J. Parallel Distrib. Comput. 73 (2013) 1627–1638

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Exploiting heterogeneous parallelism with the Heterogeneous Programming Library☆

Moisés Viñas [a,*], Zeki Bozkus [b], Basilio B. Fraguela [a]

[a] Depto. de Electrónica e Sistemas, Universidade da Coruña, Spain
[b] Department of Computer Engineering, Kadir Has Üniversitesi, Turkey

HIGHLIGHTS

- A library to improve the programmability and portability of heterogeneous systems.
- Computations are expressed with a language embedded in C++ it provides.
- Run-time code generation (RTCG) advantages and its control with C++ are discussed.
- Our library automates data transfers, task synchronization, etc.
- Performance is very similar to OpenCL while programmability largely improves.

ARTICLE INFO

Article history:
Received 15 October 2012
Received in revised form
25 April 2013
Accepted 19 July 2013
Available online 14 August 2013

Keywords:
Programmability
Heterogeneity
Parallelism
Portability
Libraries
Code generation
OpenCL

ABSTRACT

While recognition of the advantages of heterogeneous computing is steadily growing, the issues of programmability and portability hinder its exploitation. The introduction of the OpenCL standard was a major step forward in that it provides code portability, but its interface is even more complex than that of other approaches. In this paper, we present the Heterogeneous Programming Library (HPL), which permits the development of heterogeneous applications addressing both portability and programmability while not sacrificing high performance. This is achieved by means of an embedded language and data types provided by the library with which generic computations to be run in heterogeneous devices can be expressed. A comparison in terms of programmability and performance with OpenCL shows that both approaches offer very similar performance, while outlining the programmability advantages of HPL.

1. Introduction

The usage of heterogeneous computing resources that cooperate in the execution of an application has become increasingly popular as a result of improvements in runtime and power consumption achieved with respect to traditional approaches solely based on general-purpose CPUs [19]. Still, these advantages do come at a sizable cost in terms of programmer productivity and, often, code portability. The reason for this is that current hardware accelerators cannot be simply programmed using the sequential languages and semantics with which programmers are familiar. Nowadays, the most widely utilized approach to take advantage of these systems is the usage of extended versions of well-known languages [9,10,2,29] that reflect and allow for the management of the particular semantics, characteristics and limitations that these accelerators pose for programmers. Portability problems arise from the fact that the vast majority of these programming environments, in fact all of them with the exception of OpenCL [29], are vendor-specific, and sometimes even accelerator-specific. This situation has led to extensive research on ways to improve the programmability of heterogeneous systems. In light of this, researchers have proposed a rich set of libraries [13,35,4,8,22,20], each with different strengths and weaknesses, and compiler directives [23,16,30], whose performance strongly depends on compiler technology.

In this paper, we present the Heterogeneous Programming Library (HPL), a new alternative to address the problems of programmability and portability described above. Our approach relies

☆ This article is an extended version of Bozkus and Fraguela (2012) [6].

* Corresponding author.
E-mail addresses: insmvb00@gmail.com, moises.vinas@udc.es (M. Viñas), zeki.bozkus@khas.edu.tr (Z. Bozkus), basilio.fraguela@udc.es (B.B. Fraguela).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.07.013

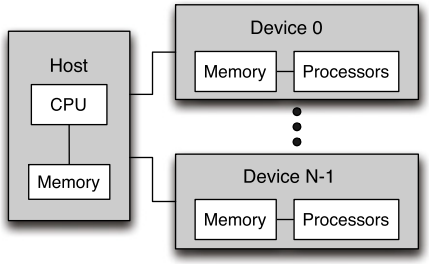


Fig. 1. Heterogeneous Programming Library hardware model.

on expressing the kernels that exploit heterogeneous parallelism in a language embedded in C++. This allows the library to capture at run-time the computations and variables required for the execution of those kernels. With this information the HPL performs run-time code generation (RTCG) in order to run those kernels on the requested device. This is currently achieved on top of OpenCL in order to maximize the portability of applications, although nothing precludes the usage of other backends in the future. Our experience with HPL indicates that it provides performance on par with OpenCL, while providing major programmability advantages.

This article is organized as follows: we begin with an overview of the hardware and programming model supported by HPL, followed by a description of the interface and implementation of our library. An evaluation in terms of programmability and performance in Section 4 is followed by a discussion on related work. The last section discusses our conclusions and future work.

2. Supported programming model

The Heterogeneous Programming Library (HPL) hardware and programming models are similar to those provided by CUDA [10] and OpenCL [29] and they are so general that they can be applied to any computing system and application. The HPL hardware model, depicted in Fig. 1, is comprised of a host with a standard CPU and memory, to which is attached a number of computing devices. The sequential portions of the application run in this host and can only access its memory. The parallel parts, which will be written using the embedded language provided by the library, run in the attached devices at the request of the host program. Each device has one or more processors, which can only operate on data found within the memory of the associated device, and which must all execute the same code in SPMD. Processors in different devices, however, can execute different pieces of code. Also, in some devices the processors are organized in groups with two properties. First, the processors in a group can synchronize using barriers, while processors in different groups cannot be synchronized. Second, each group of processors may share a small and fast scratchpad memory.

As regards the memory model of the HPL, while no special distinction is made in the host, three kinds of memory can be identified in the devices. First, we have the global memory of the device, which is the largest one, and which can be both read and written by any processor in the device. Second, the scratchpad memory which is local and restricted to a single group of processors is called local memory. Finally, a device may have a constant memory, which is read-only memory for its processors, but which can be set up by the host.

As this description of the hardware indicates, HPL applications run their serial portions in the host while their parallel regions run in SPMD mode in the attached devices. While the processors in the same device must all run the same code at a given time, different devices can run different codes. Thus, both data and task parallelism are supported. The parallel tasks are called kernels and they are expressed as functions written in the HPL embedded language. Since the device and host memories are separate, the inputs of a kernel are transferred to its device by the host, and they are provided to the kernel by means of some of its arguments. Similarly, kernels output their results through some of their arguments, which will be transferred to the host when required.

Since multiple threads in a device run the same kernel in SPMD style, an identifier is needed to univocally distinguish each thread. For this purpose, when a kernel is launched to execution in a device, it is associated to a domain of non-negative integers with between one and three dimensions called global domain. An instance of the kernel is run for each point in this domain. In this way, this point is the unique identifier (global id) of the thread, and the domain size gives the number of threads used.

Kernel executions can also be optionally associated to another domain, called local domain, whose purpose is to define groups of threads that run together in a group of device processors able to synchronize and share local memory. The local domain must have the same dimensionality as the global domain, and its size in every dimension must be a divisor of the size of that dimension in the global domain. The global domain can thus be evenly divided in regions of the size of the local domain, so that each region corresponds to a separate thread group whose threads can cooperate thanks to the barriers and the exploitation of the local memory. Each group has a unique identifier based on its position in the global domain (group id). Each thread also has a local id that corresponds to the relative position of its global id within the group's local domain.

As we will see as we develop the description of HPL, in comparison with OpenCL, its backend, HPL avoids the concepts of the context, command queues and commands submitted to the devices. There is no correspondence either for the OpenCL program and memory objects and thus for their management (explicit load and compilation, data transfers, buffer allocation, etc.). Kernel objects are not needed to refer to kernels, just their function name, as in C or C++. There are also issues that OpenCL forces to manage, while HPL can either totally hide or let the user just provide hints for optimization purposes, such as the synchronization between the devices and the host. HPL also brings generic programming capabilities to portable heterogeneous programming, as its kernels and data types support templates. Another interesting feature is that HPL supports multidimensional arrays in the kernel arguments even if their sizes are determined at runtime, giving place to a much more natural notation than the array linearization forced by the usage of raw pointers in OpenCL. Finally, HPL provides runtime code generation (RTCG) tools that can simplify the generation and selection of code versions at runtime.

3. The heterogeneous programming library

Our library supports the model described in the preceding section, providing three main components to users. First, it provides a template class `Array` that allows for the definition of both the variables that need to be communicated between the host and the devices, and the variables that are local to the kernels. Second, these kernels, as mentioned in the previous section, are functions written using the HPL embedded language, which is an API in C++ consisting of data types, functions, macros and predefined variables. This API allows our library to capture the computations requested, so that it can build a binary for them that can run in the requested accelerator. Finally HPL provides an API for the host code in order to inspect the available devices and request the execution of kernels. The entire HPL interface is made available by the inclusion of the single header file `HPL.h` and it is encapsulated inside the `HPL` namespace in order to avoid collisions with other program objects. At this point, we will turn to a discussion of the library components.

```
1 HPL_DEFINE_STRUCT( mystruct_t,
2                    { int i;
3                      float f;
4                    } );
5
6 Array<mystruct_t, 2> matrix(100, 100);
```

Fig. 2. Declaring a `struct` type to HPL in order to use it in `Arrays`.

