A Comparative Study of OpenACC Implementations

Ruymán Reyes, Iván López, Juan J. Fumero and Francisco de Sande[1]

Abstract— **GPUs and other accelerators are available on many different devices, while GPGPU has been massively adopted by the HPC research community. Although a plethora of libraries and applications providing GPU support are available, the need of implementing new algorithms from scratch, or adapting sequential programs to accelerators, will always exist. Writing CUDA or OpenCL codes, although an easier task than using their predecessors, is not trivial. Obtaining performance is even harder, as it requires deep understanding of the underlying architecture. Some efforts have been directed toward the automatic code generation for GPU devices, with different results. In this work, we present a comparison between three directive-based programming models: hiCUDA, PGI Accelerator and OpenACC, using for the last our novel `accULL` implementation.**

Key words— **OpenACC, Accelerators, GPGPU, CUDA, OpenCL, OpenMP, compiler, productivity**

I. Introduction

In recent years, the use of hardware accelerators in HPC has become ubiquitous: All computer manufacturers offer high-performance platforms that are configured as a number of conventional multicore processors with one or more connected graphics cards to relieving certain kinds of computation.

A few months ago CUDA and OpenCL were the most common tools available to develop applications in such heterogeneous environments. CUDA is an extension to ANSI C that allows to compile programs containing CUDA kernel code into executables that can use a GPU connected to a conventional computer. It is a proprietary technology that only allows to exploit NVIDIA GPUs.

OpenCL is a language for creating task-based and data-based parallel applications that can run on both CPUs as GPUs. The language used for kernels is based on C99, eliminating certain features and extending it with vector operations.

During Seattle SC2011 the new OpenACC standard for heterogeneous computing [1] was presented. As in his day the introduction of OpenMP was a major boost to the popularization of shared memory HPC systems, this new standard, adopted by industry leaders, lightens the coding effort required to develop heterogeneous parallel applications. Following the OpenMP approach, in the OpenACC API, the programmer annotates her sequential code with compiler directives, indicating those regions of code susceptible to be executed in the the GPU. The simplicity of the model, its ease of adoption by non expert users and the support received from the leading companies in this field make us believe that it is a long-term standard.

As a continuation of our recent years work [2], [3], we have developed `accULL`, an implementation of the OpenACC standard. The implementation is based on a compiler driver, `YaCF` and a runtime library (named Frangollo). `accULL` offers support for the most common used constructs and we are able to run in both CUDA and `OpenCL` platforms. To the best of our knowledge, ours is the first open-available implementation of the standard that supports `OpenCL`. In this work we compare the characteristics of the implementations of four well known algorithms using `accULL`, PGI [4] and hiCUDA [5].

The rest of the paper is organized as follows. We begin with a brief presentation of the research efforts related to directive based GPU code generation in Section II. In Section III we present the three different approaches to directive based programming evaluated in this work. Section IV describes the experimental strategy including the testbed, various benchmarks used and provides the experimental results. Finally, Section V includes the conclusions we have been able to achieve so far and ideas about future work regarding the `accULL` project.

II. Related work

The Cetus project [6] proposes a source to source compiler framework for automatic translation and optimization. In order to improve the performance, Cetus allows a wide range of loop manipulations. A set of extensions over OpenMP enable it to automatically generate and tune CUDA code. Memory transfers are analyzed and detected at compile time by a set of sophisticated interprocedural analysis. The `accULL` compiler framework, `YaCF`, requires less programming effort to develop code transformations, In addition, we tackle the generation of `OpenCL` kernels in `YaCF` and its preparation and execution in the `accULL` runtime.

The Mercurium [7] source to source compiler has been developed by the Nanos team and it is the compiler behind the `OmpSs` programming model. Starting from OpenMP or Superscalar sources and using a runtime library, the compiler produces efficient parallel code for different architectures. The Nanos team focus their attention in task parallelism and they do not implement neither loop parallelism nor automatic generation of GPU kernels.

The CAPS HMPP [8] toolkit is a set of compiler directives, tools and software runtime that supports parallel programming in C and Fortran. HMPP works based on codelets that define functions that

[1]Dept. de E. I. O. y Computación Universidad de La Laguna, 38271–La Laguna, Spain e-mail: {rreyes,ilopezro,jfumeroa,fsande}@ull.es

will be run in a hardware accelerator. These codelets can either be hand–written for a specific architecture or be generated by some code generator.

