CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE

Concurrency Computat.: Pract. Exper. 2017; **00**:2–28

Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Heterogeneous Distributed Computing based on High Level Abstractions

M. Viñas, B. B. Fraguela*, D. Andrade, R. Doallo

Universidade da Coruña, Grupo de Arquitectura de Computadores, Spain

SUMMARY

The rise of heterogeneous systems has given place to great challenges for users, as they involve new concepts, restrictions and frameworks. Their exploitation is further complicated in the context of distributed memory systems, which require the usage of additional different programming paradigms and tools. In this paper we propose a novel approach to program heterogeneous clusters that is based on high level abstractions such as tiles and hierarchical decomposition combined with the powerful APIs that data types and embedded languages can provide in languages such as C++. Rather than building our proposal from scratch, we have implemented it as a natural integration of the existing Hierarchically Tiled Arrays (HTA) and Heterogeneous Programming Library (HPL) projects, the first one being focused on distributed computing and the second one on heterogeneous processing. The result, called Heterogeneous Hierarchically Tiled Arrays (H^2TA), is very intuitive and easy to use thanks to the global view of the data and the single-threaded view of the execution that it provides at cluster level together with the transparency it provides with respect to the management of the heterogeneous devices. An evaluation comparing our proposal with MPI-based implementations shows its large programmability advantages and the reasonable overhead incurred.

Received . . .

KEY WORDS: distributed computing, heterogeneous computing, data parallelism, programming model, runtime system, OpenCL

1. INTRODUCTION

High Performance Computing (HPC) applications and systems are increasingly adopting the usage of heterogeneous devices due to their advantages in terms of performance and power consumption. A clear example of this tendency is the number of heterogeneous supercomputers found in the TOP500 list (http://www.top500.org/), as June 2017 list includes 91 systems that use accelerators/co-processors, up from 64 on June 2014, which means a 42% increase in the last three years. Unfortunately the exploitation of heterogeneity has important programming costs because of the required new restrictions and programming tools that accelerators require. The complexity of the development of applications is even larger in the context of heterogeneous clusters, as their distributed nature requires the usage of yet other tools that are usually intended for low level programming, the most common one being the standard message passing library MPI. Also, these problems are sometimes coupled with lack of portability across different families of accelerators. This latter restriction has been ameliorated by the appearance of the OpenCL standard, which suffers however from yet higher programming costs than other alternatives [32].

Given the growing relevance of heterogeneous distributed memory systems and the large development effort they pose nowadays, the research community has come up with a number of interesting proposals to facilitate their usage [23, 7, 5, 9, 20, 17]. In our opinion, the best answers to programmability problems must combine three main characteristics. First, simple semantics and high levels of abstraction must be provided to the users. This way, in the case of the heterogeneous clusters, alternatives in which the data distributed can be seen as global data structures that cover the whole cluster are preferable to those in which isolated local portions of data must be independently managed. Relatedly, it is desirable to consider the data containers as the units of representation for the data storage rather than the different buffers that can be needed to store such data in different host memories and devices. Also, enabling programmers to focus on a single thread of execution that at

*Correspondence to: Basilio B. Fraguela, Facultade de Informática, Campus de Elviña, s/n. 15071, A Coruña, Spain. Email: basilio.fraguela@udc.es

some points diverges to perform parallel independent computations in different processors is better than controlling independent execution paths in each process following the SPMD programming style typical of MPI-based applications. The second element that must be stressed is a concise and powerful API. In this regard, we favor the use of libraries over new languages and compiler directives because they facilitate code reuse and do not suffer from the limitations of compiler technology. Also, these libraries should naturally exploit the rich semantics associated to object-orientation and polymorphism when they are supported by their host language. The final third property that a proposal of this kind should have is a reasonable performance when it is compared with the low level approaches it seeks to replace or complement.

This paper presents a framework for the programming of heterogeneous clusters that has been designed based on the premises laid out above. It relies on a data type that represents a distributed array with powerful tile-based notation and semantics on which data-parallel operations can be applied. These operations can be run either on the regular CPUs or in the heterogeneous devices of a cluster depending on the specification of the user and they are encapsulated in the data type methods, which make the associated underlying management as transparent to the user as possible. The usage of our data type allows to avoid the SPMD programming style, offering a single-threaded view of the execution, where the parallelism is encapsulated in the array operations on the data type. Communications are also implicit, as they appear either in array assignments that imply portions located in different cluster nodes or in global operations that require communications such as reductions. Our proposal has been developed as an extension of the existing Hierarchically Tiled Array (HTA) project [2], as it provides a data type with all the starting properties required except the support for heterogeneous computing. Rather than reinventing the wheel, we provided the accelerator support for the HTA class by reusing the runtime and integrating the API of the Heterogeneous Programming Library (HPL) project [42] given its portability, good performance and intuitive notation. An initial experience [44] on heterogeneous clusters showed that the use of both libraries in the same application separately provided good results in terms of programmability and performance, although it required some manual management operations and a duplication of handles for arrays, which complicated the programming. This led us to focus our efforts on the development of an integrated solution. The result, which we call the Heterogeneous Hierarchically Tiled Array (H^2TA), is a high level proposal for the programming of heterogeneous clusters that presents important advantages with respect to the strategies that are currently used while presenting negligible overheads with respect to them.

The rest of this paper is organized as follows. First, our proposal is motivated through a discussion of the related work in Section 2. This is followed by an introduction to the HTA data type in Section 3 and a discussion on the Heterogeneous Programming Library and our first experiences combining it with HTAs to program heterogeneous clusters in Section 4. Our new integrated proposal is then presented in Section 5. Implementation details of the H^2TAs are provided in Section 6, which is followed by an experimental evaluation in Section 7 and our conclusions in Section 8.

2. RELATED WORK

There has been a considerable amount of work on the enhancement of the programming of heterogeneous clusters in the past few years. The approaches that operate at the lowest level take existing communication tools such as MPI and facilitate their integration with heterogeneous frameworks [23, 1, 21] keeping the same level of abstraction. When the accelerator is a Xeon Phi, it is possible to only rely on MPI, potentially combining it within each node with traditional shared memory programming tools such as OpenMP [26]. However, interesting efforts to program clusters of Xeon Phi only based on compiler directives such as OmpSs [9], discussed later, have also been made [10]. A more ambitious approach is to enable the execution of unaltered or very slightly modified heterogeneous applications written using well-known frameworks such as CUDA [25, 33, 37, 41] or OpenCL [3, 4, 7, 12, 17, 19, 20, 35, 47] on distributed systems, so that they can exploit remote accelerators in clusters, grids and the cloud, typically by virtualizing them. Since the main purpose of these proposals is not to provide higher level semantics for the programming of the distribution resources, but to simplify their exploitation as much as possible in application developed using CUDA or OpenCL, most of these tools expose abstractions, and thus

