CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; **29(12)**:e4152–n/a
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.4152

Facilitating the development of stencil applications using the Heterogeneous Programming Library

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

Universidade da Coruña, Grupo de Arquitectura de Computadores, A Coruña, Spain

SUMMARY

Stencil computations are very common in scientific codes. Heterogeneous systems achieve good results solving these problems, but their programming is complex because of the ghost regions required in multi-device implementations and the difficulty to properly exploit their hardware. The Heterogeneous Programming Library (HPL) is a recent framework that improves the programmability of heterogeneous devices. This paper describes two extensions of HPL focused on stencil computations. The first one allows to automatically update the ghost regions they involve. The second one automates the implementation of the computational kernels of these algorithms. In our evaluation the first mechanism reduces on average the number of lines of code and the Halstead programming effort of the host code of comparable HPL baselines by 34% and 64.2%, respectively, while the second contribution reduces these metrics by 72% and 79% in the computational kernels, respectively. Also, the first technique has negligible performance overheads, while the second one matches the performance of manually developed kernels. As an added benefit, the facilitation of the development of these codes thanks to these techniques helps programmers experiment with optimizations suited for this applications such as the ghost cell expansion technique, which provides speedups of up to 13% in our experiments. Copyright © 2017 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: programmability, heterogeneity, OpenCL, stencils, shadow regions

1. INTRODUCTION

Stencil computations are found in many scientific applications (simulations, PDE solvers, image processing, . . .) with computations that follow a fixed pattern. In this pattern, which can appear in computations involving any number of dimensions, the value of an element of the output array depends on the values of elements of the input array that belong to a given neighborhood. In general, the thread in charge of a given position of the output array has to access the same position in the input array as well as several of the elements surrounding it. In shared-memory environments this restriction does not cause any trouble because all the threads can access the whole arrays. Nevertheless, in distributed-memory environments such as systems with several accelerators, where both data and computations are divided among several devices with separate memories, stencil computations require that each device also accesses regions of data that are updated by other device. The traditional solution to this problem is to enlarge the data located in each device with ghost regions, which replicate pieces of data updated by other devices. Still, there is the problem of updating these regions with the correct values that reside in the owner device after each round of computations. The management of the memory coherence of these ghost regions among the different devices gives place to an exchange of information from the owner device to the one where the ghost

*Correspondence to: Basilio B. Fraguela, Facultade de Informática, Campus de Elviña, s/n. 15071, A Coruña, Spain. Email: `basilio.fraguela@udc.es` region resides. In general, stencil computations are performed repetitively in a loop [21, 28] and the result of an array in the current iteration depends on the result of this same array in the previous one. Thus, the update of the ghost regions has to be done each iteration, before starting the next stencil computation.

Heterogeneous devices have proven to be successful choices for challenging applications with stencil computations. Traditionally, the most popular approaches to program these architectures have been low level frameworks [12, 17]. These approaches present several inconveniences such as their large programming complexity and the potential lack of portability in case of choosing CUDA, for example. As a result of this situation there has been much research focusing on improving the programmability of heterogeneous systems [6, 11, 18, 21]. The Heterogeneous Programming Library (HPL) [24], built on top of OpenCL, is one of these efforts. HPL automates all the management done by the backend low level frameworks like OpenCL or CUDA. This includes the transfers between host and devices, and in fact it provides automatic memory coherency for the memory objects it manipulates, which are multidimensional arrays of a class called `Array`.

HPL allows to define pieces of Arrays, called subarrays, that map into a subregion of an existing array. This feature gives programmers a very intuitive way to update ghost regions. While this largely facilitates the development of stencil computations, it still requires users to manually build the ghost regions, select the required origin subarrays and perform the updates by hand. Also, until now HPL did not offer any specific mechanism to help users implement the heterogeneous kernels of their stencil computations. These kernels can be somewhat complex due to the need to exploit the special memories offered by many devices to reach the best performance. Namely, the OpenCL local memory of the GPUs is an excellent resource for this kind of codes due to the high locality of the accesses of the work-items in each work-group and the high speed that this memory offers, the problem being that it has to be manually managed by the programmer.

In this work, we present two enhancements to HPL that are focused on stencil computations. The first one is a new mechanism to update ghost regions that automatically detects and updates them. This new system frees users from the tedious and error-prone tasks involved by the manual update so that they only need to specify when to perform the update. In terms of performance, our proposal does not involve any penalty with respect to the HPL implementations based on manual updates. However, the addition of this automatic system noticeably improves the programmability, which is the main motivation of this work. In a similar line, the second enhancement is a series of templates that automate the efficient implementation of stencil kernels. Our templates allow to easily exploit the hardware resources of heterogeneous devices such as the OpenCL local memory.

The remainder of this paper is organized as follows. The next section explains the semantics and syntax of HPL. Section 3 discusses the problems of the ghost region update and the optimal implementation of heterogeneous stencil kernels, and the solutions we propose, which are evaluated in Section 4. Section 5 summarizes the related work, and the conclusions in Section 6 finish the paper.

2. THE HETEROGENEOUS PROGRAMMING LIBRARY

This library, which is freely available under GPL license at `http://hpl.des.udc.es`, largely facilitates the programming of heterogeneous systems. It uses the OpenCL standard as its backend so that the portability of the developed codes is guaranteed. Both approaches, HPL and OpenCL, share the same programming model and hardware model. The applications run in a general-purpose host processor and several parts of the code, called kernels, are executed on one or more OpenCL-capable devices. Each one of these devices has processors and a separate memory. The processors of a device can only access its memory, so the host is in charge of transferring data from and to each device. We now we summarize the basics of HPL and illustrate its use with simple examples. Detailed explanations on its syntax and functionality can be found in [24, 26, 27].

```
1  void simpleKernel(Array<float,1> b, Array<float,1> a)
2  {
3    b[idx] = 2 * a[idx];
4  }
5  ...
6  Array<float,1> a(N), b(N);
7
8  eval(simpleKernel)(a, b);
```

Figure 1. Example of an 1-D problem in HPL.

2.1. Basics of HPL

HPL applications follow the same scheme as OpenCL or CUDA ones. The main program runs on the host and it is in charge of the management of the parallel executions on the devices. The kernels are functions that can be run on the devices. Each kernel is executed in parallel by a number of threads defined in a space from one to three dimensions called global domain. This global domain can be divided into groups of threads of the same number of dimensions. The size of each group of threads is called local domain. All the threads share the same global and constant memories of the device. Nevertheless, only the threads belonging to the same group can share a fast on-chip memory called local memory. Additionally, each thread has a private memory which is exclusive to it.

HPL allows to express the kernels to run in the accelerators in two ways that can be mixed in the same program. One possibility is to write them in OpenCL C, storing the associated code in strings or separate files that the runtime reads, compiles and executes [27]. The second possibility is to express the kernels using a language embedded in C++ that is described in [24]. The examples in this paper are based in this second alternative, as it integrates better all the code of the application. It also has the advantage that it allows to dynamically build the codes of the kernels using runtime code generation, which can be used to adapt them to the characteristics of the devices or the data at hand, thus enabling important performance improvements [7]; although that is out of the scope of this manuscript. Following this strategy, a kernel is a regular C++ function whose arguments must have type `Array`<*type, ndim [, memoryFlag]*>, where *ndim* is the number of dimensions of the array (*ndim=0* for scalars), *type* is the C++ type of each element of the array and *memoryFlag* indicates the kind of memory (Global, Local, Constant or Private) where this array will be located in the device. If the array is declared in the host without specifying *memoryFlag*, after the kernel invocation it will be allocated at the global memory of the device. Otherwise, if the array is declared in the kernel code with no *memoryFlag* specified, it will be allocated at the private memory. Additionally, there are custom types that facilitate the definition of scalars (`Float`, `Uint`, etc.)

