J Supercomput (2015) 71:2204–2220
DOI 10.1007/s11227-014-1352-1

# Developing adaptive multi-device applications with the Heterogeneous Programming Library

**Moisés Viñas · Zeki Bozkus · Basilio B. Fraguela · Diego Andrade · Ramón Doallo**

Published online: 7 May 2015


**Abstract** The usage of heterogeneous devices presents two main problems. One is their complex programming, a problem that grows when multiple devices are used. The second issue is that even if the codes for these devices can be portable on top of OpenCL, they lack performance portability, effectively requiring specialized implementations for each device to get good performance. In this paper we extend the Heterogeneous Programming Library (HPL), which improves the usability of heterogeneous systems on top of OpenCL, to better handle both issues. First, we provide HPL with mechanisms to support the implementation of any multi-device application that requires arbitrary patterns of communication between several devices and a host memory. In a second stage HPL is improved with an adaptive scheme to optimize communications between devices depending on the execution environment. An evaluation using benchmarks with very different nature shows that HPL reduces the SLOCs and programming effort of OpenCL applications by 27 and 43 %, respectively, while improving the performance of applications that exchange data between devices by 28 % on average.

M. Viñas (✉) · B. B. Fraguela · D. Andrade · R. Doallo
Grupo de Arquitectura de Computadores, Universidade da Coruña, A Coruña, Spain
e-mail: moises.vinas@udc.es

B. B. Fraguela
e-mail: basilio.fraguela@udc.es

D. Andrade
e-mail: diego.andrade@udc.es

R. Doallo
e-mail: ramon.doallo@udc.es

Z. Bozkus
Department of Computer Engineering, Kadir Has Üniversitesi, Istanbul, Turkey
e-mail: zeki.bozkus@khas.edu.tr

**Keywords** Programmability · Heterogeneity · Parallelism · Portability · Libraries · OpenCL

## 1 Introduction

The usage of heterogeneous devices has enormously grown during the past few years. Unfortunately, the codes for these devices lack portability unless they are developed in OpenCL [12], as most frameworks are vendor- or device-specific [18]. Also, the programming of these systems is much more complex than that of traditional CPUs, as they require dealing with much more issues such as explicit memory management, transfers and synchronizations between different devices, among others. This complexity grows with the number of devices involved in the application. To make things worse, even if the codes are written in OpenCL to achieve portability, the large diversity of heterogeneous systems makes it impossible to reach good performance across different devices by applying uniform programming and optimization strategies. As a result, OpenCL applications usually need to be tuned for different kinds of devices.

There have been many proposals to simplify the programming of heterogeneous systems. A recent one is the Heterogeneous Programming Library (HPL) [23], whose unique feature is a language embedded in C++ in which the computations to run in the devices should be written. The library translates this language to OpenCL, so that it enjoys the portability of this standard together with run-time code generation capability. Furthermore, HPL avoids the high programming cost of OpenCL [17] thanks to the automation of all the tasks it requires, making them totally oblivious to the user. The result is a portable high-productivity programming tool.

An important limitation of HPL was that it lacked critical mechanisms to enable the general effective use of multiple accelerators.This way, our initial experience using several GPUs on HPL [24] relied on the mechanisms available in [23], which restricted it to algorithms in which different devices could only work on different arrays, as there was no support for coherency or data movements between devices. Relatedly, no mechanism was provided to copy data between arrays. In this paper we extend this tool to manage multiple devices while keeping its characteristics of minimum user effort and maximum performance. This extension consists of a totally general data coherency scheme for the data structures managed by HPL as well as a mechanism to make assignments between these structures so that they can be easily copied. The implementation is efficient, as it not only requires the minimum number of transfers, but also applies the most efficient mechanisms to perform these transfers. This latter characteristic implies a dynamic adaption capability of our library, as different transfer mechanisms suit better different systems. Finally, this paper evaluates for the first time HPL on the new Intel Xeon Phi systems.

The rest of this paper is organized as follows: the next Section briefly describes HPL. Section 3 describes the new extensions and Sect. 4 is devoted to the evaluation. The paper finishes with a review of the related work in Sect. 5 and our conclusions and future work plans in Sect. 6.

## 2 The Heterogeneous Programming Library

Our library, which is publicly available at http://hpl.des.udc.es, provides a programming model similar to that of CUDA [18] or OpenCL [12], in which a main application running in a host CPU can execute computational kernels in the form of functions in heterogeneous devices attached to it. Each one of these devices has its own separate memory, and the communication with the host takes place by means of the arguments to the kernel functions. The devices have a number of processors that can run in parallel the kernels in a SPMD fashion, i.e, using a number of threads that are identified by a unique identifier. When the user submits a kernel for execution to a device, she must specify the number of threads to use by means of a space of natural numbers of between one and three dimensions called global domain. A local domain with the same number of dimensions as the global domain and whose dimensions divide those of the global domain in each dimension can be specified. This domain divides the threads of the global domain in groups that can be synchronized by means of barriers and share data using a fast scratchpad. Threads that do not belong to the same local domain cannot cooperate, however.