APIs, that are at the low level of these tools, being in fact nearly identical in most cases. This also means that these approaches focus on the usage of the distributed accelerators, the exploitation of the remote CPUs being only available to the proposals based on OpenCL and requiring their use to be also based on their handling as OpenCL devices. The most outstanding efforts of abstraction in this family of proposals have been performed by the Many GPUs Package (MGP) [7] and libWater [17]. MGP allows to run unmodified OpenCL applications in clusters on top of MOSIX VCL, a cluster-wide virtual implementation of OpenCL. It also supports a C++ object-oriented API that, while simplifying the process, is still based on low-level concepts such as buffers, contexts or tasks with explicit enqueuings and synchronizations. In addition it presents important restrictions to the processing of distributed data; for example only a scatter and a gather communication patterns are supported and only one task can be associated to the data they distribute. MGP also has task-based OpenMP-like directives that are restricted to the execution of individual kernels in each node. Regarding libWater [17], it relies on explicit kernel creation processes, buffers that must be manually associated to specific devices and that require the user to specify the read and write transfers on them, as well as synchronizations based on events. Therefore it is at a considerably lower level than H^2TAs with their globally distributed data structures that abstract away any idea of buffer and make totally transparent all the management related to heterogeneous devices.

At an upper level of abstraction we find proposals based on skeletons, compiler directives and language extensions. The main restriction of skeletons, represented in this area by [11, 27], is that they support a specific set of computational patterns, thus not being universal solutions.

A proposal to program heterogeneous clusters based on compiler directives is the task-based data flow programming model of OmpSs [9], which lacks the H^2TA fine-grained control over device selection, globally distributed data structures and implicit data-parallelism across cluster nodes. The programming model in [38] is very similar to that of OmpSs, thus having similar differences with respect to H^2TAs, additional ones being that it is only intended to specific kinds of applications and CUDA-based accelerators. Also restricted to this kind of accelerators we find [39], which relies on language extensions to CUDA in combination with explicit copy requests in a distributed shared memory model in order to support the CUDA programming model in clusters of GPUs and multi-GPU systems.

Another relevant project with APIs based on libraries, directives and language extensions that supports two programming models for clusters is StarPU. While [6] operates at a lower level exposing MPI-like messages to the programmer, [5] task-based approach is quite similar to that of OmpSs. As a result it shares similar limitations, the most important difference being that it allows to define distributed arrays as a collection of tiles located in different nodes, but lacking all the tile-level semantics, advanced syntax, collective manipulation capabilities and data-parallel operations of H^2TAs. Another interesting alternative that relies on language extensions [30] and compiler directives is XcalableACC [31]. The XMP extensions are in charge of providing distributed arrays with a small subset of the array operations of H^2TAs and without their tile-level features. As for heterogeneity, the fact that XcalableACC relies on OpenACC [34] reduces its portability compared to OpenCL [28], which we use as backend, and sometimes also the performance, as OpenACC has been found to often offer considerably less performance than manually optimized kernels [16]. In addition, unlike H^2TAs OpenACC requires explicit annotations for data movements between each host and its device(s).

A final advantage of our proposal is that the reliance on HPL rather than on OpenCL allows H^2TAs to benefit from its runtime code generation capabilities, which facilitate the development of high performance kernels [13].

3. HIERARCHICALLY TILED ARRAYS (HTAS)

The HTA data type [2] represents an array that is optionally partitioned into tiles that can be either conventional arrays or lower level HTAs. The tiles can express both locality and parallelism, as different tiles can be processed in parallel following data parallel semantics that are embedded in the methods of the class. Also, the tiles of an HTA can be stored in a single node or they can be distributed across the nodes of a cluster. In this latter case the top level tiles are the ones that are distributed. For example, Fig. 1 shows how to create in C++ an HTA that is divided into 2×2 tiles

```
CyclicDistribution dist({2, 2});
auto h = HTA<float, 2>::alloc({ {7, 7}, {2, 2} }, dist);
```

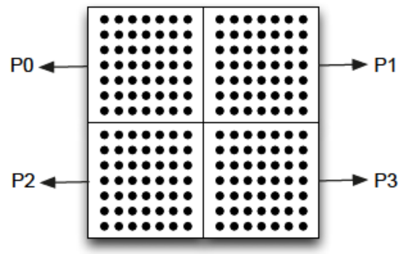
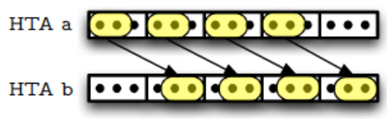


Figure 1. HTA creation



```
auto a = HTA<double, 1>::alloc({{3}, {5}});
auto b = HTA<double, 1>::alloc({{3}, {5}});
b(Triplet(1,4))[Triplet(1,2)] =a(Triplet(0,3))[Triplet(0,1)];
```

Figure 2. HTA complex indexing and assignment example

of 7×7 single-precision floating point elements each. The HTA is built in a distributed fashion using a cyclic distribution of its tiles on a grid of 2×2 processors that is specified by the object `dist` built in the first line. As a result each tile is placed in a different processor Pi, resulting in the mapping illustrated in the figure.

HTAs support two indexing operators, the parenthesis, which operate at tile level, and the brackets, that operate at element level. In addition, these operators support scalars for a single point and `Triplets` for ranges of elements. The resulting scheme is very flexible, as both kinds of indexing can be combined and applied at different levels. This is illustrated in Fig. 2, which shows how to select the first two elements of each one of the first four tiles of an HTA `a` in order to copy them to the last two elements of the last four tiles of a destination HTA `b`. Notice that in the case of distributed HTAs assignments imply communications if the tiles involved are located in different nodes. The example also illustrates the intuitive notation for point-wise operations on HTAs. For example, given the HTAs `a`, `b` and `c`, and the scalar `k`, `a = b + k * c` will add each element of `b` and the associated element of `c` multiplied by `k` into `a` on the condition that these HTAs are conformable [2], i.e., that they have the same topology (number of levels and shape of each level) and the corresponding tiles in the topology have sizes that allow to operate them.

```
void saxpy(HTA<float,1> y, HTA<float,1> x, HTA<float,1> alpha)
{
  int size = x.shape().size()[0];
  for(int i = 0; i < size; i++)
    y[i] = alpha[0] * x[i] + y[i];
}
...
auto x = HTA<float, 1>::alloc({ {N}, {M} });
auto y = HTA<float, 1>::alloc({ {N}, {M} });
auto alpha = HTA<float, 1>::alloc({ {1}, {M} });
...
hmap(saxpy, x, y, alpha);
```

Listing 1. Parallel application of a user-defined function to the tiles of HTAs

HTAs also provide many methods that allow to express more complex computations as well as higher-order operators that support the application of user-defined computations. For example, the function `hmap` applies in parallel a user function to the tiles of an HTA. When several HTAs are provided to `hmap`, each parallel invocation operates on the corresponding tiles of the input HTAs. In this situation, if the HTAs are distributed the associated tiles should be located in the same node. This way in the example in Listing 1, if we assume that `x`, `y` and `alpha` are distributed in the same way, function `saxpy` is applied in parallel to their tiles 0 in the node that owns them, their tiles 1 in their associated node, and so on. The example also illustrates that `hmap` requires that the input HTAs have the same top level structure so that their tiles can be matched, but the internal structure and size of those tiles can be different. Namely, while in Listing 1 all the HTAs have a single dimension and `M` tiles, the tiles of the HTA `alpha` have a single element, while the other ones have `N`.

