Proceedings of the 14th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2014 3–7July, 2014.

Exploting multi-GPU systems using the Heterogeneous Programming Library

Moisés Viñas[1], **Zeki Bozkus**[2], **Basilio B. Fraguela**[1], **Diego Andrade**[1] **and Ramón Doallo**[1]

[1] *Grupo de Arquitectura de Computadores, Universidade da Coruña, A Coruña, Spain*

[2] *Dept. of Computer Engineering, Kadir Has Üniversitesi, Istanbul, Turkey*

emails: moises.vinas@udc.es, zeki.bozkus@khas.edu.tr, basilio.fraguela@udc.es, diego.andrade@udc.es, ramon.doallo@udc.es

Abstract

One of the most important factors that hinder the exploitation of heterogeneous devices is their programming cost, which is much higher than that of traditional CPUs. This is particularly true when we consider the OpenCL standard, which trades complexity in the host part of the application for program portability. The Heterogeneous Programming Library (HPL) is a proposal to reduce the effort to develop applications that can use heterogeneous systems by means of a language embedded in C++ and a runtime that takes care of most of the the management of OpenCL applications. In this paper we describe for the first time the programming of multiple heterogeneous devices on top of HPL. An evaluation in multi-GPU systems shows that it achieves important programmability improvements over OpenCL with minimum overhead.

Key words: programmability, heterogeneity, parallelism, portability, libraries, OpenCL

1 Introduction

In the past few years there has been an enormous increase in the use of heterogeneous devices to perform general-purpose computations. This is due to the large advantages in terms of performance and power consumption they can achieve with respect to traditional single and even multicore CPUs for certain kinds of computations. The biggest difficulty in the usage of these kind of devices is their programming cost, as they require the specification and the management of much more details than regular computers. Another important problem is that they have been traditionally programmed with vendor-specific or even device-specific tools [7, 10], resulting in non-portable applications. While the proposal of the OpenCL standard [8] alleviates this situation, it incurs in even more programming overheads than the preceding non-portable approaches due to its effort to support a wide range of devices.

As a result of this situation there have been many proposals to reduce the programming cost of applications on heterogeneous systems, many of which are based on libraries [1, 5, 13] due to the advantages they present for their deployment. A particularly interesting one is the Heterogeneous Programming Library (HPL) [14], which operates on top of OpenCL so that it provides the same level of code portability. HPL provides a language embedded in C++ in which the kernels can be written, and a runtime that handles most of the complexity associated to the development of OpenCL applications. HPL applications present much better programmability cost metrics than regular OpenCL ones while achieving virtually the same performance.

While the publications that introduced HPL worked with a single accelerator, this paper presents the first experiences with the implementation of applications on top of HPL that use more than one accelerator. This requires a slightly more complex programming style in which the underlying host arrays are partitioned in several HPL arrays to allow the parallel operation of the devices, as we will see. An evaluation in multi-GPU systems shows that HPL provides substantial programmability improvement with respect to a baseline OpenCL implementation, the performance overhead being however negligible.

The rest of this paper is organized as follows. Section 2 introduces the basics of HPL programming. The existing support for using multiple devices and the associated programming style are explained in Section 3. An evaluation on multi-GPU systems is performed in Section 4, followed by a brief discussion on related work and our conclusions.

2 The Heterogeneous Programming Library library

The Heterogeneous Programming Library (HPL), available at `http://hpl.des.udc.es`, improves the programmability of heterogeneous systems while providing portability through an approach where the computational kernels that exploit heterogeneous parallelism are written in a language embedded in C++ that the library translates into OpenCL. This way HPL can target any device supported by OpenCL.

The HPL programming model is similar to that of CUDA and OpenCL. It considers a host with an standard CPU and memory, with a number of attached heterogeneous computing devices. The sequential portions of the code are executed in the host, while the parallel parts run in the devices. Each device, has processors that execute SPMD parallel code on data present in the memory of their device. The threads that run a kernel in a device can be grouped in sets that can be synchronized through barriers and share a small scratchpad memory.

Kernels can only work on data available in the devices, whose memories are separated from the host. The implied data transfers between the host and the devices are automatically handled by the library runtime. Kernels written in HPL can use three kinds of memories in the devices: (a) the global memory, which is read/write and shared by all the processors, (b) the local memory, which is a read/write scratchpad shared by all the processors in a group, and (c) the constant memory, which is a read-only memory for the device processors and can be set up by the host.

