Towards a High Level Approach for the Programming of Heterogeneous Clusters

Moisés Viñas, Basilio B. Fraguela, Diego Andrade, Ramón Doallo
Depto. de Electrónica e Sistemas
Universidade da Coruña
A Coruña, Spain
e-mail: {moises.vinas, basilio.fraguela, diego.andrade, doallo}@udc.es

***Abstract*—The programming of heterogeneous clusters is inherently complex, as these architectures require programmers to manage both distributed memory and computational units with a very different nature. Fortunately, there has been extensive research on the development of frameworks that raise the level of abstraction of cluster-based applications, thus enabling the use of programming models that are much more convenient that the traditional one based on message-passing. One of such proposals is the Hierarchically Tiled Array (HTA), a data type that represents globally distributed arrays on which it is possible to perform a wide range of data-parallel operations. In this paper we explore for the first time the development of heterogeneous applications for clusters using HTAs. In order to use a high level API also for the heterogenous parts of the application, we developed them using the Heterogeneous Programming Library (HPL), which operates on top of OpenCL but providing much better programmability. Our experiments show that this approach is a very attractive alternative, as it obtains large programmability benefits with respect to a traditional implementation based on MPI and OpenCL, while presenting average performance overheads just around 2%.**

***Keywords*-distributed computing; heterogeneity; data parallelism; programming model; libraries; OpenCL**

I. Introduction

Beyond the issues inherently related to parallelism, heterogeneous clusters are notoriously difficult to program for two reasons. First, every cluster has a distributed memory nature that requires to use several independent processes, at least one per node, that must communicate by means of messages. These messages can be explicitly exposed to the programmer, as in the case of MPI, which the most widely used framework for distributed memory computing, or implicitly, as in the case of PGAS (Partitioned Global Address Space) approaches [1], [2], [3], [4]. Even in the last situation the user must follow a SPMD programming style with its characteristic conditional control flows, which can be quite complex, and also be aware of the distribution of the data, as the cost of ignoring it is enormous. Second, the heterogeneity of these architectures requires the user to use special frameworks to exploit it and manage much more concepts that CPU-only applications such as blocks of threads, different kinds of memory in the device, device and host-side buffers, etc.

Both kinds of problems have received considerable attention. This way, PGAS are just one of the solutions proposed to reduce the programming cost of clusters with respect to the mainstream MPI-based approach, there being also proposals based on compiler directives [5], [6] and libraries that offer a higher level of abstraction [7]. An interesting thing is that, as far as we know, there have been few attempts to extend the existing cluster programming tools to support heterogeneous clusters [8]. Rather, these architectures have been mostly targeted by means of extensions of the tools oriented to heterogeneous computing (mostly CUDA and OpenCL) that try to support more or less transparently the access to accelerators located in different cluster nodes [9], [10], [11], [12], [13].

In this paper we evaluate for the first time the development of applications for heterogeneous clusters based on a library-based data parallel type originally targeted to homogeneous clusters. The proposal chosen is the Hierarchically Tiled Array (HTA) [7], a datatype implemented in C++ that represents an array partitioned into tiles that are distributed on the nodes of a cluster, thus providing a global view of the structure. The operations on a HTA take place implicitly in parallel across its tiles exploiting data parallelism. As a result, HTA programs do not follow an SPMD programming style. On the contrary, there is a single high-level thread of execution, the parallelism being transparently exploited in the HTA operations that involve several tiles in the same step. HTAs have been compared to optimized MPI-based codes in traditional multi-core clusters achieving large programmability improvements at the cost of reasonable overheads [14]. OpenCL was chosen for the development of the heterogeneous parts of the applications in order to maximize their portability. Rather than using its somewhat verbose host API [15], we used the Heterogeneous Programming Library (HPL) [16], [17], a C++ library that offers a very concise and high-level API for the development of OpenCL-based applications.

The rest of this paper is organized as follows. First, the HTA data type will be described. Then, Section III explains our strategy to develop applications for heterogeneous clusters using HTAs. This is followed by an evaluation on terms of performance and programmability in Section IV and a discussion on related work in Section V. Finally, Section VI concludes the paper with our conclusions and future work.

II. Hierarchically Tiled Arrays

This data type, proposed in [7] and available at `http://polaris.cs.uiuc.edu/hta`, represents an array

```
BlockCyclicDistribution<2> dist({2, 1}, {1, 4});
auto h = HTA<double, 2>::alloc({ {4, 5}, {2, 4} }, dist);
```