Figure 1 shows in lines 1-4 a simple HPL kernel that computes `b=2×a`. In line 3, each thread (identified by `idx`) takes the element of the position `idx` of `Array a` and computes the value of the element of the same position of `Array b`. HPL provides several predefined variables that contain basic information about the kernel execution. For example, `idx`, `idy` and `idz` uniquely identify each thread within the global domain. As we can see in this particular case, the differences of a HPL kernel with a regular C function are the use of the predefined variables that identify the thread running the kernel and the HPL required datatypes. Other relevant difference is that the control keywords share the same name of those of C but finished with an underscore: `if_(), while_(), for_() ...`. The associated host code, in lines 6-8, declares the kernel arguments and invokes it with syntax `eval(`*f*`)(`*args*`)` where *f* is the kernel function. Non-scalar arguments must be declared in the host code as `Array`s, while standard C/C++ scalars are directly supported. If the user does not specify the sizes of the global and local domains, as in this case, HPL chooses them by itself. It takes the size of the first argument of the kernel as the global domain and it lets the system choose the best option for the local domain. The user can specify these two parameters or the target device by means of several setter functions. For example, `eval(f).device(GPU,`

```
1  void simpleKernel(Array<float,1> b, Array<float,1> a)
2  {
3    b[idx] = 2 * a[idx];
4  }
5  ...
6  Array<float,1> a(N), b(N);
7
8  const int ndevices = getDeviceNumber(GPU);
9
10 for(i = 0; i < ndevices; i++)
11 {
12   Range elems(i * (N/ndevices), (i+1) * (N/ndevices)−1);
13   eval(simpleKernel).device(GPU,i )(a(elems), b(elems));
14 }
```

Figure 2. Example on multiple GPUs using subarrays.

`0).global(50).local(10)(a, b)` runs kernel `f` with 50 threads in parallel divided in groups of 10 threads in the GPU number 0 using the arguments `a` and `b`.

When a kernel execution starts, its inputs must be prepared in the memory of the device, and space for the outputs must also have been allocated. Thus if an `Array` does not have yet a buffer allocated in the device, HPL allocates the memory space for it before starting the execution. Otherwise, HPL will reuse the buffer previously allocated. Moreover, HPL also analyzes the accesses to the `Array`s in the kernels to know whether they are read and/or written. This allows HPL to know where is the most up-to-date copy of an array, and to perform the necessary data transfers so that both the host and the kernels always work with the most up-to-date version of every piece of data. These transfers are automatically done without any user intervention and they only happen on demand, this is, when an array is read in a memory space where no previous copy exists or that copy is outdated. This feature makes the host codes of HPL programs clean and concise while highly efficient.

2.2. *Multi-device applications: Subarrays and Subkernels*

When the main program of an HPL application launches a kernel, it continues its execution in parallel with the execution of the kernel in the device. This allows HPL to launch multiple kernels that can execute in parallel in different devices [25]. Coherency is maintained, as if during its execution the host program, or the execution of a new kernel in a device, requires the contents of an `Array`, HPL checks whether the associated memory has an updated copy according to sequential consistency. If this is not the case, the host program creates or updates that copy with the most recent version, which could imply a wait period if the `Array` is written by a kernel in execution or whose execution is pending.

The initial proposal for multi-device programming under HPL presented in [25] required the use of full `Array`s as unit of transfer and coherency, which does not adapt well to the semantics of most applications. For this reason, three new approaches that allow to create *subarrays* were presented in [26], breaking the `Array` as coherency and transfer unit, and further simplifying the development of applications that exploit multiple accelerators. We now describe the two simplest approaches.

2.2.1. Explicit subarrays: HPL allows the definition of regions inside `Array`s, which are also called subarrays. These subarrays, as their container or parent arrays, are transfer units and share the same memory coherence mechanism, so that the changes made on a subarray are automatically reflected in the container array. Subarrays are created by selecting the region of the parent `Array` they are associated to. This is achieved by indexing this array with ranges of memory positions using the notation `Range(a,b)`, which is associated to the range [a, b]. Figure 2 depicts a multi-device implementation of the example program in Figure 1. As in the single device case, lines 1-4 declare the kernel code. Multi-device execution is driven by the loop of line 10. First, in line 12, we define the range of elements of the arrays that will be processed in each device. Finally, in line 13 the kernel is launched on each GPU found with the corresponding subarrays.

Subarrays can be also used to copy portions of an `Array` using the assignment operator. Assignments are allowed between subarrays with the same size or from scalars, which provokes the replication of that scalar in each element of the target subarray.

2.2.2. Subkernels based on annotations: In the previous example, the *i*-th device works with the *i*-th subarrays of each `Array`, each array having been partitioned in as many subarrays as devices are involved in the computation and following a regular distribution. This scheme is followed by a large number of applications. For this reason, HPL provides support for it by means of annotations that specify how to partition the arrays. Once the user provides this information, HPL takes care of the partitioning, the transfers and the related coherency issues. The annotations take the form of modifiers that are applied to each array argument of a kernel in order to indicate how to partition it for that kernel. For example, the modifiers PART1, PART2 and PART3 specify that an `Array` has to be divided by its first, second or third dimension, respectively. Each one of the devices involved in the execution receives one of these pieces. The non annotated `Array`s are replicated in all the devices. This approach allows to write `eval(simpleKernel)(PART1(a), PART1(b))` for our example code, so that arrays `a` and `b` are divided among all the devices found. This is the most concise way to launch multi-device applications in HPL as in fact, the only change with respect to the single device application is the PART1 modifier.

Let us notice that the annotations just described cannot be used in stencil computations because they split the kernel in disjunct chunks of data, without overlapping. This restriction was removed by extending the annotations to allow overlapping between the subarrays generated. Namely, the annotations support the syntax `ANNOTATION(a,n)` where a is the `Array` to split and n is the number of elements of overlapping in each direction in the selected dimension. If the user needs to specify a different number of overlapped elements per dimension, she can specify them separately with `ANNOTATION(a,x,y)`, where `x` is the number of elements of the current chunk of data overlapped with the previous chunk (lower memory positions) and `y` defines the overlapped region with the following chunk (upper memory positions). This notation allows users to define stencil applications concisely and reducing the possibility of incurring in common errors in these kind of codes.

3. STENCIL COMPUTATIONS IN MULTI-DEVICE ENVIRONMENTS

Stencil computations in heterogeneous systems bring a new challenge for programmers due to the management of the special memories that these devices offer. In addition, in multi-device systems there is the extra complication of the management of the ghost regions. These regions, which replicate the border of the portion of an array involved in a stencil computation that is assigned to another device, are needed for two reasons. The first one is that in these codes the computation of each value of the output array requires accessing a neighborhood around it in its input array, which implies using data assigned to another device when we are working in the borders of the subarray assigned to each device. The second reason is that most accelerators have disjunct memories, and thus the data assigned to a computing device can only be used in another accelerator if we make a copy of it in the memory of that device. Also, since stencil computations typically happen in repetitive loops that continuously modify the values of the arrays involved, these copies must be constantly refreshed with the most up to date value computed in the owner device. Although there are proposals in the bibliography that target the two problems mentioned, we will show that our solution is the one that couples the larger generality and flexibility with the best performance. In the remaining of this section we will first illustrate the complexity of these codes, even using a high level approach such as HPL, which largely facilitates their implementation with respect to lower level frameworks such as OpenCL, and we will then present our proposals to deal with these problems.

```
 1 void simpleKernel(Array<float,1> input, Array<float,1> accum, Int nelems)
 2 { Array<float, 1, Local> tmp(64 + 2);
 3   if_(idx < nelems) {
 4     tmp[lidx + 1] = input(idx);
 5     if_(!lidx) {
 6       tmp[0] = where(idx, input[idx − 1], 0.f);
 7       tmp[65] = where(idx + 64 < nelems, input[idx + 64], 0.f);
 8     }
 9     accum[idx] = tmp[lidx − 1] + tmp[lidx] + tmp[lidx + 1];
10   }
11 }
12 void update(Array<float,1> accum, Array<float,1> array)
13 {
14   array[idx] = accum[idx];
15 }
16 ...
17 Array<float,1> input(N), accum(N);
18 ...
19 for(i = 0; i < 10; i++)
20 {
21   eval(simpleKernel).local(64)(input, accum, N);
22   eval(update)(accum, input);
23 }
```

Figure 3. HPL stencil example code for a single device.