Similar to CUDA or OpenCL, HPL kernels are run in a space or domain of between one and three dimensions of integers, so that each each point in this space corresponds to a thread that runs in parallel the kernel. Optionally, a space to define the size of the groups of threads that can be synchronized and share local memory can also be defined. HPL provides predefined variables that allow to retrieve from the kernel code the unique global and local identifiers, as well as the sizes of both spaces.

The API of the library has three main components. The first one is the `Array` template class, which allows to define all the variables used in the kernels, including those that must be transferred from or to the host. This way, kernel variables must have type `Array<type, ndim [, memoryFlag]>`, which represents an n-dimensional array of elements of a C++ `type`, or a scalar for `ndim`=0. Scalars and vectors can also be defined with special data types like `Int`, `Float`, `Int4`, `Float8`, etc. The `Array` optional `memoryFlag` either specifies one of the kinds of memory supported (`Global`, `Local` or `Constant`) or is `Private` by default, which specifies that the variable is private to the kernel. The elements that compose an array may be any of the usual C++ arithmetic types or a struct. The arrays passed as parameters to the kernels must be declared in the host using the same type. These variables are initially stored in the host memory, but when they are used as kernel parameters they are automatically transferred to the device. Similarly, the outputs are automatically transferred to the host when needed.

The second API component are predefined functions and macros that allow to write the code of the HPL kernels, including the control constructs and basic functions, together with the predefined variables mentioned before. Specifically, the embedded language uses the same constructs as C++ but their name finishes with an underscore (`if_`, `for_`, ...) and the arguments to `for` loops are separated by commas instead of semicolons. Regarding the predefined variables, for example `idx` provides the global id of the first dimension, while `szx` provides the global size of that dimension. If we add the `l` prefix to this keywords we obtain their local counterparts and if we replace the letter `x` with `y` or `z`, we obtain the same values for the second and the third dimensions respectively.

Kernels are written as regular C++ functions or functors that use these elements and whose parameters are passed by value if they are scalars, and by reference otherwise. The `saxpy` routine in Figure 1 constitutes an HPL kernel that implements the SAXPY (Single-precision real Alpha X Plus Y) vector BLAS routine, which computes $Y = a \times X + Y$. In

```
#include "hpl.h"

using namespace HPL;

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

int main(int argc, char *argv) {
  Float a;
  Array<float, 1> x(1000), y(1000);
  //x, y and a are filled in with data (not shown)
  eval(saxpy).global(1000).local(10)(y, x, a);
}
```

Figure 1: SAXPY HPL code

this kernel, each thread `idx` computes a different position of the result `y[idx]`.

Finally, there is a host-side API to inspect the devices available and to order the execution of the kernels. Its most important component is the function `eval`, which requests the execution of a kernel `f` with the syntax `eval(f)(arg1, arg2, ...)`. By default, the global domain of the execution is given by the dimensions and size of the first argument, while the local domain is automatically chosen by the library. However, these and other parameters can be detailed by inserting specifications, in the form of methods, between `eval` and the argument list. For example, the global and the local domains can be specified using methods called `global` and `local` respectively. This way, if a global domain 100×500 and a local domain 4×5 are desired, function eval must be invoked as `eval(f).global(100, 500).local(4, 5)(a, b)`. The `main` function in Figure 1 contains an example host code for the `saxpy` routine, where a global domain of 1000 elements and a local domain of 10 elements are chosen for the kernel execution.

3 Exploiting multiple heterogeneous devices

As mentioned in Section 2, HPL provides an API to find the devices available in a system. This API allows to choose specific kinds of devices, such as CPUs, GPUS or generic accelerators, as well as each individual devices within each kind, in case there are several devices of the same kind in the system. For example, `Device d(GPU,0)` would build a handle `d` of type `Device` that refers to the first GPU in the system. The number of devices of each kind can be obtained with the function `getDeviceNumber(`*`type`*`)`. A `Device` handle can be used to obtain the relevant properties of the device (memory sizes, number of processors, etc.)

by means of the the method `getProperties`, which fills in a predefined structure of type `DeviceProperties` with a field for each one of these properties. This way the interface is similar to that of CUDA's `cudaGetDeviceProperties` [10]. A device handle `d` is also needed to request the execution of a kernel on that specific device by means of the `device` modifier to an `eval` invocation using the syntax `eval(f).device(d)(...)`.

