J Supercomput (2013) 65:1063–1075
DOI 10.1007/s11227-012-0853-z

A preliminary evaluation of OpenACC implementations

Ruymán Reyes · Iván López · Juan J. Fumero · Francisco de Sande

Published online: 31 January 2013

Abstract During the last few years, the availability of hardware accelerators, such as GPUs, has rapidly increased. However, the entry cost to GPU programming is high and requires a considerable porting and tuning effort. Some research groups and vendors have made attempts to ease the situation by defining APIs and languages that simplify these tasks. In the wake of the success of OpenMP, industria and academia are working toward defining a new standard of compiler directives to leverage the GPU programming effort. Support from vendors and similarities with the upcoming OpenMP 4.0 standard lead us to believe that OpenACC is a good alternative for developers who want to port existing codes to accelerators. In this paper, we evaluate three OpenACC implementations: two commercial implementations (PGI and CAPS) and our own research implementation, accULL, to evaluate the current status and future directions of the standard.

Keywords OpenACC · PGI · HMPP · accULL · Accelerators · GPGPU · CUDA · OpenCL · OpenMP · Compiler · Productivity

1 Introduction

The importance of developer productivity in recent years should not be underestimated [4]. Much research has been conducted on reducing the programming effort

R. Reyes · I. López · J.J. Fumero · F. de Sande (✉)
Dept. de Estadística, I. O. y Computación, La Laguna, Spain
e-mail: fsande@ull.es

R. Reyes
e-mail: rreyes@ull.es

I. López
e-mail: ilopezro@ull.es

J.J. Fumero
e-mail: jjfumeroa@ull.es

required to implement or to migrate existing codes to new architectures. The US research agency DARPA launched the High Productivity Computer Systems program [5] some years ago in an effort to create Petaflop systems. However, rather than focusing on theoretical performance, it focuses on decreasing the time-to-solution. In order to achieve this target, new tools and languages need to focus on productivity, and development time needs to be taken into account. Some new languages emerged from this initiative and have been adopted by some vendors and implementors.

In the meantime, hardware accelerators, and particularly GPU systems, have gained momentum and become ubiquitous. The adoption of accelerators has gained widespread acceptance, and the aforementioned languages and initiatives still have not integrated them properly. However, the constantly growing GPU developer community requires tools to ease GPU programming, which demands a tremendous amount of effort and painstaking work.

During the Supercomputing 2011 conference in Seattle, the new OpenACC standard for heterogeneous computing [7] was presented. Just like the introduction of OpenMP was a major boost to the popularization of shared memory HPC systems in its day, this new standard, adopted by industry leaders, lightens the coding effort required to develop heterogeneous parallel applications. In keeping with the OpenMP approach, in the OpenACC API, the programmer annotates the sequential code with compiler directives, indicating those regions of code susceptible to be offloaded to the GPU. The simplicity of the model, its ease of adoption by nonexpert users and the support received from the leading companies in this field lead us to believe that it is a long-term standard.

As a continuation of our previous research [10], we have developed `accULL` [9], 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 commonly used constructs and can be run on both CUDA and OpenCL platforms [8]. Although some vendors are expected to provide implementations of OpenACC at the end of this year, at the time of writing, the only implementations available are PGI [11] and CAPS HMPP [2]. We did not have access to a Cray platform to test their compiler suite, but we have plans to evaluate the Cray OpenACC implementation in the near future. In this paper, we compare the characteristics of the implementations of four algorithms using accULL, PGI, and CAPS HMPP.

The contributions of this paper are manifold: It represents the first non-commercial implementation of the OpenACC standard. Ours is the first implementation with support for both OpenCL and CUDA platforms. We present a runtime suitable to be decoupled from our compiler and used together with a different compiler infrastructure. We validate our approach using codes from widely available benchmarks.

The rest of the paper is organized as follows. We begin with a short description of the OpenACC API in Sect. 2. In Sect. 3, we discuss some aspects of the `accULL` implementation. Section 4 describes the experimental strategy including the testbed, various benchmarks used, and provides the experimental results. Finally, Sect. 5 details the conclusions we have been able to draw so far and ideas on future work involving the `accULL` project.

2 OpenACC

The OpenACC API offers developers a set of directives that provide simple hints to the compiler, helping it to identify areas of code suitable to be run on a hardware accelerator. Developers are freed from the task of writing specific device code details. With the information provided by the developer annotations, data movement between accelerator and host memories and data caching are managed by the compiler or runtime.