Regarding the HTA implementation, the one used in this paper uses MPI as backend for the communications. This way, users spawn as many processes as desired using the same system as in any other MPI application, but the code of their application only interacts with the HTAs. The HTAs are built at run-time using the `alloc` function seen in our examples. The construction of each HTA should be performed with the same arguments by all the participating processes so that it is coherent. This builds in each process a separate HTA object located in its private memory. These objects keep the same global information on the dimensions of the top level tiles and their placement in all the processes, but the object of each process only contains the low level tiles that are mapped to the associated process. The top-level operations that affect distributed HTAs are observed by all the processes because every process executes the whole program. In each one of these operations, the process performs the computations that are associated to its local tiles. In statements with communications, since all the processes know the location and size of every tile, the HTA runtime can use a two-sided communication model where the synchronization is implicit because communication always goes from the producer to the consumer. This way eager consumers have to wait for delayed producers. As a result there is no need for synchronizations before or after a parallel computation; synchronizations take place on demand when data from other processor is needed. These communications are performed using asynchronous MPI messages whenever the runtime detects that it is safe to do so. Similarly, when the library has to perform communications that obey to a typical pattern provided by MPI collective communications, the runtime relies on them in order to optimize the performance. A detailed discussion on the implementation of HTAs can be found in [15].

As we can see, in the context of a cluster HTA programmers manipulate a data type that represents a whole data structure distributed on the cluster under a given specification. All the parallelism is encapsulated in the tile-level parallel operations supported by the data type, which can apply both standard and arbitrary user-defined functions. This way, users have a single-threaded view of the execution coupled with a global view of the data. As for communications, they are conveniently expressed by means of either assignments between tiles located in different nodes or collective operations provided by the data type. This gives place to a high level programming style that offers great programmability advantages with respect to the traditional MPI-based programming of clusters. Unfortunately, HTAs lacked until now of an integrated mechanism to exploit heterogeneity in their applications, which is the subject of this work.

4. A FIRST APPROACH TO HETEROGENEOUS COMPUTING USING HIERARCHICALLY TILED ARRAYS

Motivated by the growing usage of specialized coprocessors in HPC clusters, we seek to provide users with high level approaches to program heterogeneous clusters. We propose to explore answering this problem using HTAs because of their excellent properties for parallel distributed computing. This requires extending them to support heterogeneity. A first problem to tackle is the diversity of accelerators and frameworks to program them. Since we wanted our proposal to be as uniform and portable as possible, we decided to base our work on the OpenCL standard, as it is supported by all the most popular accelerators. Also, since there are several tools that facilitate the use of OpenCL in C++, it would not make sense to implement the HTA OpenCL support from scratch, but rather to rely on some of the existing projects. Given the discussion on OpenCL support in C++ in [14, 46, 44] we took as basis the Heterogeneous Programming Library (HPL), from which we take its runtime and some ideas for the notation of our H^{2}TA proposal. For this reason, we will first describe the basics of HPL (Section 4.1) as well as the main drawbacks of the separate use of HTAs and HPL in the same application for programming heterogeneous clusters (Section 4.2), already evaluated in [44]. Our new integrated proposal will be then discussed in Section 5.

4.1. Heterogeneous Programming Library

This framework, which is available at `http://hpl.des.udc.es`, allows to exploit heterogeneous computing in C++ on top of OpenCL. In HPL the main application runs in the host, while the pieces of code that are run in OpenCL are functions, called kernels, which can be expressed either directly in C++ by means of the HPL embedded language [42] or in native OpenCL C [46]. Both alternatives are illustrated in Listing 2 using a SAXPY kernel, which is implemented using the HPL language in lines 1–4 and a string with the OpenCL C version in line 6. In both cases, the host-side data to to use in the kernels, defined in line 10, must be contained in a data type provided by the library called `Array`. This is a class template that receives as arguments the type of the elements stored in the array and the number of dimensions of the array, which are `float` and one in SAXPY,

```
 1  void saxpy(Array<float,1> y, Array<float,1> x, Float alpha)
 2  {
 3      y[idx] = alpha * x[idx] + y[idx]; // idx is an HPL variable that contains the global thread ID
 4  }
 5  ...
 6  const char *string = "__kernel void saxpy(__global float *y, __global float *x, float alpha) { ... }";
 7
 8  void saxpy_handle(InOut< Array<float,1> > y, In< Array<float,1> > x, Float alpha) {}
 9  ...
10  Array<float, 1> x(1000), y(1000);
11  float alpha;
12
13  eval(saxpy)(y, x, alpha);
14
15  nativeHandle(saxpy_handle, "saxpy", string);
16  eval(saxpy_handle)(y, x, alpha);
```

Listing 2. HPL example code

respectively. As we see in line 1 this type is also used to represent arrays in the heterogeneous kernels written using the HPL embedded language, while scalars have types such as `Int`, `Float`, etc. Kernels are invoked using the function `eval` followed by the kernel name and arguments as lines 13 and 16 show. This requires associating a function to the kernels that are provided by means of OpenCL C strings, which is achieved by means of the function `nativeHandle`, illustrated in line 15. The associated function, shown in line 8, labels its arguments to inform HPL not only on the type, but also on whether each argument is an input, an output, or both. HPL kernels do not require this labeling because HPL automatically analyzes them to extract this information. This knowledge allows HPL to automatically manage the data transfers between the host and the devices giving transparent coherency for each `Array` across the different memories in which it may be used by the user. Also, the library runtime achieves this using the minimum possible number of transfers, getting a performance identical to that of OpenCL based applications.

The semantics of kernel executions in HPL is analogous to that of OpenCL. This way, each kernel is executed in parallel by a number of threads determined by a global index space or workspace. Users can also optionally define a local index space in order to define the size of the work-groups, which are teams of threads that can synchronize by means of barriers and share a fast scratchpad memory called local memory. Although not illustrated here for space reasons, HPL kernel invocations allow users to specify the desired global and local workspaces as well as the device where the execution should take place.