The library API is explained in detail in [23]. Here we provide a brief description of its three components so that the reader can understand the examples along the paper. The first component is the data type `Array<`*type, nd [, memoryFlag]*`>`, which represents a `nd`-dimensional array of elements of type `type`. The optional `memoryFlag` indicates where is the array located, as HPL supports global, local, constant and private memory in the devices, following the naming and characteristics of the kinds of memory supported by OpenCL [12]. When `nd` is 0, the data type represents a scalar, although the library provides a convenient naming based on an initial uppercase letter (`Int`, `Float`, etc.) to define scalars. Vector types are supported with a similar syntax (`Int4`, `Float8`, etc.). The constructor of a non-scalar `Array` receives the sizes of its dimensions. If the variable is defined in host code, it also allows as optional argument a pointer to the array data in the host memory. If that pointer is not provided, HPL automatically manages the host memory required to support the array.

The second component are a series of macros, predefined variables and functions that constitute together with `Array` a language embedded in C++ in which the HPL kernels must be written. For example, the control structs are those of C finished with an underscore, and the arguments to a `for_` loop must be separated by commas instead of semicolons. The predefined variables allow to obtain critical data for the kernels, such as their unique identifiers or the size of each dimension of the global and the local domain of the execution of the current kernel.

The last component is the host API, whose main purpose is to find the devices available and their properties and request the execution of kernels in them. This way, the execution of a kernel `f`, which is a regular C++ function written using the embedded language provided by HPL, on the arguments `arg1` and `arg2` is requested using the syntax `eval(f)(arg1, arg2)`. By default, the global domain of the execution is given by the dimensions and size of the first argument, while the local domain is automatically chosen by the library. However, these and other parameters can be detailed by inserting specifications, in the form of methods, between `eval` and the argument list.

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

**Fig. 1** Matrix product on a single device using HPL

*Example 1* The matrix product code `c=a × b` in Fig. 1 illustrates the usage of HPL. Lines 1 to 7 contain the kernel definition, which implements the work performed by each thread using the embedded language provided by HPL. The `idx` and `idy` variables identify the thread in the two dimensions of the global domain. Each thread calculates one position of the solution by multiplying a row of matrix `a` and a column of matrix `b`. In the main code, three two-dimensional arrays, *a*, *b* and *c* are defined in line 10. These three arrays correspond to the three matrices involved in the computation. The `eval` method is invoked on this kernel in line 12. As the global space is not specified, the sizes of the two dimensions of matrix *c* are used as the sizes of a two-dimensional global domain. The size of the local domain is set automatically by HPL. The device where the kernel is executed is not specified; thus, the computation will take place in the first OpenCL capable accelerator found. The kernel receives as parameters the three `Arrays` and the size of the loop whose iterations are not distributed among the threads of the global domain.

## 3 Multi-device support in HPL

The exploitation of multiple devices requires several features of the HPL library: support in its API for multiple devices, the improvement of its coherency and synchronization scheme and an easy syntax to copy data between `Arrays`. These features are first presented in turn, while implementation details are discussed in Sect. 3.1.

*Multi-device support in the HPL API* The HPL API allows to identify the devices that can be used and their characteristics, which is necessary to enable multi-device support. HPL currently classifies the devices as either CPUs, GPUs or generic accelerators (this is for example the case of the Xeon Phi). The user can obtain the number of devices of each kind (e.g. `getDeviceNumber(GPU)` provides the number of GPUs) and refer to a specific device using an object of type `Device` that can be built providing a device type and a number of device. For example, `Device(ACCELERATOR, 2)` would be the third accelerator in the system, as the numbering is zero-based. An object `d` of this type can be used to specify where to run a kernel using the syntax `eval(f).device(d)`, that is followed optionally by other execution modifiers,

and finally, the kernel arguments.The method `getProperties` of this class fills a structure of type `DeviceProperties` that has a field for each property of the associated device, thus allowing their inspection.

*Advanced coherency and synchronization scheme* When an HPL kernel execution is requested, the host copies to the memory of the selected device the kernel inputs that were not available in it, then launches the kernel, and it continues executing the main program without waiting for the kernel to finish. This allows parallelizing computations in the host and the devices as well as requesting parallel executions of kernels in different devices.

The synchronization mechanism that allowed to wait for a kernel execution to finish and retrieve its results in [23] was only based on the host accesses to the `Array`s used as arguments to the kernel executions. This way, when the host code tried to read an array that was written by a previously launched kernel, the HPL runtime waited for the kernel to finish and copied the resulting array to the host memory, after which the execution of the main thread in the host would be allowed to continue. Subsequent host accesses to the array would be immediately satisfied from the host-side copy until new kernel executions that wrote to the array were requested. Similarly, an array used as input in a kernel execution would be copied to the device only in the first usage of the array in the device or if the host had written to the array in its memory after the most recent usage of the array in the device. These mechanisms sufficed for efficient single-device executions as the evaluation in [23] shows.

However, to successfully exploit with a reasonable performance and consistent semantics several accelerators, HPL had to be extended in several ways. First, since the user can request to use the same array in multiple devices, and they do not share memory, the HPL runtime was improved to support multiple simultaneous copies of the same array, one per device where it is used, in addition to the host-side copy. The copies of each `Array` are hidden from the user, who only sees its current logical value. The underlying copies are managed following a multiple-readers/single-writer policy (MRSW) policy [21] with an invalidation protocol on writes [15] to keep a single coherent image. Let us notice that a general implementation of the data replications implied by the MRSW strategy requires the copy of data between devices, which is automatically performed by our runtime. Finally, since the host code considers in its turn each one of the kernel execution requests as well as the host accesses to the arrays, the main thread of the application provides sequential consistency [14] to all these accesses to the `Array`s, which is the simplest and most convenient model to reason about parallel programs.