III. Approaches for directive based programming

A. PGI Accelerator Model

The PGI Accelerator model proposes a set of directives, resembling those used in OpenMP, that helps compilers to generate GPU kernels. Most of the directives in the model are optional and they are used to improve performance. The only required directive is the `acc region` that indicates a region containing loops with kernels. Data-flow, alias and array region analysis are used to determine what data need to be allocated on the accelerator and copied to and from the host.

The PGI compiler maps loop parallelism to the hardware architecture using a Planner module [9], that uses information from other analysis phases present in the compiler. Strip-mining is heavily used to accomplish loop mapping and user can use optional directives to force these loop transformations.

It is worth noting that the PGI Accelerator model is available in both Fortran and C PGI compilers. In both cases, the compiler generates a single binary containing GPU and CPU versions of the code, thus, this binary can run on platforms without GPU. Although it is possible to show the intermediate kernel files, they are not easy to understand or modify.

B. hiCUDA

hiCUDA (for high-level CUDA) provides the developer with a set of pragmas that map to usual CUDA operations. Kernels are automatically extracted from the original source file, and iterations are distributed across threads and blocks according to loop partitioning clauses. It is possible to use shared memory within kernels using a specific directive. One single source file can be used for sequential and GPU versions of the code. Calls to memory management routines are replaced by pragmas and user does not need to keep track of device pointers.

The hiCUDA driver parses the original source using the GNU-3 frontend (to which hiCUDA has been added) and then operates on Open64 IR (WHIRL) to replace the pragmas, extract the loops and inject the appropriate CUDA runtime calls. Finally it uses the C code generator, with the extended CUDA syntax, to generate the target code.

The fact that the directives are almost a direct translation of the CUDA programming model, forces the users to know more details about the underlying platforms than other approaches. This also difficult hiCUDA to be ported to different accelerators, though probably it was not intended at all. The result of compiling with the hiCUDA driver is not a binary file but a directory with a single CUDA source together with some required headers. This source has to be compiled with NVIDIA tools to generate the final binary.

C. accULL

Our approach to the OpenACC implementation is a two-layer based development composed by a source to source compiler and a runtime library, in a similar fashion to other compiler infrastructures. Several of the aforementioned related works use a similar approach. The result of our compilation stage is a project tree hierarchy with compilation instructions, suitable to be modified by advanced end-users. Default compilation instructions enable average users to generate an executable without additional effort. The aim of this approach is to maintain a low development effort in the programmer side, while keeping the opportunity window for further optimizations performed by high-skilled developers.

The compiler is based on our `YaCF` research compiler framework [3], while the runtime (Frangollo) has been designed from scratch. `accULL` is the combination of the `YaCF` driver and the Frangollo runtime library.

A `YaCF`-driver translates the annotated C+OpenACC source code into a C code with calls to the Frangollo API. The `YaCF` compiler framework has been designed to create source to source translations.

User annotations are validated against data dependency analysis. A warning is emitted if variables are missing. Also, we can check whether a variable is read-only or not, to allocate the appropriate type of memory.

Source to source translation injects a set of Frangollo calls within the serial code. Whenever these calls are issued, control is deferred to the Frangollo runtime, that will execute the code of the proper API call or whatever other code it might require (for example, to handle previous asynchronous operations). Frangollo deals with two main issues of any OpenACC implementation: memory management and kernel execution.

Frangollo consists of several separate pluggable components. A common component serves as an abstract interface to all kind of components. Generic operations over devices, like memory transfers or kernel execution, are mapped on top of an abstract interface. Operations at this level refer to three main objects: Context, Devices and Variables. Components instantiate the basic operations to perform the actual work. Interfaces access the abstract layer without requiring to know which component is enabled or not.

The `YaCF` driver supports most of the syntactic constructs in the OpenACC 1.0 specification, but some of them are silently ignored. In addition, although some operations inside Frangollo runtime are handled asynchronously, support for the `async` OpenACC clause has not been implemented yet. Developers can use a runtime call to get kernel profiling information and detect potential bottlenecks.

IV. Evaluation

A. Experimentation methodology

We have done our best effort to implement the codes using all the programming model capabilities. Despite this best effort, resulting implementations, not being naive, are not what an highly skilled GPU developer could produce.

We believe that this is the real scenario for directive-based GPU programming, as GPU experts might find better ways to exploit accelerators, whereas scientist and engineers with no particular experience will prefer to improve performance without requiring excessive amount of development time.

The Figures show the performance of the main section of codes (i.e, that were most of the time is spent). The initialization of CUDA devices is always performed outside time measurement and it is constant across all compiler tools.