3.1. *Motivating example : A stencil written with HPL*

We will use as running example a simple stencil computation that can be implemented in a single device using HPL as Figure 3 shows. This code computes in each iteration of the loop at line 19 the expression `input[idx] = input[idx-1] + input[idx] + input[idx+1]` by means of two kernels: `simpleKernel`, which computes the accumulation in the temporary array `accum` and `update`, which stores it into `input`. This code verifies the condition of stencil because it is a 1-D problem where the value of `input[idx]` depends on `input[idx]` and its neighborhood. This is a single-device version, where only one device does all the work and therefore there are no ghost regions. Despite this simplicity, the fact that the kernel tries to optimize the memory accesses by exploiting the local memory of the GPU and that we must correctly handle the computations in the borders of the array complicate the implementation. Namely, the kernel uses a temporary storage in the local memory called `tmp` which can store 64 floats, one for each thread in the group (notice the size is defined in the invocation of the kernel in line 21), plus the neighbors needed at the left of the first element and the right of the last one. This storage is usually accessed using the local index of each work-item within its work-group, provided by the predefined variable `lidx`. In line 5, the local thread 0 branches to take care of the loading of these neighbors in lines 6 and 7. These statements use the HPL function `where(a,b,c)` that represents the C expression `a ? b : c`, so that either the neighbors, or the border conditions are loaded. In this example we have assumed that a fixed value of 0 must be used for the borders. As the number of dimensions of the problem and/or the size of the stencil grows, the complexity of the manipulation of the local memory and the boundary conditions also increases. Similarly, the execution of the kernel using multiple devices further complicates the implementation of the kernel. For instance, in our example the element at the left of thread 0 would be a boundary condition in the device 0, but it would be a ghost region that replicates the last element owned by device i for all the devices $i > 0$. In addition, these multi-device executions require a non trivial mechanism of memory management due to the ghost regions. To understand the details of this mechanism, a multi-device version of the example in Figure 3 is presented in Figure 4.

```
1  int obtain_subarrays_sizes(std::vector<int>& s_s, std::vector<int>& s_e, const int ndevices)
2  {
3    int accum_work, current_work;
4    accum_work = 0;
5    const int last = ndevices-1;
6    const float work = 1.0f / ndevices;
7    for(int j = 0; j < ndevices; j++)
8    {
9      if(j == last)
10       current_work = N - accum_work;
11     else
12       current_work = N * work;
13
14     int r_begin = accum_work - (j!=0);
15     int r_end = accum_work + (current_work + (j!=(last))) -1;
16     s_s.push_back(r_begin);
17     s_e.push_back(r_end);
18     accum_work += current_work;
19   }
20 }
21 ...
22 Array<float,1> input(N), accum(N);
23 vector<int> s_s;       // Subarray start points
24 vector<int> s_e;       // Subarray end points
25 const int ndevices = getDeviceNumber(GPU);
26 obtain_subarrays_sizes(s_s, s_e, ndevices);
27 ...
28 for(i = 0; i < 10; i++)
29 {
30   for(int j = 0; j < ndevices; j++)
31     eval(simpleKernel).device(GPU, j).local(64)(input(Range(s_s[j],s_e[j])), accum(Range(s_s[j],s_e[j])), s_s[j]-
           s_e[j]+1);
32
33   for(int j = 0; j < ndevices; j++)
34     eval(update).device(GPU, j)(accum(Range(s_s[j],s_e[j])), input(Range(s_s[j],s_e[j])));
35
36   for(int j = 1; j < ndevices; j++)
37   {
38     input(Range(s_s[j],s_e[j]))(Range(0)) = input(Range(s_s[j-1],s_e[j-1]))(Range(s_e[j-1]-s_s[j-1]-1));
39     input(Range(s_s[j-1],s_e[j-1]))(Range(s_e[j-1]-s_s[j-1])) = input(Range(s_s[j],s_e[j]))(Range(1));
40   }
41 }
```

Figure 4. HPL stencil example in multi-device using subarrays.

The code exploits the advantage that HPL allows the definition of overlapped subarrays [26], that is, subarrays with shared parts of their memory spaces. When the subarrays are updated in different devices, their shared parts are called ghost regions and they must be synchronized to guarantee that the most up-to-date values are available in each device when they are needed. In our example, this involves manually updating the ghost regions of `input` in each iteration as lines 36-40 in Figure 4 show. Another issue is the correct computation of the limits of each subarray needed in any multi-device problem, which is performed in function `obtain_subarrays_size()` (lines [1-20]). The bounds, stored in the vectors `s_s` and `s_e` in the code, are computed taking into account the size of the problem and the number of devices involved, and they are used to select the correct subarray sizes in the kernel executions in the lines 31 and 34. When these kernels finish, the ghost regions of the input array are outdated. Lines 36-40 perform the required update by exploiting HPL facilities such as the use of subarray assignments and nested subarrays, this is, subarrays defined inside another subarray. For example, when `j` and `ndevices` are 1 and 2 respectively, in line 39 the

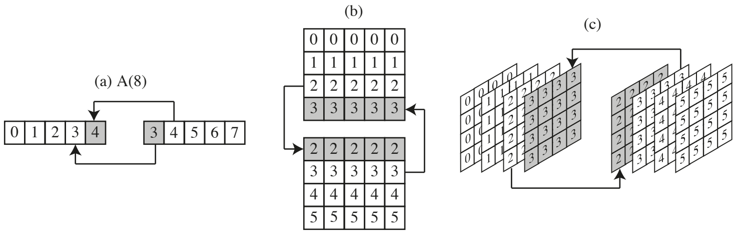

Figure 5. Examples of ghost regions for (a) 1-D, (b) 2-D and (c) 3-D problems in a problem divided by two devices.

2nd element of subarray `input(Range(N/2-1, N-1))` is copied in the last element of subarray `input(Range(0,N/2))`. This is the update of the ghost region of the GPU 0. Similarly, the update of the ghost region of GPU 1 takes place in line 38. Once the ghost regions have been updated, a new iteration can start. Notice that this update process becomes even more complex and error-prone as the number of dimensions of the problem involved increases.

3.2. Automatic update of the shadow regions: syncGhosts

Ghost regions always appear in the frontiers of the subarrays. Figure 5 illustrates typical examples of ghost regions, which are the shadowed cells, for three spaces (1-D, 2-D and 3-D). For example, in the case 5(a), where the vector `A` of eight elements is distributed, the device in charge of processing the elements [0-3] has one extra element that replicates A(4), which is updated in parallel by the other device. The same happens in the second device with A(3). When the parallel executions of both devices have finished, these ghost regions must be updated because they are outdated. We now explain the details of the novel automatic mechanism to update ghost regions in HPL.

A first thing to take into account is that ghost regions are naturally generated in HPL by selecting subarrays that overlap in their borders and which are used in computations in different devices. No matter the subarrays are generated by means of explicit indexing, as in the example in Figure 4 or by means of an automatic distribution of an `Array` using annotations, the HPL runtime keeps track of their size and relative position within the global `Array` where they are defined. In addition, as explained in [26], the HPL coherency system knows where the current and the outdated versions of every `Array` are, no matter it is a subarray of another `Array` or not. With this information it is possible to infer which are the ghost regions and which device owns each portion on them. For example, in Figure 5(a) since one device would work on A(Range(0,4)) and the other one on A(Range(3,7)), HPL would realize that A(Range(3,4)) is shared between both devices, A(3) being actually owned by the first device and A(4) by the second one. In this case the overlapped area has just two cells, but it could be more complex, containing elements in several dimensions when multidimensional problems are addressed. Also, the ghost regions can have a width of more than one element, which is achieved simply by making bigger the region shared between the subarrays or by using a larger value for the overlapping extent when using annotations. The HPL runtime has been enhanced to correctly identify all these situations and determine which part of each shared region is owned by each device based on its relative position within the region. With this information, HPL can apply its automatic update algorithm, called syncGhosts, whose pseudo-code for the one-dimensional version is shown in Figure 6.