3.1. The *Array* data type

Like any function, HPL kernels have parameters and private variables. Both kinds of variables must have type `Array`<*type*, *ndim*[, *memoryFlag*]>, which represents an *ndim*-dimensional array of elements of the C++ type *type*, or a scalar for *ndim* = 0. The optional *memoryFlag* either specifies one of the kinds of memory supported (`Global`, `Local` and `Constant`, in the order used in Section 2) or is `Private`, which specifies that the variable is private to the kernel and which is the default for variables defined inside kernels. The *type* of the elements can be any of the usual C++ arithmetic types or a `struct`. In this latter case, the `struct` definition must be made known to HPL using the syntax shown in Fig. 2, where `mystruct_t` is the name we want to give to the `struct`.

When the host code invokes a kernel, it provides the arguments for its execution, which must also be `Arrays`. In this way `Arrays` must be declared in the host space as global variables or inside functions that run in the host, in order to specify the inputs and outputs of the kernels. These variables, which we call host `Arrays`, are initially stored only in the host memory. When they are used as kernel arguments, our library transparently builds a buffer for each one of them in the required device if no such buffer exists yet. The library also automatically performs the appropriate transfers between host and device memory, again only if needed. When a host array or kernel argument declaration specifies no *memoryFlag*, `Global` is assumed. Variables defined inside kernels do not allow the `Global` and `Constant` flags. By default they follow the standard behavior of data items defined inside functions, being thus private to each thread in its kernel instantiation. The exceptions are `Arrays` with the `Local` flag, which are shared by all the threads in a group even if they are defined inside a kernel.

While scalars can be defined using the `Array` template class with *ndim* = 0, there are convenience types (`Int`, `Uint`, `Float`, ...) that simplify the definition of scalars of the obvious corresponding C++ type. Vector types are also supported both in the kernels (e.g. `Int2`, `Float4`, ...) and the host code (correspondingly `int2`, `float4`, ...). These vectors can be indexed to access their components and manipulated with several functions, including the standard operators. Computations can be performed between vectors as well as between vectors and scalars.

An important characteristic both of `Arrays` and HPL vector types is that while they are indexed with the usual square brackets in kernels, their indexing in host code is made with parenthesis. This difference visually emphasizes the fact that while `Array` accesses in the host code experience the usual overheads found in the indexing of user-defined data types [14], this is not the case in the kernels. The reason is that HPL kernels are dynamically captured and compiled into native binary code by our library, so that the array accesses have no added overheads.

One reason for the extra cost of the `Array` accesses in the host code is that they track the status of the array in order to maintain a consistent state for the computations. In this way an array that has been modified by a kernel in a device is refreshed in the host when an access detects the host copy is non-consistent. If the array is written, it is marked as obsolete in the devices.

Table 1
Predefined HPL variables.

Meaning	First dimension	Second dimension	Third dimension
Global id	`idx`	`idy`	`idz`
Local id	`lidx`	`lidy`	`lidz`
Group id	`gidx`	`gidy`	`gidz`
Global domain size	`szx`	`szy`	`szz`
Local domain size	`lszx`	`lszy`	`lszz`
Number of groups	`ngroupsx`	`ngroupsy`	`ngroupsz`

The other crucial point for the maintenance of the consistency is at kernel launch. Input arrays are updated in the device only if there have been most recent writes to them in the host or another device. Also, output arrays are marked as modified by the device, but they are not actively refreshed in the host after the execution. Rather, as explained above, an access in the host will trigger this process. Overall this leads to a lazy copying policy that minimizes the number of transfers.

While this automated management is the default, it can be avoided in order to improve the performance. For example, the user may get the raw pointer to the array data in the host through the `Array` method `data` and perform the accesses through the pointer. This method has as an optional argument a flag to indicate whether the array will be read, written or both through the pointer; if not provided, both kinds of accesses are assumed. With this information the host data is updated if necessary, and the status of the array is correctly tracked. Fig. 3 illustrates both possibilities. In the case of Fig. 3(a) HPL automatically tracks the state of the arrays and makes the required updates, but the check is performed in every access. In Fig. 3(b), however, the user explicitly indicates in lines 3 and 4 that `Array a` will be overwritten in the host, while `b` should be brought from the device with the newest version, unless such version is of course the one in the host. Data are then accessed through pointers in line 7, incurring no overhead.

3.2. Computational kernels

The second requirement for writing HPL kernels, after the usage of the HPL data types, is to express control flow structures using HPL keywords. The constructs are the same as in C++, with the differences that an underscore finishes their name (`if_`, `for_`, ...) and that the arguments to `for_` are separated by commas instead of semicolons.[1]

Given the SPMD nature of the execution of kernels, an API to obtain the global, local and group ids as well as the sizes of the domains and numbers of groups described in Section 2 is critical. This is achieved by means of the predefined variables displayed in Table 1.

Kernels are written as regular C++ functions that use these elements and whose parameters are passed by value if they are scalars, and by reference otherwise. For example, the SAXPY (Single-precision real Alpha X Plus Y) vector BLAS routine, which computes $Y = aX + Y$, can be parallelized with a kernel in which each thread `idx` computes `y[idx]`. This results in the code in Fig. 4.

The kernel functions can be instantiations of function templates, i.e., C++ functions that depend on template parameters. This is a very useful feature, as it facilitates generic programming and code reuse with the corresponding boost in productivity. In fact, templates are one of the most missed features by OpenCL developers, who can finally exploit them on top of OpenCL, the current

[1] The initial version of HPL presented in [6] required that the end of a block was explicitly marked either with the keyword `end_`, or with a structure-specific keyword (`endif_`, `endfor_`, ...). This is no longer needed.

```
1 Array<float, 1> a(N), b(N);
2 ...
3 for(int i = 0; i < N; i++)
4     a(i) = b(i);
```

(a) Automated management.

```
1 Array<float, 1> a(N), b(N);
2 ...
3 float *pa = a.data(HPL_WRITE);
4 float *pb = b.data(HPL_READ);
5
6 for(int i = 0; i < N; i++)
7     pa[i] = pb[i];
```

(b) Manual management.

Fig. 3. Usage of `Arrays` in host code.

```
1 void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
2     y[idx] = a * x[idx] + y[idx];
3 }
```

Fig. 4. SAXPY kernel in HPL.

```
1 template<typename T>
2 void addmatrices(Array<T,2> c, Array<T,2> a, Array<T,2> b) {
3     c[idx][idy] = a[idx][idy] + b[idx][idy];
4 }
```

Fig. 5. Generic HPL kernel to add bidimensional arrays of any type.

backend for our library, thanks to HPL. A small kernel to add two 2-D arrays `a` and `b` into a destination array `c`, all of them with elements of a generic type `T`, is shown in Fig. 5. The kernel will be executed with a global domain of the size of the arrays, and the thread with the global id given by the combination of `idx` and `idy` takes care of the addition of the corresponding elements of the arrays.

HPL provides several functions very useful for the development of kernels. For example, `barrier` performs a barrier synchronization among all the threads in a group. It accepts an argument to specify whether the local memory (argument `LOCAL`), the global memory (argument `GLOBAL`) or both (`LOCAL|GLOBAL`) must provide a coherent view for all those threads after the barrier. Fig. 6(a) illustrates its usage in a kernel used in the computation of the dot product between two vectors `v1` and `v2`. An instance of the kernel, which is run using groups (local domain size) of `M` threads, is executed for each one of the elements of the vectors so that thread `idx` multiplies `v1[idx]` by `v2[idx]`. The reduction of these values is achieved in two stages. First, a shared vector `vec` of `M` elements located in the local memory stores the partial result computed by each thread in the group. Once the barrier ensures all the values have been stored, the thread with the local id 0 reduces them. There are more efficient algorithms to perform this reduction, but our priority here is clarity. The result is stored in the element of the output vector `pSum` associated to this group, which is selected with the group id `gidx`. In a second stage, when the kernel finishes, the host reduces the contents of `pSum` into a single value.