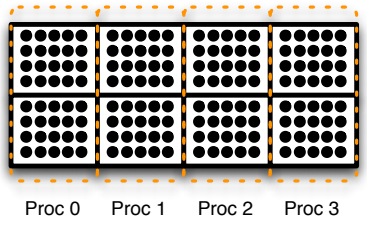

Fig. 1. HTA creation

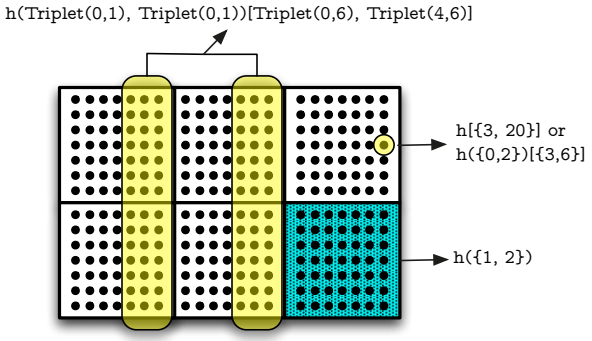

Fig. 2. HTA indexing

that can be partitioned into tiles. These tiles can be low level (i.e., unpartitioned) arrays or they can be also internally partitioned, giving place to the hierarchical nature of this class. HTAs can specify a distribution of their tiles (cyclic, block cyclic, etc.) on a mesh of processors, which allows the representation of distributed arrays. The recursive tiling can be used to express locality as well as lower levels of distribution and parallelism. For example one could use the topmost level of tiling to distribute the array between the nodes in a cluster and the following level to distribute the tile assigned to a multi-core node between its CPU cores. However, the most common practice is to express the partitioning in a single level. Figure 1 illustrates the C++ notation used to build an HTA of 2×4 top level tiles, each one of them composed by 4×5 double precision floating point elements that is distributed on a mesh of 1×4 processors so that each processor gets a block of 2×1 tiles.

A very interesting feature of HTAs is the enormous flexibility they provide for their indexing due to the support of two indexing operators, one for tiles (parenthesis) and another one for scalars (brackets). In addition, both indexings can be combined, as Fig. 2 shows, where the `Triplet` objects allow to express inclusive ranges, so that `Triplet(i,j)` is the range of indices between `i` and `j`, both included. As we can see, the scalar indexing disregards the tiled nature of an HTA, and when it is applied within a tile or set of selected tiles, it is relative to the beginning of each one of those tiles.

HTAs support a wide range of array-based expressions that can express computations, data movements or both. All the HTA expressions take place in parallel on their tiles, thus

```
1  void mxmul(HTA<float,2> a, HTA<float,2> b,
2             HTA<float,2> c, HTA<float,1> alpha)
3  {
4    int rows = a.shape().size()[0];
5    int cols = a.shape().size()[1];
6    int commonbc = b.shape().size()[1];
7
8    for(int i = 0; i < rows; i++)
9      for(int j = 0; j < cols; j++)
10       for(int k = 0; k < commonbc; k++)
11         a[{i,j}] += alpha[0] * b[{i,k}] * c[{k, j}];
12 }
13 ...
14 hmap(mxmul, a, b, c, alpha);
```

Fig. 3. HTA `hmap` example

providing implicit parallelism and communications. When two HTAs are operated jointly, they must be conformable, that is, they must have the same structure and the corresponding tiles that will be operated together will be required to have sizes that allow to operate them. The exceptions are the operations with scalars, which are conformable to any HTA by replication, and untiled arrays, under the condition that such array is conformable to all the leaf tiles of the HTA it is operated with. This way, the conformability rules of HTAs are a generalization of those of Fortran 90.

When the corresponding tiles used in an HTA operation (including assignments) are located in different cluster nodes, communications are automatically performed to complete the operation. This way, for example, if we had two HTAs `a` and `b` built with the structure and distribution of the example in Fig. 1, and assuming that each processor is located in a different node, the expression

```
a(Tuple(0,1), Tuple(0,1)) = b(Tuple(0,1), Tuple(2,3))
```

would imply that processor 2 would send its tiles of `b` to processor 0, while processor 3 would send its tiles to processor 1, both communications taking place in parallel. The other mechanism to perform communications is the invocation of HTA methods that implement array-based typical operations that imply movement of data within the array, examples being transpositions, circular shifts, permutations, etc.

Computations can be directly performed using the standard arithmetic operators (e.g. `a=b+c`, where all the operands are HTAs) thanks to C++ operator overloading. There are also methods that allow to express more complex computations as well as higher-order operators that support the application of user-defined computations. The most widely used operator of this kind is `hmap`, which applies a user function in parallel to the tiles of one or more HTAs. When `hmap` receives more than one argument HTA, the corresponding tiles of these HTAs are operated together. For this reason all the argument HTAs to an `hmap` invocation must have the same top level structure and distribution. Figure 3 illustrates the application of this function to four HTAs in order to compute `a=a+alpha×b×c` by tiles so that in each tile-level operation `a`, `b` and `c` provide matrices with the appropriate sizes to be operated and `alpha` provides

a scalar.

As we can see, HTAs provide high-level semantics that largely increase the degree of abstraction with respect to the traditional message-passing approach used in clusters. They have in common with PGAS solutions that they provide a unified view of globally distributed arrays coupled with information on their distribution so that users can make their accesses as efficiently as possible. Nevertheless, HTAs avoid the SPMD programming style typical of the other approaches and provide rich array-based operations, indexing possibilities, and tile-based manipulations, which further simplify the tasks of cluster programmers.