Several steps are performed for each ghost region of the `Array`. First, its size is computed in line 3. It deserves to be mentioned that regions always have an even size, because the number of ghost elements that each subarray has, matches with those of the neighbor in charge of their update and vice versa. Then, the overlapped arrays are selected: the SL or lower positions subarray, and the SU or upper positions subarray (lines 4 and 5). After this, two memory copies are done. First, in line 6, the data of SU needed to update the ghost region in SL are transferred. Then, in line 7, the appropriate data of SL are copied in the ghost region of SU. HPL takes into account the location of the most up-to-date copy of these subarrays so that only the minimum number of memory copies

```
1 Algorithm syncGhosts($A$)
      input: Array with overlapped subarrays
2     foreach $ghost\_region$ in $A$ do
3         $N = size(ghost\_region)$
4         $SL = ghost\_region.getLowerSubarray()$
5         $LU = ghost\_region.getUpperSubarray()$
6         $SL(Range(size(SL) - N)) = SU(Range(N/2, N))$
7         $SU(Range(0, N/2)) = SL(Range(size(SL) - N/2))$
8     end
```

Figure 6. 1-D `syncGhosts` algorithm.

```
int obtain_subarrays_sizes(std::vector<int>& s_s, std::vector<int>& s_e, const int ndevices)
{...}
...
Array<float,1> input(N), accum(N);
vector<int> s_s;        // Subarray start points
vector<int> s_e;        // Subarray end points
const int ndevices = getDeviceNumber(GPU);
obtain_subarrays_sizes(s_s, s_e, ndevices);
...
for(i = 0; i < 10; i++)
{
  for(int j = 0; j < ndevices; j++)
    eval(simpleKernel).device(GPU, j).local(64)(input(Range(s_s[j],s_e[j])), accum(Range(s_s[j],
         s_e[j])), s_s[j]−s_e[j]+1);

  for(int j = 0; j < ndevices; j++)
    eval(update).device(GPU, j)(accum(Range(s_s[j],s_e[j])), input(Range(s_s[j],s_e[j])));

  input.syncGhosts();
}
```

Figure 7. Example on multiple GPUs using subarrays with `syncGhosts`.

will be done to keep updated the ghost regions. The algorithm shown in the figure does not express all the details for the different shapes of the ghost regions. The algorithm for 2-D and 3-D problems is much more complex due to the introduction of one or two additional dimensions respectively. The process is also optimized so that these memory copies do not necessarily imply a true exchange of data between two devices. Namely, they will be done only if they are strictly necessary, i.e., only ghost regions that are not up-to-date will be updated.

The *syncGhosts* algorithm is run on an `Array` when the user invokes the new method `syncGhosts()` on it. Figure 7 shows the code of Figure 4 after replacing the manual updates with a *syncGhosts* call. The `obtain_subarray_sizes()` method has the same body as in the example of Figure 4. It deserves to be mentioned that the benefit of the `syncGhosts()` call is not exclusively related to the lower number of lines of code, but also with the complexity of the computations of the regions of data to exchange. Using the array distributing annotations, the same example of Figure 7 could be rewritten as Figure 8 shows.

Notice that coupling the use of annotations with `syncGhosts()`, the code related with the definition of subarrays, such as function `obtain_subarrays_sizes()`, is no longer needed. As we can see, the complexity related to the creation of subarrays and their exchange is largely reduced using this new approach.

```
Array<float,1> input(N), accum(N);

for(i = 0; i < 10; i++)
{
  eval(simpleKernel).local(64)(PART1(input,1), PART1(accum,1), input1.part(1));
  eval(update)(PART1(accum,1), PART1(input,1));
  input.syncGhosts();
}
```

Figure 8. Example on multiple GPUs using annotations with `syncGhosts`.

3.3. *HPL stencil templates*

As discussed in Section 3.1, the implementation of stencil kernels in the kind of environment we are considering has many complications. Some of them, such as the boundary conditions, are in common with general purpose CPUs. Others, such as the different treatment that the boundaries require in different devices, depending on the chunk of the computation assigned to each device, appear also in any distributed-memory system. Finally, other issues are very particular of heterogeneous systems such as GPUs. This is the case of the decomposition in groups of threads that can only synchronize with the other threads in the same group or the need to manually manage local memories of a restricted size that can only be shared by the threads within each group. The fact that HPL offers a language embedded in C++ allows to write on top of it complex software that exploits advanced C++ features, including not only classes and templates, but even the variadic templates provided by C++11. By exploiting these features, we have developed a series of templates that automate the generation of heterogeneous kernel codes, only requiring from the user basic informations such as the nature of the boundary conditions, the size of the stencil in each direction, and of course, the stencil computation to perform.

Figure 9 illustrates the implementation of the computational kernel of well-known Conway's Game of Life game, which simulates through time the evolution of the existence of life in each cell in a matrix depending on the number of neighbors that are alive and the current status of the cell. The user is in charge of writing a basic function in HPL that performs the computation for a single point (lines 1-8). The first argument of this function is always an `AliasArray` object, which is basically an indexable pointer to the element to process in the input. In a problem with N dimensions, this pointer is followed by $N-1$ integers that contain the displacements required to access elements through this pointer in the $N-1$ non-first dimensions (the displacement in the less significant dimension being always obviously 1). Finally, this can be followed by arbitrary arguments needed by the computation; in this example no extra parameters are needed. It is important to mention that although this API must be written in HPL, parts or all of the internals can be written in OpenCL C just by first associating the OpenCL code to an interface function as explained in [27] and then invoking it from the HPL function using the `call` function, discussed in [24].

The user must also define the kernel function (line 10) so that HPL learns which are the arguments it requires. Our new proposed HPL feature is used inside this kernel: the programmer just invokes a suitable stencil template, for example here `StencilKernel2DF` for a 2D stencil that uses a fixed constant value for its boundary conditions. All the templates are parameterized by the size of the stencil in each dimension. In this case the stencil extends one step in the four directions. This is followed by the list of arguments. All our templates require the input, the output, and the array in local memory to use in the stencil, as well as a helper array with values used by the internals of the template. In this case, because the boundary condition consists of a fixed value, it is the next argument, here a zero. The last compulsory argument of the template is the computational function. After this point any additional argument provided to the stencil template is considered as as additional argument for the computational function. These arguments are provided after the standard ones described before.

```
 1  Int life(AliasArray<int> tmp_v, Int lcl_row_sz)
 2  {
 3    Int tmp = tmp_v[−lcl_row_sz − 1] + tmp_v[−lcl_row_sz] + tmp_v[−lcl_row_sz + 1] +
 4              tmp_v[−1] + tmp_v[1] +
 5              tmp_v[lcl_row_sz − 1] + tmp_v[lcl_row_sz] + tmp_v[ lcl_row_sz + 1];
 6
 7    return (tmp == 3) || (tmp_v[0] && (tmp == 2));
 8  }
 9
10  void kernel(Array<int, 2> in, Array<int, 2> out, Array<int, 2, Local> tmp, Array<int, 1> helper)
11  {
12    StencilKernel2DF<1,1,1,1>(in, out, tmp, helper, 0, life);
13  }
14
15  //Partition 'in' by the 1st dimension on nGPUs GPUs, use a 16x16 local domain and process 4 elems/thread
16  auto s = PrepareStencilKernel2D<1,1,1,1>(in, 1, nGPUs, Domain(16, 16), 4);
17  eval(kernel).global(s.global)local(s.local)(PART1(in, 1), PART1(out, 1), s.temp, s.helper);
```

Figure 9. Kernel of Conway's Game of Life based on HPL stencil templates

The helper array is initialized in the host by a HPL function that prepares and object with information to launch and execute the kernel. The user provides to this function, called `PrepareStencilKernel2D` in line 16, the extent of the stencil in each dimension, the input array, the dimension that will be partitioned, the number of devices to use, the work-group size and the the desired number of elements that each parallel thread will process. The function simultaneously initializes the helper vector, which just contains a few integers with offsets and sizes to use in each device, a temporary vector located in local memory with the appropriate size, and vectors with the global and local domains for the kernel executions. All of them are stored in the same information object, called `s` in the figure. Finally, line 17 launches to execution the stencil kernel using the elements prepared by this function. As we see, for the user both the helper and the local memory arrays operate as black boxes that just must be provided to the kernel and the stencil template.