B. Compiler tools

This research has been performed using the PGI Compiler toolkit version 12.2, which implements the PGI Accelerator Programming Model version 1.3. The API call `acc_init` was used in order to hide initialization costs.

The subversion release of the hiCUDA compiler was obtained from the hiCUDA project website. hiCUDA does not offer the possibility of preinitializing the device. To prevent device initialization costs to hinder hiCUDA performance, we manually inject initialization code to hiCUDA sources before starting the timers.

Evaluation of the OpenACC directives using commercial compilers was not feasible at the time of writing. To illustrate the capabilities of this programming model, we use our `accULL` implementation, described in Section IV-C. that represents a major step in the direction of a complete and efficient implementation of the OpenACC standard. The `accULL` runtime automatically starts devices before running the program.

C. Experimental platform

For these experiments, an NVIDIA Tesla 2050 GPU with 4GB memory was used. A desktop computer with an Intel Core i7 930 processor, running at 2.80GHz, with 1MB of L2 cache and 8MB of L3 cache serves as the host.

D. Motivating example: Matrix Multiplication

Matrix multiplication (`MxM`) is a basic kernel frequently used to showcase the peak performance of GPU computing. In this Section, we use a naive matrix multiplication implementation to show the differences and similarities among the three different programming models evaluated.

With these three implementations we aim to show the basic syntax and usage mode of the programming models. No loop transformation has been performed manually and results shown in Figure 1 showcase the performance that an average user without deep knowledge of the CUDA programming model would obtain. For the matrix product, the PGI compiler produces the best GPU codes. All implementations outperform OpenMP implementation for all problem instances, maintaining a similar development effort.

D.1 PGI Accelerator implementation

The naive `MxM` matrix multiplication implementation is shown in Listing 1. Data transfers are declared using copy directives. Array regions to be transferred are declared using a syntax that closely resembles the Fortran 90 array indexes.

The philosophy behind PGI seems to be *better safe than sorry*. Whenever the compiler can not determine whether a loop can be run in parallel or not, it warns the programmer and do not generate CUDA code. While compiling this `MxM`, the compiler emitted several warnings, notifying that it cannot guarantee that loops were independent. We had to specify the optional `independent` clause to force the kernel generation.

Listing 1: Sketch of `MxM` in PGI Accelerator Model

```
1 #pragma acc data copyin(b[0:N*N],c[0:N*N
      ]) copy(a[0:N*N])
2 {
3   #pragma acc region
4   {
5   #pragma acc for independent parallel
6   for ( j = 0; j < N; j++)
7   {
8   #pragma acc for independent parallel
9     for ( i = 0; i < N; i++ )
10    {
11      double sum = 0.0;
12      for ( k = 0; k < N; k++ )
13      {
14        sum += b[i+k*N] * c[k+j*N];
15      }
16      a[i+j*N] = sum;
17    }
18  }
19  }
20 }
```

At the time of compilation, PGI generates a single binary file, containing the CUDA kernel. Kernel parameters, like block grid and threads, are computed at compile time. If requested, the PGI compiler is able to show occupancy, block grid and thread configuration at compile time, among other information. It is also possible to show detailed profiling information after execution. This enables the users to have a global idea about the quality of the implementation.

D.2 hiCUDA implementation

A sketch of the `MxM` code in hiCUDA is shown in Listing 2. Global arrays have to be allocated on GPU using the appropriate clause. In this case, the compiler was not able to properly detect the dimensions of the matrix, thus requiring the `shape` directive to indicate how the data are distributed in memory. The usage of shared memory is exposed through the `shared` directive. However, for this particular case, where matrices are dynamic vectors, we were not able to use it inside the kernel, even applying manually

strip mining to the inner loop. It seems that this directive requires the matrix to be a two dimensional static array and does not work properly with linear vectors.

Grid dimensions, specified by `tblock` and `thread` clauses, were selected so that each thread only performs one iteration. Varying `tblock` and `thread` allows the users to fine-tune the kernel configuration. To get maximum performance in different GPU architectures, these values have to be determined by hand. hiCUDA does a great job leveraging loop partitioning from the user and it offers several scheduling possibilities for loop iterations.