Kernel runs are asynchronous in HPL so that after requesting them, the host program continues the execution of the main thread in parallel with the kernel executions. As a result, a series of `eval`s on different devices give place to parallel executions of the associated kernels in the requested devices.

The HPL synchronization mechanism that allows to wait for a kernel execution to finish and then retrieve its results is based on the host accesses to the `Array`s used as arguments to the kernel executions. This way, when the host code tries to read an array that was written by a previously launched kernel, the HPL runtime waits for the kernel to finish and copies the resulting array to the host memory, after which the execution of the main thread in the host is allowed to continue. Subsequent host accesses to the array are immediately satisfied from the host-side copy until new kernel executions that write to the array are requested. Similarly, an array used as input in a kernel execution is copied to the device only in the first usage of the array in the device or if the host has written to the array in its memory after the most recent usage of the array in the device.

These mechanisms were originally designed only to support single-device executions in [14]. However, they also allow to run kernels in parallel on multiple devices when the kernels of each device operate on different `Array`s, which is the most common situation. The reason is that if each `Array` is only used either in the host or in a single device, it has exactly the same coherency requirements as the execution in a single device, since in that case there is no need to move data or keep the coherency of the `Array` across different devices.

Interestingly, with the aforementioned implementation it is also possible to implement applications in which the very same array is shared by multiple devices, provided it is only read in the devices. It just suffices to define a different `Array` to use for each one of the devices, but with all the `Array`s pointing to the same memory location in the host. Our library allows this because, although not detailed in its short description in Section 2, the host-side constructors for `Array`s support a last optional argument that provides a pointer to the location in the host memory where the underlying `Array` data resides in the host. As a result, several `Array`s that point to the same host data can be defined. This simple technique further extends the scope of application of our current HPL runtime to exploit multiple heterogeneous devices.

Just as in the case of single-device executions, in multi-device environments HPL also automatically takes care of all the transfers while minimizing them. For the reason that in these environments there are more transfers and buffers, the advantages for the users in

```
 1 float x[N], y[N];
 2 Float a;
 3 Array<float,2> **vx, **vy;
 4
 5 int nGPUs = getDeviceNumber(GPU);
 6 vx = new Array<float, 1> * [nGPUs];
 7 vy = new Array<float, 1> * [nGPUs];
 8
 9 for(i = 0; i < nGPUs; i++) {
10   vx[i] = new Array<float, 1>(N/nGPUs, x + i*(N/nGPUs));
11   vy[i] = new Array<float, 1>(N/nGPUs, y + i*(N/nGPUs));
12 }
13
14 for(i=0; i< nGPUs; i++)
15   eval(mxProduct).device(Device(GPU,i))(*vy[i], *vx[i], a);
```

Figure 2: SAXPY HPL code parallelized on multiple GPUs

terms of programming effort reductions are proportionally larger.

Since the `Array` is the unit of consistency and each `Array` can only be used in a single device, when a user wants to run in parallel kernels that compute the values of different portions of a problem array, he or she must use a separate HPL `Array` for each portion to use in a different device. Building several HPL `Array`s associated to different portions of the same C/C++ underlying array is facilitated by the already mentioned fact that the constructor of these objects allows a final optional argument to specify the location in host memory of the data managed by the `Array`. This way, different `Array`s can start in different positions within the same C array.

The resulting programming style is exemplified in Figure 2 with a multi-GPU implementation of SAXPY that uses the kernel shown in Figure 1, so that only the host-side code is shown. To allow maximum flexibility, the code allocates at runtime arrays of pointers to `Array`s (`vx` and `vy` in this example) with one entry per available GPU, as provided by `getDeviceNumber(GPU)`. In order to simplify the example code we assume that the length N of the arrays is divisible by the number of available devices, stored in $nGPUs$. This way, the loop in lines 9-12 allocates the `Array`s for the i-th GPU to start in the position $i * (N/nGPUs)$ within the underlying host vectors x and y defined in line 1, with a fixed length of $N/nGPUs$ elements. Finally, the loop in lines 14-15 requests the execution of the kernel in the i-th GPU using its associated `Array`s. It deserves to be mentioned that since x is a read-only input to SAXPY, its partition in multiple `Array`s is not required to effectively parallelize its execution on multiple GPUs. Instead, we could have simply defined multiple non-partitioned `Array`s copies that point to the same underlying host memory array, each

Table 1: Benchmarks and programmability evaluation.