Another example of useful HPL function is `call`, used for invoking functions within kernels. For example, `call(f)(a,b)` calls function `f` with the arguments `a` and `b`. Of course the routine must also be written using the HPL data types and syntax. HPL will internally generate code for a routine and compile it only the first time it is used; subsequent `call`s will simply invoke it. It should be mentioned that routines that do not include a `return_` statement can also be called with the usual `f(a,b)` syntax. The difference is that they will be completely inlined inside the code of the calling function.

This behavior of `call` raises the issue of how HPL kernels are transformed into a binary suitable to run on a given device. This is a two-step process that is hidden from the user. In the first stage, called instantiation, the kernel is run as a regular C++ code compiled in the host application. The fact that this code is written using the embedded language provided by HPL allows the library to capture all the data definitions, computations, control flow structures, etc. involved in the code, and build a suitable internal representation (IR) that can be compiled, as a second step, into a binary for the desired device. Our current implementation relies on OpenCL C [29] as IR because, as the open standard for the programming of heterogeneous systems, it provides the HPL programs with portability across the wide range of platforms that already support it. There are not, however, any restrictions that preclude the usage of other IRs and platforms as backend.

```
1  #define M 64
2
3  void dotp(Array<float,1> v1,
4            Array<float,1> v2,
5            Array<float, 1> pSum) {
6    Int i;
7    Array<float, 1, Local> vec(M);
8
9    vec[lidx] = v1[idx] * v2[idx];
10
11   barrier(LOCAL);
12
13   if_( lidx == 0 ) {
14     for_( i = 0, i < M, i++ ) {
15       pSum[gidx] += vec[i];
16     }
17   }
18 }
```

(a) Basic manual reduction.

```
1  #define M 64
2
3  void dotp(Array<float,1> v1,
4            Array<float,1> v2,
5            Array<float, 1> pSum) {
6
7    reduce(pSum[gidx],
8           v1[idx] * v2[idx],
9           "+").groupSize(M).inTree();
10 }
```

(b) Using `reduce` and binary tree reduction.

Fig. 6. Dot product kernels in HPL.

In fact efforts were made in the development of the library to facilitate this possibility, for example by placing most OpenCL-dependent code in a separate module. The aim is for heterogeneous applications written in HPL to have the potential both to preserve the effort spent in their development even in environments where OpenCL is not available and to exploit more efficient backends where possible.

Since the kernel is run as a regular C++ routine during the instantiation, variables of standard C++ types can appear in the kernel. These variables will not appear in the kernel IR; rather, they will be replaced by a constant with their value at the points of the kernel in which they interact with the HPL embedded language elements. By taking advance of this property, the macro `M` used in lines 7 and 14 of Fig. 6(a) and defined as a constant in line 1, could have been instead defined as an external integer variable. The best value for the group size could have been chosen at runtime and stored in this variable before the kernel was instantiated, which happens when it is invoked for the first time. At that point, any reference to `M` in the kernel would be replaced by its actual value in the IR.

For the reasons explained above, standard C++ code, such as computations and control flow keywords, can also appear in kernels. Just as the variables of a type other than `Array`, they will not appear in the IR. In their case, they will simply be executed during the instantiation. In this way, they can be used to compute at runtime values that can become constants in the kernel, to choose among different HPL code versions to include in the kernel or to simplify the generation of repetitive codes. This is illustrated in Fig. 7, where `r`, `a` and `b` are 2-D `Arrays` of $\mathtt{m} \times \mathtt{n}$, $\mathtt{m} \times \mathtt{m}$ and $\mathtt{m} \times \mathtt{n}$ elements, respectively, and in which `m` and `n` are C++ integers whose value is only known at runtime, but remains fixed once computed, and the matrix `a` is known to be upper triangular. The code computes $\mathtt{r} = \mathtt{a} \times \mathtt{b}$ avoiding computations on the zeros of the lower triangle of `a`. HPL first helps by allowing the direct usage of `m` and `n` in the kernel without having to pass them as arguments. If the number of iterations of the innermost loop is above some threshold `C`, the matrix product is computed using HPL loops whose optimization is left to the backend compiler. Otherwise the code runs the loops in C++ so that they get completely unrolled during the instantiation, which should enhance the performance in GPUs given the properties of these devices. This gives place to $(m \times (m+1) \times n)/2$ lines of code with the shape of line 11 in the figure, each one with a combination of the values of `i`, `j` and `k`. In CUDA or OpenCL the compiler may have trouble applying this optimization due to the triangular loop, the variable nature of `m` and `n` or both, so the programmer would have to perform this tedious and error-prone process by hand. Nevertheless, the HPL user can write the code in Fig. 7, knowing that the loops will only run during the instantiation, generating the required versions of line 11 with the appropriate frozen values of `i`, `j` and `k`. These lines will be part of the kernel IR due to the use of the variables of type `Array`.

As can be seen, regular C++ embedded inside HPL kernels acts as a metaprogramming language that controls the code generated for the kernels. This provides HPL with advanced run-time code generation (RTCG) abilities that simplify the creation of versions of a kernel optimized for different situations as well as the application of several optimizations. This property is particularly valuable for HPL given the diversity of heterogeneous devices on which the kernels could be run and the high dependence of their performance on the exact codification chosen. This metaprogramming approach, also called generative metaprogramming [11,17], is much more powerful than other well-known metaprogramming techniques such as those based on C++ templates [36,1]. For example, templates are very restricted by the requirement to perform their transformations only with the information available at compile-time. Another problem is their somewhat cumbersome notation (specializations

```
1  if( ( (m * (m + 1) ) / 2 ) * n > C ) {
2      Int i, j, k;
3      for_( i = 0, i < m, i++ )
4          for_( j = 0, j < n, j++ )
5              for_( k = i, k < m, k++ )
6                  r[i][j] += a[i][k] * b[k][j];
7  } else {
8      for( int i = 0; i < m; i++ )
9          for( int j = 0; j < n; j++ )
10             for( int k = i; k < m; k++ )
11                 r[i][j] += a[i][k] * b[k][j];
12 }
```

Fig. 7. Using regular C++ in a kernel to generate an unrolled matrix product.

of functions or classes are used to choose between different versions of code, recursion rather than iteration is used for repetitive structures, etc.), which complicates their application. This contrasts with our approach, which takes place at run-time and uses the familiar control structs of C++. Template metaprogramming has been widely used though in the internal implementation of HPL in order to optimize the HPL code capture and the backend code generation, moving computations to compile time whenever possible. Still, most of the process is performed at runtime, although its cost is negligible, as we will see in Section 4.

The advantages of RTCG are not only provided by HPL as a feature to be manually exploited by the programmer. Rather, the interface includes facilities to express common patterns of computation whose codification can be built at runtime in order to tailor it to the specific requirements needed. An example is `reduce`, which accepts as inputs a destination, an input value and a string representing an operator and which performs the reduction of the input value provided by each thread in a group using the specified operator into the destination. This routine actually builds an object that generates at run-time the code for the reduction. This object accepts, by means of methods, a number of optional hints to control or optimize the code generated. As an example, the dot product kernel in Fig. 6(a) is simplified using this feature in Fig. 6(b). In this case, we provide the optional hint that the kernel will be run using groups of `M` threads to help generate a more optimized code. We also request the reduction to be performed using a binary tree algorithm, which often yields better performance than the alternative used in Fig. 6(a), at the cost of a more complex codification. As of now `reduce` supports nine code generation modifiers. Other examples of modifiers are requesting a maximum amount of local memory to be used in the reduction process, indicating a minimum group size rather than the exact group size, or specifying whether only one thread needs to write the result in the destination, which is the default, or whether all of them must do it. The object builds an optimized code that tries to fulfill the requests performed while using the minimum number of resources, computations and thread synchronizations, and inserts it in the kernel. This mechanism is thus equivalent to having a library of an infinite number of functions to perform reductions in a thread group, each one optimized for a specific situation.

Finally, it should be pointed out that our library does not merely translate the HPL embedded language into an IR in a passive way. On the contrary, during this process the code can be analyzed by the library, which enables it to act as a compiler, gathering information and performing optimizations in the generation of the IR. As of now, HPL does not yet automatically optimize the IR. Nevertheless, kernels are analyzed during the instantiation in order to learn which arrays are only read, only written or both, and in this case, in what order. This information is used by the runtime to minimize the number of transfers required for the kernel and host accesses between the host and the device memories in use without

```
1  void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
2      y[idx] = a * x[idx] + y[idx];
3  }
4
5  int main(int argc, char **argv) {
6      float myvector[1000];
7      Float a;
8      Array<float, 1> x(1000), y(1000, myvector);
9
10     //the vectors and a are filled in with data (not shown)
11
12     eval(saxpy)(y, x, a);
13 }
```

Fig. 8. Array creation and SAXPY kernel usage.

any user intervention, as discussed in the previous section. It also allows to learn the dependences of each kernel submitted to execution, so that HPL automatically ensures they are satisfied before it runs, which results in an automatic and effortless synchronization system.