4. EVALUATION

In this section we evaluate the two improvements of the HPL library proposed in this paper. Both of them aim to improve the programmability of the stencil computations, the automatic mechanism for updating ghost regions being only of interest in multi-device environments. Our proposals have also the virtue of reducing the errors typically related to the management of boundary conditions, local memory manipulation, and ghost regions. It is also important to ensure that these improvements do not hurt the performance of applications. Therefore in this section both the performance and the programmability impact of our extension are considered.

For these studies, we have selected five well-known benchmarks that use stencil computations. The main characteristics of each benchmark are shown in Table I as follows. After the name, the number of dimensions of each problem appears in the second column and the third one defines the size of the problem tested. The last three columns are the shape of the stencil used in each benchmark, the number of iterations needed in each execution and finally, the number of kernels launched in each iteration/execution. Briefly, *CANNY* and *GAUSS* are two image processing benchmarks. *CANNY* is an algorithm used for detecting edges in images. A simple solution consists in four steps, one kernel per step. First, a Gaussian filter is applied to smooth the image. Then, an edge detection operator is applied (e.g. Sobel). Third, a non-maximum suppression is performed as an edge thinning technique. Finally, a double threshold is applied to reduce the variety of output values. *GAUSS* is an image processing algorithm aimed at reducing the noise in images. Regarding their structure, the main difference between GAUSS and *CANNY*, is that the former one is usually

Table I. Benchmark details.

Benchmark	Dimensions	Problem size	Stencil shape	Iterations	Kernels
CANNY	2-D	4096×4096	5×5	1	4
GAUSS	2-D	4096×4096	11×11	1000	1
JACOBI3D	3-D	512×512×512	3×3×3	1	1
LIFE	2-D	2048×2048	3×3	5000	1
SHWA	2-D	2000×2000	3×3	100000	3

Table II. Programmability improvement without and with the syncGhosts mechanism of HPL programs based on annotations with respect to versions based on explicit subarrays, and relative improvements due to the use of the syncGhosts mechanism in the codes based on annotations.

Benchmark	HPL annotations with manual sync vs HPL subarrays with manual sync		HPL annotations with syncGhosts vs HPL subarrays with manual sync		HPL annotations with syncGhosts vs HPL annotations with manual sync	
	ΔSLOC	ΔPE	ΔSLOC	ΔPE	ΔSLOC	ΔPE
CANNY	18.18	23.48	67.90	93.90	60.77	92.03
GAUSS	15.63	20.40	47.66	79.01	37.96	73.63
JACOBI3D	12.28	20.79	37.72	75.42	29.00	68.97
LIFE	17.05	14.33	49.61	73.11	39.25	68.61
SHWA	20.86	34.61	23.31	46.18	3.10	17.69

performed inside a loop instead of a fixed number of steps. JACOBI3D is an algorithm that is very used in scientific computations, as it is the simplest approach to a numerical solution of the 3-D Laplace Equation via relaxation. LIFE is the Conway's Game of Life, a game that simulates the evolution of a 2-D environment. Each cell can be dead or alive and following several behavior rules each cell changes its state during the game. Finally SHWA is a shallow water simulator with pollutant transport presented in [28]. This iterative benchmark needs three stages/kernels to compute the evolution of a mesh of finite volumes: in the first one, the flux variation among the elements of the mesh is computed. Then, the global time step is computed for each iteration. Finally, the new flux value is computed taking into account the variations computed at the first stage.

4.1. *Programmability*

We will first focus on the programmability advantages of the mechanism to automatically synchronize ghost regions. HPL has proven to be a good alternative to low level approaches like OpenCL or CUDA in terms of programmability and performance as we can see in [25, 24]. Also, its evaluation in [26] showed that the usage of subarrays largely improved the programmability while having a negligible impact on performance when compared to native OpenCL. Thus, the baseline for our evaluation is an HPL implementation using subarrays written following the strategy explained in Section 2.2.1. We use two objective metrics based on the source code to measure the programmability of the different approaches tested, the source lines of code (SLOCs) and the Halstead's programming effort [10] (PE, expressed in thousands) of the host side. This latter metric is an estimation of the cost of the development of a code taking into account the number of unique operands, unique operators, total operands and total operators found in the code. In our opinion, this metric is a fairer measurement than the SLOCs because the length and complexity of a SLOC can widely vary. We measured only the host code because the feature presented does not affect the kernels.

Table II shows the percentage reduction of SLOCs and programming effort of two HPL versions based on annotations (Section 2.2.2) with respect to versions written using subarrays (Section 2.2.1) and manual updates. One of the versions based on annotations performs manually the update of the ghost regions , while the other one uses *syncGhosts*. Also, the last two columns show the relative SLOC and programming effort reduction among the two codes based on annotations. These two versions reduce significantly the programming effort (PE) with respect to the use of subarrays for

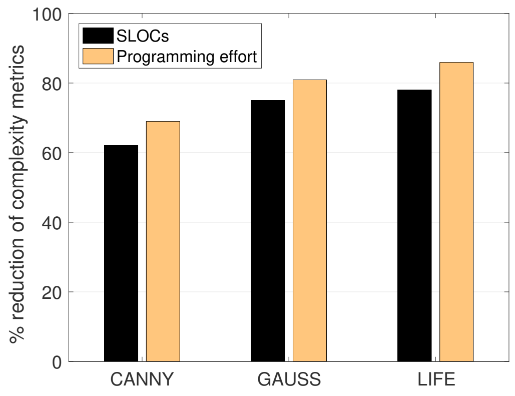

Figure 10. Programmability improvement of the computational kernels thanks to the usage of HPL stencil templates.

all the benchmarks. The largest programmability improvement achieved by annotations without *syncGhosts* takes place in SHWA, where the SLOCS and PE are reduced by 20.86% and 34.61%, respectively. The improvements are much bigger in all the benchmarks when *syncGhosts* is used, reaching its maximum values in CANNY, which requires 67.9% and 93.9% fewer SLOCS and PE than the subarray-based HPL version, respectively. In fact, while the average PE reduction when using annotations instead of subarrays is 22.72%, this reduction grows to 73.52% (3.2 times larger) when *syncGhosts* is also used. In terms of SLOCS, the average reduction achieved is 16.8% for the first case, and 45.24% for the second one. As the last two columns show, when the use of subarrays is factored out of the comparison, the use of *syncGhosts* reduces the number of SLOCs and the programming effort of the version based on annotations by a noticeable average of 34% and 64.2%, respectively.