Since the `Array` is the unit of consistency, the usage of the same `Array` in several kernel executions serializes them, even if each kernel operates on disjoint parts of its data, unless of course if the `Array` is only a read-only input to all these kernels. This way, to successfully parallelize executions of kernels that update different portions of the same array in several devices, a different HPL `Array`, associated with the specific portion updated in the device, must be defined for each device. This policy also makes sense for read-only arrays when each device only needs to read a portion of the array. The reason in this case is not to avoid the serialization of the tasks, but to minimize the data transferred, as the `Array` is also the unit or allocation and transfer. The

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

**Fig. 2** Matrix product on multiple GPUs using HPL

construction of HPL `Arrays` associated with different portions of the same host array is facilitated by the fact that their constructor supports an optional argument to specify the location in host memory of the data managed by the `Array`, as commented in Sect. 2. This way, different `Arrays` can start in different positions within the same C array.

*Example 2* Figure 2 shows a multi-device implementation of the same matrix product as Example 1 where the work is splitted among the available GPUs by rows. This example uses the features provided by the multi-device support in the HPL API and takes advantage of the extended coherency and synchronization algorithm by working with `Arrays` associated with different parts of the original underlying matrices. For simplicity the code assumes that the number of rows `M` is a multiple of the number `ndevices` of GPUs, obtained in line 4. The underlying matrices are declared in line 1 as regular arrays. Since the whole matrix `bx` is used in each one of the parallel kernel executions, a single HPL `Array b` is declared in line 2 that contains it.

As explained before, the kernel executions in different devices will only take place in parallel if separate `Arrays` for `cx` are used in each one of them; thus an array of `ndevices` pointers to `Arrays` is built in line 5. Then each `Array` of the appropriate size is created, associating its storage with the corresponding portion of matrix `cx` in line 9. The same approach is followed with respect to matrix `ax`, as this minimizes the amount of data transferred to each device. Finally, the kernel executions in lines 13–14 use the $i$-th `Arrays` of `c` and `a` for the run in the $i$-th GPU. After the kernels are launched, the host continues executing the code after line 14. It will only stop and wait for a kernel execution to finish when either the host code tries to read the associated output `Array c[i]` or a kernel execution in a different device requires `c[i]` as input. In the latter case, the host will wait for `c[i]` to be computed, and then it will transfer it to the other device.

The totally general coherency support implemented in HPL together with its automated movement of data enables to program algorithms that require transfers between devices in a very natural way. For example, stencil codes are usually parallelized by means of ghost regions [9] that replicate a portion of an array that is updated by

```
1 bool f1_was_run = false;
2 while(read(input0)) {
3   eval(f0).device(d0)(input0, output0);
4   if(f1_was_run) write(output1);
5   eval(f1).device(d1)(output0, output1);
6   f1_was_run = true;
7 }
8
9 if(f1_was_run) write(output1);
```

**Fig. 3** Data exchange to implement a pipeline between devices

another processor or accelerator. These structures need to be refreshed with the most recent version of the data they replicate after each update and before the next round of computations begins. When accelerators are used, this gives place to a data exchange between them [25]. Another example are pipelines, in which data proceed through a series of tasks that transform them and handle the results to the next task in the sequence. The parallelism comes from the fact that different tasks work in parallel in different processors or devices on different sets of data.

*Example 3* Figure 3 shows an multi-device pipeline implemented with HPL. In this code we assume that the first argument of each task is only read, and the second one is only written. The pipeline iterates while there are new inputs to read in the initial `Array input0` (line 2). Device `d0` runs task `f0` on this input to generate the intermediate result `output0`. If this is not the first iteration of the pipeline, the boolean `f1_was_run` is true, so in line 4 we write to a file the final result of the pipeline, contained in `Array output1`. When the host accesses `output1` in function `write` through its API, HPL checks this `Array` status, so that if there are pending writes to it (from the execution of `f1` in line 5 in the previous iteration), HPL waits for them, updates the host copy with the current value and finally provides the data to the user code. When line 5 requests to run `f1` on device `d1` taking as input `output0`, the HPL coherency system waits for the most recent execution of `f0` to finish to generate the most up-to-date value, which is then transferred to `d1`. Both operations are blocking for the host, so lines 5 and 6 are only executed once `output0` has been safely copied to a buffer in `d1`. Line 9 ensures that the last result generated is written. Notice that since `f1` only reads the copy of `output0` in the device `d1`, its execution does not delay the next execution of `f0`, which just writes to its local copy of `output0` in device `d0`, located thus in a separate buffer, as HPL knows there is no dependency between both tasks.

*Copy data between arrays* A final programmability improvement has been the implementation of an intelligent assignment operator (see Sect. 3.1) that allows to easily copy data between `Array`s using the natural notation `a=b` in the host code.