Listing 2: Sketch of `MxM` in hiCUDA

```
1 #pragma hicuda shape a [N] [N]
2 #pragma hicuda shape b [N] [N]
3 #pragma hicuda shape c [N] [N]
4 #pragma hicuda global alloc a [*] [*]
5 #pragma hicuda global alloc b [*] [*]
      copyin
6 #pragma hicuda global alloc c [*] [*]
      copyin
7
8 #pragma hicuda kernel mxm tblock(N/16,N
      /16) thread(16,16)
9  #pragma hicuda loop_partition
       over_tblock over_thread
10  for ( i = 0; i < N; i++ ) {
11  #pragma hicuda loop_partition
       over_tblock over_thread
12   for ( j = 0; j < N; j++)
13     {
14       double sum = 0.0;
15       for ( k = 0; k < N; k++ )
16       {
17         sum  +=  b[i+k*N] * c[k+j*N];
18       }
19       a[i+j*N] = sum;
20     }
21   }
22 #pragma hicuda kernel_end
23 #pragma hicuda global copyout a [*] [*]
24 #pragma hicuda global free a b c
```

D.3 OpenACC Implementation

Listing 3 shows a potential OpenACC implementation of the `MxM` code. In this implementation, we choose to use an external `kernels` construct. This construct creates a data region and sets the variables required inside and/or outside the region. Inside the `kernels` construct, we define two loops that the `accULL` implementation translates into GPU kernels. The first loop (line 4) deals with matrix initialization. The `collapse` clause in line 3 indicates to the compiler driver that the loop is suitable to be extracted as a 2D kernel. We use the loop nest at line 13 to generate a kernel with the inner loop.

Listing 3: Sketch of `MxM` in OpenACC

```
1 #pragma acc kernels name("mxm") pcopy(a[N
      *N]) pcopyin(b[N * N],c[N * N])
2 {
3  #pragma acc loop private(i, j,k)
       collapse(2)
4  for ( i = 0; i < N; i++ )
5   for ( j = 0; j < N; j++)
6     {
7       double sum = 0.0;
8       for ( k = 0; k < N; k++ )
9       {
10         sum  +=  b[i+k*N] * c[k+j*N];
11       }
12       a[i+j*N] = sum;
13     }
14
15 }
```

One of the most important aspects of CUDA tuning is an appropriate thread and kernel block selection. The CUDA component of Frangollo deals with this issue computing the appropriate thread/block combination through an estimator fed with the compute intensity information extracted by the `YaCF` driver.

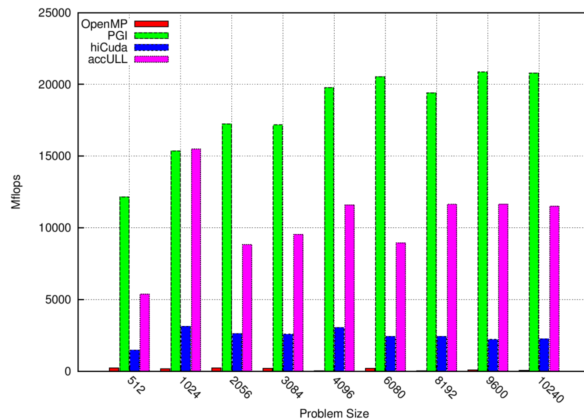

Fig. 1: Floating point performance comparison between OpenMP, `accULL`, PGI and hiCUDA for `MxM`

E. Rodinia

The Rodinia Benchmark suite [10] comprises compute-heavy applications meant to be run in the massively parallel environment of a GPU and cover a wide range of applications. OpenMP, CUDA and OpenCL versions are available for most of the codes in the suite. In this contribution we provide computational results for three codes: a LU decomposition (`LUD`), a thermal simulation tool (`HS`) and a nonlinear global optimization method for DNA sequence alignments (`NW`).

HotSpot (`HS`) is a thermal simulation tool used for estimating processor temperature based on an architectural floor plan and simulated power measurements. The Rodinia implementation includes the 2D transient thermal simulation kernel of `HS`, which iteratively solves a serie of differential equations for block temperatures. The inputs to the program are power and initial temperatures. Each output cell in the grid represents the average temperature value of the corresponding area of the chip.

It is worth noting that, although OpenACC and OpenMP have great similarities, a major conceptual difference between them exists: *Data management is independent from parallelism.* A developer used to work with OpenMP could attempt to parallelize both loops separately, using a `acc parallel loop` construct for each one in the same way as a `omp parallel for` from `OpenMP`. This would force the compiler to create separate data regions for each loop and therefore data would be copied in and out the device when entering and exiting from both loops.

As these loops are executed once per iteration step, data would be transferred twice in and out the device per step. However transferring in an out the whole matrix is not required, as the whole computation can be done inside the GPU device. An important optimization to this particular piece of code is to allocate and transfer the matrices outside the outermost loop. In plain CUDA code (i.e, not using directives) this would require writing the allocation and memory transfer calls before the loop.