The execution model targeted by OpenACC is host-directed execution with an attached accelerator device, such as a GPU. Computationally intensive regions of the code are offloaded to the accelerator device. The devices execute *parallel regions,* which usually contain work-sharing loops, or *kernel regions* which mostly contain one or more loops that are executed as kernels in the devices.

Up to three different levels of parallelism are available on accelerators. Fine-grain parallelism is handled by multiple threads within a single execution unit. Coarse-grain parallelism is handled by running groups of multiple threads in different execution units. SIMD-like operations are also available on accelerators inside each thread or by combining a set of threads. These three levels have to be properly mapped to the hardware in order to extract the maximum performance from the accelerator, and directives are provided to allow this. The programmer also has to take into account that synchronization on some levels is not possible.

The memory is modeled assuming separate address spaces between the host and the accelerator. This means that in order to compute anything inside the device, a memory transfer has to be performed. The developer has to be aware of this situation when writing OpenACC, and cannot ignore the fact that memory bandwidth and size limits may impede the automatic offloading of code segments.

CUDA [6] exposes an SPMD programming model using a large number of threads organized into blocks. All blocks run the same program, referred to as a kernel. The blocks are automatically split into warps, consisting of 32 threads. The blocks are scheduled to the streaming multiprocessors at runtime, and each warp is executed in SIMD fashion.

The OpenACC execution model has three levels: gang, worker, and vector. The target architecture is a collection of processing elements (PEs), which can run in parallel. Each PE also has the ability to efficiently perform vector-like operations. For an NVIDIA platform, we could envision that each PE is a streaming multiprocessor, and that an OpenACC gang is a block of threads. A vector could be a form of CUDA threads. An schedule clause such as `gang vector(256)` on the top of a loop indicates that the iterations of the loop will be partitioned on 256 chunks and that each block will execute one of this chunks. However, the standard does not define a particular way of mapping the gang, worker, and vector levels onto the hardware. This situation causes that different compilers may generate different code for these clauses, thus, values that are optimal for one compiler may not be ideal for others.

The OpenACC API relies on directives implemented using the pragma mechanism to identify different region types inside the code. Three main directives can be identified: `data` to mark data regions inside the code, `kernels` to define groups of loop nests to be offloaded to the accelerator, and `parallel` which allows for more fine-grain control over the code to be offloaded.

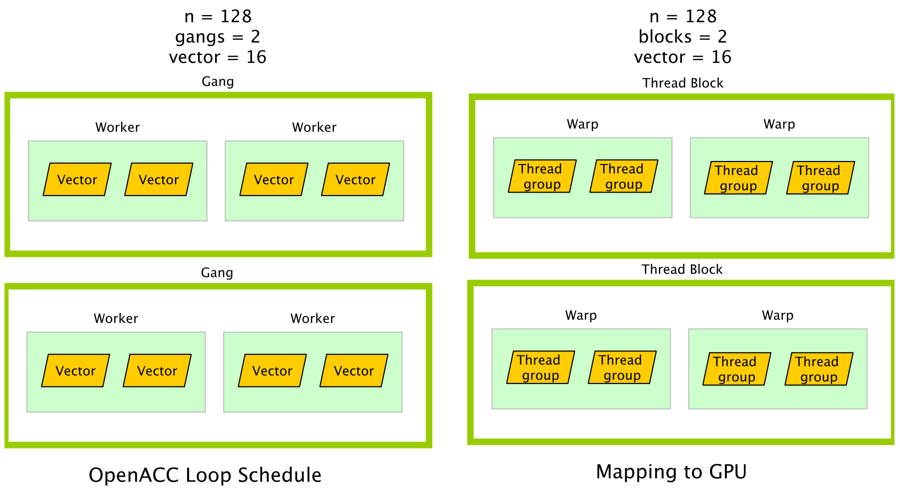

Fig. 1 Relationship between OpenACC concepts and elements in the GPU architecture

Both `parallel` and `kernels` OpenACC constructs may contain `loop` constructs. This construct precedes a loop or a loop nest and gives indications to the compiler on how the loop should be parallelized. Figure 1 relates the gangs and workers OpenACC concepts (left-hand side) with their CUDA counterparts (right-hand side).

Additional directives and API runtime calls are available. A detailed description of the OpenACC API is beyond the scope of this paper. In Sect. 4, whenever we use a clause or directive not listed here, its usage is explained. Some additional clauses, required for the algorithms implemented, are also explained in that section.