### 3.1 Implementation details

HPL `Array`s were extended to support multiple simultaneous copies of the same `Array`, one per device where it is used, each copy being supported by an underlying

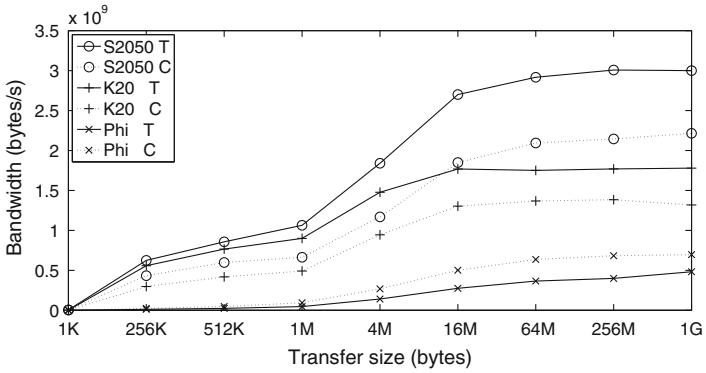


**Fig. 4** Bandwidth of copies between devices. *T* stands for two transfers in sequence and *C* for clEnqueueCopyBuffer usage

OpenCL buffer, in addition to the host-side copy, which is located in plain host memory. HPL builds the buffer images of `Arrays` on demand so that an `Array` is only allocated in a device if it is used in a kernel execution in the device. The potential existence of copies in several devices implied the need for a coherency strategy, which is the multiple-readers/single-writer policy (MRSW) policy [21] with copy invalidations on writes [15]. Our implementation takes into account the new situation that an array in a device could become outdated not only by host-side modifications as in [23], but also by executions of kernels in other devices that wrote to the array. This required in turn a new update mechanism that implied device-to-device transfers of `Arrays`, in addition to the transfers between host and device considered in [23].

A very important issue that we have not seen discussed in the bibliography is how to transfer data between OpenCL buffers in different devices, which corresponds to the copies of `Arrays` between devices.There are two possibilities to perform this transfer in OpenCL, which is our backend. One is to use the OpenCL function `clEnqueueCopyBuffer`, which performs a copy between two buffers. The other possibility is to first transfer the data from the source device to a host location and once it has finished, transfer the data from the host to the destination buffer. Common sense suggests that the first option should be the best one, since it uses a specific runtime function defined for this purpose, which enables it to exploit better possibilities when they are available and fall back on the second option when that is not possible. In fact, the families of OpenCL benchmarks that support multiple devices that we know of, such as the SNU NPB suite [19], use this approach to exchange data between devices. Also, the benchmarks to characterize OpenCL [22] have never compared these two possibilities as far as we know. We have found, however, that `clEnqueueCopyBuffer` can be in fact much slower than the two sequenced transfers possibility in some systems. Figure 4 shows the bandwidth observed in transfers between two devices of the same type in the S2050, K20 and Xeon Phi systems that will be described in Sect. 4 using the two copy mechanisms described, two transfers (T) and `clEnqueueCopyBuffer` (C). We can see that there is a substantial difference between both approaches, and while the Xeon Phi systematically favors `clEnqueueCopyBuffer`, the situation is the opposite in the Nvidia GPUs.

In order to cope with this variability, HPL follows an adaptive approach. The library runs tests making a few transfers using both copy mechanisms to choose the best one for each kind of device when it is installed in a system. The chosen copy strategy is stored in a configuration file that is read whenever a HPL application begins its execution. The HPL runtime then uses the best copy mechanism as a function of the device involved in the communication.

The assignment operator `a=b` to copy data between arrays enjoys many optimizations. Its data transfer is performed in a smart way so that the data from `b` is copied to the image of `a` that holds its most recent version (no matter it is in a device or in the host), as it is expected that subsequent uses of `a` will take place in the same place. If there are multiple updated copies of `a`, the host copy is updated, and it is later transferred to the devices under demand when needed. Also, the copy is automatically performed by means of a kernel when the source and the destination are in the same device.

The synchronization mechanism was also updated. If a kernel execution $Y$ in a device requires an array written by another kernel execution $X$ in another device, $Y$ must be delayed until $X$ finishes to gather the correct results. This is in contrast with [23], which never had to delay a kernel execution, as they were all run sequentially in the only device available.

Finally, we must stress that the complexity of the extended environment is totally hidden by our runtime so that users are not concerned by the existence of the multiple copies and they do not even need to specify when to perform any transfers or updates, all the analysis of dependencies and other details being automatically managed by HPL. This way programmers are just given the simple and intuitive semantics that an `Array` data are (sequentially) consistent across all their usages in the host and the multiple devices available.

## 4 Evaluation

This section evaluates the programmability and performance of our proposal. Since HPL seeks to provide wide portability, using OpenCL as backend for this purpose, this is the standard tool with which it is fairer to compare our library. We have chosen the C++ OpenCL API for the comparisons, as this is the language in which HPL and the benchmarks using it have been developed, and this way both approaches enjoy the same base language.

The evaluation is based on six benchmarks described in Table 1 in terms of the number of source lines of code excluding comments and empty lines (SLOCs) of their OpenCL C++ implementation, the number of kernels involved in unique (only once) invocations and in repetitive invocations (i.e. inside a loop, so that each kernel is invoked several times) and finally the pattern of communication between subtasks when they are split among several devices. These baselines do not contain the cumbersome initialization of OpenCL (device selection, creation of context and command queue, loading and compilation of kernels, etc.), which we have encapsulated in routines that are invoked from the baselines. This way these baselines contain the minimum amount of code that users need to write using the OpenCL host C++ API.

