A Portable High-Productivity Approach to Program Heterogeneous Systems

Zeki Bozkus
Dept. of Computer Engineering
Kadir Has Üniversitesi
Istanbul, Turkey
zeki.bozkus@khas.edu.tr

Basilio B. Fraguela
Depto. de Electrónica e Sistemas
Universidade da Coruña
A Coruña, Spain
basilio.fraguela@udc.es

***Abstract*—The exploitation of heterogeneous resources is becoming increasingly important for general purpose computing. Unfortunately, heterogeneous systems require much more effort to be programmed than the traditional single or even multi-core computers most programmers are familiar with. Not only new concepts, but also new tools with different restrictions must be learned and applied. Additionally, many of these approaches are specific to one vendor or device, resulting in little portability or rapid obsolescence for the applications built on them. Open standards for programming heterogeneous systems such as OpenCL contribute to improve the situation, but the requirement of portability has led to a programming interface more complex than that of other approaches. In this paper we present a novel library-based approach to programming heterogeneous systems that couples portability with ease of use. Our evaluations indicate that while the performance of our library, called Heterogeneous Programming Library (HPL), is on par with that of OpenCL, the current standard for portable heterogeneous computing, the programming effort required by HPL is 3 to 10 times smaller than that of OpenCL based on the authors' implementation of five benchmarks.**

***Keywords*-programmability; heterogeneity; portability; libraries**

I. Introduction

The relevance of the usage of computing devices with very different characteristics that cooperate in a computation has increased exponentially in the past few years. The reason for this has been the appearance of accelerators which can be programmed to perform general-purpose computations which can achieve large speedups over traditional CPUs and even multicore CPUs. These accelerators, such as GPUs [1] or the Synergistic Processing Elements (SPEs) of the Cell BE [2], always coexist with one or more general-purpose CPUs, giving place to heterogeneous computing systems.

Unfortunately this hardware heterogeneity is also reflected in the software required to program these systems since, unlike in the case of regular CPUs, with these types of accelerators programmers are typically exposed to a number of characteristics and limitations that must be handled. For this reason, they cannot be programmed using the standard languages used for general-purpose CPUs, but rather require extended versions [3][4][5][6] which demand different semantics and the specification of many details such as buffers, transfers and synchronizations. Moreover, most of these tools are specific to one vendor or even to one family of devices, which severely restricts the portability of the codes and places into question the effort needed for their development. Libraries that complement some of these languages in order to improve programmability have been developed [7][8][9], but either their scope of application is restricted or their interface to program arbitrary computations in the GPU is inconvenient and requires the usage of the GPU-specific languages. Lastly, proposals have been put forward based on compiler directives [10][11][12], which obviously require special compiler support and whose performance is highly dependent on compiler technology. Additionally, all of the alternatives that we are aware of are either solutions restricted to a single vendor or have not yet been implemented.

In this paper we present a portable library-based approach for the usage of heterogeneous systems that focuses on delivering high programmer productivity while allowing low level control. Our library, called Heterogeneous Programming Library (HPL) and developed in C++, is built on two key concepts. First, it allows for the definition of functions that are evaluated in parallel by multiple threads on different inputs for regular CPUs, hardware accelerators, or both. We call these functions *kernels*, as they are analogous to those found in [4][5][6]. The second concept is data types that allow for the expression of both scalars and arrays of any number of dimensions that can be used both in serial portions of the code as well as in kernels.

The rest of this paper is structured as follows. The following section introduces the hardware view and the programming model provided by our library. Its programming interface is explained in Section III, followed by an illustration with examples of increasing complexity in Section IV. Then, Section V evaluates our proposal. This is followed by a discussion on related work in Section VI. The paper presents our conclusions and future work in Section VII.

II. Hardware and Programming Model

The main purpose of the Heterogeneous Programming Library (HPL) is to improve the productivity of programmers

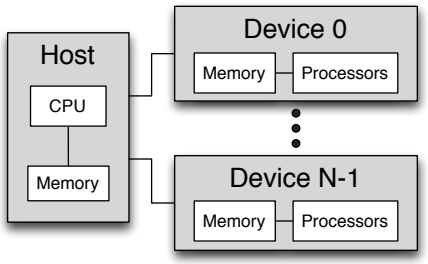

Figure 1. Underlying hardware model for HPL

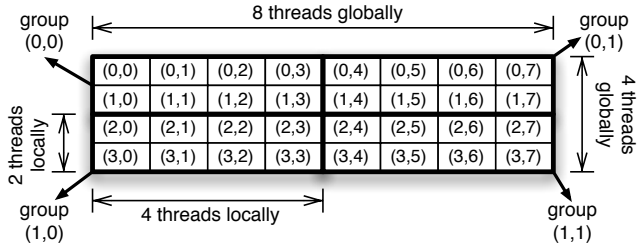

Figure 2. Global and local domains for the threads that execute in parallel a HPL kernel

who want to be able to exploit the computational power of heterogeneous devices without compromising the portability of their applications. In this way, HPL reveals a programming model which enables the rapid development of parallel applications, which is suitable for any computational device, from sequential processors to many-core systems, and which is focused first on the application parallelism, and only later on its mapping on a specific platform. For these reasons, the HPL programming model is quite simple and intuitive, and it does not result in complicated patterns of parallelism and interdependencies between tasks. Rather, the expression of parallelism is limited to a reduced and well-structured set of constructs that are effective and platform-independent. These ones are in fact the characteristics that were found to be more desirable for a unified programming model for many-core systems in [13]. According to their classification, the HPL programming model belongs to the family of the application-centric generic programming models.