3 The `accULL` implementation

As with other compiler infrastructures, our approach to the OpenACC implementation consists of two layers comprised of a source-to-source compiler and a runtime library. The result of our compilation stage is a project tree directory hierarchy with compilation instructions that can 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 on the programmer side while providing highly skilled developers the opportunity for further optimizations.

The compiler is based on our `YaCF` research compiler framework [10], while the runtime (Frangollo) was designed from scratch. `accULL` is the combination of the `YaCF` driver and the Frangollo runtime library.

`accULL` provides support only for C99 code and `YaCF` translates the annotated C+OpenACC source code into C code with calls to the Frangollo API. The `YaCF` compiler framework has been designed to create source-to-source translations. It is intended to be a fast prototyping tool, which allows compiler developers to write portable source-to-source transformations using just a few lines of Python code. The framework has been made available as an open source tool [9].

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

Source-to-source translation injects a set of Frangollo calls within the serial code. When the control is deffered to the runtime, in addition to executing the code for the current call, it might also execute other operations (e.g., previous asynchronous operations).

Frangollo addresses two of the main issues of any OpenACC implementation: memory management and kernel execution. To handle the separate memory spaces transparently, Frangollo uses a base pointer address detection mechanism to match each host variable to its device counterpart (if any). Using this mechanism, we are able to track accesses to variables across interprocedural calls.

Memory transfers are handled on demand by Frangollo. No assumption can be made with respect to the time order of these transfers, apart from their completion before kernel execution. It is possible to use Frangollo without our compiler framework, and its software architecture based on components and interfaces would facilitate porting the runtime to other types of devices or creating new bindings for different languages.

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 the runtime are handled asynchronously, support for the `async` OpenACC clause has not been implemented yet in the CUDA component, although it is available for OpenCL. Table 1 showcases the compliance of the implementations used in this paper with the OpenACC 1.0 standard, and particularly that of `accULL`.

4 Evaluation

In order to evaluate the different OpenACC implementations, we have developed a variety of algorithms using the OpenACC API, from which we have chosen an illustrative subset to showcase in this paper. The methodology we followed has been to port existing OpenMP codes to OpenACC. This could be the most straightforward way to port code using directives, though it might not be the most efficient.

The performance comparisons shown in this section could result in some controversy. In implementing the algorithms exposed, we did so from the perspective of a nonexpert GPU developer; that is, we assumed some knowledge of the CUDA/GPU architecture, particularly the requirement of memory transfers and the kinds of algorithms and instructions that can be run on a GPU.

For these experiments, our testbed consisted of an NVIDIA C2050 GPU with 3GB of memory and an Intel Core i7 930 quad core running at 2.80 GHz with 1MB of L2 cache and 8MB of L3 cache serving as the host. We used CUDA version 4.1 with compute capability 2.0.

This kind of platform (a semipowerful desktop computer with a GPU card attached) is typical for GPU developers, and it is the potential target platform for an

Table 1 Compliance with the OpenACC 1.0 standard (directives). PGI stands for the PGI Compiler toolkit v. 12.5 and HMPP stands for HMPP v. 3.2.3

Construct	Supported by	Notes
`kernels`	PGI, HMPP, `accULL`	(`accULL`) Kernels for OpenCL and CUDA are generated for each loop inside the scope
`loop`	PGI, HMPP, `accULL`	–
`kernels loop`	PGI, HMPP, `accULL`	–
`parallel`	PGI, HMPP	–
`update`	Implemented	(`accULL`) Mixing host and device clauses in the same device does not work, they must be in separate directives
`copy`, `copyin`, `copyout`, ...	PGI, HMPP, `accULL`	Runtime handles memory transfers dynamically
`pcopy`, `pcopyin`, `pcopyout`, ...	PGI, HMPP, `accULL`	Runtime handles memory transfers when required dynamically
`async`	PGI	(`accULL`) Only for OpenCL
`deviceptr` clause	PGI	–
`host`	`accULL`	Our framework generates the right code, but we still have to solve portability issues between OpenCL and CUDA
`name`	Not in standard (`accULL` only)	Optional clause to name a particular acc region or loop and refer it from an external optimization file at compile time.
`private`, `firstprivate`	PGI (private only), HMPP (private only), `accULL`	–
`collapse`	`accULL`	Generates a 2D (or 3D) kernel
`gang`, `worker`, `vector`, `independent`	PGI, HMPP, `accULL`	Different levels of support, check Sect. 4.1

initial validation of code rewritten using OpenACC directives. It is a relatively cheap platform in comparison to a multinode cluster and it can achieve an aggregate peak theoretical performance of 478.36 GFLOPs in double precision.