3.3. Host interface

The most important part of the host interface is function `eval`, which requests the execution of a kernel with the syntax `eval`(f)($arg1, arg2, \ldots$) where f is the routine that implements the kernel. As mentioned before, scalars are passed by value and arrays are passed by reference, and thus allow the returning of results.

Specifications in the form of methods to parameterize the execution of the kernel can be inserted between `eval` and the argument list. Two key properties are the global and local domains associated with the kernel run explained in Section 2, which can be specified using methods `global` and `local`, respectively. For example, if kernel `f` is to be run on arguments `a` and `b` on a global domain of 100 × 200 threads with a local domain of size 5 × 2, the programmer should write `eval(f).global(100, 200).local(5, 2)(a, b)`.

By default the global domain corresponds to the sizes of the first argument, while the local domain is chosen by the library. Fig. 8 illustrates a simple invocation of the SAXPY kernel of Fig. 4, also included in this figure for completeness, by means of its function pointer in line 12. The global domain requires one point per element of `y`, which is the first argument, while the local domain needs no specification. The example also shows that host arrays can be created from scratch (`x`), making the library responsible for the allocation and deallocation of its storage in the host, or they can use already allocated host memory by providing the pointer to this memory as last argument to the constructor (`y`). In this latter case the user is responsible for the deallocation too.

Also, although not detailed here due to space limitations, our library provides a simple interface to identify and inspect the devices in the system and their attributes (number of threads supported, amount of memory of each kind, etc.) and to obtain a handle of type `Device` to make reference to each one of them. A final method to control the execution of a kernel is `device`, which takes as argument one of these handles in order to choose the associated device for the execution. If none is specified, the kernel is run in the first device found in the system that is not a standard CPU. If no such device is found, the kernel is run in the CPU.

The sequence of steps performed by HPL when a kernel is invoked is described in Fig. 9. In the first place, an IR of the kernel suitable for the chosen device is sought in an internal cache. If such IR is not found, the kernel is instantiated following the process described in the previous section. Once the required IR is available, it could have been already compiled to generate a binary for the chosen device or not. This is checked in a second cache, which is updated with that binary after the corresponding compilation if it is not found. At this point, HPL transfers to the device those and only those data needed for the execution. This is possible thanks to the information that is automatically tracked on the status of the HPL arrays, and the knowledge of which of the kernel arguments are inputs, which is obtained during the kernel instantiation. As a final step, the kernel is launched for execution.

As we can see in Fig. 9, the kernel evaluation request finishes in the host side when the device is asked to run the kernel, without further synchronizations with the device. In this way, HPL kernel runs are asynchronous, i.e., the host does not wait for their completion before proceeding to the next statement. This enables overlapping computations among the host and the device, as well as among several devices in a straightforward way. There are several ways to synchronize with the kernel evaluations. As discussed in Section 3.1, whenever the host code accesses an array or submits for execution a kernel that uses it, our runtime analyzes the dependences with preceding uses of the array, enforces them and performs the required transfers. There are also explicit synchronization mechanisms such as the `data` method of `Arrays` or the `sync` method of `Devices`, which waits for the completion of all the kernels sent to the associated device and then updates those that have been modified in the host memory.

Another conclusion from our description of Fig. 9 is that kernel instantiations and compilations are minimized, because each kernel is only instantiated the first time it is used, and an IR is only compiled when an associated binary does not exist yet for the chosen device. However, a user might want to reinstantiate a kernel in some situations. For example, as we mentioned in Section 3.2, the instantiation could depend on the values of external C++ variables, and the user could be interested in generating several possible instantiations and comparing their performance in order to choose the best one. For this reason, there is a function `reeval` with the same syntax as `eval`, but which forces the instantiation of a kernel even if there were already a version in the HPL caches. Also, our library allows the user to retrieve the string with the IR generated for any kernel, so that it can be inspected and/or directly used on top of the corresponding backend.

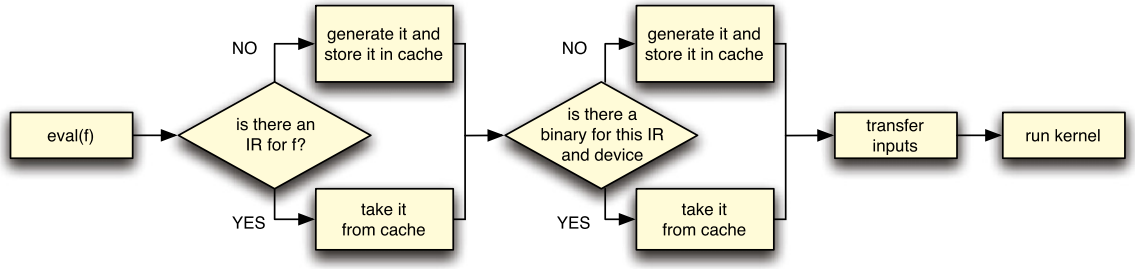


Fig. 9. Kernel invocation algorithm.

Table 2
Benchmark characteristics.

Benchmark	SLOCs OpenCL	Routines	Repetitive invocation	Cooperation	Arithmetic intensity
Spmv	500			Medium	Low
Reduction	399		1 kernel	High	Low
Matrix transpose	373			Low	Low
Floyd–Warshall	407		1 kernel	No	Low
EP	605	✓		Low	High
Shallow water	965	✓	3 kernels	Low	High

4. Evaluation

This section evaluates the programmability benefits and the performance achieved by HPL. The baseline of our study is OpenCL, since this is the only tool that provides the same degree of portability. Also, as it is the current backend for HPL, the comparison allows for the measurement of the overhead that HPL incurs.

The evaluation is based on six codes, namely the sparse matrix vector multiplication (spmv) and reduction benchmarks of the SHOC Benchmark suite [12], the matrix transpose and Floyd–Warshall codes from the AMD APP SDK, the EP benchmark of the NPB suite [26], and the shallow water simulator with pollutant transport (shwa) first presented in [37], whose OpenCL version is thoroughly described and evaluated in [24]. The first five codes were already used in a preliminary evaluation in [6]. This study relied on the original OpenCL implementations from the corresponding suites, which include several non-basic routines and use the C interface of the OpenCL framework. Although EP had not been taken from any distribution, the baseline code suffered similar problems. The HPL versions of spmv and reduction also had some unneeded routines inherited from the original OpenCL implementation.

We have now streamlined and cleared all the codes. The OpenCL baselines have also been translated to C++ in order to use the much more succinct OpenCL C++ interface, so that by avoiding the C++ versus C expressivity difference in the host interface, the programmability comparison is much fairer. The same policies were followed in the translation of the shallow water code from the original CUDA implementation [37]. The result is that now the number of source lines of code excluding comments (SLOC) of our OpenCL baselines is up to 3.3 times smaller than in [6], as Table 2 indicates. The HPL codes were also improved with features implemented after the publication of [6], such as the customized `reduce` mechanism described in Section 3.2.

Table 2 further characterizes the benchmarks indicating whether their kernels use subroutines, whether there is a single kernel invocation or repetitive invocations (and in this case of how many kernels), the degree of cooperation between the threads in the kernels and the arithmetic intensity. The repetitive invocation of kernels is interesting for the analysis of the cost of the kernel executions and synchronizations with the host, including the effectiveness of the mechanisms to avoid unneeded transfers between host and device memory. Reduction and Floyd–Warshall repetitively invoke in a loop a single kernel, while the shallow water simulator performs a simulation through time in a sequential loop in which in each iteration three different kernels are run one after another, there being also computations in the host in each time iteration.