III. Heterogeneous Applications based on HTAs

While most applications that rely on accelerators are written using either CUDA or OpenCL, since our aim is to explore the programming of heterogenous clusters using high level approaches, we had to choose a tool that offers richer abstractions. We preferred one based on OpenCL for portability reasons, as there are several families of accelerators that are becoming increasingly important in HPC clusters. Also, we wanted the solution chosen to be as general as possible, i.e., to impose minimum restrictions on the kernels, and to provide performance similar to the one provide by native OpenCL. Finally, since it had to work jointly with HTAs, the approach chosen should integrate nicely in C++. Although there is a proposal of a specification to facilitate the use of OpenCL in C++ [18], to this date there are no public implementations of it that can run on accelerators. Thus, given the discussions in [17], [19], the Heterogeneous Programming Library (HPL) [16] was chosen as the best option that fulfilled the desired conditions. This section now briefly introduces HPL, and then explains our strategy to develop applications for heterogeneous clusters using HTAs and HPL.

A. Heterogeneous Programming Library

This project consists of a C++ library, distributed under GPL license from `http://hpl.des.udc.es`, which allows to develop OpenCL-based applications using two mechanisms that can be combined. The first one, introduced in [16], is a language embedded in C++ that expresses the heterogeneous kernels to execute in the accelerators (also the CPU, if it supports OpenCL) by means of a number of macros, predefined variables and data types provided by HPL. Since it is embedded in C++, this naturally gives place to single-source applications with very interesting properties such as the use of C++ variables inside the kernels. Its most powerful property is, however, that the kernels written with this language are built at runtime, exploiting runtime code generation. This allows to write kernels that self-adapt at runtime to the underlying hardware or the inputs, as illustrated in [20]. The second mechanism enables the use of traditional string or separate file-based OpenCL C kernels [17] using the same simple host API as the HPL embedded language, which will be described later.

```
1  void mxmul(Array<float,2> a, Array<float,2> b,
2             Array<float,2> c, Int commonbc, Float alpha)
3  { Int k;
4
5    for_(k = 0, k < commonbc, k++)
6        a[idx][idy] += alpha * b[idx][k] * c[k][idy];
7  }
8  ...
9  Array<float, 2> a(N, N), b(N, N), c(N, N);
10
11 eval(mxmul)(a, b, c, N, alpha);
```

Fig. 4. HPL example

HPL follows the same semantics as OpenCL regarding the execution of a host-side application that can execute kernels, expressed as functions, in the different devices attached to the host. Also, the kernels are executed in parallel by a number of threads that is defined by a space of indices of between one and three dimensions that is called global space in HPL. Another similarity is that the threads can be organized in teams (work-groups in OpenCL; thread groups in HPL) that can synchronize by means of barriers and share a fast scratchpad called local memory. The size and dimensions of these teams are specified by an index space of the same number of dimensions as the global space and that must divide it in every dimension that is called local space in HPL.

Contrary to OpenCL, however, HPL raises the level of abstraction by avoiding both the definition of separate buffers in the different memories involved in this process and the transfers related to the management of these buffers. Rather, HPL users have an unified view of their memory objects, the underlying details being transparently taken care of by the library runtime. All the HPL memory objects belong to the `Array<`$type$`,`N`>` data type, which expresses an N-dimensional array (0 for scalars) of elements of the type $type$. The objects of this kind can be defined in the host side of the application and they are used by the kernels that are executed in the devices when they appear as arguments to those kernels. Since all the accesses to these objects take place by their member functions, HPL knows when they are read or written, which allows it to know the state of their copies and provide a coherent view of the object to the user across the different memories involved. The runtime is optimized so that transfers are only performed when they are strictly necessary.

Figure 4 illustrates the implementation of a simple dense matrix product similar to the one in Fig. 3 using the HPL embedded language. The analogous implementation based on native OpenCL C code is not explained here for brevity, but it requires very simple steps explained in [17]. The kernel, expressed as a regular C++ function in lines 1-7, computes a single element of the destination array `a`. Namely, this kernel is written to be executed in a two-dimensional space of $X \times Y$ threads where X and Y are the number of rows and columns of the destination matrix, respectively. The HPL predefined variables `idx` and `idy` provide the global identifier

of the thread executing the kernel in the global space of threads in the first and the second dimension of this space, respectively. As a result, thread (idx,idy) is in charge of computing a[idx][idy]. The kernel receives as arguments the three matrices involved in the product, and number of elements of the dimension that is common to the two matrices to be multiplied, and the constant alpha. Following our explanation above the type of these two latter elements should be Array<int,0> and Array<float,0>, and in fact this is the case, because Int and Float are just alias provided for conveniency. The host side code starts after line 8. There we can see the definition of the arrays to use in the heterogeneous computations in line 9 and the syntax used to request the execution of a kernel in line 11. It deserves to be mentioned that the scalars used in the list of kernel arguments in the host side can belong to the regular C++ data types.