A programming model requires a minimal model of the underlying hardware. The abstract view of the hardware on which HPL applications run is depicted in Figure 1. There is a host with a memory and a single CPU in which the sequential portions of the application are executed. Attached to it, there are a number of computing devices, each one of them with its own memory and a number of processors that can only access the memory within their device. While different devices can run different codes, all the processors in the same device must execute the same code in an SPMD fashion. In some devices the processors are subdivided in groups that share a scratchpad memory of a limited size and can synchronize by means of barriers, this being the only mechanism available to synchronize processors within a device. This model is basically a simplified version of the one proposed by OpenCL [6], which also aims for maximum portability. Notice that the computational devices in this model need not be special hardware accelerators. A traditional cluster of multi-core nodes, either homogeneous or heterogeneous, could fit perfectly in this model. One possibility would be to map each node to a device whose processors are the cores within the node. Another possibility would be to conceptualize each core in the cluster as an independent device.

Given this view, a HPL application consists of traditional sequential regions, which are executed in the host, and portions of code that are run in an SPMD fashion in one or several devices. The main program run in the host manages the transfers of data between the host and the devices and requests the execution of the parallel regions of the application in the different devices. These parallel portions of the application are expressed as functions that are evaluated in parallel by the processors in the selected device. These functions are called *kernels*, since they are analogous to the kernels found in [4][5][6]. Each thread that runs a copy of a kernel needs a unique identifier so that it can identify the work it is responsible for. To allow for this, kernels are executed on a domain of integers of up to three dimensions, called a global domain. Each point in this domain is assigned a unique identifier that is associated to an instance of the requested kernel, and therefore the size of this domain is the total number of parallel threads running the requested kernel. The user can optionally specify a local domain, which must have the same number of dimensions as the global domain and whose size in every dimension must be a divisor of the size of the corresponding dimension of the global domain. The threads whose identifiers belong to the same local domain can share scratchpad memory and synchronize by means of local barriers. These threads form what we call a *group* of threads, each group also having an n-dimensional identifier. Figure 2 represents the unique global identifiers of the 32 threads to run for a global domain of 4×8 threads. The identifiers of threads that belong to the same local domain (or group) of 2×4 threads are surrounded by a thicker line. The unique identifier of each thread group is also indicated. As we see again these concepts are very similar to those in [4][6] and can be mapped to any computational device. This execution model supports in a straightforward manner the data parallel programming model. Task parallelism can be provided by requesting the parallel evaluation of different kernels on different devices.

Lastly, regarding memory, kernels can only access the processor's registers and the memory available inside the device where they are run. HPL distinguishes three kinds of

device memory. First, we have the standard memory, which is shared by all the processors in the device both for reading and writing. HPL calls this memory global memory because it is accessible by all the threads in the same global domain. Second, there is the scratchpad memory, which can be only accessed by the threads that belong to the same local domain. We call it the local memory for this reason. Finally, there is a memory for data that can be written by the host but which the kernels can only read, which is therefore called constant memory.

III. Programming Interface

As explained in the preceding section, our C++ HPL library allows for expression and execution of arbitrary user-defined kernels on any of the computational devices available in the system. Since these kernels must be compiled so that they can be run on any device requested by the user, they cannot be expressed using the native C++ data types and control structures, as this would result in their regular compilation as standard code to be run in the host. Rather, they are written in standard C++, but using datatypes, functions and macros provided by HPL. Thanks to the usage of these tools, our library is able to build from the original C++ expressions code that can be compiled at runtime for the desired device.

Our current implementation of the library generates OpenCL C [6] versions of the HPL kernels, which are then compiled to binary with the OpenCL compiler. As a result, our library can be used to perform computations on any device that is supported by OpenCL. Since this is the open standard for the programming of heterogeneous systems, and it is already supported by a large number of heterogeneous systems including GPUs, the Cell Broadband Engine and standard multicores, this allows in turn the widespread and portable usage of HPL. This was indeed the main reason for choosing this platform as the backend for HPL, although we do not exclude using other backends for different platforms in the future.

We now explain in turn the HPL datatypes, the syntax required to build its kernels, and the interface to request their execution, followed by example programs that illustrate all these points. All the HPL keywords and types are provided to the user program by the inclusion of the header file `HPL.h`, and they are encapsulated inside the `HPL` namespace in order to avoid collisions with other program objects.

A. Data Types

The HPL datatypes encapsulate and provide to the library the information it needs to manipulate the data involved in the heterogeneous computations as automatically as possible. This includes array sizes, kind of memory where a data structure should be allocated, availability of copies of a data structure in different devices, etc.

The arrays that are used in kernels must belong to the HPL datatype `Array<`*type, ndim [, memoryFlag]*`>`. This is a C++ template where the *type* is the standard C++ type for the elements of the array and *ndim* is the number of dimensions of the array. The third argument indicates the kind of device memory in which the array must be allocated. It is only needed when the array is to be located in constant or local memory, their respective flag values being `Constant` and `Local`. When this flag is not specified, the array is allocated in global memory (`Global` flag). A special situation takes place when an `Array` is defined within a kernel. Since a kernel is a function, all the variables defined within it are local, and therefore, private to each given evaluation of the kernel. For this reason, all the variables defined inside a kernel that do not specify a memory flag are totally private, even if their final physical location is the global memory of the device. The constructor for an `Array` takes as inputs the sizes of its dimensions, and, optionally, a pointer to the raw data in the host memory in case the array had been previously allocated, which also implies the user is responsible for its deallocation. Otherwise HPL takes care of the allocation and deallocation of the storage required by the array.