**Table 1** Benchmarks characteristics

| Benchmark | SLOCs OpenCL | Unique invocation | Repetitive invocation | Data exchanges |
|---|---|---|---|---|
| EP | 440 | 1 kernel | | |
| FT | 2,450 | 3 kernels | 7 kernels | All to all |
| MatmulRow | 243 | 1 kernel | | |
| Summa | 320 | | 1 kernel | |
| ShaWa | 961 | | 3 kernels | Stencil |
| N-body | 209 | | 1 kernel | All to all |

The EP and FT benchmarks come from the SNU NPB suite [19], an optimized implementation of the NAS Parallel Benchmarks in OpenCL. EP is an embarrassingly parallel application that is easy to distribute among several devices. FT is a more complex benchmark that uses three kernels for its initialization before entering an iterative process that invokes 7 kernels in each iteration, all of them parallelizable among all the devices available. This benchmark computes the Fourier transform of a 3-D array along its three dimensions. Since the array is partitioned along one of its dimensions to split the work among the devices, when the Fourier transform is to be computed along that dimension, the array has to be permuted or rotated so that the array becomes partitioned by other of its dimensions, and the originally distributed dimension fits locally in each device, enabling the local computation. This leads to an all-to-all pattern of communication between the devices.

MatmulRow is the matrix multiplication distributed by rows used as example in Fig. 2. Summa implements the Summa algorithm for matrix multiplication [6], which divides the three matrices in tiles and interleaves stages of local multiplication in each device with stages of communications consisting of broadcasts across columns and across rows of tiles of the two input arrays. The efficient implementation of these broadcasts in our case does not involve copies between devices, but transfers of different portions of the input arrays from the host to each device in each step. This way this benchmark stresses the communications between the host and each device.

Benchmark ShaWa is a shallow water simulator with transport of contaminants developed in [16]. This application divides a surface into square volumes that interact with their neighbor volumes through their four edges, having a pattern of computation in stencil. This way, its kernels are parallelized using the well-known approach of ghost or shadow regions [9] that replicate a portion of the data in another processor. These regions need to be refreshed in each new time step as the original data are modified. Our baseline exchanges the data between devices by means of device to host, and then host to device, transfers, as they were the best method for our GPUs in Sect. 3.1. Finally, N-body is a simulation of a dynamical system of particles that presents an all-to-all communication pattern because in each time step of the algorithm each particle influences the behavior of all the other particles. Its data exchanges are implemented using `clEnqueueCopyBuffer`, as it is the natural way to make data copies in OpenCL programs.

Figure 5 measures the programmability improvement provided by HPL with respect to the OpenCL C++ baseline in terms of the reduction of programming effort metrics

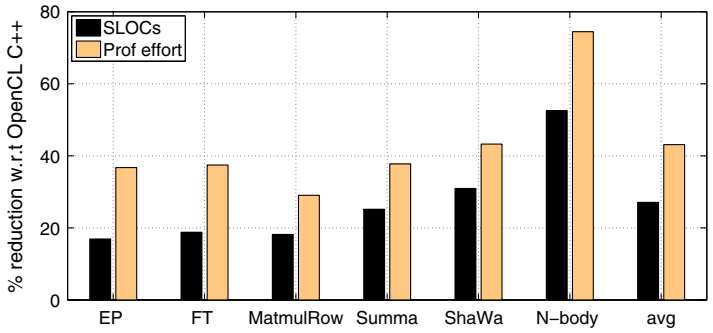


**Fig. 5** Reduction in the number of SLOCs and programming effort of the host side of the application when using HPL with respect to the OpenCL C++ baseline (color figure online)

**Table 2** The hardware platform details

| | System#0 | System#1 | System#2 |
|---|---|---|---|
| CPU | | | |
| Processor | Intel Xeon X5650 | 2x Intel E5-2660 | 2x Intel E5-2660 |
| Frequency (GHz) | 2.67 | 2.20 | 2.20 |
| #Cores | 6 (12 HT) | 8 (16 HT) | 8 (16 HT) |
| Memory capacity (GB) | 12 | 64 | 64 |
| Peak memory bandwidth (GB/s) | 32 | 51.2 | 51.2 |
| GPU | | | |
| Processor | Nvidia S2050 (2x Nvidia M2050) | Nvidia K20m | Intel Xeon PHI 5110P |
| Frequency (GHz) | 1.55 | 0.705 | 1.053 |
| #Cores | 448 | 2496 | 60 (240 HT) |
| Memory capacity (GB) | 3 | 5 | 8 |
| Peak memory bandwidth (GB/s) | 148 | 208 | 320 |

measured in the code of the host side of the application. The kernels have not been included in the measurement because their code is very similar both between OpenCL and HPL and between single-device and multi-device versions of the applications; thus the extensions described in this paper play a small role in them. The first metric is the well-known SLOC. The second one is the programming effort [10], which considers also the complexity of these lines taking into account in a reasoned formula the number of unique operands, unique operators, total operands and total operators found in the code. We can see that the effort is consistently much smaller in HPL, particularly if we take into account the relative complexity of each line of code. On average, HPL

reduces the SLOCs and the programming effort of the baseline by 27.1 and 43.1 %, respectively, even when the baseline is a streamlined version with minimal code for the initialization, as we have explained.