Kernel invocations use by default a global space of threads defined by the number of dimensions and elements per dimension of the first argument of the kernel, which suits our example. Also, users are not required to specify the local space, letting the underlying OpenCL runtime choose it. HPL allows however to define these spaces as well as the device to use in the execution by means of specifications inserted between the eval and its list of arguments. For example, eval(f).global(10,20).local(5,2).device(GPU,3)(...) runs f using a global space of 10×20 threads subdivided in teams of 5×2 threads in the GPU number 3 using the provided arguments. Although not explained here, HPL also provides a rich API to explore the devices available and their properties, profiling facilities and efficient multi-device execution in a single node.

B. HTA-HPL joint usage

As we can see, HTA and HPL serve very different purposes. While HTAs are well suited to express the top-level data distribution, communication and parallelism across cluster nodes, HPL largely simplifies the use of the heterogeneous computing resources available in a node. Their joint usage in one application requires solving two problems that we discuss in turn in this section followed by a small example.

1) Data type integration: These frameworks require different data types to store the data they can manipulate. Once we are forced to handle these two types (HTAs and HPL Arrays), and since the top-level distribution of data of the HTAs is made at tile level, the best approach would be to build an Array associated to each (local) tile that will be used in heterogeneous computations. The situation would be ideal if we managed to use the same host memory region for the storage of the local HTA tile data and the host-side version of its associated HPL Array, as this would avoid the need for copies between both storages. Fortunately, the API of these datatypes is very rich, which enables programmers to achieve this using a relatively simple strategy illustrated in Fig. 5. First, HTAs provide several methods to identify the tiles that are local to each process. In most situations, however, the identification is extremely simple, as the most widely pattern

```
1 const int N = Traits::Default::nPlaces();
2 auto h = HTA<float, 2>({100, 100}, {N, 1});
3
4 const int MYID = Traits::Default::myPlace();
5 Array<float, 2> local_array(100, 100, h({MYID, 1}).raw());
```

Fig. 5. Joint usage of HTAs and HPL

for the usage of HTAs is to make the distribution of the HTA along a single dimension, defining one tile per process. This is the case in our example, where line 1 gets the number of processes in variable N using the API of the HTA framework and line 2 builds a distributed HTA that places a 100×100 tile in each process, so that all the tiles together conform a $(100 \times \mathtt{N}) \times 100$ HTA that is distributed by chunks of rows. Line 4 obtains the id MYID of the current process, so that choosing h(MYID, 1) will return the tile that is local to this process. Once this tile is identified, obtaining its storage is trivial, as HTAs provide a method raw() that returns a pointer to it. The final step involves making sure that the associated HPL Array uses the memory region that begins at that memory position for storing its host-side version of the array it handles. This is very easy to achieve in HPL, as the Array constructors admit a last optional argument to provide this storage. This way, the Array can be built using the syntax shown in line 5. From this point, any change on the local tile of HTA h will be automatically reflected in the host-side copy of the Array local_array and viceversa.

2) Coherency management: While HPL can automatically manage the coherency of its Arrays across all their usages in HPL, the changes that are due to HTA activities must be explicitly communicated to HPL. Again, this did not require any extension to the existing HPL API, as HPL Arrays have a method data that allows to do this. The original purpose of this method was to obtain a pointer to the host-side copy of an Array so that programmers could access its data at high speed through this pointer, rather than by the usual indexing operators of the Array. The reason is that these operators check and maintain the coherency of the Array in every single access, thus having a considerable overhead with respect to the usage of a native pointer. The data method supports an optional argument that informs HPL of whether the pointer will be used for reading, writing or both, which is the default assumption when no specification is made. This is all the information HPL needs to ensure that the users will get coherent data from the pointer, and the devices will access a coherent view of the Array when it is used in the subsequent kernel invocations. Thus this simple mechanism also suffices to make sure that HTAs have a coherent view of the Arrays that have been modified by heterogeneous computations as well as to guarantee that HPL pushes to the heterogeneous devices fresh copies of those Arrays whose host-side copy has just been modified by an HTA operation.

3) Example: Our ongoing matrix product example is used in Fig. 6 to illustrate the joint usage of HTA and HPL. Here

```
1  // N is the number of processes and MY_ID the local id
2
3  auto hta_A = HTA<float,2>::alloc({{(HA/N), WA}, {N, 1}});
4  Array<float,2> hpl_A((HA/N), WA, hta_A({MY_ID}).raw());
5
6  auto hta_B = HTA<float,2>::alloc({{(HB/N), WB}, {N, 1}});
7  Array<float,2> hpl_B((HB/N), WB, hta_B({MY_ID}).raw());
8
9  auto hta_C = HTA<float,2>::alloc({{HC, WC}, {N, 1}});
10 Array<float,2> hpl_C(HC, WC, hta_C({MY_ID}).raw());
11
12 hta_A = 0.f;
13 eval(fillinB)(hpl_B);
14 hmap(fillinC, hta_C);
15 eval(mxmul)(hpl_A, hpl_B, hpl_C, HC, alpha);
16
17 hpl_A.data(HPL_RD); // Brings A data to the host
18 auto result = hpl_A.reduce(plus<double>());
```