Our stencil templates have an even clearer impact on the development cost of this kind of applications. The reason is that while HPL, even as described in its original version in [24], largely reduced the complexity of the host-side of heterogeneous applications, this is not the case in the computational kernels, which are either written in native OpenCL C, or in the HPL language, which has basically a one-to-one correspondence to OpenCL C. Figure 10 shows the reduction of the complexity metrics considered in the kernels of three of our benchmarks when they are written using stencil templates instead of manually. JACOBI3D is not included because according to [22] the use of local memory for 3D stencils is of little help and sometimes even counterproductive, thus our kernel directly access the data from the global memory. Since this noticeably simplifies the implementation of the kernel, a stencil template for it would be of little use, the programming cost being very similar to that of the development by hand. For this reason, the kernel does not involve any template. Regarding SHWA, although it can benefit from local memory [28], the implementation using the generic stencil templates we developed would be less efficient than a manual implementation, and thus the kernel used was written manually. There are two main reasons why the stencil templates are not a good match for the stencil kernel of this problem. First, this kernel follows a stencil pattern to access two arrays, not a single one. This problem could be solved fusing both arrays in a single one, as they have the same shape. But this would result in decreased performance because one of them is read-only, and is only used in this kernel, while the other one is modified in the third kernel. Fusing them would thus result in worse locality in the third kernel and in an increase of unnecessary data to transfer in the ghost region updates. The second reason is that the SHWA stencil kernel has two different outputs. Again, they could be fused because they have the same shape, but one of them is only used in the second kernel, while the other one is only used in the third kernel of this application, so again, much locality would be lost. Also, in both cases the memory requirements of the application would grow considerably because one of the arrays is made of `float4` vectors, while the other one is a matrix of scalar `float`s, thus, due to alignment requirements, a C struct composed by both elements would require 32 bytes,

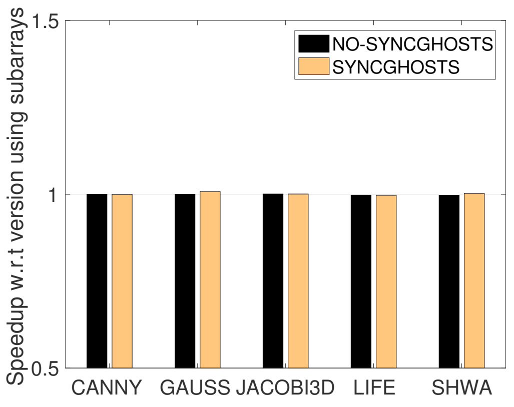

Figure 11. Performance in the Fermi system using both GPUs.

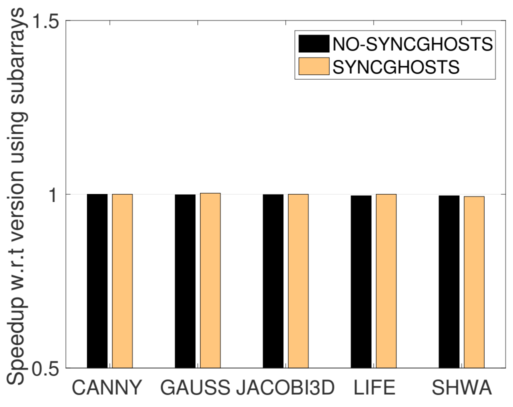

Figure 12. Performance in the K20 system using two GPUs.

compared to the 20 bytes of the separate implementation, thus requiring a 60% larger footprint. It is interesting to notice that the fact that HPL supports the use of arbitrary manual kernels, allows users to attain the best performance while enjoying the programmability benefits of the automatic ghost region synchronization just evaluated. Nevertheless, solutions based on skeletons [6, 21] enforce the assumption of a single stencil input and a single stencil output, with the negative consequences just described. As we can see in Figure 10, in those situations in which our stencil templates are applicable, there is a drastic reduction in the complexity of the implementation of the kernel. The reason for the slightly smaller reduction in CANNY is that its fourth kernel is not actually an stencil, but a map operation, and thus this enhancement is not applicable to it.

4.2. *Performance of the automatic ghost synchronization*

The performance evaluation took place in two systems called Fermi and K20. Fermi has an Intel Xeon X5650 CPU with 6-core and 12 GB of memory. Additionally, it has two GPUs Nvidia M2050 with 3 GB of memory. K20 has two 8-core Intel E5-2660 with 64 GB of memory connected to 3 GPUs NVIDIA K20m with 5 GB per GPU. The compiler used to obtain our measurements is g++-4.7.2 with optimization level -O3. The size of the input problem chosen for each benchmark is reflected in the third column of Table I. Regarding performance, another relevant parameter to take into account in this kind of applications is the shape of the stencil used in each case (fourth column). Finally, the fifth column includes the number of iterations of each benchmark. In summary, CANNY and GAUSS need 4096×4096 pixels image as input. CANNY uses a stencil shape's extent of 2 in each direction and GAUSS a stencil with 5 elements in each direction and 1024 iterations. A 3-D matrix of 512 elements per dimension was used in JACOBI3D. A 2048×2048 mesh and 5000 iterations is the configuration of LIFE, while SHWA performed simulations of a mesh of 2000×2000 cells. In these three last benchmarks the extent of the stencil shape was of one element in each direction.

Figures 11 and 12 show the speedup of the HPL versions based on annotations with respect to the subarray versions based on manual updates when two devices are used in our Fermi and K20 platforms, respectively. As we expected, the *syncGhosts* mechanism does not add any overhead to the manual mechanism. In fact, the performance differences among the three versions do not reach 1% for any benchmark in both systems.

In order to demonstrate that the good performance obtained is independent of the number of devices, the same benchmarks were run using the three devices available in our K20 system. It is remarkable that in this configuration, there is a device that exchanges ghost regions with the other two. Figure 13 shows the speedup of the two versions based on annotations that update the ghost regions manually (*no-syncGhosts*) and automatically (*syncGhosts*) with respect to our baseline HPL version based on subarrays and manual updates in this environment. The differences among the three versions are again minimal, as the maximum performance difference observed between any two

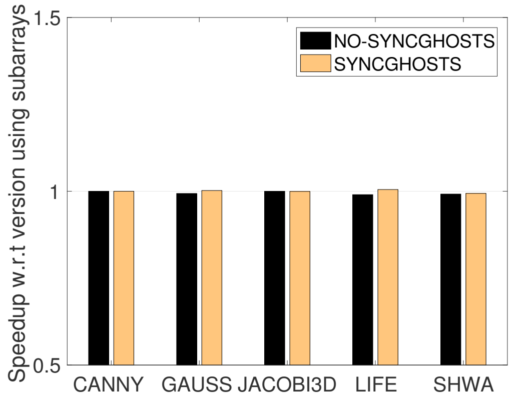

Figure 13. Performance improvement using 3 GPUs in the K20 system.

versions of the same benchmark is below 2%. These results indicate that our *syncGhosts* mechanism does not cause any meaningful performance penalty in comparison with manual versions.

4.3. Performance of the stencil templates

Our stencil templates efficiently generate the code of the kernel using the HPL runtime code generation properties, which have proved to have a negligible impact on performance [24]. Indeed, we have observed exactly the same performance in the code generated by our templates and the versions we wrote for comparison purposes by hand, which is not surprising since they follow the same strategy. Therefore this section compares the quality of the code of our stencil templates with the code generated by the tool with the most similar properties, which is SkelCL [21], a family of skeletons for OpenCL that includes two skeletons related to stencil computations. One of them, *MapOverlap*, which is also present in the SkePU library [6], generates partially overlapping subarrays in the heterogeneous devices, i.e., with overlapped regions, but it offers no mechanisms to keep them updated other than rebuilding the whole array in the host side and redistributing it again. As a result it is not suitable for iterative stencils. The second skeleton, called *Stencil*, offers automatic ghost region updates, although only for stencils that are executed in sequence or repetitively. Although this is much better than MapOverlap, it is still far from the capabilities of HPL. For example, in SHWA the iterative process involves a stencil computation (kernel 1), followed by a global reduction (kernel 2), and a map operation (kernel 3), the ghost region update being required after this third kernel. Therefore it is impossible to efficiently implement this sequence using the Stencil skeleton in [21]. Regarding the kernel used in the heterogeneous devices, both skeletons automatically generate it from a function that describes the computation for each point of the domain and a description of the properties of the stencil such as the extent in each dimension or the boundary conditions. Their approach is thus very similar to that of our stencil templates.