As we can see, HPL largely improves the usability of heterogeneous systems replacing the verbose host API of OpenCL [32] with high-level abstractions such as system-wide coherent `Arrays` and kernels, which in turn allow to hide many low level concepts such as contexts, programs, or buffers, as well as tasks such as kernel compilations, data transfers or synchronizations.

4.2. Separate use of HTA and HPL

A first approach to program heterogeneous clusters taking advantage of HTAs and HPL would be to use both libraries separately in the same application. This possibility was successfully evaluated in [44], where the steps to make them properly work together were explained. Despite the positive results, this decoupled design presented significant limitations. The most relevant ones are related to the management of the two kinds of array containers used by both libraries independently. This lack of integration forces users to manually maintain two memory spaces for each regular array and to use *ad-hoc* HPL arrays as a workaround in order to perform efficient copies of regions of those arrays. Namely, these applications require global HTAs to partition the data and perform communications among processes. Also, locally at each process, the kernels are executed in the accelerators by means of HPL, which requires the user to convert the local data of HTAs into local per-process HPL Arrays. Similarly, if the results of kernel executions have to be communicated through the mechanisms provided by HTAs, the HTA arrays have to be manually updated with the data of the associated HPL Arrays. In addition, the update of subregions of HTAs with the data of the HPL Arrays also suffers the lack of integration of the involved libraries. A possibility would be to update the whole host side of the HPL Arrays, which could have an unacceptable overhead, as it involves copying all the data when only a subset is needed. HPL also provides mechanisms to copy portions of HPL arrays between the devices and the host, but while they avoid this performance overhead, they involve additional coding and thus higher cost in terms of programmability.

```
1  auto hta_x = HTA<float, 1>::alloc({ {N}, {M} });
2  Array<float,1> hpl_x(N, hta_x({MY_ID}).raw());
3  auto hta_y = HTA<float, 1>::alloc({ {N}, {M} });
4  Array<float,1> hpl_y(N, hta_y({MY_ID}).raw());
5  float alpha;
6
7  eval(fillin_x)(hpl_x);
8  hmap(fillin_y, hta_y);
9  eval(saxpy)(hpl_y, hpl_x, alpha);
10
11 hpl_y.data(HPL_RD); //Brings y data to the host
12 float accum = hta_y.reduce(plus<float>());
13
14 if(MY_ID < M − 1) {
15   eval(user_GPU_kernel)(hpl_y, accum);
16   hpl_y.data(HPL_RD); //Brings y data to the host
17 }
```

Listing 3. HTA+HPL example code

Many of the problems that arise with this programming style are exemplified in Listing 3, which uses the SAXPY computation also illustrated in the preceding examples. It assumes that there are M parallel processes and that in each process we build a tile of N elements for each one of the two vectors involved, `x` and `y`. Besides the `hta_x` and `hta_y` HTAs that represent these distributed memory structures, we also need in each process an `hpl_x` and an `hpl_y` HPL Array. These HPL containers are associated to the corresponding local tile of the global HTA, and their purpose is to allow to operate on that tile in an accelerator. As explained in [44], each HPL Array has its host-side storage in the same memory area as the associated tile, which is obtained by applying the method `raw` to the local tile, that is, the one with the index `MY_ID`, which is the global identifier of the process. The example assumes that the initialization algorithm for `x` is more efficiently run in the accelerator than in the regular CPU, while the opposite happens for the initialization of `y`. This way, line 7 fills `x` using an HPL `eval` on its HPL Array, while line 8 applies a traditional HTA `hmap` on `hta_y`. The SAXPY computation takes place in line 9 in the accelerator. HPL knows that `hpl_x` was initialized in this device, and thus no data movement is required for this array. Nevertheless, `hpl_y` has never been been used in the accelerator, and thus the runtime builds a buffer for it in the device and fills it with the host-side version, which matches the storage of the local tile of the HTA `hta_y`. After this point the only valid copy of this array is the one contained in the buffer image of `hpl_y` in each accelerator. For this reason, and since there is no automatic communication between the runtimes of HTA and HPL, when we want to make some operation using `hta_y`, we have to explicitly request the update of the host-side copy of `hpl_y`, which as we have just explained is associated to the same memory as the local tile of `hta_y`. As detailed in [44], this is achieved using method `data` on this HPL Array, indicating the kind of access which will be performed; in this case only a read access to perform a reduction across the whole HTA in line 12. As a final step, the example assumes that some computation must be performed in the device in all the tiles of `hta_y` except the last one. The natural way to do this using HTAs would have been to select the required tiles, that this, those in the range $[0, M-2]$, and then operate on the selected HTA using `hmap`. Because this function only operates in the CPU and we want to perform the operation in the accelerator, this is not possible. Instead, the user has to take a local view of the problem and apply a SPMD programming style, asking the processes with the appropriate `MY_ID` to perform the computation in their local `hpl_y` HPL Array. Finally, this container is updated again in the host memory, as the example assumes that further computations will require to operate on it using HTAs.

5. HETEROGENEOUS HIERARCHICALLY TILED ARRAYS

The new data type Heterogeneous Hierarchically Tiles Array (H^2TA) avoids the shortcomings described in Section 4.2 thanks to a total integration of HTA and HPL. We will describe the properties of H^2TAs illustrating them with Listing 4, which shows the high level programming style enabled by our extension for the example shown in Listing 3.

5.1. A single unified data type

H^2TAs can exploit the general CPUs of a cluster using the same mechanisms as HTAs. In addition, they allow to use the heterogeneous devices available by means of kernels defined using any of the two strategies explained in Section 4.1. In both situations the H^2TAs, which have the same syntax as the original HTAs, are the only data structure required in the host side, while the heterogeneous kernels are written using the HPL `Array` data type (or OpenCL C strings) because no hierarchical sub-partitioning or tile-level manipulation is supported inside them. This representation allows

```
auto x = HTA<float, 1>::alloc({ {N}, {M} });
auto y = HTA<float, 1>::alloc({ {N}, {M} });
float alpha;

evalHTA(fillin_x)(x);
hmap(fillin_y, y);
evalHTA(saxpy)(y, x, alpha);

float accum = hta_y.reduce(plus<float>());

evalHTA(user_GPU_kernel)(hpl_y(Triplet(0, M−2)), accum);
```

Listing 4. Example from Listing 3 written using H^{2}TAs

to seamlessly mix in the same application parallel computations that are run in the CPUs and operations that are performed in the accelerators. This can be seen in Listing 4, where only HTA objects, which are actually H^{2}TAs from our new library, are built. The syntax is the same because H^{2}TAs keep the same name and creation process as the original HTAs. Also, their usages in the CPUs as well as the communications by means of assignments or high level collective operations follow exactly the same notation as in the original HTA.