Scalars can be defined by using the `Array` template with *ndim=0*, but HPL provides for convenience a series of types (`Int`, `Uint`, `Double`, ...) to define scalars of the obvious corresponding C++ type.

HPL arrays are indexed inside kernels using square brackets, just like standard C++ arrays. Nevertheless, they are indexed with parenthesis in the host code. The reason for this difference is that while the code in HPL kernels is dynamically compiled, and therefore optimized, this is not the case for the host code, which is only statically compiled. As a result, accesses to variables that belong to HPL datatypes within HPL kernels have no overheads, while the accesses in host code suffer the typical overhead associated with user-defined datatypes [14]. The usage of a different kind of indexing helps the programmer to be aware of this cost and to identify more quickly whether a portion of C++ code is a parallel kernel or not. The user can avoid the indexing overhead in the host code by requesting the native pointer to the `Array` data in the host memory, which is provided by the method `data()`, and accessing the data through the pointer.

B. Kernel Syntax

As discussed at the beginning of this section, the control flow structures used inside kernels must be written using HPL keywords so that the library can capture them and thereby generate the appropriate code for them. HPL provides the usual C++ control flow constructs (`if`, `for`, ...) with three differences. First, their names finish with an underscore (`if_`, `for_`, ...). Second, the end of each block must be closed with a corresponding statement (`endif_`,

`endfor_`, ...). Lastly, the arguments to `for_` are separated by commas instead of semicolons.

A second point of interest for writing HPL kernels is the existence of variables with predefined meanings. As Section II explained, kernel executions take place on a global domain of up to three dimensions on which local domains can be optionally defined. The predefined variables `idx`, `idy` and `idz` correspond to the value of the global domain associated with the current execution of the kernel in the first, second, and third dimension of the global domain, respectively. In this way, these variables allow for a unique identification of the current execution of the kernel. Similarly, `lidx`, `lidy` and `lidz` provide the local identification of the thread within its local domain in the first, second and third dimensions of the problem, respectively. For example, in Figure 2, the threads with global id (1,2), (1,6), (3,2) and (3,6) all have the local id (1, 2) within their local domain. The current group of threads can be identified by means of the variables `gidx`, `gidy` and `gidz`, each one of them providing the identification in each one of the dimensions of the domain. HPL also provides similar variables for the global and the local sizes of the execution of the current kernel as well as the number of groups of threads in every dimension of the domain.

Finally, HPL provides a series of functions to perform typical computations and tasks within the kernels. A particularly important function is `barrier`, which performs a barrier synchronization between all the threads in a group. This function supports an optional argument to specify whether the threads need to have a consistent view of the local memory (argument `LOCAL`), the global memory (argument `GLOBAL`), or both (`LOCAL|GLOBAL`), after the barrier. Notice that the global and the local memory are different and separate, thus the consistency of one of them does not imply anything on the state of the other one.

C. Kernel Invocation

HPL kernels are functions that are evaluated in parallel on a domain. These functions communicate with the host by means of their arguments, which are HPL arrays or scalars. While the scalars are always passed by value, the arrays are passed by reference, and therefore are the mechanism to return the results of the computation. The syntax to request the parallel evaluation of a kernel is `eval(`*kernelfunction*`)(`*arg1*, *arg2*, ...`)`.

By default the kernel is evaluated in the first device found in the system that is not a standard general-purpose CPU, the global domain of the evaluation of a kernel is given by the dimensions of its first argument, and the local domain is chosen by the library. The optional methods `global` and `local` in between `eval` and the kernel arguments can be used to specify the global domain and the local domain of the evaluation, respectively. For example, in order to evaluate kernel `f` with the argument `a` in the default device using the global and local domain sizes illustrated in Figure 2 one would write `eval(f).global(4,8).local(2,4)(a)`.

```
1  #include "HPL.h"
2
3  using namespace HPL;
4
5  double myvector[1000];
6
7  Array<double, 1> x(1000), y(1000, myvector);
8
9  void saxpy(Array<double,1> y, Array<double,1> x, Double a) {
10     y[idx] = a * x[idx] + y[idx];
11 }
12
13 int main(int argc, char **argv) {
14     Double a;
15
16     //the vectors and a are filled in with data (not shown)
17
18     eval(saxpy)(y, x, a);
19 }
```

Figure 3. SAXPY implementation in HPL

As we explained in Section II, the main program that runs in the host is responsible for managing the data transfers and launching the kernels for execution in the different devices. For this reason, the `eval` function can only be used in host code.

IV. HPL Example Codes

Three codes of increasing complexity are now used in turn to illustrate the HPL syntax described in Section III as well as the usual programming style and strategies implied by the programming model explained in Section II. The description of the codes is detailed enough to try to enable any C++ programmer with an average proficiency and no previous experience with the programming of heterogeneous systems to begin to exploit the advantages of these systems thanks to HPL. This is in fact one of the main purposes of our work.

A. SAXPY

Let us begin with an HPL implementation of SAXPY, which computes $S = aX + Y$, where S, X and Y are vectors and a is a scalar. A complete program in HPL for this computation using double-precision floating point data is shown in Figure 3. After including the `HPL.h` header, we indicate that we will operate with objects defined in the C++ `HPL` namespace in line 3. Then two vectors suitable for use in HPL kernels are defined in line 7. In one of them, `x`, the library is responsible for its allocation and deallocation. For the second one, `y`, an existing regular C++ vector `myvector` provides the storage. The scalar variable used in SAXPY is defined in line 14 with the suitable HPL type `Double`.