Since MapOverlap cannot be competitive in iterative stencils, we have compared in Figure 14 our proposal with the SkelCL Stencil skeleton using the benchmarks that consist of a single stencil kernel. Notice that even if we had written JACOBI3D with our stencil templates, we could not have included it in this comparison because it cannot be implemented with SkelCL. The reason is that it does not support arrays of 3 dimensions, and even if we used a flattened representation to try to avoid this limitation, the stencil kernels built by SkelCL and the indexing functions the user must use in her custom computational function only consider 2 dimensions. The two problems used have a very different nature, as GAUSS is based on single-precision floating point operations, has a stencil size of 11, and it requires a mask array that is used in the processing of each input pixel, while LIFE has only integer operations, a stencil extent of just one element, and no additional inputs besides the universe to simulate. We can see that HPL achieves speedups between 6.1 and 19.7 with respect to SkelCL. The figure classifies this speedup in three regions. The first one, labeled basic speedup, is achieved by putting each parallel thread of the kernel in charge of the computation of a single point of the domain, and in the case of GAUSS, locating the mask array used in the computations

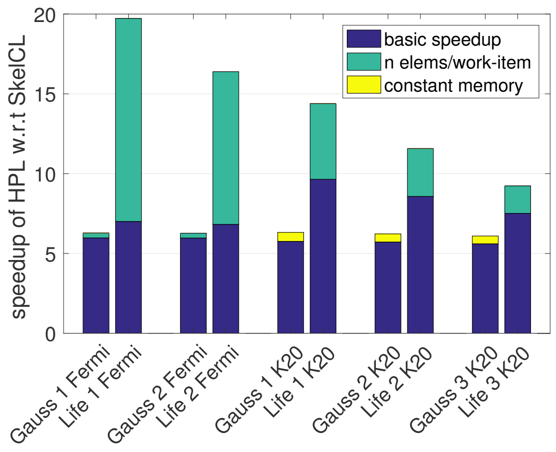

Figure 14. Performance comparison between HPL and SkelCL stencil kernels.

in global memory. Our stencil template, contrary to the stencil kernels of SkelCL and SkePU, also supports the calculation of multiple points of the domain by the same thread. We can see that this provides a small performance benefit for GAUSS, although only in the Fermi GPUs. Nevertheless, it is responsible for most of the speedup of LIFE in the Fermi GPUs and an important portion of it in the K20 units. Finally, we also experienced with declaring the mask used by GAUSS in the GPU constant memory, something that again neither SkelCL nor SkePU support as of today. While this had no impact in the executions in the Fermi GPUs, it was beneficial for the executions in the K20 GPUs. Regarding the huge performance difference between the HPL stencil template kernels and the SkelCL Stencil skeleton, an analysis of the latter revealed that the problem is concentrated in the OpenCL kernel built by the skeleton. Besides losing optimization opportunities such as the ones discussed before, the SkelCL kernel entrusts the filling of the local memory to a single row of threads of each bidimensional group of threads. This is in contrast with our implementation, in which this process involved all the threads in the group.

4.4. Ghost Cell Expansion

One of the most common techniques applied to applications with stencil computations in distributed memory systems is known as Ghost Cell Expansion [4]. In those systems, ghost region updates are done at the process level and they involve message passing among processes. These messages typically reduce the performance of the application. The ghost cell expansion (GCE) technique reduces their impact by decreasing their frequency. This is achieved using larger ghost regions so that more iterations can be performed without requiring an update, as the size of the updated region shrinks in each iteration. The price to pay is that each update is more expensive, as it involves a wider ghost region. For this reason the best performance is typically found with an intermediate ghost region width that balances the number of updates and their weight.

The GCE technique can be also used in multi-device systems since they are a kind of distributed memory systems. Using this technique in multi-device systems, the user performs less but heavier memory copies between devices. For example, in a 2-D problem with a ghost region of one row, each device needs to update its ghost region every iteration. With two rows per region, in the first iteration, one of the ghost rows can be updated avoiding its update in that iteration. In general, with N rows per ghost region, the ghost rows that will be read in the next iteration can be updated N-1 times without perform any memory copy between devices. The implementation of this technique in HPL using annotations is straightforward thanks to the freedom that the user has to locate the *syncGhosts* calls where it is necessary, either once per iteration or each N iterations.

The syncGhosts algorithm and the notation introduced in this paper largely simplify the application of GCE in heterogeneous applications involving several devices. For this reason, we tested its application to our set of benchmarks using our notation, the results for two GPUs in the

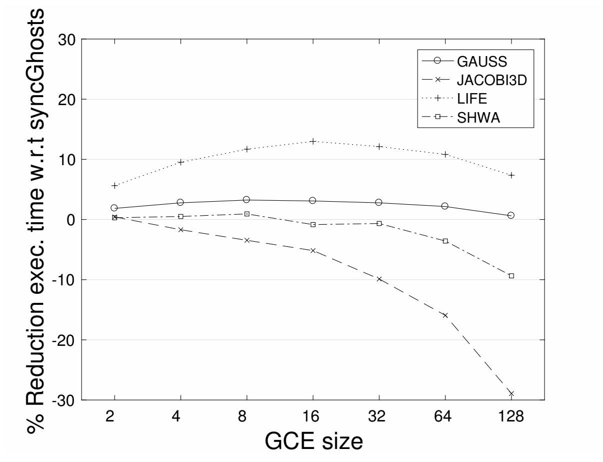

Figure 15. Performance of *syncGhosts* versions varying the ghost region sizes and using two GPUs in K20 system.

K20 systems being shown in Figure 15. Notice that CANNY is not included because it does not have an iterative nature, and thus GCE is not appropriate for it. The y axis indicates the percentage of reduction of the execution time of the *syncGhosts* version using ghost regions of different sizes with respect to the *syncGhosts* version with a ghost region of only one row. The x axis specifies the extra width of each ghost region. With ghost regions of width 2, the *syncGhosts* call is located every 2 iterations; with width 4, every 4 iterations, and so on.

GCE is beneficial for all the benchmarks but JACOBI3D. The reason is that this is a 3-D problem, and thus the increase of the width of the ghost region in one unit involves adding a whole slice (in this case of 512×512 elements) to the region. In this situation the exchange of regions becomes too expensive to compensate for the reduced update frequency. All the other benchmarks show some degree of improvement for moderate amounts of GCE, ranking from a small 1% improvement of SHWA for 8 elements to a worthy 13% for LIFE with 16 elements, in both cases each element implying a row of the underlying array. The effect of GCE on the different benchmarks is quite different because it depends on many parameters such as the increase of the ghost region size in bytes it implies (not only because of the number of elements, but also because of the number of bytes required to represent each element), the computational cost associated to the larger ghost regions, whether there is a need to synchronize the devices because of other kernels used in the application, etc.

As we can see, GCE can often improve the performance of stencil multi-device codes, and our notation makes it very easy to implement it and experiment with its width. For example, using annotations it is just enough to increase the extent of the overlapped region, by changing a single number, and to reduce the frequency of invocations to the *syncGhosts* method just by adding an appropriate conditional.

Overall, the good results achieved by our solution both in terms of performance, this is, the absence of overhead, and programmability, turn HPL into an excellent tool for the development of stencil codes.

5. RELATED WORK

A first source of related work has to do with the efforts to support ghost regions in traditional distributed memory systems. Some approaches managed to totally hide their existence [13, 2], although at the risk of failing to provide good performance or asking too much from the existing compiler technology. For this reason, most proposals have required some indications from the user, either by asking to partition the computation [20, 1], or by associating the ghost regions to (semi)automatically managed data structures rather than to regions of code [19, 16, 9, 15].

There has been much research aimed at improving the programmability of heterogeneous systems. Some of these efforts, like ours, are specifically targeted to stencil codes, given the difficulties for their development, particularly when multiple heterogeneous devices are used. In particular, HLSF [5] provides a high-level interface for describing stencils hiding the low level details on single-device systems. It is built on top of CUDA so that only CUDA-capable devices are supported. PARTANS [14] automatically optimizes the distribution of data taking into account both the problem and the device characteristics. It presents an interface consisting of two main classes, *Volume* and *Stencil*, which define the elements of a grid and the operation to do with each one of them, respectively. PARTANS is more focused on auto-tuning stencil applications than on improving their programmability, which is the main goal of HPL. PATUS [3] is an auto-tuner of stencil applications in the same fashion as PARTANS but while it allows the user to define stencils and strategies to parallelize and optimize them, it only supports single-device environment based on either CPUs by means of OpenMP, or CUDA-capable devices.