Fig. 6. HTA-HPL example code

the distributed result HTA `hta_A` and the left HTA of the product `hta_B` are allocated by chunks of rows, while the right HTA `hta_C` replicates the whole matrix in each process. The example assumes for simplicity that the number of rows of `hta_A` and `hta_B` is divisible by the number of processes `N`. Next to each HTA is built the `Array` that allows to use the local tile for heterogeneous computing. Lines 12, 13 and 14 fill in the local portion of the A, B and C matrices, respectively. They illustrate how both accelerators under HPL, in the case of B, as well as the CPU under the HTA, in the case of A and C, can be used following this strategy. The matrix product itself happens in line 15 using the kernel shown in Fig. 4. Notice how the usage of `hpl_A` and `hpl_C` as arguments allows the kernel to automatically use the data of the local tile of `hta_A` and `hta_C`, respectively, both initialized by the CPU.

Until this point it has not been necessary to invoke the `data` method on any `Array` because of the default assumption that `Array`s are initially only valid in the CPU together with the automated management of their coherency for the execution of accelerator kernels in the `eval` invocations by HPL. In order to exemplify its use, our example assumes that as last step we want to reduce all the values of the distributed HTA `hta_A` by means of a global addition. This can be achieved by means of the HTA method `reduce`, which will take care of both the communications and the computations required. However, if `reduce` is invoked right after the matrix product in the accelerator, the result will be wrong. The reasons are that (1) the HTA library only has access to the host side copy of the tiles, as it is the only one it knows, and (2) this copy of `hta_A` is outdated, because the result of the matrix product is in the accelerator, as HPL only moves data when it is strictly required. The `data` invocation in line 17 informs HPL that the host-side version of `hpl_A`, which is exactly located in the same host memory as the local tile of `hta_A`, is going to be read. This updates the host side, allowing the HTA reduction to operate successfully.

IV. Evaluation

We will evaluate the development of applications for heterogeneous clusters by means of the HTA and HPL high level approaches using five benchmarks. The first two benchmarks are two of the OpenCL codes developed in [21], namely EP and FT. The first one gets its name from being embarrassingly parallel, although it requires inter-node communications for reductions that happen at the end of the main computation. The second one repetitively performs Fourier Transforms on each one of the dimensions of a 3D array. This requires fully rotating the array in each main iteration of the algorithm, which implies an all-to-all communication between the cluster nodes. The third problem, Matmul, is a distributed single precision dense matrix product in which each node computes a block of rows of the result matrix. The fourth benchmark is a simulation on time of the evolution of a pollutant on the surface of the sea depending on the tides, oceanic currents, etc. called ShWa and parallelized for a cluster with distributed GPUs in [22]. The simulation partitions the sea surface in a matrix of cells that interact through their borders. Thus in every time step each cell needs to communicate its state to its neighbors, which implies communications when they are assigned to different nodes. The distributed arrays are extended with additional rows of cells to keep this information from the neighbor cells in other nodes, following the well known ghost or shadow region technique. The fifth application is Canny, an algorithm that finds edges in images by following a series of four steps, each one implemented in a different kernel. The parallelization comes from the processing or different regions of the kernel in parallel. Communications between neighboring regions of arrays used in the computations are required for some of the kernels. This gives place to the application of the already mentioned shadow region technique, which replicates portions of the borders of the distributed arrays which need to be updated when the actual owner of the replicated portion (rows, in the case of this algorithm) modifies it.

The fact that most applications for heterogeneous clusters use MPI for their communications led us to use this tool in our baseline for this purpose. Regarding the heterogeneous computations, since HPL is a C++ library based on OpenCL, and most OpenCL-based codes rely on its standard API, our baselines are written using OpenCL with the C++ API. Given that the main aim of using tools that provide a higher level of abstraction is the improvement of the programmability and reduction of the programming cost of heterogeneous clusters, this will be the first aspect we will evaluate. The final part of this section will demonstrate that the performance overhead of these tools is negligible.

A. Programmability evaluation

Comparing the ease of use of two approaches is extremely complex. The ideal situation would be to measure the development time and collect the impressions of teams of programmers with a similar level of experience [23]. Unfortunately this is seldom available, so another widely used alternative is to rely on objective metrics extracted from the source code. Our

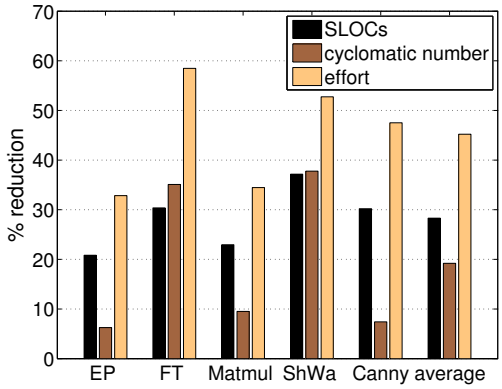

Fig. 7. Reduction of programming complexity metrics of HTA+HPL programs with respect to versions based on MPI+OpenCL.

programmability comparison will be based on three metrics of this kind. The first one is the well-known source lines of code excluding comments and empty lines (SLOCs). The second one is the cyclomatic number [24] $V = P + 1$, where P is the number of predicates (conditionals) found in the program. The foundation of this metric is that the more complex the control flow and the larger the number of execution paths in a program, the more difficult its development and maintenance are. The last metric is the programming effort, proposed in [25] as a function of the total number of operands (constants and identifiers) and operators (symbols that affect the value or ordering of the operands) and the number of unique operands and unique operators.