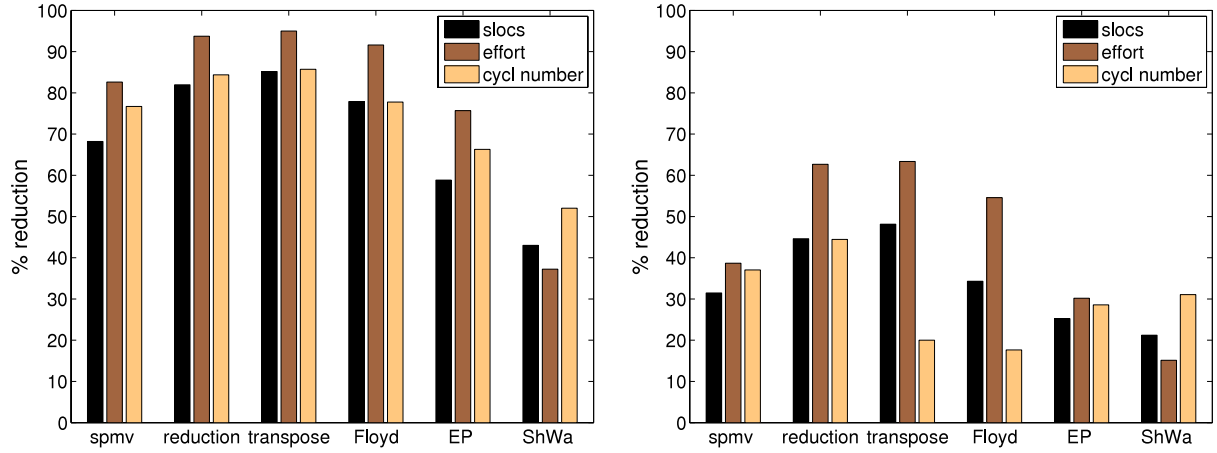
The cooperation column qualitatively represents the weight of synchronizations and data exchanges between threads in the kernels. For example, in spmv each thread first performs part of the product of the compressed row of a matrix by a vector, and then performs a binary tree reduction with other threads in its group to compute the final value for the row. The reduction benchmark focuses intensively in reductions that imply continuous data sharing and synchronization among threads. In matrix transpose, each thread group loads the local memory with a sub-block of the matrix to transpose, then synchronizes once with a barrier, and finally copies the data from the local memory to the transposed matrix. In Floyd–Warshall, each thread performs its own computations without the use of local memory or barriers. In EP, each thread runs the vast majority of the time working on its own data, there being a final reduction of the results of each thread. The situation is similar in the shallow water simulator, in which threads only need to cooperate in a reduction in the most lightweight kernel.

Finally, the arithmetic intensity, which measures the ratio of computations per memory word transferred, is a usual indicator for characterizing applications run in GPUs. Due to the much higher cost of memory accesses compared to computations in these devices, high arithmetic intensity is a very desirable property for GPGPU computing. As can be seen in Table 2, our evaluation relies on codes with a wide range of different characteristics.

4.1. Programmability analysis

Productivity is difficult to measure, thus most studies try to approximate it from metrics obtained from the source code such as the SLOCs. Lines of code, however, can vary to a large extent in terms of complexity. Therefore, other objective metrics have been proposed to more accurately characterize productivity. For example, the programming effort [15] tries to estimate in a reasoned way the cost of developing a code by means of a formula that takes into account the number of unique operands, unique operators, total operands and total operators found in the code. For this, it regards as operands the constants and identifiers, while symbols or combinations of symbols that affect the value or ordering of operands constitute the operators. Another indicator of the complexity of a program is the number of conditions and branches it contains. Based on this idea, [25] proposed as a measure of complexity the cyclomatic number $V = P + 1$, where P is the number of decision points or predicates.

Fig. 10 shows the reduction of SLOCs, programming effort [15] and the cyclomatic number [25] of HPL with respect to an OpenCL implementation of the considered benchmarks. Fig. 10(a) takes as the baseline an OpenCL program including the initialization code to choose a suitable platform and device, build the context and command queue used by this framework, and load and compile the kernels. The initialization code is written in a very generic way, so that it maximizes portability by supporting environments with multiple OpenCL platforms installed and/or several devices, and it controls all the errors that can appear during the process. The code is in fact taken with minor adaptations from the internals of HPL in order to provide exactly the same high degree of portability and error control. This is the OpenCL version whose SLOCs appear in Table 2. Nevertheless, the initialization of the OpenCL environment as well as the loading and compilation of the kernels can be easily placed in routines that can be reused across most applications, thus avoiding much programming cost. For this reason Fig. 10(b) takes as the baseline for the improvement in productivity metrics provided by HPL a factorized OpenCL code that replaces with a few generic routine calls this heavy initialization of ∼270 SLOCs. The two baselines considered thus represent a reasonable maximal



(a) Full OpenCL baselinet. (b) Factorized OpenCL baseline.

Fig. 10. Productivity metrics reduction in HPL with respect to two OpenCL baseline implementations.

and minimal programming cost of the OpenCL version of each application, even if the minimal one is somewhat unfair to HPL, as the removed code has still to be written at some point.

Even if we consider the most demanding scenario, HPL reduces the SLOCs between 21% and 48%, the programming effort between 15% and 63% and the cyclomatic number between 18% and 44%. While these numbers are very positive, complexity measurements on the code do not tell the whole story. Our experience when programming with HPL is that it speeds up the development process in two additional ways not reflected in the code. The first way is by moving the detection of errors to an earlier point. Concretely, since OpenCL kernels are compiled at runtime, the application needs to be recompiled (if there are changes in the host code), sent to execution, and reach the point where the kernel is compiled to find the usual lexical and syntactical errors, fix them and repeat the process. With HPL the detection of the most common problems of this kind (missing semicolons, unbalanced parenthesis, mistyped variables, ...) happens right at the compilation stage, as in any C++ program. Besides in many integrated development environments (IDEs) the integration of the compiler with the editor allows quickly going through all the errors found by the compiler and fix them. We have seen a productivity improvement thanks to the faster response time enabled by HPL.

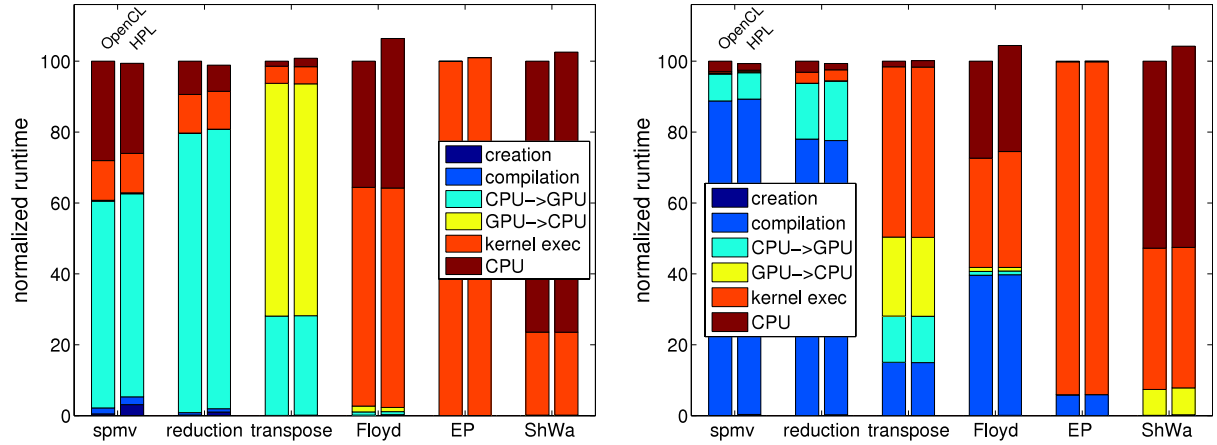
The second way how HPL further improves productivity is by providing better error messages. This way, sometimes the error messages obtained from some OpenCL compilers were not helpful to address the problem. For example, some errors detectable at compile or link time, such as invoking a nonexistent function due to a typo, were reported using a line of a PTX assembly file, without any mention of the identifier or line of OpenCL where the error had been made. Obviously, this is a hit to the productivity of the average user who has to track the source of this problem and fix it. With HPL, the C++ compiler always clearly complains about the unknown identifier in the point of the source code where it is referenced or, in the worst case, when the error is detected during linking, at least it indicates the object file and name of the missing function, largely simplifying the programmer's work.

4.2. Performance analysis

This section compares the performance of the baseline OpenCL applications with those developed in HPL in two systems. The first is a host with 4xDual-Core Intel 2.13 GHz Xeon processors that are connected to a Tesla C2050/C2070 GPU, a device with 448 thread processors operating at 1.15 GHz and 3GB of DRAM. This GPU operates under CUDA 4.2.1 with an OpenCL 1.1 driver. In order to evaluate the very different environments and test the portability of the applications, the second machine selected was an Intel Core 2 at 2.4 GHz with an AMD HD6970 GPU with 2 GB of DRAM and 1536 processing elements at 880 MHz operating under OpenCL 1.2 AMD-APP. The applications were compiled with g++ 4.7.1 using optimization level O3 on both platforms.