The HPL kernel for SAXPY is the `saxpy` function in lines 9 to 11. As we explained in Section III-C, HPL kernels only communicate with the host by means of their arguments. For this reason the elements that participate

in the computation must be parameters of the kernel and the return type of the function must be `void`. In our implementation each execution of the kernel computes a single element of the destination vector. This way, to perform SAXPY on vectors of N elements, a global domain of a single dimension and N elements must be used, so that the kernel with unique identification $0 \leq$`idx`$< N$ is in charge of the computation for the `idx`-th elements of the vectors, as reflected in line 10. Let us remember that `idx` is a predefined variable that provides the value of the first dimension of the global domain associated to the current execution of the kernel. Since this problem has a single dimension, `idx` suffices to identify uniquely a kernel execution. Note that we use the vector Y to store the result S.

The invocation of the kernel takes place in line 18, where neither the global nor the local domain for the execution of the kernel are provided. As we explained in the preceding section, by default the global domain is given by the number of dimensions and sizes of the first argument, which perfectly fits this example. As for the local domain, it can be chosen by the library, as this code does not use or make assumptions on it because the computation of each kernel execution is completely independent, that is, there is no cooperation between the threads that belong to the same group or local domain.

B. Dot Product

The program shown in Figure 4 is a somewhat more complex example that illustrates the usage of most HPL features introduced in the preceding sections. This program computes the dot product of two vectors of single-precision floating point elements of length `N` in two stages. First, a HPL kernel computes in parallel the partial dot products associated to subregions of `M` consecutive elements of the arrays. The result is an array of `nGroup = N/M` floating point values which are reduced in the host in the second stage. Notice how this array, called `pSums`, is indexed with square brackets in the HPL kernel in line 19, but with round parenthesis in the host code in line 33. The reasons for this have been explained in Section III-A.

The vectors `v1` and `v2` whose dot product will be computed as well as the intermediate vector `pSums` are defined in line 25 as HPL arrays, since they will be used in the kernel. The kernel, written in function `dotp`, is invoked in line 30 with the syntax we have just explained. The strategy followed by our implementation is to launch `N` parallel kernel executions, so that the `idx`-th thread will be in charge of reading and multiplying the `idx`-th elements of the input arrays. Then, the threads in each group of `M` threads, which is uniquely identified by the variable `gidx`, will cooperate to compute the partial dot product by means of the scratchpad memory they share. For this reason, our evaluation specifies a global domain of `N` elements and a local domain of `M` elements.

```
1  #include "HPL.h"
2  #define N 256
3  #define M 32
4  #define nGroup (N/M)
5
6  using namespace HPL;
7
8  void dotp(Array<float,1> v1, Array<float,1> v2,
9            Array<float, 1> pSums) {
10   Int i;
11   Array<float, 1, Local> sharedM(M);
12
13   sharedM[lidx] = v1[idx] * v2[idx];
14
15   barrier(LOCAL);
16
17   if_( lidx == 0 ) {
18     for_( i = 0, i < M, i++ ) {
19       pSums[gidx] += sharedM[i];
20     } endfor_
21   } endif_
22 }
23
24 int main(int argc, char **argv) {
25   Array<float, 1> v1(N), v2(N), pSums(nGroup);
26   float result = 0.0;
27
28   //v1 and v2 are filled in with data (not shown)
29
30   eval(dotp).global(N).local(M)(v1, v2, pSums);
31
32   for(int i = 0; i < nGroup; i++)
33     result += pSums(i);
34
35   std::cout << "Dot = " << result << "\n";
36 }
```

Figure 4. Dotproduct example in the HPL syntax.

The `dotp` function is written with the HPL syntax. Here, the array `sharedM` is declared with the `Local` flag to place this array on the scratchpad memory shared by the threads that belong to the same local domain. Its purpose is to store the result of multiplication operations of the input arrays. A barrier is used in line 15 to synchronize the threads in the local domain and ensure that the writing of the `sharedM` array in the local memory has been completed after the barrier. After this, the first thread of each group, whose `lidx` is zero, performs the partial sums in the location associated with the group. We could have implemented a much more efficient reduction, using for example a binary tree of parallel reductions. However, we have followed this approach for the sake of clarity.

C. Sparse Matrix Vector Product

Sparse matrix vector multiplication (spmv) is a common primitive at many scientific applications. For example, this operation is the most computationally expensive part of the Conjugate Gradient (CG) code of the NAS Parallel Benchmarks suite, and in fact it is part of the benchmarks chosen by the SHOC Benchmark [15] suite to characterize heterogeneous systems (although it does not appear in [15] it

```
for ( i = 0; i < nRows; i++) {
  for ( j = rowptr[i]; j < rowpt[i+1]; j++)
    out[i] += A[j] * val[cols[j]];
}
```