The performance evaluation relies on three systems that are described in Table 2: a system with a NVIDIA Tesla Fermi S2050, another one with 3 Nvidia Tesla Kepler K20m, and one with two Intel Xeon Phi 5110P. The compiler was g++ 4.7.2 with optimization level `O3`.

Figures 6, 7 and 8 show the speedup of our baseline and HPL versions when using all the devices with respect to an OpenCL single-device implementation using a single

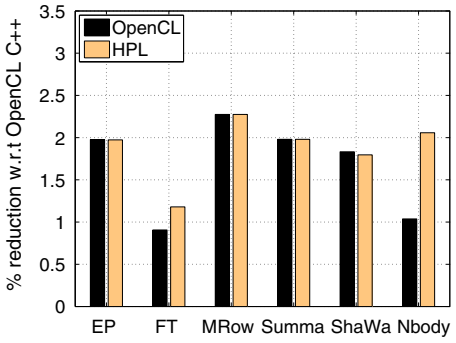


**Fig. 6** Speedups with S2050 (color figure online)

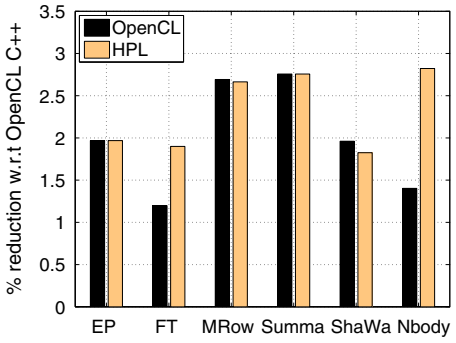


**Fig. 7** Speedups with K20 (color figure online)

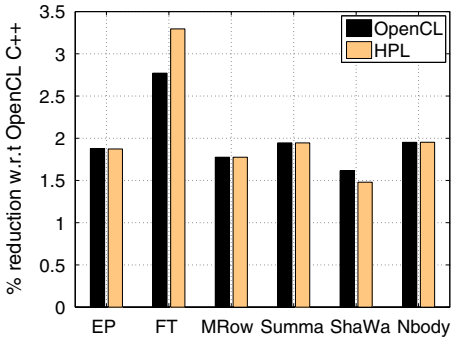


**Fig. 8** Speedups with Xeon Phi (color figure online)

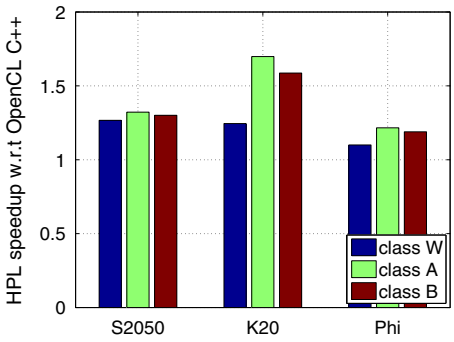


**Fig. 9** Speedup of HPL over OpenCL for different problem sizes of FT (color figure online)

device in each one of the systems just described. This implies using 2 GPUs and 2 Xeon Phis in Figs. 6 and 8, respectively. The SNU NPB, just as the original NPB, requires a number of devices that is a power of two; thus EP and FT only use two K20 in Fig. 7, while the other benchmarks use three. EP and FT were run for classes D and B, respectively. The matrix products used matrices of $6{,}000 \times 6{,}000$ double-precision floating point elements. Finally, ShaWa was run with a $1{,}000 \times 1{,}000$ mesh representing an actual stuary and N-body worked on 192K particles.

As we can see, HPL matches or outperforms OpenCL in most applications, sometimes experiencing some degradation introduced by its runtime. In ShaWa there is an additional overhead derived from the unavailability of mechanisms in the current version of HPL to select a portion of an array for a copy or as argument for a kernel execution. For this reason, HPL ShaWa must make more work to copy the rows that must be exchanged between the devices to and from separate buffers that are used for the exchanges and it is the benchmark with the largest overhead, reaching a maximum of 9 % in the Xeon Phi. In FT, however, HPL is noticeably faster than OpenCL in all the systems (up to 59 % in the K20 system) for two reasons. One is that in the transfers between GPUs the HPL runtime uses the two-transfer mechanism described in Sect. 3.1, instead of the slower `clEnqueueCopyBuffer` found in the SNU NPB. The second reason is that some of the FT array copies take place between arrays that are actually located in the same device. While the SNU NPB implementation always uses the same `clEnqueueCopyBuffer` mechanism, the HPL runtime detects this situation and avoids any transfer, just making a copy inside the device by means of a kernel. The impact of these optimizations is large because FT requires many array transfers, making HPL the winner in terms of average speedup in every device for this benchmark. Something similar happens with N-body, whose data exchanges are an important part of its runtime and are much faster under the policy applied by the adaptive HPL in the GPUs. As a result, HPL is on average 21.4, 25.7 and 2.1 % faster than the OpenCL baseline across the applications tested in the S2050, K20 and Xeon Phi systems, respectively.

