and the host memory, as well as partial copies, due to the usage of subarrays, when this is the case. The only exception to this systematic unified plain view takes place when the programmer selects subarrays that partially overlap along their borders. This situation happens in kernels with stencil patterns that require the replication of neighboring data in the piece assigned to each device, forming what is known as ghost regions. When this happens, HPL does not enforce the strict coherency of the overlapped ghost regions so that it is possible to run in parallel the related kernel in several devices, letting the user in charge of updating the ghost regions when needed, as shown in the detailed example in Appendix A. When the subarrays are brought to a memory that contains the whole array, such as the main host memory, the user does not need to worry about the potentially incoherent ghost regions because HPL knows which is the owned region of each subarray and thus only updates it.

HPL provides the unified view of arrays to the main thread of the application, which orders the execution of the different kernels and array operations. Although these activities can be run in parallel because kernels are run asynchronously, i.e., the main thread does not wait for a kernel to finish when it orders its execution, the user sees their execution as if it happened in a sequential order. Therefore HPL provides sequential consistency [22] to all the accesses to its arrays in the host application, as it is the simplest model to reason about parallel programs. With respect to the internal management of the copies, as we will see along this explanation, the HPL runtime follows a multiple-readers/single-writer policy (MRSW) policy [41] with an invalidation protocol on writes [25] in order to keep a single coherent image. Along the explanation, the terms *valid* and *updated* will be used to indicate that a given copy or buffer has the most up to date version of the data it is associated to. We will also use the term *region* to refer to a set of positions within an array, typically resulting from an indexing with ranges. Finally, an array is said to cover a region of another array when that set of positions is shared by both arrays.

We first describe the main data structures used in the algorithms. Every `Array` keeps a list of the subarrays of it that exist at a given point in time called `subarrays`. Also, all the `Arrays` have a field `parent` that points to their parent, or ϕ if they are not subarrays, as well as three vectors called `buffer`, `valid` and `childrenValid` that are indexed with a device number. The first vector provides the physical buffer associated to that array in that device. The second one is a boolean that indicates whether that buffer has a valid version of the portions of the array not covered by its children in the device. The last vector plays a similar role, indicating whether all the regions covered by subarrays are updated in the

```
1  Array Array::buildArray(rangesList) {
2    tmpArr = subarrays.find(rangesList);
3    if(tmpArr == φ) {
4      tmpArr = newSubarray(rangesList);
5      subarrays.add(tmpArr);
6    } else {
7      if(tmpArr.superSet(rangesList)) {
8        return tmpArr.buildArray(tmpArr.relative(rangesList));
9      }
10   }
11   return tmpArr;
12 }
```

Figure B.18: Array indexing algorithm

device. The host is considered for the sake of these structures as another device, namely, the one with index 0. The `buffer` entry in a device is ϕ if no such buffer exists.

We now describe in turn the three basic top-level algorithms involved in this mechanism: the identification or creation of a subarray, a generic access to an array, and finally, an assignment between arrays. The algorithm in Fig. B.18 shows how HPL builds a subarray when an existing `Array` is indexed with ranges. This is achieved by the member function `buildArray`, belonging to the `Array` class, whose `rangesList` input argument stores the ranges used for the indexing. The pseudocode is written in a C++ style, in which class member functions are invoked on an object of their class and they can directly access the data members of the objects of their class. As a first step, the `subarrays` container of the `Array` is searched in line 2 for a subarray that covers this range. If more than one subarray covered this region the search would return an error, as portions of an array that are common in different subarrays can be in different states in these subarrays. The typical example for this situation are stencil computations, in which the neighboring region is replicated and has different values in the subarray used in each device. The mechanism followed in HPL to avoid this ambiguity when addressing these common regions is to select them by indexing the subarray in which we are interested. This allows in practice the HPL subarrays to overlap, and thus to enable the efficient implementation of stencil computations. If no subarray covers this region, a subarray structure is created and added to the list. If a subarray covers the range specified by the user, it could be the case that this range is a subarray inside a subarray. In this case, detected in line 7, the `rangesList` is adapted so that its indices are relative to the found subarray and the function `buildArray` is invoked on the new parent array. Otherwise `tmpArr` represents the subarray requested by the user and is thus returned by the function.