(a) Spmv with C++

```
1  #include "HPL.h"
2  #define nRows 1024
3  #define NZ 128 // numNonZeroes
4  #define M 8
5  #define N nRows*M
6
7  using namespace HPL;
8
9  void spmv(Array<float, 1> A, Array<float, 1> vec,
10           Array<int, 1> cols, Array<int, 1> rowptr,
11           Array<float, 1> out) {
12   Int j;
13   Float mySum = 0;
14
15   for_ ( j = rowptr[gidx] + lidx,
16              j < rowptr[gidx + 1], j += M) {
17     mySum += A[j] * vec[cols[j]];
18   } endfor_
19
20   Array<float, 1, Local> sdata(M);
21   sdata[lidx] = mySum;
22   barrier(LOCAL);
23
24   // Reduce sdata
25   if_ ( lidx < 4 ) {
26     sdata[lidx] += sdata[lidx + 4];
27   } endif_
28
29   barrier(LOCAL);
30
31   if_ ( lidx < 2 ) {
32     sdata[lidx] += sdata[lidx + 2];
33   } endif_
34
35   barrier(LOCAL);
36
37   if_ ( lidx == 0 ) {
38     out[gidx] = sdata[0] + sdata[1];
39   } endif_
40 }
41
42 int main(int argc, char **argv) {
43    Array<float, 1> A(NZ), vec(nRows), out(nRows);
44    Array<int, 1> cols(NZ), rowptr(nRows+1);
45
46   //A and vec are filled in with data (not shown)
47   //cols and rowptr are calculated with CSR format
48
49   eval(spmv).global(N).local(M)(A, vec, cols, rowptr, out);
50 }
```

(b) Spmv with HPL

Figure 5. Sparse matrix vector multiplication example.

was added later to the suite). For these reasons we will also use this computation in our evaluation in the next section. Figure 5(a) shows the main loop of the spmv kernel for a sequential code where the sparse matrix is stored using the compressed sparse row (CSR) format. Figure 5(b) presents the corresponding HPL code for spmv. This code is a good example of heterogeneous computing using HPL. Here, the CPU works sequentially to make the CSR format, as it is more suitable to perform this task; later, the heavy duty and naturally parallel computation part is written with HPL so that it can be run on a device. In this code, a group of local threads identified by the predefined variable `gidx` is responsible for the multiplication of a row from the sparse matrix `A` with vector `vec`. Each group performs the reduction required to compute the result in the `out` vector for the row by summing the elements of the vector `sdata` on the local memory.

V. Evaluation

This section evaluates HPL using OpenCL as comparison point for two reasons. Firstly, since it is the open standard for programming heterogeneous systems, it is the natural alternative to HPL for the portable development of applications for these systems. Secondly, since OpenCL is the backend that HPL currently uses, it is natural to wonder which is the overhead that HPL imposes with respect to the manual usage of OpenCL.

Our evaluation consists of three categories. First, we measure the programmability provided by our library by comparing some benchmark examples written both with HPL and OpenCL. The second set of experiments compares the runtime performance of HPL with that of OpenCL. The third category evaluates the HPL performance on different platforms for portability.

We wrote all the HPL versions of our codes by ourselves. This was also the case for the OpenCL version of the EP benchmark from the NAS Parallel Benchmark suite [16]. The OpenCL versions of the Floyd-Warshall algorithm and Matrix transpose were taken from the AMD APP SDK. Finally, the sparse matrix vector multiplication (spmv) and reductions OpenCL benchmarks were extracted from the SHOC Benchmark suite [15]. We chose these five benchmarks because they vary largely in terms of ratio of computations to accesses to memory, access patterns, and degree of interaction required between the parallel threads that evaluate the kernels. This way they cover a wide spectrum of application behaviors and, as we will see in Section V-B, they achieve very different degrees of improvement when their parallel portions are executed on a GPU compared to their serial execution in a regular CPU. Moreover, we chose them from different sources in order to minimize the impact of coding style differences on the programmability. The compiler used in all the tests was g++ 4.3.3 with optimization level `O3`.

A. Programmability

There is not a straightforward or universally accepted way to determine the benefits for programmability of the usage of a given programming approach. In this paper we have used Sloccount [17], which counts the number of source lines

Benchmark	OpenCL	HPL	Reduction
EP	1151	281	75.6%
Floyd-Warshall	1170	107	90.9%
Matrix transpose	455	52	88.6%
Spmv	1637	517	68.4%
Reduction	773	218	71.8%

Table I
SLOCS FOR THE OPENCL AND HPL VERSIONS OF THE BENCHMARKS AND REDUCTION IN SLOCS DUE TO THE USAGE OF HPL

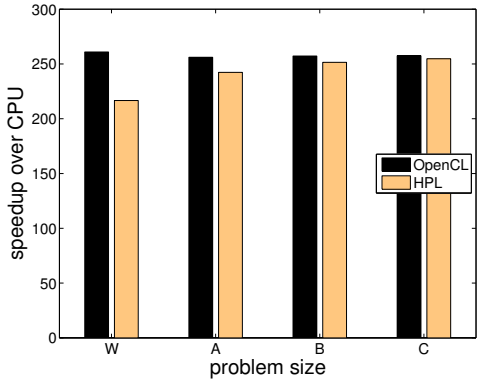

Figure 6. Speedups of the GPU executions of the OpenCL and HPL versions of EP over the sequential execution in a CPU for different problem sizes

of code excluding comments and empty lines (SLOC), to measure the programmability of HPL and OpenCL. SLOC is a very effective software metric to estimate the amount of effort that will be required to develop a program, as well as to forecast the programming productivity or maintainability once the software is produced. The SLOC results are reported in Table I for the five different benchmarks written with both HPL and OpenCL. From this data, we can see that HPL outperforms OpenCL with programs that are 3 to 10 times shorter. The main reason for this result is that OpenCL requires the manual setup of the environment, management of the buffers both in the device and host memory and the transfers between them, explicit load and compilation of the kernels, etc. On the other hand, all these necessary steps are highly automated and hidden from the user in HPL.