Figures 9, 10 and 11 show the speedup of HPL with respect to OpenCL for different problem sizes of FT, ShaWa and N-Body, respectively, as they are the three algorithms that exchange data between devices. Since FT and N-body are based on `clEnqueueCopyBuffer`, which offers bad performance in GPUs but is the best

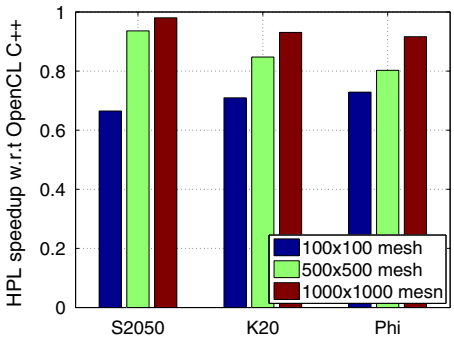


**Fig. 10** Speedup of HPL over OpenCL for different problem sizes of ShaWa (color figure online)

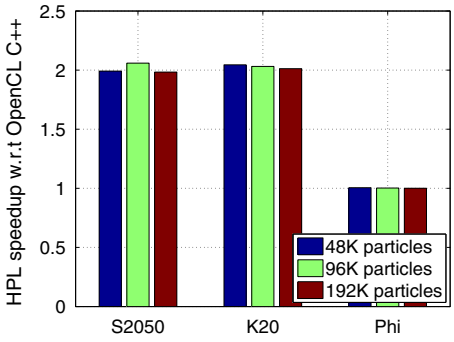


**Fig. 11** Speedup of HPL over OpenCL for different problem sizes of N-Body (color figure online)

option in the Phi, HPL clearly outperforms OpenCL in the GPUs for all the sizes. It also outperforms OpenCL FT in the Xeon Phi because of the second advantage mentioned in the previous paragraph: HPL detects that some copies that the SNU NPB FT code always blindly performs by means of `clEnqueueCopyBuffer` have their source and destionation in the same device, so HPL performs them by a faster copy inside a kernel. The N-body baseline only copies between devices the data that are strictly needed, so HPL and OpenCL get exactly the same performance on the Xeon Phi. Finally, the baseline OpenCL ShaWa is optimal in the GPUs because it uses the two transfers mechanism, so HPL performs worse due to the library overheads and the additional copies that its restriction to operate on whole arrays imply in this algorithm that only exchanges one row between neighboring devices. In fact, since the amount of data exchanged is small, the adaptive nature of HPL, which allows it to use in the Xeon Phi the faster `clEnqueueCopyBuffer` alternative (see Sect. 3.1), does not help it to reach the baseline performance in this accelerator. We see, however, that as the problem size grows, these overheads become a smaller and smaller portion of the runtime, thus reducing the overhead of HPL. In FT and N-body, however, HPL advantage remains basically constant across problem sizes because the whole arrays used in the problem are exchanged. The only exception is FT in the K20, where when the problem size grows from W to A we get a HPL relative speed bump, probably because W is a small problem size with many kernels and the K20 is a powerful accelerator, so the overheads of HPL do not allow it to reach its maximum advantage for a small

size. Overall, HPL was 28% faster than OpenCL across this set of experiments, clearly showing its advantage in applications that exchange data between devices.

## 5 Related work

Many works have focused on the exploitation of clusters of heterogeneous nodes. They allow a thread of execution running in a host to allocate buffers and submit tasks to all the devices in the cluster, avoiding the use of communication APIs such as MPI. While some of these approaches [3] are based on CUDA, many [2,8,11,13] have been built closely following the OpenCL API and concepts, thus requiring a much lower level management than HPL. Only [2,8] abstract away some low-level details of OpenCL. However, the API layer of [2], which supports unmodified OpenCL applications, involves compiler directives that must indicate the inputs and outputs of each task and synchronize them, or an object-oriented API that in addition to these specifications also explicitly uses contexts and buffers. Similarly, libWater [8] still relies on explicit kernel creation processes, buffers associated with devices that are explicitly read and written and synchronizations based on OpenCL-like events. HPL is currently restricted to the exploitation of the devices found in a single node, but it offers programmers a much higher level view based on $n$-dimensional arrays rather than buffers in a given memory or device. Also, it automates all the kernel compilation, task synchronizations, buffer allocations, data transfers and coherency management.

The fact that HPL tasks synchronizations and scheduling are defined by their data dependencies expressed through their arguments relates it to the task superscalar paradigm, which has been proposed to manage heterogeneous computations through the OmpSs programming model [4]. This requires a compiler and the user must explicitly annotate the tasks inputs and outputs, contrary to the library-based and fully automated extraction the dependencies of HPL. OmpSs does not provide either convenient array classes. Similar to HPL, DepSpawn [7] provides such classes and also automatically extracts the data dependencies of the parallel tasks it allows to define, but it has no support for heterogeneous systems.

The exploitation of heterogeneous parallelism across the devices existing in a node by combining OpenMP and OpenACC, or ad-hoc directives has been explored in [26] and in [5], respectively. The result and the differences with respect to HPL are similar to those of OmpSs, with the addition that these solutions do not automatically schedule and synchronize the tasks based on their data dependencies.

Finally, skeleton libraries [1,20] are another approach to use multiple heterogeneous devices with reduced programming effort. The Heterogeneous Programming Library has a wider scope of application than these tools, as they only allow to exploit parallelism in computations whose structure conforms to one of their skeletons.