Using hiCUDA, a pair of `shape` and `global alloc` directives for each matrix used inside the loop, with the appropriate `copyin` clause, would suffice. After this loop, the results can be transferred back to the host with the `copyout` directive. Global memory has to be deallocated after loop termination with the appropriate directive.

The PGI accelerator directives allow developers to use the `data` clause outside the outermost loop, which defines an explicit data region for the loop. Data regions may be nested and usually contain compute regions. Data regions enclose a C compound statement and when the end of this compound statement is reached, data are copied back to the host. The clauses `copy`, `copyin` and `copyout` are used to indicate variable directionality.

In OpenACC it is possible to use the `kernels` directive, which defines a data region containing a set of loops that will be executed on the accelerator device.

It is not necessary to inline the subroutine if developer is using our `accULL` implementation. Replacing the `kernels` directive with a `data` directive, and then using the `kernels` inside the subroutine is enough for the runtime to track the usage of the host variables and handle properly their device counterparts. A sketch of this approach is shown in Listing 4.

Figure 2 shows the efficiency of each directive-based implementation over the native CUDA code. This native CUDA implementation is the best for all problem sizes. The PGI compiler performs badly for small problem sizes while hiCUDA and `accULL` show a performance between 40% and 75% of the native approach. It is worth to mention that for small problem sizes, better performance is achieved using OpenMP over the CPU than using directive-based GPU programming. This make us believe that exploring combinations of OpenMP and GPU directives for different problem sizes might be interesting. Also, it is noticeable that the native GPU version of the code has 366 lines of code, while the PGI implementation has 275, the `accULL` 250 and the hiCUDA 291 lines.

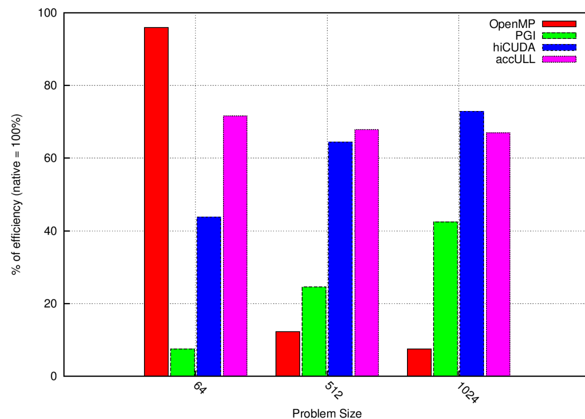

Fig. 2: Performance comparison of different `HS` implementations, showing efficiency over native implementation.

Listing 4: Sketch of `HS` using OpenACC

```
1  void do_iteration(double * temp,...) {
2   #pragma acc kernels ....
3     { /* Compute current temperatures */
4       #pragma acc loop ...
5       for (r = 0; r < row; r++)
6         for (c = 0; c < col; c++)
7           ...
8       /* Update */
9       #pragma acc loop ...
10      for (r = 0; r < row; r++)
11          for (c = 0; c < col; c++)
12              ....
13    }
14 }
15 void routine(...) {
16 ...
17 #pragma acc data copy(temp,...)
18 for (i = 0; i < n_it ; i++)
19     do_iteration(temp ... )
20 }
```

This increased number of lines of code (LOC), mainly due the requirement of manually writing the CUDA runtime calls and restructuring the code, represents a remarkable increment of the development effort. OpenACC has been designed with a general concept of accelerator in mind, rather than CUDA-enabled GPUs. Taking advantage of this general design, our `accULL` implementation supports also OpenCL platforms with the same set of directives.

P. Size	Native	`accULL`	PGI	OpenMP	hiCUDA
64	0.0603	0.0087	0.0924	0.0016	0.0625
256	0.0600	0.0644	0.2074	0.0059	0.0975
512	0.0637	0.2699	0.3981	0.0458	0.2461
2048	0.1345	8.7984	4.4857	8.9619	11.1810

TABLE I: Execution time (s.) for four different instances of the LU decomposition code from Rodinia.

`Needleman-Wunsch (NW)` is a nonlinear global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2D matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. The optimum alignment is the pathway through the array with maximum score, where the score is the value of the maximum weighted path ending at that cell. Thus, the value of each data element depends on the values of its northwest-, north- and west-adjacent elements. In the second step, the maximum path is traced backward to deduce the optimal alignment. Properly scheduling iterations within each loop is critical to improve performance. Performance comparison of PGI, hiCUDA, `accULL` and OpenMP implementations with respect to the native CUDA implementation is shown in Figure 3.