B. Runtime Performance

In this section we performed some experiments to show the performance differences of HPL and OpenCL. We used a Tesla C2050/C2070 GPU as experimental platform. The device has 448 thread processors with a clock rate of 1.15 GHz and 6GB of DRAM and it is connected to a host system consisting of 4xDual-Cores Intel 2.13 GHz Xeon processors.

Figure 6 shows the speedup of the execution on the GPU of EP both when using OpenCL and HPL with respect to the serial execution of a standard C++ version of the code in the CPU for different problem sizes. The speedup is computed taking into account the generation of the backend code (in the case of HPL) and the compilation and execution of the kernel, but not the transfers between the GPU and the main memory. The reason is that the transfer time is basically the same for OpenCL and HPL, as they both use the same OpenCL functions and runtime for this. Since the main purpose of this evaluation is to analyze the performance difference between these two approaches, disregarding the transfers allows to identify the difference between them more clearly, even if it is a bit unfair to HPL. Still, in most of our benchmarks, and particularly in EP, the transfer time is minimal compared to the computation time, which is why we have chosen this benchmark to illustrate the performance difference between both programming environments as a function of the problem size.

Given the embarrassingly parallel nature of EP and its regular access patterns the GPU always provides large speedups with respect to the CPU in Figure 6. However, what interests us most here is that the HPL performance is very similar to that of OpenCL. For the smallest problem size, W, HPL is 20.5% slower than OpenCL, but in absolute terms the execution time only goes from 0.044 to 0.053 seconds. It is very important to outline at this point that HPL stores internally and reuses the binaries of the kernels it generates. This way, second and later invocations of an HPL kernel do not incur in overheads of analysis, backend code generation and compilation, and as a result they achieve runtimes virtually identical to those of OpenCL when reusing a previously compiled kernel. Kernels that require short computing times are usually written to be run in heterogeneous devices only if the program will use them several (typically, many) times. Therefore this behavior of our library dilutes the overhead of the first invocation on all the subsequent usages of the kernel that are finally performed.

The absolute difference in runtime between OpenCL and HPL remains in similar values for larger problem sizes. This results in run-time slowdowns for HPL, that is, increases of its runtime with respect to the OpenCL version, of only 5.7%, 2.3% and 1.1% for the classes A, B and C, respectively. This happens even when the largest runs are not long either. For example the GPU run for class C with OpenCL is just 2.81 seconds.

Figure 7 shows the speedup of all the benchmarks we implemented when they are run in the GPU using OpenCL and HPL, the baseline being a serial execution in the CPU of our system of the corresponding benchmark written and compiled with regular C++. The benchmarks and problems sizes are: EP class C, the Floyd-Warshall algorithm applied on 1024 nodes, the transposition of a 16K$\times$16K matrix, the spmv code for a 16K$\times$16K matrix with a 1% of non zeros and the addition of 16M single-precision floating point values. The speedups were computed as in Figure 6 for the reasons explained above. We can see that depending on the degree of parallelism, the regularity of the accesses and ratio of computations to memory accesses, we have

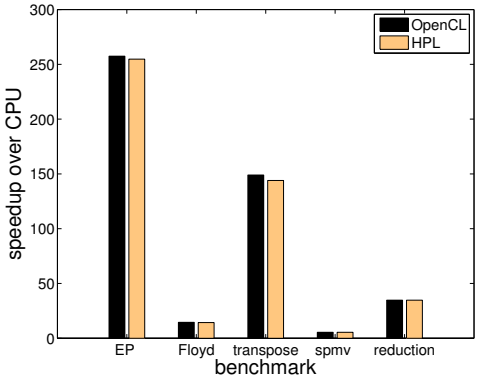

Figure 7. Speedups of the executions in GPU of the OpenCL and HPL versions of the benchmarks over their sequential execution in the CPU

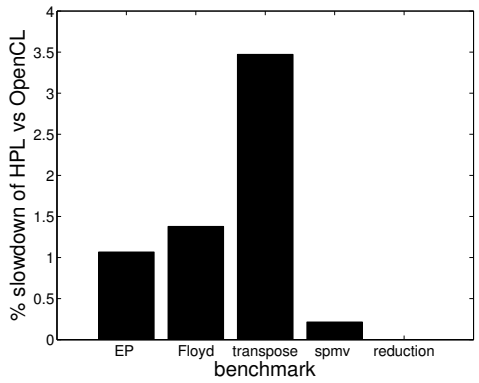

Figure 8. Slowdown of HPL with respect to OpenCL for the different benchmark executions in the GPU

chosen benchmarks with a wide range of performances in an accelerator such as a GPU. This way the speedups found for the OpenCL codes range from 5.4 for spmv to 257 for EP. The most interesting fact for us is that for all of them the performance achieved by HPL is very similar to that of OpenCL. This can be seen more clearly in Figure 8, which represents the slowdown of HPL with respect to OpenCL in these executions in the GPU. We can see that the typical degradation is below 4%. This degradation is mostly due to time required by our library to capture the computations expressed in the HPL kernels, analyze them to decide which data transfers between memories will be needed due to their execution, and finally generate the corresponding OpenCL C codes. Additionally, these codes may also be slightly less efficient than OpenCL C versions written by hand in some situations.