Benchmark	SLOCs host OpenCL	SLOCs kernel OpenCL	SLOCs host HPL	SLOCS kernel HPL	Reduction host (%)	Reduction total (%)
EP	325	115	269	115	17.2	12.7
Matmul	220	23	181	22	17.7	16.5
Coulombic	178	17	118	17	33.7	30.8
Spmv	263	23	172	22	34.6	32.2

one of them being used in a different device. Nevertheless it is better to use partitioned `Array`s of `x` for two reasons. First, this allows to copy to each GPU only the portion of the vector that it needs for its computations, rather than the whole vector. The second reason is that this allows to use the same indexing for the vectors y and x within each kernel execution, as they both begin in the same relative position within the corresponding global vector.

4 Evaluation

We now evaluate the implementation of multi-GPU applications on top of our HPL runtime both in terms of programmability and performance using as baseline the corresponding OpenCL codes. We wrote the baseline using the OpenCL C++ API, which exploits the advantages of this language, such as object oriented programming, so that the base language characteristics do not impact the comparison. The codes used in the evaluation are the EP benchmark from the SNU NPB suite [12], a dense matrix-matrix product that splits the work in chunks of rows between the GPUs (Matmul), the Coulombic potential application delivered with SkePU [5], and the sparse matrix-vector kernel from the SHOC Benchmarks [4] (spmv), which although not discussed in [4], was added later to the suite. All the benchmarks run kernels that write on different `Array`s on different devices, that is, no `Array` is modified in more than one device. Also, all the benchmarks but EP have one array that is read by all the devices, which allows to test the sharing of host read-only data by multiple devices.

Table 1 compares the programming effort of OpenCL and HPL for the codes used in the evaluation using an objective metric derived from the source code. Namely, we have applied the well-known metric of the SLOCs, which counts the number of source lines of code excluding comments and empty lines. The measurements consider separately the host side and the kernel side of the applications because the kernel code is virtually the same for OpenCL and HPL. In fact we can see that HPL kernels have the same length or are slightly shorter than OpenCL kernels. In addition, the kernels remain mostly unaffected when the

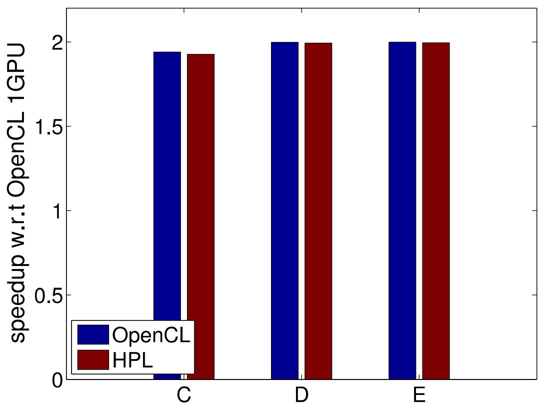

Figure 3: EP speedups using 2 GPUs of OpenCL and HPL with respect to OpenCL on one GPU for several problem class sizes

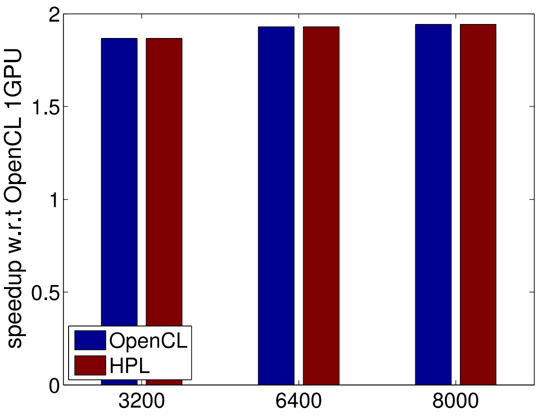

Figure 4: Matmul speedups using 2 GPUs of OpenCL and HPL with respect to OpenCL on one GPU for several matrix sizes

applications change to support multiple devices, with at most one or a few offsets being added to the arguments of the kernels for the multi-device version. Finally, these small changes are identical in OpenCL and HPL. Regarding the host code, it is important to underline that the lengthy initialization of OpenCL (platform and device selection, creation of context and command queue, loading and compilation of kernels) was removed from the baseline code because it is easy to encapsulate it in common routines that can be reused across most applications. This way, our baseline replaces these costly operations with calls to a predefined library, so that our baseline corresponds to the minimum amount of code that a user has to write when using the OpenCL host C++ API. Despite this fact, the OpenCL host code is substantially longer than its HPL counterpart, as we have observed reductions of the measured programming cost between 17.2% and 34.6% when migrating from OpenCL to HPL as Table 1 shows. The variability depends on multiple factors including the length of the host code needed to initialize the data and process the results, which is the same in OpenCL and HPL, and the number of arguments to the kernels, playing a much more important role buffers than scalars.