As we explained, the baselines are also written in C++ and use the OpenCL C++ API. Another decision taken to make the comparison as fair as possible was to encapsulate in our baselines the relatively burdensome initialization of OpenCL in reusable routines and perform this initialization just invoking these routines, so that it is not part of the programmability evaluation. As a result the baselines have the minimum amount of code required to write these applications using MPI and the OpenCL host C++ API.

Figure 7 shows the percentage of reduction of the metrics we have just described when the applications are written using HTA and HPL instead of MPI and OpenCL. The comparison only considers the host-side of the application, because kernels are identical in both cases. We can see that the high level approaches have systematically reduced the programming cost for all the metrics, and even simple codes like Matmul achieve reductions of over 20% for the SLOCs and 30% for the programming effort, which is always the metric that experiences the largest improvement. This latter observation is particularly positive, as since this metric considers every single operand and operator used in the program, it seems a much better indication of complexity than SLOCs, which do not take into account that source lines of code can vastly vary in terms of complexity. On average, the comprehensive high level approach requires 28.3% fewer SLOCs, 19.2% less conditionals, and a 45.2% smaller programming effort than the versions based on MPI and the OpenCL API. These results largely support the combined usage of HTA and HPL, provided that it does not involve important performance overheads.

B. Performance evaluation

Two clusters were used for the performance comparisons. Fermi is a cluster of 4 nodes with a QDR InfiniBand network. Each node has an Intel Xeon X5650 with 6 cores, 12 GB of memory and two Nvidia M2050 GPUs with 3GB per GPU. The K20 cluster has 8 nodes connected by a FDR InfiniBand. Each node has two Intel Xeon E5-2660 8-core CPUs, 64 GB of RAM and a K20m GPU with 5 GB. The compiler used in both systems is g++ 4.7.2 with optimization level O3 and the MPI implementation is the OpenMPI 1.6.4. The problem sizes used for EP and FT were the classes D and B, respectively. Matmul multiplied matrices of 8192 $\times$ 8192 elements, ShWa simulates the evolution of a mesh of 1000 $\times$ 1000 volumes and Canny processes an image of 9600 $\times$ 9600 pixels.

Figures 8 to 12 show, for each benchmark, the speedup of the executions using multiple devices with respect to an execution using a single device for the MPI+OpenCL baseline and the HTA+HPL version. The executions in Fermi were performed using the minimum number of nodes, that is, the experiments using 2, 4 and 8 GPUs involved one, two and four nodes, respectively. The execution using a single device is based on an OpenCL code targeted to a single device, that is, without any MPI or HTA invocations. The most important conclusion from these plots is that the high level approach has very small overheads with respect to the low level one, which is what we wanted to prove. In fact, the average performance difference between both versions is just 2% in the Fermi cluster and 1.8% in the K20 cluster. Not surprisingly, the overhead is more apparent the more intensively HTAs are used, which is the case of FT (around 5%) and ShWa (around 3%), where the communication between nodes happens in a repetitive loop. In FT in addition the HTA takes care of a very complex all-to-all communication pattern with data transpositions. This is also the reason why this was the application with the largest programming effort reduction overall (58.5%) and very strong reductions of the SLOCs (30.4%) and cyclomatic number (35.1%).

V. Related Work

The programmability of heterogeneous clusters has been explored by most researchers by means of expansions of the CUDA and OpenCL paradigm, which are well suited for the management of multiple accelerators in a node, by means of tools that enable the access to accelerators located in other nodes [9], [10], [11], [12], [13], [26]. As a result, The level of abstraction of these tools is analogous to that of CUDA or OpenCL, which force programmers to manage numerous low level details. Some approaches like [27] avoid some of these tasks with their related boilerplate code, but they still keep many others such as the explicit kernel creation, allocation of buffers associated to devices, event-based synchronizations, etc. thus resulting in a much lower level, and therefore worse