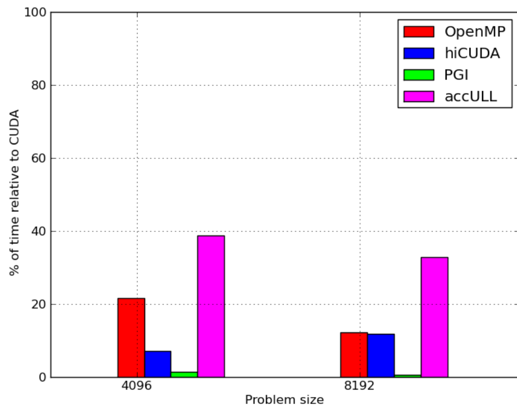

Fig. 3: Performance comparison of `NW`.

When working with hiCUDA, it is possible to schedule loop iterations with different combinations of blocks and threads. We explored several different thread number to get the maximum performance. Despite of `NW` loops being triangular, hiCUDA was able of properly scheduling the iterations across threads. The `accULL` loop scheduling for this particular algorithm is similar to the hiCUDA version, thus performance is comparable.

Performance of the PGI implementation is the worst in this case. Despite our best efforts to force the compiler to schedule the loops in the GPU, its dependency analysis created sequential GPU kernels.

`LU` Decomposition is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This application has many row-wise and column-wise interdependencies and requires significant optimization to achieve good parallel performance. Execution times for the different implementations of this algorithm are shown in Table I. Although in this case performance for smaller problem size is achieved using OpenMP, it is important to highlight the outstanding performance achieved with larger problem sizes with the native implementation. However, the complexity of dependences within the loop greatly difficult the implementation using directive-based programming. Native implementation heavily uses shared memory, which `accULL` currently does not use. PGI compiler caches some array accesses on shared memory, thus its performance better than `accULL` in this case. hiCUDA performance is lower because we were not able to use shared memory directives.

V. Conclusions and future work

In this work we have presented three different directive-based GPU programming models. These models somewhat represent the evolution of this approach during the last years. HiCUDA came first and presented a simplification of the CUDA model with 1:1 translation of directives, whereas PGI came later with a more complex and powerful model. OpenACC represents the final effort to push forward a standard based on the previous experience.

Our implementation of the OpenACC standard can be used as a fast-prototyping tool to explore optimizations and alternative runtime libraries. Frangollo runtime can be fully detached from the compiler environment and used together with a different commercial or production-ready compiler, like LLVM or Open64, to implement the OpenACC standard in a short time. Memory allocation, kernel scheduling, data splitting, overlapping of computation and communications or parallel reduction implementation are some of the issues that can be tackled within the runtime independently from the compiler. In addition, performance of `accULL`, despite of being an openly-available project, in some cases outperforms results of other approaches. As future work, we would like to increase the number of algorithms implemented in OpenACC and other directive models and to add other OpenACC compliant compilers to the performance comparison.

Acknowledgment

This work has been partially supported by the EU (FEDER), the Spanish MEC (Plan Nacional de I+D+I, contracts TIN2008-06570-C04-03 and TIN2011-24598), HPC-EUROPA2 (project number 228398) and the Canary Islands Government, ACI-ISI (contract PI2008/285).

References

[1] "OpenACC directives for accelerators," 2011.

[2] R. Reyes and F. de Sande, "Automatic code generation for GPUs in `llc`," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 349–356, Mar 2011.

[3] R. Reyes and F. de Sande, "Optimization strategies in different CUDA architectures using `llCoMP`," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 36, no. 2, pp. 78–87, Mar 2012.

[4] The Portland Group, "PGI Fortran & C accelerator programming model v1.2," March 2010.

[5] Tianyi David Han and Tarek S. Abdelrahman, "hiCUDA: High-level GPGPU programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.

[6] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[7] Roger Ferrer, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé, "Unrolling Loops Containing Task Parallelism," in *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, 09/2009 2009.

[8] Francois Bodinand Stephane Bihan, "Heterogeneous multicore parallel programming for graphics processing units," *Sci. Program.*, vol. 17, pp. 325–336, December 2009.

[9] Michael Wolfe, "Implementing the PGI accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, New York, NY, USA, 2010, GPGPU '10, pp. 43–50, ACM.

[10] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, Washington, DC, USA, 2010, IISWC '10, pp. 1–11, IEEE Computer Society.