The performance of OpenCL and HPL applications is compared for the NVIDIA and AMD GPU based systems in Fig. 11(a) and (b), respectively. The runtime of both versions was normalized to that which was achieved by the OpenCL version and it was decomposed in six components: kernel creation, kernel compilation, time spent in CPU to GPU transfers, time required by GPU to CPU transfers, kernel execution time, and finally host CPU runtime. The kernel creation time accounts in the OpenCL version for the loading of the kernel source code from a file, as this is the most usual approach followed, particularly for medium and large kernels. In the HPL columns, it corresponds to the time our library required to build the kernel IR from the C++ embedded language representation. The other portions of the runtime correspond to the same basic steps in both versions. The measurements were made using synchronous executions, as most profilers do for accuracy reasons, thus there was no overlapping between host computations and GPU computations or transfers. It should be pointed out that it is particularly easy to obtain this detailed profiling for the HPL codes because when our library and the application are compiled with the flag `HPL_PROFILE`, HPL automatically gathers these statistics for each individual kernel invocation as well as for the global execution of the program. The user can retrieve these measurements by means of a convenient API.

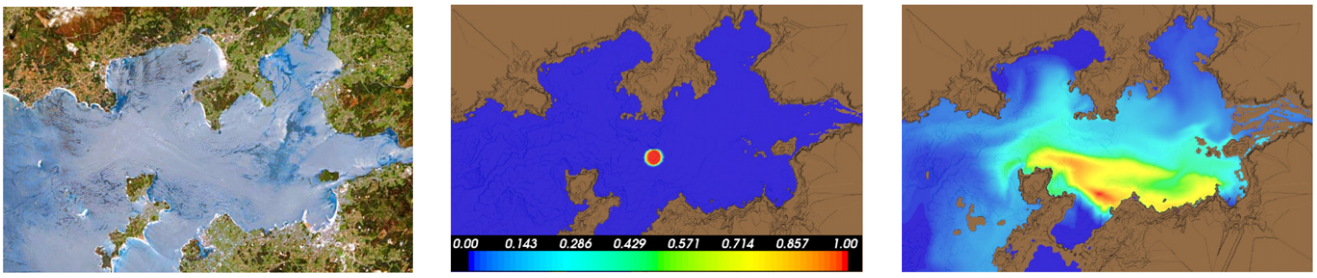
The experiments consisted of multiplying a 16K $\times$ 16K sparse matrix with a 1% of nonzeros by a vector in spmv, adding 16M values in reduction, transposing a 8K $\times$ 8K matrix, applying the Floyd–Warshall algorithm on 1024 nodes, running EP with class C, and finally simulating the evolution of a contaminant during one week in a mesh of 400 $\times$ 400 cells that represents an actual estuary (*Ría de Arousa*, in Galicia, Spain) using real terrain and bathymetry data. The input and the visual representation of the results of this real-world application are illustrated in Fig. 12(a), with the Google Maps satellite image of the region where the simulation takes place, the illustration of the initial setup in Fig. 12(b), in which the contaminant is concentrated in a small circle with a radius of 400 m, and Fig. 12(c) where we see how it evolved after eight days via a color scale which indicates the normalized concentration of the pollutant. All the benchmarks operate on single-precision values, the exceptions being Floyd–Warshall, which works with integers, and EP, which is based on double-precision floating point



(a) Tesla C2050/C2070 system.

(b) HD6970 system.

Fig. 11. Performance of the OpenCL and the HPL versions of the codes, normalized to the runtime of the OpenCL version.



(a) Satellite image.

(b) Pollutant drope.

(c) Situation after 8 days.

Fig. 12. Simulation of evolution of a pollutant in Ría de Arousa.

computations. It should also be mentioned that the shallow water simulation kernels largely rely on vector types both in the OpenCL and HPL versions.

The most relevant conclusion that can be drawn from Fig. 11 is that the performance of HPL applications is very similar to that of the corresponding native OpenCL code. The average slowdown of HPL with respect to OpenCL across these tests was just 1.5% and 1.3% in the NVIDIA and AMD GPU based systems, respectively. The maximum overhead measured has been 6.4% for Floyd–Warshall in the NVIDIA system, followed by a 4.4% for this same algorithm in the AMD system, and it is mostly concentrated in the CPU runtime in both cases. The reason is that this application launches 1024 consecutive kernel executions of very short length (0.1 ms) to the GPU, without any array transfer (only a scalar is sent), and unsurprisingly the HPL runtime incurs in additional costs in the kernel launches with respect to the OpenCL implementation. This was also the main overhead found in the HPL versions of the shallow water simulator, as it is the other application that launches many kernel executions. At this point it is relevant to remember that the measurements were taken using synchronous executions for the benefit of the detailed analysis of all the execution stages. However, HPL runs by default in asynchronous mode, which enables partially overlapping this overhead with GPU computations and transfers. This way, the overhead, in a non-profiled run of HPL with respect to an OpenCL implementation of Floyd–Warshall that also exploits asynchronous execution, is 5% and just 0.44% in the NVIDIA and AMD systems, respectively.

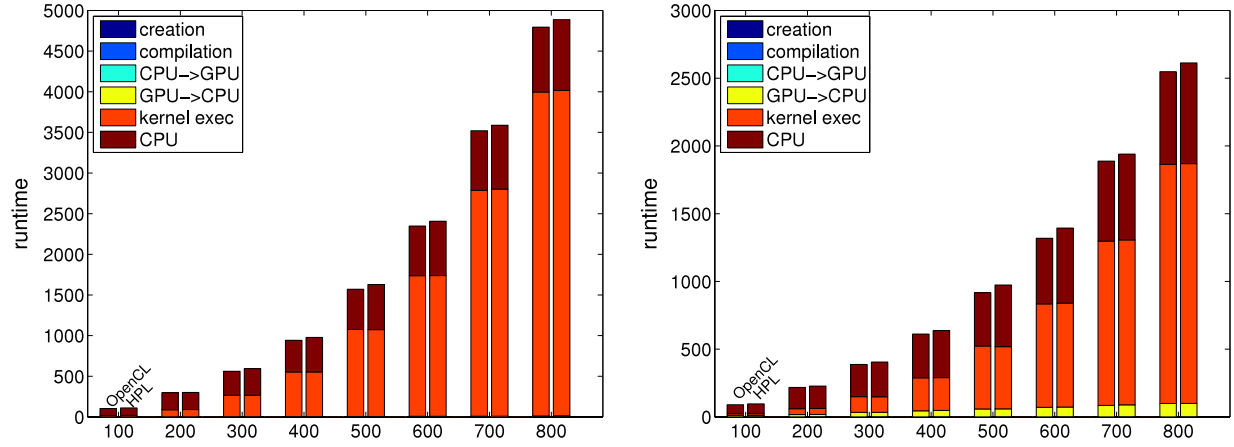
It is interesting to note in Fig. 11 that the same code spends its runtime in quite different activities in the two platforms tested. For example, compilation consumes much more resources in the AMD than in the NVIDIA system. Also, the kernel creation time is always negligible.

Table 3
NAS parallel benchmark EP runtimes for class C (in seconds).

Benchmark	Tesla C2050/C2070		HD6970	
	OpenCL	HPL	OpenCL	HPL
SNU NPB EP	2.905	2.930	4.513	4.536
Locally developed EP	2.745	2.772	4.408	4.428

While our sparse matrix vector product, reduction, matrix transpose and Floyd–Warshall algorithm baseline OpenCL codes are existing works taken from well-known external sources, this is not the case for NAS Parallel Benchmark EP and the shallow water simulator, which we have developed ourselves. Thus it can be interesting to compare these baselines with other works in order to evaluate their quality. Although it is not feasible to find another shallow water simulator with exactly the same characteristics, the quality of our OpenCL implementation can be assessed in our recent publication [24]. As for EP, Table 3 shows the total runtime for problem size C of the SNU NPB suite [34] EP and the EP we developed in the two platforms tested, both when written in OpenCL and in HPL. We can see that HPL has a minimal overhead of around 0.85%–1% for both EP versions in the NVIDIA system, which drops to 0.5% in the AMD GPU. Regarding the performance of our implementation, it is competitive with respect to the SNU NPB implementation, and in fact it outperforms it by a small margin of 5.7% and 2.4% in the NVIDIA and AMD systems, respectively. As a result, our HPL version slightly outperforms the SNU OpenCL native implementation in both platforms.