Whenever an `Array` is used in a kernel, HPL identifies whether the kernel will read, write, or both read and write the data structure, as well as the device where the kernel will be run. Similarly, when an `Array` is accessed in the host, the API knows the kind of access and the host is considered as another device. In both cases, the member function `access`, shown in Fig. B.19 is invoked on the `Array` with the device and access mode requested.

```
1  Buffer Array::access(device, mode) {
2    getBuffer(device);
3    if( (mode.isRead() || mode.isReadWrite()) &&
4        !(childrenValid[device] && valid[device]) ) {
5      refresh(device);
6    }
7    if(mode.isWrite() || mode.isReadWrite()) {
8      markOnlyOwner(device, true);
9    }
10   return buffer[device];
11 }
```

Figure B.19: Main access algorithm

```
1  void Array::getBuffer(device) {
2    if(buffer[device] == φ) {
3      if(parent == φ) {
4        buffer[device] = separateBuffer(device);
5        mapExistingChildren(device);
6      } else {
7        if(parent.buffer[device] != φ) {
8          buffer[device] = map(parent, device);
9        } else {
10         buffer[device] = separateBuffer(device);
11       }
12     }
13   }
14 }
```

Figure B.20: Algorithm for building a buffer in a device

The aim of this function is to prepare the `Array` for the access by the kernel as well as to update its coherency state. The function follows three steps: it builds the physical buffer for the array in the device if it does not exist, it updates the buffer with valid contents if it is outdated and the array is going to be read, and finally it marks the copy in this device as the only valid one for this array if it is going to be written. The `true` argument to function `markOnlyOwner` indicates that the array on which it is invoked is the level of array hierarchy where the write operation takes place. This function is explained later in this Section.

The `getBuffer` function that builds the buffer of an `Array` in a given device is shown in Fig. B.20. If there is already a buffer in the device, there is no need to allocate one. Otherwise, a top level `Array` will require for sure an actual buffer allocation, which is achieved by means of `separateBuffer`. This latter function checks whether there is space for the buffer, performing deallocations of existing buffers if necessary. The deallocation order is LRU in order to favor temporal locality. In this process, buffers that were the only valid copy of a (sub)array would be saved in the host before their deallocation. When a parent Array is allocated in a device, `mapExistingChildren` maps into it the children Arrays that might

```
1  void Array::refresh(device) {
2    if(!valid[device]) {
3      for(lastValidDevice = 0; (lastValidDevice < NDevices)
4                                 && !valid[lastValidDevice]; lastValidDevice++);
5      updateNonChildren(lastValidDevice);
6      valid[device] = true;
7    }
8
9    if(!childrenValid[device]) {
10     for(s = subarrays.begin(); s != subarrays.end(); s++) {
11       if(!s.valid[device] || !s.childrenValid[device]) {
12         s.refresh(device);
13       }
14     }
15     childrenValid[device] = true;
16   }
17 }
```

Figure B.21: Algorithm for updating a (sub)array in a device

already exist in the device in order to free their memory and thus optimize the memory space. Regarding subarrays, they would be also allocated as a separate buffer if their parent does not have a buffer in the device. Otherwise they would be mapped inside this buffer, thus saving memory and facilitating coherency.

The update of a buffer with the current version of an array is performed by member function `refresh`, displayed in Fig. B.21. This function first checks whether the portions of the array not covered by subarrays are valid or not in this device. In the second case, they are updated from the `lastValidDevice` device, where they have their most up to date value by means of the `updateNonchildren` function. In the second step, if not all the subarrays are valid in the device, it iterates on them refreshing those that need it by means of a call of this member function on them. As explained at the beginning of this appendix, in the case of overlaps between subarrays each subarray knows which is the portion of the parent it has to update, therefore avoiding inconsistencies.

The last step related to the array accesses not associated to array assignments is the algorithm to manage the state of the copies due to write operations, shown in Fig. B.22. In the first stage of the function, it invokes itself on the subarrays of this `Array` in order to mark the copy in this device as only valid copy of them. In the second stage the specified device is marked as the only valid source for both the regions not covered by any subarray (`valid` vector) and those covered by the array subarrays (`childrenValid` vector). The third stage updates the state of the parents of this array. This only needs to be done in the topmost level where the write operation takes place. For this reason it is only made in the topmost invocation of `markOnlyOwner`, which is the one that receives a `true istoplevel`, as the invocations to lower levels in line 3 make it `false`. The state that is modified corresponds to the `childrenValid` vectors of these ancestors, which are updated by `propagateChildrenValidUp` to indicate that not all their children are valid in

```
1  void Array::markOnlyOwner(device, istoplevel) {
2    for(s=subarrays.begin(); s != subarrays.end(); s++) {
3      s.markOnlyOwner(device, false);
4    }
5
6    for(i = 0; i < NDevices; i++) {
7      valid[i] = (i == device);
8      childrenValid[i] = (i == device);
9    }
10
11   if(istoplevel) {
12     this_child = this;
13     for(p = parent; p != φ; p = p.parent) {
14       p.propagateChildrenValidUp(this_child, device);
15       this_child = p;
16     }
17   }
18 }
```