The performance evaluation took place in an NVIDIA Tesla Fermi 2050 with 3GB whose host has an Intel Xeon X5650 (6 cores) at 2,67GHz and 12GB RAM. The compiler was g++ 4.7.2 with optimization level `-O3`. Figures 3 to 6 show the speedups that the OpenCL and HPL versions achieve when running on two GPUs with respect to the baseline OpenCL implementation executed on a single GPU for three problem sizes for each one of our four benchmarks. It deserves to be mentioned that EP, Matmul and Coulombic work with double-precision data, while Spmv operates on single-precision floating point values. Also, Spmv only has a 1% of non zeros in its matrix, therefore it presents a very

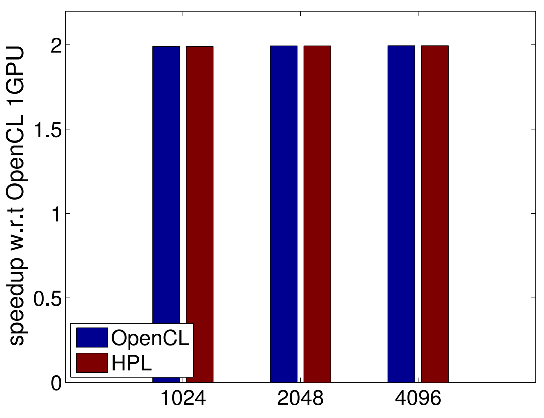

Figure 5: Coulombic speedups using 2 GPUs of OpenCL and HPL with respect to OpenCL on one GPU for several numbers of atoms

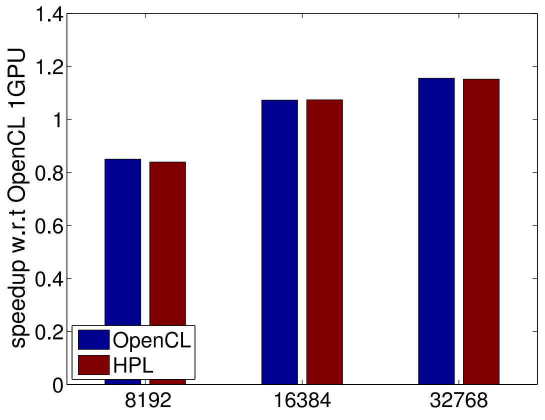

Figure 6: Spmv speedups using 2 GPUs of OpenCL and HPL with respect to OpenCL on one GPU for several matrix sizes

sparse access pattern, which together with the small proportion of computations in relation to the number of data items accessed gives place to smaller speedups than in the other kernels. In fact we can see that for the smallest problem size, two GPUs are slower than a single one. This run is also the shortest one measured, as it accounts for just 2.23 ms. in a single GPU, which becomes 2.62 when using two GPUs under OpenCL and 2.64 when using HPL. This is an overhead of 0.76% of HPL with respect to OpenCL, which is in fact the maximum overhead measured in our tests. This way, the most important conclusion from the performance measurements is that the overhead of HPL with respect to OpenCL is totally negligible, as the speedups are almost identical in all the situations.

5 Related work

There are multiple proposals that try to improve the programmability of heterogeneous devices with respect to the most widely used tools [10, 8]. This way, some approaches annotate sequential code with directives that guide the creation of the device kernels and the related data and synchronization management by the compiler [9, 6, 11]. Compiler directives usually require little programming effort, but their expressiveness is seldom enough to specify all the details needed to attain the maximum performance our of the heterogeneous devices, which offer a much larger number of knobs to tune than regular CPUs and whose performance is very sensitive to the implementation decisions taken. These facts, together with the lack of a clear performance model and the sometimes suboptimal quality of the code generated, which are matters traditionally associated to tools that rely on compiler transformations, together with the requirement of specific compilers, has precluded the widespread usage of this approach. Embedded languages [3] share basically the same restrictions, with the additional cost that codes must be rewritten in the new language.