The runtime of the OpenCL and HPL versions of the shallow water simulator is shown for varying problem sizes in the two platforms considered in Fig. 13. This code was chosen because it is the largest and unlike the others has several kernels, which are invoked repetitively during the simulation, and also because it



(a) Tesla C2050/C2070 system. (b) HD6970 system.

Fig. 13. Runtime of the OpenCL and the HPL versions of the shallow water simulator for different problem sizes.

is an actual complete application. The figure shows the runtimes for mesh sizes from 100 × 100 to 800 × 800 in steps of 100 cells. The runtimes of the OpenCL version went from 103.5 and 90 s for the smallest mesh, to 4794 and 2548 s for the largest in the NVIDIA and AMD systems, respectively. In all of the cases the runtime was mostly dominated by the execution times of the kernels, followed by the operations in the host CPU. The periodic transfers of data from the GPU to CPU are only noticeable in the AMD system. The runtimes were very similar for both versions for all the problem sizes, the average slowdown of HPL with respect to OpenCL being 3.4% and 4.6% in the NVIDIA and AMD GPU based systems, respectively. The HPL overhead is concentrated in the host CPU usage implied by its runtime. As we previously explained, this is a maximal bound of the actual overhead found in a non-profiled run, in which the asynchronous execution between host and device hides part of it.

5. Related work

Much research has been devoted to improving the programmability of heterogeneous systems. This is the case for example of mpC [21], a high level programming language for parallel computing in heterogeneous networks of computers, inspired by HPF but more focused on performance models. The interest of this field has further grown with the rise of modern hardware accelerators. This way, CuPP [7] and EPGPU [22] facilitate the usage in C++ programs of CUDA [10] and OpenCL [29], respectively, by providing a better interface and a runtime that takes care of low level tasks such as memory management and kernel invocation. A higher degree of abstraction is provided by CUDPP [33], a library of data-parallel algorithm primitives that can only run a predefined set of operations and only in CUDA-supported devices. ViennaCL [31] mainly focuses on a convenient C++ API for running linear algebra operations on top of OpenCL, although it also simplifies the execution of custom kernels provided as strings in OpenCL C.

Thrust [4] provides an interface inspired in STL to perform operations on 1D vectors in either CPUs of CUDA-supported GPUs. Its user-defined operations are restricted to being one-to-one, that is, each element of the output vector is computed using a single value from each input vector and the user cannot control basic execution parameters such as numbers of threads or kinds of memory to use.

SkePU [13] and SkelCL [35] further explore the idea of using skeletons to express computations in heterogeneous systems. They can run on top of OpenCL (SkePU also supports CUDA and OpenMP) and they support up to 1D (OpenCL) or 2D (SkePU) arrays. Their skeletons accept user functions in the form of strings for OpenCL, or class member functions for the CUDA and OpenMP backends. However, since these latter functions must be representable as strings, they have in practice the same restrictions as strings. In this way, contrary to HPL kernels, which can capture external variables and perform RTCG even under the control of the programmer, their code must be constant at compile time and include all the definitions of the values they use. An additional restriction in the case of SkePU is that since all its backends use the same user function code, only the common denominator of the language supported by all the backends can appear in the user code, which can preclude many important optimizations. These libraries, which also support the usage of multiple GPUs in a straightforward way, are excellent options to run in heterogeneous devices for those computations whose structure naturally conforms to one of their skeletons.

The PyCUDA and PyOpenCL [20] toolkits simplify the usage of hardware accelerators in the high-level scripting language Python to perform many predefined computations. Custom user functions in the form of strings are also supported, although they are restricted to element-to-element computations and reductions. These projects also emphasize RTCG, although in their case it is based on string processing in the form of keyword replacement, textual templating and syntax tree building. These approaches require learning a new, and sometimes quite complex, interface to perform the corresponding transformations. This contrasts with the natural integration of HPL kernels in C++ and the direct and simple use of this language to control RTCG.

The kind of RTCG provided by HPL is supported by TaskGraph [3] and Intel Array Building Blocks (ArBB) [27] because they also build at runtime their kernels from a representation using a language embedded in C++. TaskGraph combines code specialization with runtime dependence analysis and restructuring optimizations. It has been used to build active libraries that can compose and optimize sequences of kernels [32], and while it exposes no parallel programming model, its authors have explored parallel schemes using it. Regarding ArrBB, it only targets multicore CPUs, however, and it has a very different programming model, with special instructions to copy data in and out of the space where the kernels are run and does not offer the possibility of controlling the task granularity, optimizing the memory hierarchy usage or cooperating between parallel threads.

Copperhead [8] is an embedded language that allows the exploitation of heterogeneous devices, although only NVIDIA GPUs, in order to run computations expressed with data-parallel primitives and a restricted subset of Python, which is its host language. It is a powerful tool for expressing computations that adjust to the usual data-parallel abstractions and in which all the code generation and execution parameters are transparently controlled by the

Copperhead runtime. This results in a high level of abstraction that benefits programmability, but which provides little or no programmer control on the result, which largely depends on the ability of the compiler. These characteristics are typical of compiler directives, which have been also explored in the area of heterogeneous programming [5,23,16,30]. The number of directives and clauses that some of these approaches require to generate competitive code is sometimes on par with or even exceeds the programming costs of library-based approaches. More importantly, the lack of a clear performance model [28] and the suboptimal code generated by compilers in many situations have already led to the demise of promising approaches of this kind such as HPF [18]. The state of affairs is even worse in the case of heterogeneous systems because they allow for more possible implementations for the same algorithm, they have a large number of parameters that can be chosen, and their performance is very sensitive to small changes in these parameters.

6. Conclusions and future work

Heterogeneous systems are becoming increasingly important in the computing landscape as a result of the absolute performance and performance per watt advantage that devices such as GPUs achieve with respect to the standard general-purpose CPUs for many problems. Nevertheless, an improvable aspect of these systems is their programmability and the portability of the codes that exploit them. This paper addresses these issues proposing the Heterogeneous Programming Library (HPL), whose most characteristic component is a language embedded inside C++ to express the computations (kernels) to run in heterogeneous environments. This language allows our library to capture the kernels so that it can translate them into a suitable IR that is then compiled for the device where they will be run. The host C++ language can be naturally interleaved with our embedded language in the kernels, acting as a metaprogramming language that controls the code generated using a syntax that is much more convenient and intuitive than other metaprogramming approaches such as C++ templates. This results in a very powerful run-time code generation (RTCG) environment that is particularly useful for heterogeneous systems, in which users often need to tune the kernels to the specific characteristics of each device to achieve good performance. HPL also provides very convenient interfaces to exploit RTCG to generate highly specialized code for common patterns of computation such as reductions.

During the generation of the IR for a kernel, our library has the opportunity of analyzing and potentially optimizing it, as a compiler would do. While our current implementation performs no code transformations, it does analyze the kernels in order to determine their inputs and outputs. This information allows HPL to track the data dependences between the tasks that the user requests to run in the devices exploiting the asynchronous execution model of the library, as well as between these tasks and the host. This way HPL provides automatic task synchronization while minimizing the number of data transfers. In a related manner, HPL provides rather handy classes to represent data for use in the heterogeneous kernels whose management (creation and deallocation of buffers, tracking of the state and synchronization as required among physical buffers associated with the same logical array in different memories, etc.) is completely automated by our library.

As of today HPL uses as only backend OpenCL, as it maximizes the portability of the applications. The programmability advantages of HPL over OpenCL are not restricted to the points discussed above. The ability to exploit C++ templates in kernels, the detection of errors at compile time, at times with clearer messages, the natural embedding in the kernels of runtime constants, and the transparent indexing of multidimensional arrays further boost HPL programmer productivity.

An evaluation using codes with quite different natures and taken from different sources indicates that HPL provides significant programmability improvements with respect to OpenCL while achieving nearly the same performance in different platforms. In fact, even if we take as the baseline a streamlined version of OpenCL codes in which the initialization and program compilation stages typical of this platform have been removed, the average reduction in terms of SLOCs, programming effort and cyclomatic number achieved by HPL are 34%, 44% and 30%, respectively. Nevertheless, the typical performance overhead is below 5%.

The Heterogeneous Programming Library is an ongoing project in our group. Our future lines of work include the addition of more mechanisms to help further exploit RTCG and ease the exploration of the search space for different versions of HPL computational kernels. We also plan to support distributed memory systems, so that HPL applications can run with minimal effort on clusters. Finally, we have found that skeletons are an appealing alternative for expressing numerous computations, and the integration of typical ones in our framework is thus an interesting extension.