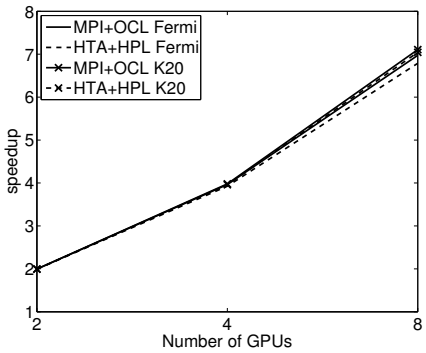

Fig. 8. Performance for EP

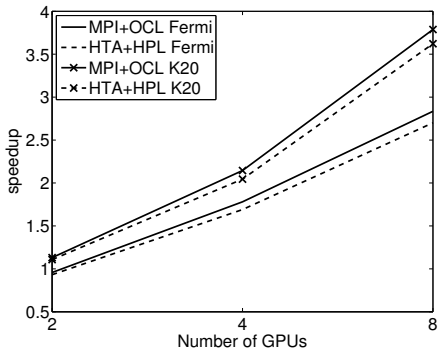

Fig. 9. Performance for FT

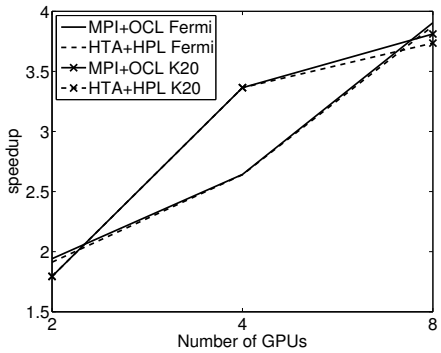

Fig. 10. Performance for Matmul

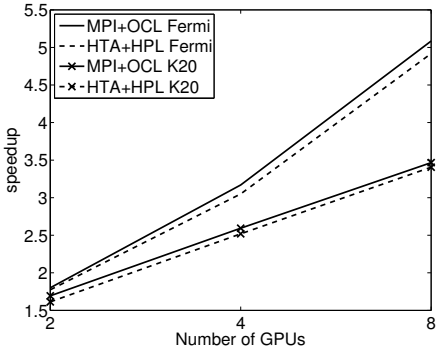

Fig. 11. Performance for ShWa

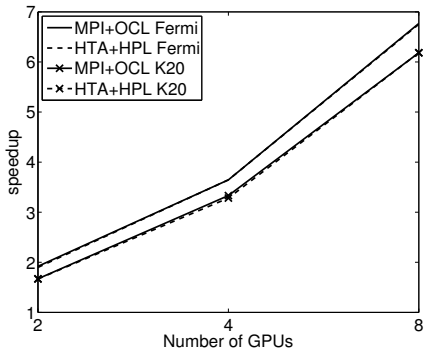

Fig. 12. Performance for Canny

programmability, than the HTA/HPL combination proposed in this paper. A common problem of these approaches is that since they are based on the extension of the CUDA/OpenCL model to access heterogeneous resources across a cluster, contrary to our proposal, they do not provide a solution to the efficient exploitation of the CPUs in the cluster and the communication of the data to be used only in the CPUs. In the OpenCL-based approaches, this could be addressed by writing the CPU codes as OpenCL kernels that are run in those CPUs considering them as OpenCL devices. This clearly requires much more effort than directly using those resources by means of C/C++ functions that work on the tiles of a distributed HTA.

An elegant higher level solution, [28], extends node-level skeletons for heterogeneous programming to clusters. Its main limitation is that it can only support applications in which all the computational patterns are covered by the skeletons. Another project that started considering the computational units in a single node and was later extended to support clusters is SkePU, which developed alternatives based on explicit messages [29] and a task-based model [30] where a single thread of execution enqueues tasks that can be executed in the different accelerators in a cluster respecting the dependencies marked by the user. The data-parallel semantics of HTAs, which provide both predefined common array operations and higher-order functions such as `hmap`, automating the parallel computations and/or complex communications required by many of these operations (e.g. matrix permutations), offer a much more convenient notation thanks to the higher level of abstraction involved. This important advantage also holds with respect the rest of the alternatives discussed in this section.

OmpSs deserves a separate mention, as it targeted traditional clusters [6] before being extended to support heterogeneous ones [8], which enables it to exploit all the parallelism across these systems. OmpSs, however, requires users to indicate which are the inputs and outputs of each parallel task. Worse, it lacks distributed structures, forcing programmers to manually partition in chunks, similar to the HTA tiles, the program arrays and to individually specify the computation to perform with each chunk.

A proposal that combines a language for the programming of clusters [31] with OpenACC [32], which facilitates the programming of accelerators, is [33]. Its main drawbacks are the current reduced portability of OpenACC [34] compared to OpenCL, on which HPL is based, and the dependence on the quality of the compiler of OpenACC, which often reaches much less performance than manually developed kernels [35].

Also related to this work, although to a lesser point, are the efforts for enhancing the integration between MPI and the accelerators [36], [37] and the large body of improvements to facilitate the usage of accelerators in a single node. This includes both native features such as the CUDA-unified address space available since CUDA 6 and the large number of research proposals in this field, many of which are discussed in [16], [17], [19].