5.2. *Heterogeneous computing*

The API for the heterogeneous executions is based on that of HPL because it facilitates the specification of details such as the kernel global and local spaces when the default values are not suitable or the best ones. This way, the most important component of the new API is the function `evalHTA(`f`)`, where f is the C++ function associated to a heterogeneous kernel, which plays a role analogous to that of `eval(`f`)` in HPL, but accepting as inputs H^{2}TAs or scalars. Just as in `hmap`, the H^{2}TAs should have the same top-level structure, that is, number of dimensions and top level tiles per dimension, so that the associated tiles of each one of the H^{2}TAs would be processed together in the same kernel execution. As we can see in Listing 4, operations on H^{2}TAs can be performed interchangeably either in the CPU, using the traditional HTA `hmap`, or in the accelerator, using the new `evalHTA` mechanism, all the complexity (buffer creations, transfers, kernel compilations, etc.) being hidden from the user. The heterogeneous kernels can be implemented by means of any of the two mechanisms exemplified in Listing 2 using exactly the same notation. For example the `saxpy` kernel used can be the one from Listing 2. Also, in between `evalHTA` and the kernel arguments it is possible to insert the same modifiers as in the HPL `eval` invocations in order to specify the workspaces and devices to use. Some improvements have been performed to enhance the programmability of these modifiers. For example, the device specification allows to provide a kind of device (GPU, CPU or accelerator) rather than a specific device. In this case the tiles are processed in their home node using devices of that kind. If there are several tiles and more than one device, the tiles in the same node are evenly distributed on the existing devices to maximize the parallelism of the runtime.

5.3. Automatic coherency

Following the spirit of both the HTA and the HPL projects, the integration automatically keeps a coherent view of the H^2TAs across the distributed memory nodes, the CPUs and the devices of the cluster, avoiding explicit copies. In order to achieve this, the runtime automatically monitors any access to H^2TAs, ensuring that correct versions of the data are always used in any computation, no matter where it is performed. This is reflected in Listing 4 by the lack of any kind of statement to maintain the coherency.

5.4. Indexing

As exemplified by the last line in Listing 4, one can choose to operate on a subset of the tiles of any H^2TA, and thus only in some nodes of the cluster. In addition, although not illustrated in this example, in each tile used it is possible to choose between operating either on the whole tile or only on a portion by using the high level indexing notation of HTAs. With these properties, the requirements in Listing 3 to swap between global and local view of the data along our application and to resort sometimes to a SPMD programming style are gone, the outcome being noticeably cleaner and more maintainable.

5.5. Summary

As we can see, the resulting programming style is very powerful and easy to use thanks to the intuitive and simple semantics of H^2TAs. Users manipulate the abstract arrays required by their application rather than the underlying different copies of them that are required because of the disjunct memories of each host and its devices. The data objects seen by the programmer enjoy a simple sequential consistency model [22] that is automatically provided by their data type. This feature, coupled with the single threaded view of the programming model and the global view of the distributed data structures that H^{2}TAs inherit from HTAs, largely simplify the reasoning about the parallel applications.

6. IMPLEMENTATION DETAILS

H^{2}TAs are basically HTAs that have been extended with HPL Arrays associated to their tiles. Just as HTAs only store in each process the tiles that belong to it, H^{2}TAs only keep in each process the HPL Arrays associated to the tiles owned by the process. The runtime exploits the ability of HPL to place the host-side storage of HPL Arrays in any memory location, thus internally applying the policy explained in Section 4.2 of placing the host-side memory of the HPL Arrays it builds in exactly the same location as their associated HTA tiles in order to avoid memory copies.

The copies of the same tile that are used in different memories are managed by the runtime under a multiple-readers/single-writer (MRSW) policy [40] with an invalidation protocol on writes [24], which together with a lazy copy policy that only updates a copy when it is actually required, minimizes the transfers between the host and the devices. A lazy policy is also applied to the creation of buffers for the tiles, so that they are only allocated in the devices when necessary. This way, H^{2}TAs that are only used in the host CPUs do not require more resources than the original HTAs. The cost of data copies was also reduced to the minimum possible one by ensuring that whenever only a portion of a tile is required in a memory where it is outdated or inexistent, only that region is copied, rather than the whole tile. Relatedly, the H^{2}TA runtime remembers which portions of each tile are updated or outdated in each memory, so that the coherency mechanism granularity dynamically adjusts to the size of the tile regions manipulated by the user, which is needed to ensure a minimum number of transfers with the smallest possible cost. The coherency between the operation performed on the CPUS, either computational or due to communications, and the ones performed in the heterogeneous computations is kept by means of the mechanism explained in Section 4.2, with the difference that now the H^2TA runtime is the one in charge of the invocations to the `data` method of the HPL Arrays in order to kept this coherency.

Most optimizations of the H^2TA runtime are inherited from the runtimes it is built on. For example, the HPL runtime provides critical performance optimizations such as the caching of buffers and kernels to avoid repetitive creation processes, while the HTA implementation provides performance enhancement techniques such as the caching of HTAs or asynchronous communications between nodes [15]. Finally, just as the libraries it integrates, H^2TAs also heavily rely on the compile-time polymorphism and optimizations enabled by C++ templates [8] rather than in the more expensive dynamic polymorphism also supported by this language.

At this point it can be also interesting to discuss how to implement this proposal on top of CUDA given its popularity. The implementation would be much simpler than the one presented here, since CUDA naturally provides for single-source applications and a cleaner and simpler host API than OpenCL. Two main changes would be required. The first one would consist in changing the host API used used inside HPL to initialize the heterogeneous environment, manage memory and transfers, etc. by the corresponding and usually much simpler equivalents of CUDA. The second one would be writing the kernels as standard CUDA kernels and compiling their code with the CUDA compiler, so that the runtime could request their execution.

7. EVALUATION

A high level approach to program a system must show programmability improvements with respect to existing alternatives to motivate their interest. Also, its abstractions must incur in reasonable performance costs. Thus, both sides of the problem are tackled in this evaluation. In [44] we already showed that the separate usage of HTA and HPL, which we call for short HTA+HPL, largely improved the programmability of heterogeneous clusters while incurring in negligible overheads. This way, in order to better assess the advantages of the integrated H^2TA with respect to HTA+HPL,

Table I. Benchmarks characteristics.

Benchmark	SLOCs host	Unique invocation	Repetitive invocation	Data exchanges
EP	248	1 kernel		final reduction
FT	1263	3 kernels	7 kernels	all to all
Matmul	184	1 kernel		none
ShWa	386		3 kernels	stencil and reduction
Canny	209	4 kernels		stencil

our evaluation takes as baseline MPI+HPL versions, so that the improvement that HPL means with respect to the usage of standard OpenCL bindings is already present in the baseline.