If the transfer time between CPU and GPU is taken into account in the performance comparison between HPL and OpenCL, the results are basically the same as in Figure 8 except for matrix transpose. In this benchmark these transfers consume a long time compared to the transposition itself, and since they require the same time in HPL and OpenCL,

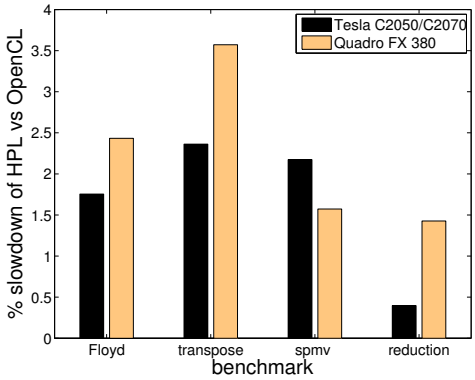

Figure 9. HPL overhead with respect to OpenCL for the different benchmark executions in the Tesla C2050/C2070 and the Quadro FX 380 GPUs

the overhead of HPL compared to OpenCL is reduced to only 0.41%. This is in contrast with the 3.47% shown in Figure 8.

C. Portability Results

In order to illustrate the portability of HPL across different devices, we run our benchmarks choosing for the execution of the kernels a Quadro FX 380 (16 thread procesors with a clock rate of 700 MHZ and 256 MB of DRAM) that is connected to the same host. EP was not part of this set of experiments because it requires double-precision floating point calculations, which are not supported by this device. Also, due to its smaller memory we had to reduce the problem size of Floyd-Warshall to 512 elements, and the matrix transposition was performed on matrices of 5K$\times$5K elements. The spmv code was performed on a 8K$\times$8K matrix with a 1% of non zeros.

Figure 9 shows the overhead of these HPL runs compared to those of the same benchmark under OpenCL in our two GPUs. It is clear that HPL performance is again very similar to that of manually programmed OpenCL. The precision of the measurement of such a minimal performance difference is subject to the usual small variations observed in different performance measurements for the same code and inputs. This explains the small changes ($\leq$ 2%) with respect to Figure 8 in the Tesla. The relevant conclusion is that HPL overhead is minimal for both devices.

VI. Related work

The most widely used tools to program computing systems with accelerators are new languages which are extended versions of C (sometimes C++), with a series of related libraries and a runtime system. Brook+ [5] and the C/C++ extensions for the Cell BE [3] are good examples of this trend, although CUDA [4] has been the most successful to date. All these tools force programmers to write their applications with new languages, to deal with varying levels

```
1  GPU_FILLKERNEL_2D(float,
2    naive_transpose,(__global<float *> src),
3      result = src[j+i*h];
4  );
5
6  gpu_array2d<float> src(h, w), dst(w, h);
7  dst = naive_transpose(src);
```

(a) EPGPU transpose

```
1  void naive_transpose(Array<float, 2> dest,
2                       Array<float, 2> src) {
3    dest[idy][idx] = src[idx][idy];
4  }
5
6  Array<float, 2> src(h, w), dst(w, h);
7  eval(naive_transpose)(dest, src);
```

(b) HPL transpose

Figure 10. Naïve implementations of matrix transpose

of low level detail (depending on the language), and worse, to be restricted to a single kind of accelerator or, in the best case, the devices provided by a single vendor. A separate mention should be made of the more recent OpenCL [6] which, although in this group, contrary to the others, is an open royalty-free standard for general purpose parallel programming across regular CPUs and all kinds of hardware accelerators. OpenCL has been chosen for this reason as the backend for the current implementation of our library.

Some of these environments come with libraries that can interoperate with them and which improve programmability for certain kinds of problems. For example Thrust [7] facilitates the expression on CUDA of many algorithms, but it has numerous restrictions compared to our library. It only allows for the manipulation of unidimensional arrays, its computations must always be one-to-one, i.e., a single element from each input array can be processed to generate a single element of one output array, it does not allow for the use of local or constant memory or the specification of the number of threads to run, etc.

EPGPU [9] is an interesting library focused on OpenCL with fewer limitations than Thrust. In exchange, the user-defined computations to be run in OpenCL must be written in that language inside macros that build the complete kernels. This implies that EPGPU kernels must not only be constant at compile time, but also include inside them all the definitions of the constants they use, as they are actually only strings. HPL nevertheless captures in its kernels variables and macros that are defined outside them, which makes programming more natural and less verbose. For similar reasons, EPGPU does not analyze the kernels it generates, as it would amount to developing a compiler for the OpenCL C strings it manipulates. HPL nevertheless can and does analyze the kernels it builds, the aim of that analysis currently being the minimization of the data transfers due to the execution of the kernels. The different focus between EPGPU and HPL is partially illustrated by the naïve matrix transpose implementations[1] shown for them in Figures 10(a) and 10(b), respectively. EPGPU facilitates enormously the usage of OpenCL when its restriction are fulfilled. In its code OpenCL is clearly displayed with the usage of some of its keywords (`__global`) or the appearance of its limitations in the kernels, such as the requirement to use linear indices to access the multidimensional arrays not defined inside the kernels (see line 3). HPL on the other hand abstracts away completely the backend used for the kernels and avoids these restrictions, resulting in a much more natural integration in the host language.

Other related libraries are PyCUDA and PyOpenCL [8], which provide convenient interfaces to Python to perform numerous predefined computations on accelerators. They also allow for the expression of custom computations on these devices, although they require strings of CUDA or OpenCL code and they must be element-to-element computations or reductions.