As a result, most approaches have relied on the usage of libraries that naturally integrate into existing widespread languages. Each library presents its own focus and restrictions. For example, [2] only works on CPUs and Nvidia GPUs, and it only allows to work with unidimensional arrays in one-to-one computations in which the user cannot request to use local or constant memory or specify the number of threads to use. A more general solution is provided by libraries [5, 13] that provide classes for arrays similar to HPL `Array`s and higher order functions that implement typical computational patterns, also known as skeletons. These functions can be parameterized with user-defined kernels in the form of strings or member functions, thus facilitating the exploitation of the heterogeneous devices for those calculations with a high level structure that fits one of the predefined available skeletons.

The Heterogeneous Programming Library (HPL) belongs to the family of library-based proposals to improve the programmability of heterogeneous systems. It provides the unique feature of building its kernels at runtime from the language embedded in C++ described in Section 2. Although not discussed in this paper, this largely facilitates the use of run-time code generation (RTCG) [14], which is a valuable tool to adapt kernels to the features of the specific device to use. This coupled with the usage of OpenCL as backend turns HPL into a valuable resource to implement heterogeneous applications with good levels of performance portability. Contrary to previously discussed libraries, HPL is not restricted to particular patterns of computation, thus it can be applied in all situations.

6 Conclusions

While heterogeneous devices offer many advantages in comparison with traditional general-purpose CPUs, their programming, which requires the specification of many details, also requires much more user effort. The situation is even worse in the case of the OpenCL standard, as the effort to make it portable has given place to a highly generic API that requires many steps and the management of many elements. When multiple heterogeneous devices rather than a single one are used in a program, this complexity is even larger, as each device has unique data structures and must be managed separately. In this paper we have described for the first time the programming of multiple heterogeneous devices in the same application on top of the Heterogeneous Programming Library (HPL), a project to facilitate the programming of heterogeneous devices. While the implementation is restricted to allowing modifications on each portion of an array only by a single device, this still suffices to parallelize a large class of algorithms. A comparison with baseline OpenCL versions of several codes reveals that HPL reduces the SLOCs of the host code between 17% and 34% while achieving virtually the same performance in multi-GPU applications. In fact the overhead of HPL with respect to the baseline never exceeded 0.76% even for very short executions of a few milliseconds. As a result, we think that HPL is a very promising approach to develop applications that exploit heterogeneity.

As future work we plan to implement a general coherency system that allows to arbitrarily use the same HPL array in any number of devices, always keeping a consistent view of its data, and providing optimized transfers between devices when necessary. In a second step, we will extend the expressiveness of HPL to support assignments of data between different arrays and the usage of portions of arrays in the kernel executions and in the assignments. We also want to explore the adaptation of the runtime of HPL to each specific environment to maximize the performance it provides.

Acknowledgements

This work was supported by the Xunta de Galicia under the Consolidation Program of Competitive Reference Groups (Ref. GRC2013/055) and by the Spanish Ministry of Science and Innovation (Ref. TIN2010-16735), both of them cofunded by FEDER funds of the European Union. Zeki Bozkus is funded by the Scientific and Technological Research Council of Turkey (TUBITAK; 112E191).

References

[1] A. Acosta and F. Almeida. Skeletal based programming for dynamic programming on multigpu systems. *The Journal of Supercomputing*, 65(3):1125–1136, 2013.

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

[3] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. 16th ACM symp. on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, 2011.

[4] 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)*, pages 63–74, 2010.

[5] 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.

[6] T.D. Han and T.S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. on Parallel and Distributed Systems*, 22:78–90, 2011.

[7] IBM, Sony, and Toshiba. *C/C++ Language Extensions for Cell Broadband Engine Architecture*. IBM, 2006.

[8] Khronos OpenCL Working Group. The OpenCL Specification. Version 2.0, Nov 2013.

[9] 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)*, pages 1–11, 2010.

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

[11] OpenACC-Standard.org. The OpenACC Application Programming Interface Version 2.0a, Aug 2013.

[12] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Proc. 2011 IEEE Intl. Symp. on Workload Characterization*, IISWC '11, pages 137–148, 2011.

[13] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - a portable skeleton library for high-level GPU programming. In *2011 IEEE Intl. Parallel and Distributed Processing Symp. Workshops and Phd Forum (IPDPSW)*, pages 1176 –1182, may 2011.

[14] 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.