HPL is publicly available under GPL license at http://hpl.des.udc.es.

Acknowledgments

This work was funded by the Xunta de Galicia under the project "*Consolidación e Estructuración de Unidades de Investigación Competitivas*" 2010/06 and the MICINN, cofunded by FEDER funds, under grant TIN2010-16735. Zeki Bozkus is funded by the Scientific and Technological Research Council of Turkey (TUBITAK; 112E191). We also thank Prof. Paul H.J. Kelly for his valuable comments and the Consellería do Mar of Xunta de Galicia and the Centro Tecnolóxico do Mar (CETMAR) for providing the ocean currents and topographic data of Ría de Arousa. Basilio B. Fraguela is a member of the HiPEAC European network of excellence and the Spanish network CAPAP-H3, in whose framework this paper has been developed.

References

[1] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming, Addison-Wesley, 2004.
[2] AMD. Stream computing user guide, 2008.
[3] O. Beckmann, A. Houghton, M. Mellor, P.H.J. Kelly, Runtime code generation in C++ as a foundation for domain-specific optimisation, in: Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23–28, 2003, Revised Papers, in: Lecture Notes in Computer Science, vol. 3016, Springer Verlag, 2004, pp. 291–306.
[4] N. Bell, J. Hoberock, GPU Computing Gems Jade Edition, Morgan Kaufmann, 2011 (Chapter 26).
[5] S. Bihan, G.E. Moulard, R. Dolbeau, H. Calandra, R. Abdelkhalek, Directive-based heterogeneous programming. a GPU-accelerated RTM use case, in: Proc. 7th Intl. Conf. Con Computing, Communications and Control Technologies, July 2009.
[6] Z. Bozkus, B.B. Fraguela, A portable high-productivity approach to program heterogeneous systems, in: 2012 IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops Ph.D. Forum, IPDPSW, May 2012, pp. 163–173.
[7] J. Breitbart, CuPP — a framework for easy CUDA integration, in: IEEE Intl. Symp. on Parallel Distributed Processing, IPDPS 2009, May 2009, pp. 1–8.
[8] B. Catanzaro, M. Garland, K. Keutzer, Copperhead: compiling an embedded data parallel language, in: Proc. 16th ACM symp. on Principles and Practice of Parallel Programming, PPoPP'11, 2011, pp. 47–56.
[9] IBM, Sony, and Toshiba. C/C++ Language Extensions for Cell Broadband Engine Architecture, IBM, 2006.
[10] Nvidia. CUDA Compute Unified Device Architecture, Nvidia, 2008.
[11] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley Professional, 2000.
[12] A. Danalis, G. Marin, C. Mccurdy, J.S. Meredith, P.C. Roth, K. Spafford, J.S. Vetter, The Scalable HeterOgeneous Computing (SHOC) benchmark suite, in: Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU3, 2010, pp. 63–74.
[13] J. Enmyren, C.W. Kessler, SkePU: a multi-backend skeleton programming library for multi-GPU systems, in: Proc. 4th intl. workshop on High-level Parallel Programming and Applications, HLPP'10, 2010, pp. 5–14.

[14] B.B. Fraguela, G. Bikshandi, J. Guo, M.J. Garzarán, D. Padua, C. von Praun, Optimization techniques for efficient HTA programs, Parallel Comput. 38 (9) (2012) 465–484.
[15] M.H. Halstead, Elements of Software Science, Elsevier, 1977.
[16] T.D. Han, T.S. Abdelrahman, hiCUDA: high-level GPGPU programming, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 78–90.
[17] J. Herrington, Code Generation in Action, Manning Publications, 2003.
[18] High Performance Fortran Forum. High Performance Fortran Specification Version 2.0, January 1997.
[19] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, D. Glasco, GPUs and the future of parallel computing, IEEE Micro 31 (5) (2011) 7–17.
[20] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation, Parallel Comput. 38 (3) (2012) 157–174.
[21] Alexey Lastovetsky, Adaptive parallel computing on heterogeneous networks with mpC, Parallel Comput. 28 (10) (2002) 1369–1407.
[22] O.S. Lawlor, Embedding OpenCL in C++ for expressive GPU programming, in: Proc. 5th Intl. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC 2011, May 2011.
[23] S. Lee, R. Eigenmann, OpenMPC: Extended OpenMP programming and tuning for GPUs, in: Proc. of 2010 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC), 2010, pp. 1–11.
[24] J. Lobeiras, M. Viñas, M. Amor, B.B. Fraguela, M. Arenaz, J.A. García, M. Castro, Parallelization of shallow water simulations on current multi-threaded systems, Intl. J. High Perform. Comput. (2013) accepted for publication.
[25] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 2 (1976) 308–320.
[26] National Aeronautics and Space Administration. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/ (last accessed 05.09.12).
[27] C.J. Newburn, B. So, Z. Liu, M.D. McCool, A.M. Ghuloum, S. Du Toit, Z-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, D. Zhang, Intel's array building blocks: A retargetable, dynamic compiler and embedded language, in: 9th Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization, CGO 2011, April 2011, pp. 224–235.
[28] T.A. Ngo, The Role of Performance Models in Parallel Programming and Languages. Ph.D. thesis, Dept. of Computer Science and Engineering, University of Washington, 1997.
[29] Khronos OpenCL Working Group. The OpenCL Specification. Version 1.2, Nov 2011.
[30] OpenACC-Standard.org. The OpenACC Application Programming Interface Version 1.0, Nov. 2011.
[31] K. Rupp, F. Rudolf, J. Weinbub, ViennaCL — a high level linear algebra library for GPUs and multi-core CPUs, in: Intl. Workshop on GPUs and Scientific Applications, GPUScA, 2010, pp. 51–56.
[32] F.P. Russell, M.R. Mellor, P.H.J. Kelly, O. Beckmann, DESOLA: An active linear algebra library using delayed evaluation and runtime code generation, Science of Computer Programming 76 (4) (2011) 227–242.
[33] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, Scan primitives for gpu computing, in: Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS symp. on Graphics hardware, GH'07, 2007, pp. 97–106.
[34] S. Seo, G. Jo, J. Lee, Performance characterization of the NAS Parallel Benchmarks in OpenCL, in: Proc. 2011 IEEE Intl. Symp. on Workload Characterization, IISWC'11, 2011, pp. 137–148.
[35] M. Steuwer, P. Kegel, S. Gorlatch, SkelCL — a portable skeleton library for high-level GPU programming, in: 2011 IEEE Intl. Parallel and Distributed Processing Symp. Workshops and Phd Forum, IPDPSW, May 2011, pp. 1176–1182.
[36] T.L. Veldhuizen, C++ templates as partial evaluation, in: Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'99, January 1999, pp. 13–18.
[37] M. Viñas, J. Lobeiras, B.B. Fraguela, M. Arenaz, M. Amor, R. Doallo, Simulation of pollutant transport in shallow water on a CUDA architecture, in: 2011 Intl. Conf. on High Performance Computing and Simulation, HPCS, July 2011, pp. 664–670.

Moisés Viñas is a Ph.D. student at the Computer Architecture Group (GAC) in the Departamento de Electronica e Sistemas of the Universidade da Coruña, Spain. He received his Computer Science Graduate in 2011 and the Master's degree in High Performance Computing in 2012, both of the Universidade da Coruña. Before being Ph.D. student, he worked on fluid simulation using GPUs. Currently, he is devoted to make easier the programming of heterogeneous architectures through the use of libraries.

Zeki Bozkus received the M.S. and the Ph.D. degrees in computer science from Syracuse University, NY, USA, in 1990 and 1995, respectively. He worked as a senior compiler engineer at the Portland Group, Inc. for six years. He worked as a senior software engineer at Mentor Graphics for the parallelization of Calibre product line for 11 years. He is now an assistant professor at the Computer Engineering Department of Kadir Has University since 2008. His primary research interests are in the fields of parallel programming algorithms, parallel programming languages, and compilers.

Basilio B. Fraguela received the M.S. and the Ph.D. degrees in computer science from the Universidade da Coruña, Spain, in 1994 and 1999, respectively. He is an associate professor in the Departamento de Electronica e Sistemas of the Universidade da Coruña since 2001. His primary research interests are in the fields of programmability, analytical modeling, design of high performance processors and memory hierarchies, and compiler transformations. His homepage is http://gac.udc.es/~basilio.