VI. Conclusions

Accelerators are becoming increasingly important for large-scale scientific and engineering applications, which has led to their increasing adoption in HPC clusters. As a result, developers have to cope simultaneously with the traditional problems derived from the distributed-memory nature of these systems and the added complexity inherent to heterogeneous computing. Most approaches to deal with this situation have simply enabled to access remote accelerators, making little to raise the level of abstraction of the tools provided to the programmer. In this paper we explore the development of applications for heterogeneous clusters based on two high levels tools. The first one, Hierarchically Tiled Arrays (HTAs), provides arrays distributed by tiles that can be manipulated using data-parallel semantics. HTA applications have a global view of the distributed data and a single thread of control, which splits to operate on parallel on different tiles using high level operators. Some of these operators express global HTA changes, such as permutations and rotations, while assignments between tiles located in different nodes imply data communications, reinforcing the power of the HTA notation for the programming of clusters. The second tool is the Heterogeneous Programming Library (HPL), a framework that allows to develop heterogeneous applications on top of OpenCL avoiding its verbose host API and all all its low level details such as buffers, compilation processes, transfers, synchronizations, etc.

We have shown that these tools can be combined in order to develop programs targeted to heterogeneous clusters following relatively simple strategies. An evaluation based on benchmarks with very different nature has shown that comparing with a baseline written using MPI and OpenCL, the strategy proposed achieves remarkable average programmability metric improvements between 19% and 45%, with a peak of 58%. Also, both tools had already shown very small overheads with respect to low-level versions, and this was reflected in our tests, in which their combination was on average just around 2% slower than the baselines. This way the high-level approach they constitute considerably raises the degree of abstraction of the applications without implying noticeable performance penalties, thus being a very attractive alternative for the programming of heterogenous clusters.

Our future work is to effectively integrate both tools into a single one so that the notation and semantics are more natural and compact and operations such as the explicit synchronizations or the definion of both HTAs and HPL arrays in each node are avoided. This should also expose opportunities for optimizing the runtime, and thus reducing the overheads of the resulting approach.

Acknowledgements

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

References

[1] R. W. Numrich and J. Reid, "Co-array Fortran for Parallel Programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[2] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, 1999.

[3] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proc. 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2005)*, Oct 2005, pp. 519–538.

[4] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *Intl. J. of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[5] C. Koelbel and P. Mehrotra, "An overview of High Performance Fortran," *SIGPLAN Fortran Forum*, vol. 11, no. 4, pp. 9–16, 1992.

[6] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with OmpSs," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, vol. 6852, pp. 555–566.

[7] 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)*, 2003, pp. 162–176.

[8] J. Bueno, J. Planas, A. Duran, R. 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)*, 2012, pp. 557–568.

[9] 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)*, 2008, pp. 49–56.

[10] 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)*, 2012, pp. 341–352.

[11] 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*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7640, pp. 112–122.

[12] P. Kegel, M. Steuwer, and S. Gorlatch, "dOpenCL: Towards uniform programming of distributed heterogeneous multi-/many-core systems," *J. Parallel Distrib. Comput.*, vol. 73, no. 12, pp. 1639–1648, 2013.

[13] 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*, vol. 40, no. 10, pp. 574–588, 2014.

[14] B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzarán, D. Padua, and C. von Praun, "Optimization techniques for efficient HTA programs," *Parallel Computing*, vol. 38, no. 9, pp. 465–484, Sep. 2012.

[15] R. V. Nieuwpoort and J. W. Romein, "Correlating radio astronomy signals with many-core hardware," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 88–114, 2011.

[16] M. Viñas, Z. Bozkus, and B. B. Fraguela, "Exploiting heterogeneous parallelism with the Heterogeneous Programming Library," *J. of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1627–1638, Dec. 2013.

[17] 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)*, 2015, pp. 110–119.

[18] Khronos OpenCL Working Group-SYCL subgroup, "SYCL 2.2 Specification," Feb 2016.

[19] P. Faber and A. Größlinger, "A comparison of GPGPU computing frameworks on embedded systems," *IFAC-PapersOnLine*, vol. 48, no. 4, pp. 240–245, 2015, 13th IFAC and IEEE Conf. on Programmable Devices and Embedded Systems (PDES 2015).

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

[21] 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*, ser. IISWC '11, 2011, pp. 137–148.

[22] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J. García, M. J. Castro, and R. Doallo, "A multi-GPU shallow-water simulation with transport of contaminants," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1153–1169, Jun. 2013.

[23] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan, "Evaluation of UPC programmability using classroom studies," in *Proc. Third Conf. on Partitioned Global Address Space Programing Models*, ser. PGAS '09, 2009, pp. 10:1–10:7.

[24] T. McCabe, "A complexity measure," *IEEE Trans. on Software Engineering*, vol. 2, pp. 308–320, 1976.

[25] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[26] 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)*,, Aug 2015, pp. 47–56.

[27] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "A uniform approach for programming distributed heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3228–3239, 2014.

[28] 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.

[29] 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)*, Dec 2010, pp. 291–298.

[30] 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*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7490, pp. 298–299.

[31] 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)*, May 2012, pp. 402–409.

[32] OpenACC-Standard.org, "The OpenACC Application Programming Interface Version 2.5," Oct 2015.

[33] 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)*, Nov 2014, pp. 27–36.

[34] 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*, ser. PMAM'16, 2016, pp. 1–10.

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

[36] 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)*, June 2012, pp. 647–654.

[37] J. Kraus, "An introduction to CUDA-aware MPI," March 2014, https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/ [Online; accessed 28-May-2016].