Figure B.22: Algorithm for marking a device as only valid location for an array

```
1  void Array::assign(rhs_array) {
2    where_lhs = whereLastUsage();
3    where_rhs = rhs_array.whereValid(where_lhs);
4    copy(rhs_array, where_rhs, this, where_lhs);
5    markOnlyOwner(where_lhs, true);
6  }
```

Figure B.23: Algorithm for array assignments

other devices, and that they could all be valid in this device. This would be the case if the only invalid child before this access were this array (in the lowest level), or its corresponding parent (in the upper levels), if also all the other children of that ancestor of this subarray were valid in this device.

Accesses to arrays in the host have a treatment similar to that of kernels in the devices. Namely, when a (sub)array is read or written in the host, the operations are analogous to those of a read or write access in a kernel run in a device, correspondingly. The library knows the kind of access because the interactions with the data take place through its API, as explained in [43].

Finally, Fig. B.23 summarizes the steps followed for an assignment between arrays, which is the other source of movements and coherency changes besides kernel executions and accesses in the host. The member function `assign` is invoked on the array in the left hand side (lhs) of the assignment, providing as argument the right hand side (rhs) array. Function `whereLastUsage` returns the device where an array, or its most recently used parent array was used for the last time. This device is the most likely location for the next usage of the array, and thus the best place where it should be updated. Function `whereValid` looks for

a device where an array has a valid copy, starting from the device provided as argument so that it has higher priority in the search. If no device has a valid view of the whole array implied, this function builds a valid copy in the priority device, updates correspondingly the validity vectors, and returns the associated id. Once we have decided the source and the target devices, `copy` performs the data transfer between them. Finally, since the destination array has been written with its most up to date version in the destination device, it is marked as the only owner of the lhs array by means of the `markOnlyOwner` member function.

Regarding data transfers, HPL can make them in three different ways. First, whenever the source and the destination are in the same device, an optimized copy kernel is executed in the device to copy the data. When inter-device communications are needed, the adaptive algorithm explained in [44] chooses the best strategy to perform the copy. As mentioned in Section 4.2, the best strategy can rely on using either two transfers in sequence using a temporary buffer in the host memory, or the built-in OpenCL `clEnqueueCopyBuffer` function, which copies data between buffers in the same or in different devices. As shown in [44], while some accelerators favor the first alternative, in others the second strategy offers the best performance.

[1] A. Acosta and F. Almeida. Skeletal based programming for dynamic programming on multiGPU systems. *The Journal of Supercomputing*, 65(3):1125–1136, 2013.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2010 IEEE Intl. Conf. on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, pages 1–7, 2010.

[4] N. Bell and J. Hoberock. *GPU Computing Gems Jade Edition*, chapter 26. Morgan Kaufmann, 2011.

[5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. 16th ACM symp. on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, 2011.

[6] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE Intl. Symp. on Workload Characterization (IISWC 2010)*, pages 1–11, Dec 2010.

[7] Y. Chen, X. Cui, and H. Mei. PARRAY: A unifying array representation for heterogeneous parallelism. In *17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '12, pages 171–180, 2012.

[8] A. Duran, E. Ayguadé, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[9] S. Ernsting and H. Kuchen. Algorithmic skeletons for multicore, multiGPU systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138, 2012.

[10] B. Eskikaya and D.T. Altilar. Distributed OpenCL distributing OpenCL platform on network scale. *IJCA Special Issue on Advanced Computing and Communication Technologies for HPC Applications*, ACCTHPCA(2):26–30, July 2012.

[11] J. F. Fabeiro, D. Andrade, and B. B. Fraguela. Writing a performance-portable matrix multiplication. *Parallel Computing*, 52:65–77, 2016.

[12] S. Ghike, R. Gran, M. J. Garzarán, and D. Padua. *27th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2014)*, chapter Directive-Based Compilers for GPUs, pages 19–35. Springer International Publishing, Cham", 2015.