Although, as of today, it only targets general-purpose multicore systems, the Intel Array Building Blocks framework [18] is similar to HPL in that it also compiles at runtime arbitrary computations that the programmer expresses in standard C++ using a series of data containers and macros it provides. Other differences with HPL are its programming model (no local domains, groups of threads, etc.) and features, as for example it does not allow for the control of the task granularity, nor specify different kinds of memory or synchronizations in the parallel codes. Finally, contrary to our application-centric approach, it is a hardware-centric programming model according to [13].

Proposals to program heterogeneous systems by means of compiler directives [10][11] that try to replicate the success of OpenMP [19] in homogeneous multicore systems have also been put forth. The limitations of compiler directives are well known. When the directives do not allow the programmer to specify with sufficient detail the transformations desired, the user does not have enough information about the transformations performed by the compiler. The result is a lack of a clear performance model, and, therefore, of the ability to reason on the performance attained by the application [20]. Second, the compiler technology might not be developed enough to find the best low level implementation for the application in many situations. These two problems, which were behind the lack of success of HPF [21], are particularly important for hardware accelerators, as they allow for many kinds of optimizations and are very sensitive to them. An approach based on compiler directives that tries to avoid these problems will probably require a non-negligible number of directives, clauses and specifications

[1]These implementations do not correspond to the matrix transpose benchmark used in Section V, which optimizes the process by making contiguous reads and transposing blocks of the matrix in the local memory shared by each group of threads

in order to achieve good performance in an heterogeneous system. This is particularly true given the enormous gap between the semantics of the regular sequential code in which the directives are to be inserted and the execution models in the accelerators, as well as the large number of possible implementations of the same algorithm, and even of specifiable parameters for each implementation, in these devices. Lastly, the alternatives mentioned above only generate CUDA code, and therefore they can only target the accelerators of a single vendor. A standard interface for the parallelization on heterogeneous systems based on compiler directives has been recently proposed [12] but to date it has not been implemented.

VII. Conclusions and Future Work

In this paper we have presented the design and implementation of the Heterogeneous Programming Library (HPL), which provides a programming environment with an interface embedded inside C++ for the programing of heterogeneous platforms. HPL is designed to maximize the programmability of these systems by hiding from the user the complexities related to the usage of these platforms (buffers, transfers, synchronizations, ...) that are found in other approaches. Our proposal also avoids the learning curve of new languages. Rather, it uses only standard C++ features, so that programmers can continue to use the compilers and tools they are familiar with. Despite this, HPL provides a very expressive and powerful syntax with an impressive abstraction to write parallel functions to take advantage of parallel architectures. Our experiments demonstrate that HPL is a powerful alternative to OpenCL, the current standard for portable heterogeneous computing. HPL outperformed OpenCL by 3 to 10 times on programmability and productivity metrics, while it typically only experienced a degradation on performance way below 5%. We believe that the enormous benefit of programmability of HPL outweighs this minor performance degradation. HPL or future similar approaches will increase the much needed usability of high performance heterogeneous platforms.

We are working to add new features to HPL in order to improve further the programmability by providing functions for typical patterns of computation. Additionally, we plan to extend the high-productivity features of HPL to handle distributed memory parallelism by running HPL on a cluster of SMP nodes in which each node can contain multiple heterogeneous computing devices.

Acknowledgment

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 the grant with reference TIN2010-16735. Basilio B. Fraguela is a member of the HiPEAC European network of excellence and the Spanish network CAPAP-H, in whose framework this paper has been developed.

References

[1] GPGPU, "General Purpose Computation on Graphics Processing Units," http://www.gpgpu.org, last access December 5, 2011.

[2] IBM, Sony, and Toshiba, *Cell Broadband Engine Architecture*. IBM, 2006.

[3] ——, *C/C++ Language Extensions for Cell Broadband Engine Architecture*. IBM, 2006.

[4] Nvidia, *CUDA Compute Unified Device Architecture*. Nvidia, 2008.

[5] AMD, "Stream computing user guide," 2008.

[6] Khronos OpenCL Working Group, "The OpenCL Specification. Version 1.2," Nov 2011.

[7] N. Bell and J. Hoberock, *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011, ch. 26.

[8] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," Scientific Computing Group, Brown University, Providence, RI, USA, Tech. Rep. 2009-40, Nov. 2009. [Online]. Available: http://arxiv.org/abs/0911.3456

[9] 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.

[10] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.

[11] S. Lee and 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.

[12] OpenACC-Standard.org, "The OpenACC Application Programming Interface Version 1.0," Nov 2011.

[13] A. L. Varbanescu, P. Hijma, R. van Nieuwpoort, and H. E. Bal, "Towards an Effective Unified Programming Model for Many-Cores," in *IPDPS Workshops 2011*, 2011, pp. 681–692.

[14] B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzarán, D. Padua, and C. von Praun, "Optimization Techniques for Efficient HTA Programs," *Parallel Computing*, accepted for publication.

[15] A. Danalis, G. Marin, C. Mccurdy, J. S. Meredith, P. C. Roth, K. Spafford, and 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.

[16] National Aeronautics and Space Administration, "NAS Parallel Benchmarks," http://www.nas.nasa.gov/Software/NPB/, last access May 20, 2011.

[17] D. A. Wheeler, "Sloccount," 2004. [Online]. Available: http://www.dwheeler.com/sloccount/

[18] C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and 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.

[19] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1," July 2011.

[20] T. A. Ngo, "The Role of Performance Models in Parallel Programming and Languages," Ph.D. dissertation, Department of Computer Science and Engineering, University of Washington, 1997.

[21] High Performance Fortran Forum, "High Performance Fortran Specification Version 2.0," January 1997.

BIOGRAPHIES

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 Electrónica 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`