The main characteristics of the benchmarks used in the evaluation are shown in Table I, where the first column represents the number of source lines of code excluding comments and empty lines (SLOCs) of the host side of their baseline HPL+MPI version. The size of the kernels has been here dismissed because the H^{2}TA versions use exactly the same OpenCL kernels, so they play no role in the comparison. The remaining columns contain the number of kernels that are invoked just once during their execution, the number of kernels that are invoked inside loops and the nature of the data exchanges between processes they have.

As we can see, these programs present very different patterns, going from codes with no exchange of information among processes to iterative applications with several data exchanges in each iteration. EP and FT are two NAS Parallel Benchmark (NPB) implemented in OpenCL in [36]. EP is a embarrassingly parallel application that finishes with reductions. FT computes the Fourier Transform of a 3-D array along its three dimensions. Since the array is partitioned along one of its three iterations, it needs to be rotated to complete the Fourier Transform for the remaining dimensions. This requires an all-to-all pattern of communications among the processes. Matmul is a single precision floating point dense matrix product distributed per rows among the processes involved. ShWa is a shallow water simulator with pollutant transport parallelized for multiple GPUs in [43]. This application computes the evolution of a mesh of volumes on time, where each volume has several parameters associated such as the amount of pollutant or the strength of the oceanic currents in each direction. Each time iteration step executes three kernels in sequence and requires exchanges between GPUs both to correctly update the volumes that are in the frontiers of the region assigned to each GPU, as well as to perform reductions necessary to compute global values. The last application used in this evaluation is an algorithm aimed at finding edges in images called Canny, which consists of four stages, each one of them implemented in a kernel. The first stage applies a Gaussian filter to smooth the image. Then, the edges are highlighted with an edge detection benchmark (e.g. Gauss, Prewitt, . . .). This is followed by an edge thinning technique and finally the variety of the output image is reduced with a double threshold kernel. It deserves to be mentioned that the stencil kernels developed in HPL in this paper were not based on the mechanisms described in [45], which were developed later.

7.1. Programmability

The ideal way to compare several programming approaches from the point of view of the effort they require from the users is to ask teams of programmers to try them and compare their opinions and time taken developing some applications using them. This is usually difficult to achieve, so we have resorted to three objetive metrics automatically extracted from the source code to estimate the programmability. The first one is the SLOCs, already introduced in the description of Table I. The second one is Halstead's programming effort [18], which is a value computed by means of a reasoned formula that takes into account the total number of operands (constants and identifiers) and operators (symbols that affect the value or ordering of the operands) in the code, as well as the number of unique operands and operators. The third one is the cyclomatic number $V = P + 1$, where P is the number of decision points or predicates, which was proposed as a measurement of code complexity in [29], as the more branches and conditions a program has, the more complex it is.

Figure 3 shows the reduction of the three metrics just described in applications written using separately HTA and HPL (HTA+HPL) and the proposed H^2TA library, compared to the baseline counterparts written using MPI and HPL (MPI+HPL). The measurements are based on the host side of the applications, since kernels are identical in the three versions. There are two kinds of benchmarks attending to the strength of the reduction. In the programs that have no or very little communication, which are EP and Matmul, the programmability of the three versions are

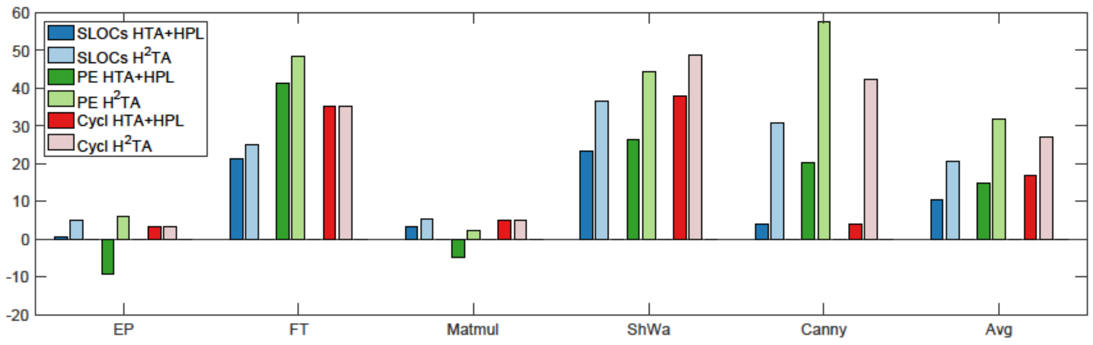


Figure 3. Reduction of programming complexity metrics of HTA+HPL and H^{2}TA programs with respect to versions based on MPI+HPL (higher is better).

very similar because the sources have few differences and HPL already provides very good programmability metrics to the baseline for the exploitation of heterogeneity. However, while H^{2}TA always obtains a positive result, the HTA + HPL version requires more programming effort than the baseline because of the duality of the arrays used. This bad behavior is mitigated in the rest of the benchmarks because of their larger complexity. The other group consists of the benchmarks with more complex communication patterns, which make the applications benefit more from the high level semantics of the HTAs. In the case of ShWa and Canny this complexity is mainly due to the management of the ghost regions needed in these benchmarks with distributed stencil computations. Finally, the rotation of the 3D array that requires FT, which implies an all-to-all communication coupled with transpositions, is well covered by the HTA interface, as it includes a rich set of global collective operations. In this second group, although HTA+HPL usually obtains good results, H^{2}TA always improves them, except in the cyclomatic number of FT, where they achieve the same value. The large improvement that H^{2}TA obtains with respect to HTA+HPL for all the metrics in ShWa and Canny is particularly outstanding. The most important reason is related to the synchronization of the ghost regions of these two applications based on stencil computations. While HTA+HPL and MPI+HPL need a more manual user management to keep the memory coherence of the ghost regions, H^{2}TA allows a more convenient and concise programming thanks to its better integration.

7.2. *Performance*

Two different heterogeneous clusters were used to assess the performance of our proposal. The first one, called Fermi, has four nodes with an Intel Xeon X5650 CPU with 6 cores and 12 GB of memory each. Additionally, each node is connected to 2 Nvidia M2050 GPUs with 3GB per GPU. The interconnection network is a QDR InfiniBand. The second system, called K20, has eight nodes. Each one has a 2xIntel Xeon E5-2660 8-core CPUs and 64 GB of RAM. In this case, the accelerator present in each node is a K20m GPU with 5 GB. The interconnection network for this system is an FDR InfiniBand. The g++ 4.7.2 compiler was used with optimization level O3 in both systems, the underlying MPI library being OpenMPI 1.6.4. The problem sizes used for the NPB tests were classes D and C for EP and FT, respectively. Matmul multiplies two matrices of 8192 $\times$ 8192 elements, ShWa computes the evolution of a mesh of 1000 $\times$ 1000 volumes and Canny filters an image of 9600 $\times$ 9600 pixels.