## 6 Conclusions

In this paper we have extended HPL with an automated and optimized coherency system for arrays that can be used across multiple accelerators as well as the host of a computing node. The extension is also adaptive, as it chooses the most efficient

mechanism to perform the copies and it avoids transfers when it detects the source and the destination are in the same device. The resulting tool reduces on average the programming cost metrics of SLOCs and programming effort of multi-device OpenCL C++ baselines by 27 and 43 %, respectively. As for performance, while HPL can present large overheads for small applications that require features not yet implemented such as the copy of a subarray, its overheads are quite small for medium and large applications, where they peak at 9 %. Furthermore, its adaptive nature allows to obtain noticeable speedups with respect to hand-coded OpenCL in applications that exchange data between devices, achieving an average and a maximum speedup on a series of tests for this kind of applications using different problem sizes of 28 and 106 %, respectively. We plan to extend HPL to heterogeneous clusters and to further enhance programmability by allowing to define subarrays that can be used both in the data transfers and the kernels.

**Acknowledgments** This work was supported by the Xunta de Galicia under the Consolidation Program of Competitive Reference Groups (GRC2013/055), the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (TIN2013-42148-P), both of them cofunded by FEDER funds of the EU, and the Scientific and Technological Research Council of Turkey (TUBITAK; 112E191). This work is also partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

## References

1. Acosta A, Almeida F (2013) Skeletal based programming for dynamic programming on multiGPU systems. J Supercomput 65(3):1125–1136
2. Barak A, Ben-Nun T, Levy E, Shiloh A (2010) A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In: 2010 IEEE international conference on cluster computing workshops and posters (CLUSTER WORKSHOPS), pp 1–7
3. Duato J, Pena A, Silla F, Mayo R, Quintana-Ortí E (2010) rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In: 2010 International conference on high performance computing and simulation (HPCS 2010), pp 224–231
4. Duran A, Ayguadé E, Badia R, Labarta J, Martinell L, Martorell X, Planas J (2011) OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett 21(2):173–193
5. Fraguela BB, Renau J, Feautrier P, Padua D, Torrellas J (2003) Programming the FlexRAM parallel intelligent memory system. ACM SIGPLAN Not 38(10):49–60
6. Geijn RAVD, Watts J (1997) SUMMA: scalable universal matrix multiplication algorithm. Concurr Comput Pract Exp 9(4):255–274
7. González C, Fraguela B (2013) A framework for argument-based task synchronization with automatic detection of dependencies. Parallel Comput 39(9):475–489
8. Grasso I, Pellegrini S, Cosenza B, Fahringer T (2013) LibWater: heterogeneous distributed computing made easy. In: International conference on supercomputing (ICS'13), pp 161–172
9. Guo J, Bikshandi G, Fraguela B, Padua D (2009) Writing productive stencil codes with overlapped tiling. Concurr Comput Pract Exp 21(1):25–39
10. Halstead MH (1977) Elements of software science. Elsevier Science Inc., New York, USA
11. Kegel P, Steuwer M, Gorlatch S (2013) dOpenCL: towards uniform programming of distributed heterogeneous multi-/many-core systems. J Parallel Distrib Comput 73(12):1639–1648
12. Khronos OpenCL Working Group (2013) The OpenCL specification. Version 2
13. Kim J, Seo S, Lee J, Nah J, Jo G, Lee J (2012) SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM international conference on supercomputing (ICS'12), pp 341–352
14. Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans Comput 28(9):690–691

15. Li K, Hudak P (1989) Memory coherence in shared virtual memory systems. ACM Trans Comput Syst 7(4):321–359
16. Lobeiras J, Viñas M, Amor M, Fraguela B, Arenaz M, García J, Castro M (2013) Parallelization of shallow water simulations on current multi-threaded systems. Int J High Perform Comput Appl 27(4):493–512
17. Nieuwpoort RV, Romein JW (2011) Correlating radio astronomy signals with many-core hardware. Int J Parallel Program 39(1):88–114
18. Nvidia (2008) Nvidia: CUDA compute unified device architecture
19. Seo S, Jo G, Lee J (2011) Performance characterization of the NAS parallel benchmarks in OpenCL. In: Proceedings of the 2011 IEEE international symposium on workload characterization, IISWC '11, pp 137–148
20. Steuwer M, Gorlatch S (2014) SkelCL: a high-level extension of OpenCL for multi-GPU systems. J Supercomput 69(1):25–33
21. Stumm M, Zhou S (1990) Algorithms implementing distributed shared memory. Computer 23(5):54–64
22. Thoman P, Kofler K, Studt H, Thomson J, Fahringer T (2011) Automatic OpenCL device characterization: guiding optimized kernel design. In: Euro-Par'11, LNCS, vol 6853. Springer, pp 438–452
23. Viñas M, Bozkus Z, Fraguela B (2013) Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. J Parallel Distrib Comput 73(12):1627–1638
24. Viñas M, Bozkus Z, Fraguela B, Andrade D, Doallo R (2014) Exploiting multi-GPU systems using the Heterogeneous Programming Library. In: 14th International conference on computational and mathematical methods in science and engineering (CMMSE 2014), pp 1280–1291
25. Viñas M, Lobeiras J, Fraguela B, Arenaz M, Amor M, García J, Castro M, Doallo R (2013) A multi-GPU shallow-water simulation with transport of contaminants. Concurr Comput Pract Exp 25(8):1153–1169
26. Xu R, Chandrasekaran S, Chapman B (2013) Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model. In: 2013 IEEE 27th International parallel and distributed processing symposium workshops Ph.D. forum (IPDPSW), pp 1169–1176