[13] C. H. González and B. B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475–489, September 2013.
[14] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. LibWater: heterogeneous distributed computing made easy. In *Intl. Conf. on Supercomputing (ICS'13)*, pages 161–172, 2013.
[15] D. Grewe and M. F. P. OBoyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 286–305. Springer Berlin Heidelberg, 2011.
[16] X. Guo, J. Wu, Z. Wu, and B. Huang. Parallel computation of aerial target reflection of background infrared radiation: Performance comparison of OpenMP, OpenACC, and CUDA implementations. *IEEE J. of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(4):1653–1662, April 2016.
[17] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.
[18] T.D. Han and T.S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. on Parallel and Distributed Systems*, 22:78–90, 2011.
[19] P. Kegel, M. Steuwer, and S. Gorlatch. dOpenCL: Towards uniform programming of distributed heterogeneous multi-/many-core systems. *J. Parallel Distrib. Comput.*, 73(12):1639–1648, 2013.
[20] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.
[21] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proc. 27th Intl. ACM Conf. on Supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.
[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.
[23] O.S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *IEEE Intl. Conf. on Cluster Computing and Workshops (CLUSTER'09)*, pages 1–8, 2009.
[24] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proc. 2010 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010.
[25] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.
[26] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. M. Chapman. *9th Intl. Workshop on OpenMP (IWOMP 2013)*, chapter Early Experiences with the OpenMP Accelerator Model, pages 84–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
[27] C-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. 42Nd Annual IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.
[28] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *Proc. 41st Intl. Conf. on Parallel Processing (ICPP 2012)*, pages 48–57, Sept 2012.
[29] M. Majeed, U. Dastgeer, and C. Kessler. Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In *Proc. Intl. Conf. on Parallel and Distr. Processing Techniques and Applications (PDPTA 2013)*, July 2013.
[30] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin. An evaluation of emerging many-core parallel programming models. In *7th Intl. Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'16, pages 1–10, 2016.
[31] M. Norman, J. Larkin, A. Vose, and K. Evans. A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel. *Journal of Computational Science*, 9:1–6, 2015.
[32] OpenACC-Standard.org. The OpenACC Application Programming Interface Version 2.5, Oct 2015.
[33] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, Nov 2015.
[34] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computing*, 40(10):574–588, 2014.

[35] T. Ramashekar and U. Bondhugula. Automatic data allocation and buffer management for multi-GPU machines. *ACM Trans. Archit. Code Optim.*, 10(4):60:1–60:26, December 2013.

[36] A. J. Rueda, J. M. Noguera, and A. Luque. A comparison of native GPU computing versus OpenACC for implementing flow-routing algorithms in hydrological applications. *Computers & Geosciences*, 87:91 – 100, 2016.

[37] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Proc. 2011 IEEE Intl. Symp. on Workload Characterization*, IISWC '11, pages 137–148, 2011.

[38] J. Shen, A. L. Varbanescu, and H. J. Sips. Look before you leap: Using the right hardware resources to accelerate applications. In *Proc. 2014 IEEE Intl. Conf. on High Performance Computing and Communications, (HPCC 2014)*, pages 383–391, Aug 2014.

[39] M. Steuwer and S. Gorlatch. SkelCL: a high-level extension of OpenCL for multi-GPU systems. *The Journal of Supercomputing*, 69(1):25–33, 2014.

[40] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: Compute unified device and systems architecture. In *Eurographics Symp. on Parallel Graphics and Visualization (EGPGV 2008)*, pages 49–56, 2008.

[41] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.

[42] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[43] M. Viñas, Z. Bozkus, and B. B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. *J. of Parallel and Distributed Computing*, 73(12):1627–1638, December 2013.

[44] M. Viñas, Z. Bozkus, B. B. Fraguela, D. Andrade, and R. Doallo. Developing adaptive multi-device applications with the Heterogeneous Programming Library. *The Journal of Supercomputing*, 71(6):2204–2220, 2015.

[45] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J.A. García, M.J. Castro, and R. Doallo. A multi-GPU shallow-water simulation with transport of contaminants. *Concurrency and Computation: Practice and Experience*, 25(8):1153–1169, June 2013.

[46] R. Xu, X. Tian, S. Chandrasekaran, and B. M. Chapman. Multi-GPU support on single node using directive-based programming model. *Scientific Programming*, 2015:621730:1–621730:15, 2015.