Figures 4 to 8 show the speedups of the MPI+HPL, the HTA+HPL and the H^{2}TA versions when using a varying number of accelerators of each cluster taking as baseline an HPL version that uses a single accelerator of the corresponding cluster. All the values plotted as well as the baselines were obtained as the minimum of five executions. While the effort required for programmers was very different for the three approaches, their performance behavior is very similar. This way, the maximum slowdown of H^{2}TAs with respect to MPI-based applications happens in the execution of FT, where the overhead reaches a maximum value of 2.9% in the Fermi cluster and 2.4% in the K20 system. The next benchmark with the largest overheads is ShWa, which reaches slowdowns of just 1.1% and 1.9% with respect to the MPI+HPL baseline in the Fermi and K20 systems, respectively, while the maximum overhead for the other three benchmarks remains well below 1%. This way, when we look at the big picture, the performance overhead of H^{2}TAs with respect to MPI combined with HPL is minimal, with an average of just 0.5% in both clusters. This overhead is very similar, and even a bit better than that of the manual integration of HTA with HPL, whose average slowdown with respect to the same baseline is 0.8% in both clusters. As a result, it is clear that the large

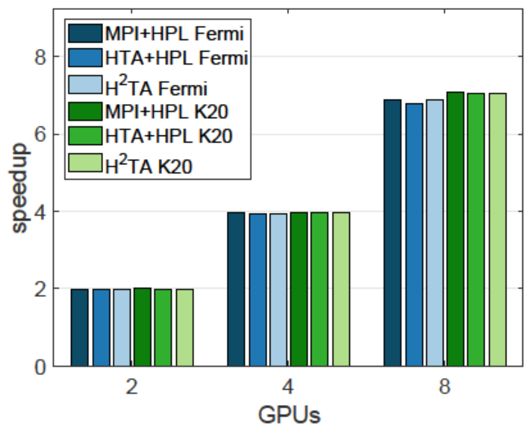


Figure 4. Speedups for EP

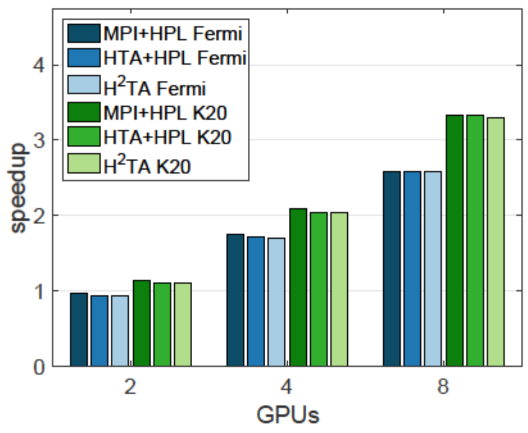


Figure 5. Speedups for FT

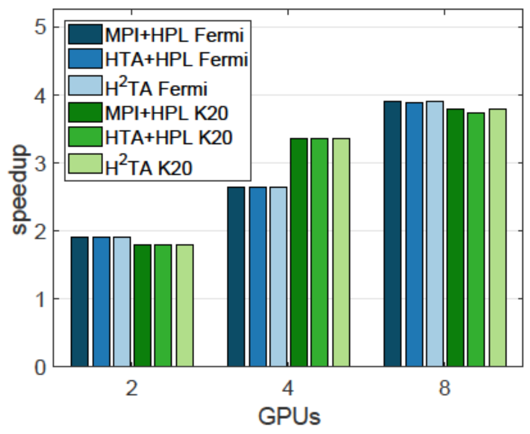


Figure 6. Speedups for Matmul

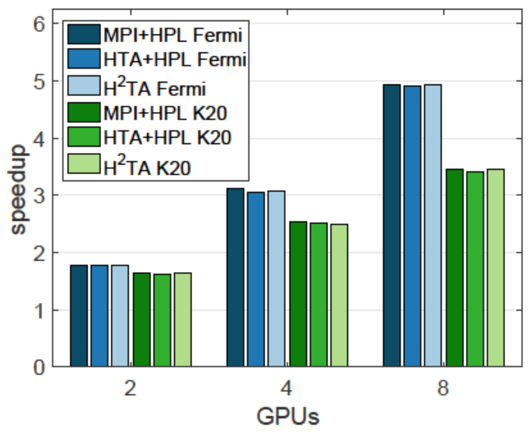


Figure 7. Speedups for ShWa

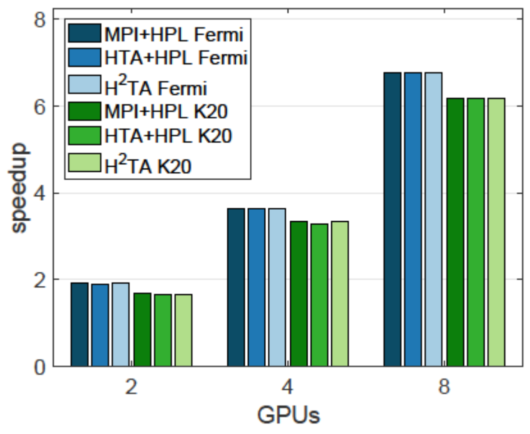


Figure 8. Speedups for Canny

programmability improvements measured in Sect. 7.1 totally justify the reduced overhead of our proposal.

8. CONCLUSIONS

Developing parallel applications for heterogeneous clusters requires simultaneously facing the complexity inherent to distributed memory environments and heterogeneous systems, which leads to increased development times, debugging difficulty, and maintenance costs. In this paper we propose a high level approach to program these systems that is based on an abstract data type that represents an array partitioned into tiles. Such tiles can be distributed on a cluster and processed in parallel following data-parallel semantics, giving a global view of the distributed data structure and exposing a single high-level thread of execution to the user. The data type, called Heterogeneous Hierarchically Tiled Array (H^2TA), extends the existing Hierarchically Tiled Array (HTA), which was oriented to traditional distributed memory clusters, adding support for arbitrary computing devices that support OpenCL, thus maximizing the portability of our solution. Rather than exposing the user to the raw OpenCL API, H^2TA relies on the Heterogeneous Programming Library (HPL), which substantially reduces the development complexity of OpenCL-based applications. H^2TAs inherit the high level notation of HTAs for communications between cluster nodes and add total transparency and automated management of the kernels, buffers, transfers between host and devices memory, etc. required by heterogeneous computing.