The programming of stencils on single or multiple accelerators can also be facilitated by skeletons [6, 21]. An important limitation of approaches based on skeletons is that they can only be applied to applications in which all the kernels have computational patterns that can be accommodated to one of the available skeletons. Also, although sometimes applications can be modified to match these specific patterns, this can result in important performance costs and/or the requirement of many more resources, as we explained in Section 4.1. In contrast, HPL is a completely general solution. Many other restrictions of the stencil skeletons provided by these libraries compared to HPL were already discussed in Section 4.3.

Another limitation of multi-device solutions like [6, 14, 21] is that they do not allow to separate the kernel launch from the synchronization of the ghost regions. This prevents users from delaying the synchronization, thus making it impossible to take advantage of the gap between the kernel execution and the synchronization in order to let the host to do useful work in the meantime. With HPL users can freely locate useful work between the kernel execution and the swapping of the ghost regions. This is a very interesting option when GCE is applied because the kernels are even heavier because of the recomputed cells.

The parallelization of heterogenous computations on multiple devices, including support for stencils, has also been explored by approaches based on compiler directives [23, 29]. However, as discussed in [29], the lack of certain directives often makes the exploitation of multiple accelerators under this paradigm challenging for programmers. However, the main concern with this strategy is that compiler-based approaches strongly depend on the quality of the compiler, often lacking a reasonable performance model and, worse, strongly underperforming with respect to other alternatives due to missing optimization opportunities [8].

Finally, HPL has the unique feature with respect to all the preceding works that its kernels can be written in a language embedded in C++. This allows to exploit runtime code generation under the control of the programmer and thus to dynamically adapt and optimize the kernel codes for different platforms and inputs, which can enable large performance improvements [7].

6. CONCLUSIONS

The Heterogeneous Programming Library (HPL) has proven to be an interesting choice to reduce the effort of developing heterogeneous applications without incurring in meaningful performance penalties. In this paper we have extended HPL to improve the programmability of stencil applications both in single and multi-device environments. Our first proposal takes into account that the complexity of these codes increases when several devices are used because of the tasks associated to the data distribution and the synchronization of the ghost regions. Thus, the simplification and automation of these tasks becomes a very attractive option.

The new mechanism, called syncGhosts has been evaluated with five very different applications, showing that it largely reduces the programming effort without increasing the execution time. For example, while the use of HPL annotations to express stencil multi-device applications reduces on average the programming effort on 22.72% with respect to the usage of HPL subarrays, this average reduction grows up to 73.52% when the syncGhosts mechanism described in this paper is also used. The peak reduction, which is 93.9%, is achieved in CANNY, an application with four stencil kernels. Something similar can be said about the lines of code.

The experiments also show that the performance overhead introduced by the new syncGhosts feature is negligible. Concretely using two devices, the performance differences are always below 1% and using three GPUs, in which more memory transfers are needed to maintain the coherence of the arrays, they peak below 2% thanks to the underlying careful implementation.

We have also proposed HPL stencil templates as a mechanism to simplify the development of high-performance stencil kernels. Our templates try to extract the best performance from current heterogeneous systems by properly exploiting both the memory system and the parallelism they provide, and by enabling potentially important optimizations such as the processing of multiple elements per parallel thread. Their use reduced between 62% and 86% the complexity metrics of the kernels of the applications to which we applied them. As for performance, in our test they generated codes with a performance identical to the ones developed by hand and between 6.1 and 19.7 times faster than those generated by the most similar tool we found in the bibliography.

In addition, we applied the ghost cell expansion (GCE) technique in HPL obtaining interesting results. Namely, with almost no programmability costs we improved the performance of multi-device executions using ghost regions of different sizes. In particular, we achieved reductions of up to 13% of the execution time in one benchmark, using ghost regions of 16 rows in comparison to the same benchmark with ghost regions of a single row.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (Project TIN2016-75845-P).

REFERENCES

1. V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High Performance Fortran compilation techniques for parallelizing scientific codes. In *IEEE/ACM Conf. on Supercomputing (SC'98)*, page 11, 1998.
2. B. L. Chamberlain, S. Choi, E. Lewis, C. Lin, S. Snyder, and W. Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.
3. M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, May 2011.
4. C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. In *Proc. Supercomputing 2001*, SC 2001, pages 50–50, 2001.
5. F. Dütsch, K. Djelassi, M. Haidl, and S. Gorlatch. HLSF: A High-Level; C++-Based Framework for Stencil Computations on Accelerators. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*, WOSC '14, pages 41–4, New York, NY, USA, 2014. ACM.
6. J. Enmyren and 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, pages 5–14, 2010.
7. J. F. Fabeiro, D. Andrade, and B. B. Fraguela. Writing a performance-portable matrix multiplication. *Parallel Computing*, 52:65–77, 2016.
8. S. Ghike, R. Gran, M. J. Garzarán, and D. Padua. Directive-based compilers for GPUs. In *27th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2014)*, pages 19–35, 2014.
9. J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua. Writing productive stencil codes with overlapped tiling. *Concurrency and Computation: Practice and Experience*, 21(1):25–39, 2009.
10. M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.
11. T.D. Han and T.S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. on Parallel and Distributed Systems*, 22:78–90, 2011.
12. Khronos OpenCL Working Group. The OpenCL Specification. Version 2.0, Nov 2013.
13. C. Koelbel and P. Mehrotra. An overview of High Performance Fortran. *SIGPLAN Fortran Forum*, 11(4):9–16, 1992.
14. T. Lutz, C. Fensch, and M. Cole. PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, January 2013.
15. J. Milthorpe, D. Grove, B. Herta, and O. Tardieu. Exploring the APGAS programming model using the LULESH proxy application. Technical Report RC25555, IBM Research Technical Reports, 2015.

16. J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Intl. J. of High Performance Computing Applications*, 20(2):203–231, 2006.
17. Nvidia. *CUDA Compute Unified Device Architecture*. Nvidia, 2008.
18. OpenACC-Standard.org. The OpenACC Application Programming Interface Version 2.0a, Aug 2013.
19. J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn. POOMA: A framework for scientific simulations of parallel architectures. In *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.
20. Aaron Sawdey and Matthew O'Keefe. Program analysis of overlap area usage in self-similar parallel programs. In *10th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'97)*, pages 79–93, Aug 1997.
21. M. Steuwer, M. Haidl, S. Breuer, and S. Gorlatch. High-level programming of stencil computations on multi-GPU systems using the SkelCL library. *Parallel Processing Letters*, 24(3), 2014.
22. H. Su, N. Wu, M. Wen, C. Zhang, and X. Cai. On the GPU performance of 3D stencil computations implemented in OpenCL. In *28th Intl. Supercomputing Conference (ISC 21013)*, pages 125–135, 2013.
23. R. Veldema, T. Blaß, and M. Philippsen. Enabling multiple accelerator acceleration for java/openmp. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar'11)*, 2011.
24. M. Viñas, Z. Bozkus, and B. B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. *J. of Parallel and Distributed Computing*, 73(12):1627–1638, December 2013.
25. M. Viñas, Z. Bozkus, B. B. Fraguela, D. Andrade, and R. Doallo. Developing adaptive multi-device applications with the Heterogeneous Programming Library. *The Journal of Supercomputing*, 71(6):2204–2220, 2015.
26. M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo. High productivity multi-device exploitation with the Heterogeneous Programming Library. *J. of Parallel and Distributed Computing*, 101:51–68, march 2017.
27. M. Viñas, B. B. Fraguela, Z. Bozkus, and D. Andrade. Improving OpenCL programmability with the Heterogeneous Programming Library. In *Proc. Intl. Conf. on Computational Science (ICCS 2015)*, pages 110–119. Elsevier, 2015.
28. M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J. A. García, M. J. Castro, and R. Doallo. A multi-GPU shallow-water simulation with transport of contaminants. *Concurrency and Computation: Practice and Experience*, 25(8):1153–1169, June 2013.
29. R. Xu, X. Tian, S. Chandrasekaran, and B. M. Chapman. Multi-GPU support on single node using directive-based programming model. *Scientific Programming*, 2015:621730:1–621730:15, 2015.