H^2TA vastly improves the programmability of heterogeneous clusters with respect to existing approaches. Even if we consider baselines that exploit the advantages of HPL but resort to the traditional MPI library for communications, H^2TAs reduce their programming complexity metrics by an average of 20.5%, 31.8% and 26.9% in terms of SLOCs, Halstead's programming effort and cyclomatic number, respectively. These improvements are twice larger than those achieved by separately using the HTA and HPL libraries, which further justifies the interest of this proposal. Also, the H^2TA runtime is very light, with average slowdowns below 1% that peak at 2.9%, thus making our proposal a very appealing approach for the programming of current complex heterogeneous clusters.

A possible ambitious line of future work for this project would be to implement an HTA-aware compiler that further improves the programmability of these systems and applies optimizations that are more difficult to identify using a library-based implementation.

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) as well as by the Xunta de Galicia (Centro singular de investigación de Galicia accreditation 2016-2019) and the European Union (European Regional Development Fund - ERDF) under grant Ref. ED431G/01.

REFERENCES

1. A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W. c. Feng, K. R. Bisset, and R. Thakur. MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems. In *14th IEEE Intl. Conf. on High Performance Computing and Communication & 9th IEEE Intl. Conf. on Embedded Software and Systems (HPCC-ICESS)*, pages 647–654, June 2012.
2. G. Almási, L. De Rose, B. B. Fraguela, J. E. Moreira, and D. A. Padua. Programming for locality and parallelism with Hierarchically Tiled Arrays. In *16th Intl. Workshop on Languages and Compilers for Parallel Computing, (LCPC 2003)*, pages 162–176, 2003.
3. A. Alves, J. Rufino, A. Pina, and L. P. Santos. clOpenCL - supporting distributed heterogeneous computing in HPC clusters. In *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 112–122. Springer, 2013.
4. R. Aoki, S. Oikawa, T. Nakamura, and S. Miki. Hybrid OpenCL: Enhancing OpenCL for distributed processing. In *IEEE Intl. Symp. on Parallel and Distributed Processing with Applications (ISPA 2011)*, pages 149–154, 2011.
5. C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 298–299. Springer Berlin Heidelberg, 2012.
6. C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *IEEE 16th Intl. Conf. on Parallel and Distributed Systems (ICPADS 2010)*, pages 291–298, Dec 2010.
7. 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.

8. John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Longman Publishing Co., Inc., 1994.
9. J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *2012 IEEE 26th Intl. Parallel Distributed Processing Symp. (IPDPS)*, pages 557–568, 2012.
10. N. Eicker, T. Lippert, T. Moschny, and E. Suarez. The DEEP project an alternative approach to heterogeneous cluster-computing in the many-core era. *Concurrency and Computation: Practice and Experience*, 28(8):2394–2411, 2016.
11. S. Ernsting and Kuchen H. Algorithmic skeletons for multicore, multiGPU systems and clusters. *Intl. J. of High Performance Computing and Networking*, 7(2):129–138, 2012.
12. 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.
13. J. F. Fabeiro, D. Andrade, and B. B. Fraguela. Writing a performance-portable matrix multiplication. *Parallel Computing*, 52:65–77, 2016.
14. P. Faber and A. Größlinger. A comparison of GPGPU computing frameworks on embedded systems. *IFAC-PapersOnLine*, 48(4):240–245, 2015. 13th IFAC and IEEE Conf. on Programmable Devices and Embedded Systems (PDES 2015).
15. B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzarán, D. Padua, and C. von Praun. Optimization techniques for efficient HTA programs. *Parallel Computing*, 38(9):465–484, September 2012.
16. 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, 2015.
17. I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. A uniform approach for programming distributed heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 74(12):3228–3239, 2014.
18. M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.
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. J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proc. 26th ACM Intl. Conf. on Supercomputing (ICS'12)*, pages 341–352, 2012.
21. J. Kraus. An introduction to CUDA-aware MPI, March 2014. https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/ [Online; accessed 11-July-2017].
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. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.
25. T-Y. Liang, Y-W. Chang, and H-F. Li. A CUDA programming toolkit on grids. *Intl. J. of Grid and Utility Computing*, 3(2/3):97–111, 2012.
26. G. R. Luecke, N. T. Weeks, B. M. Groth, M. Kraeva, L. Ma, L. M. Kramer, J. E. Koltes, and J. M. Reecy. Fast epistasis detection in large-scale GWAS for Intel Xeon Phi clusters. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 228–235, Aug 2015.
27. 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.
28. 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.
29. T.J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2:308–320, 1976.
30. M. Nakao, J. Lee, T. Boku, and M. Sato. Productivity and performance of global-view programming with XcalableMP PGAS language. In *12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 2012)*, pages 402–409, May 2012.
31. M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato. XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In *1st Workshop on Accelerator Programming using Directives (WACCPD)*, pages 27–36, Nov 2014.
32. R. V. Nieuwpoort and J. W. Romein. Correlating radio astronomy signals with many-core hardware. *International Journal of Parallel Programming*, 39(1):88–114, 2011.
33. M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi. DS-CUDA: A middleware to use many GPUs in the cloud environment. In *SC Companion*, 2012.
34. OpenACC-Standard.org. The OpenACC Application Programming Interface Version 2.5, Oct 2015.
35. R. Ozaydin and D.T. Altilar. OpenCL Remote: Extending OpenCL platform model to network scale. In *2012 IEEE 14th Intl. Conf. on High Performance Computing and Communication & 2012 IEEE 9th Intl. Conf. on Embedded Software and Systems (HPCC-ICESS)*, pages 830–835, 2012.
36. 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.
37. F. Silla, S. Iserte, C. Reaño, and J. Prades. On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation: Practice and Experience*, 29(13):e4072, 2017.
38. F. Song and J. Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *24th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'12)*, pages 91–100, 2012.
39. 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.

40. M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.
41. B. Varghese, J. Prades, C. Reaño, and F. Silla. Acceleration-as-a-service: Exploiting virtualised GPUs for a financial application. In *IEEE 11th Intl. Conf. on e-Science (e-Science),*, pages 47–56, Aug 2015.
42. 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.
43. 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.
44. M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo. Towards a high level approach for the programming of heterogeneous clusters. In *45th Intl. Conf. on Parallel Processing Workshops (ICPPW 2016)*, pages 106–114, 2016.
45. M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo. Facilitating the development of stencil applications using the Heterogeneous Programming Library. *Concurrency and Computation: Practice and Experience*, 29(12):e4152, 2017.
46. M. Viñas, B. B. Fraguela, Z. Bozkus, and D. Andrade. Improving OpenCL programmability with the Heterogeneous Programming Library. In *Intl. Conf. on Computational Science (ICCS 2015)*, pages 110–119, 2015.
47. S. Xiao and W-C. Feng. Generalizing the utility of GPUs in large-scale heterogeneous computing systems. In *2012 IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, IPDPSW'12, pages 2554–2557, 2012.