The figures show the performance of the main section of the codes (i.e, those where most of the time is spent). The time measurement does not include the initialization of the CUDA devices (*acc_init()* OpenACC API function). According to the standard, device initialization should occur when calling this function. Memory copying times are included.

The results obtained with the PGI Compiler toolkit used version 12.5, which features initial OpenACC support. The version of the CAPS HMPP compiler with OpenACC support was 3.2.3; the latest available at the time of writing. The `accULL` implementation used was the latest available from our development repository at the time of writing. For the OpenMP code implementations, we also used the PGI compiler and four OpenMP threads, each thread bound to a core. In the following paragraphs, whenever we refer to the current version of each compiler, we are referring to the aforementioned versions. We are in contact with the PGI and CAPS support

```
 1 #pragma acc kernels pcopy(a[0:n*l]) pcopyin(b[0:l*m],c[0:m*n]...
 2 {
 3 #pragma acc loop private(i, j) collapse(2)
 4 for (i = 0; i < l; (i++))
 5  for (j = 0; j < n; (j++))
 6   a[(i * l) + j] = 0.0;
 7  /* Iterate over blocks */
 8 for (ii = 0; ii < l; ii += tile_size)
 9  for (jj = 0; jj < n; jj += tile_size)
10   for (kk = 0; kk < m; kk += tile_size)
11    {
12    /* Iterate inside a block */
13    #pragma acc loop private(j) gang independent
14    for (j = jj; j < min (n, jj + tile_size); (j++))
15      #pragma acc loop private(i) worker independent
16     for (i = ii; i < min (l, ii + tile_size); (i++))
17      for (k = kk; k < min (m, kk + tile_size); (k++))
18               a[(i * l) + j] += (b[(i * l) + k] * c[(k * m) + j]);
19    }
20 }
```

Listing 1 Sketch of M × M in OpenACC

teams, who have assured us that some of the missing features or the performance issues will be addressed in future releases.

For the computational experiment, we chose four different codes: a blocked Matrix multiplication and three algorithms from the Rodinia Benchmark Suite. Matrix multiplication (M × M) is a basic kernel frequently used to determine peak performance with GPU computing. The blocked matrix multiplication traditionally performs better in situations where memory locality is critical to performance [1].

The Rodinia Benchmark suite [3] comprises compute-heavy applications meant to be run in the massively parallel environment of a GPU, and covers a wide range of applications. OpenMP, CUDA, and OpenCL versions are available for most of the codes in the suite. As stated before, we started from the OpenMP versions of the codes, evaluated the granularity of the different sections, and migrated the suitable sections of the code to OpenACC annotated code. In this section, we provide computational results for three codes from the suite: A LU decomposition (LUD), a thermal simulation tool (HS) and a dynamic programming algorithm (PF).

4.1 Optimizing loop nests

Listing 1 shows a potential OpenACC implementation of the M × M code, where we chose 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 will be translated 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 have used the loop nest at line 14 to generate a kernel with the inner loop.

The `collapse` clause is not implemented either in the PGI or the CAPS HMPP current compiler versions. We thus had to choose a slightly different implementation

Table 2 FLOPs delivered by the $\mathtt{M} \times \mathtt{M}$ OpenACC implementation for each compiler using four different problem sizes

Dimension	PGI	`accULL`	HMPP
1024	10569	24555	12749
2048	15748	29601	18725
4096	19629	32704	21005
8192	20399	33283	21997

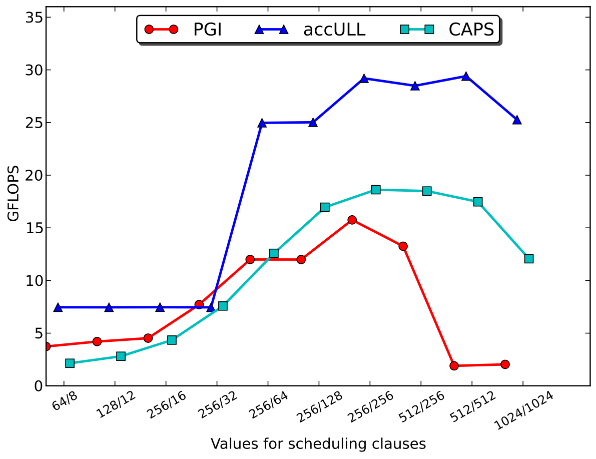

Fig. 2 Effect of varying the values of the OpenACC scheduling clauses

for them, where the loop at line 16 is also annotated with a `loop` directive. Both j-based and i-based loops feature a `gang` and `independent` to force the extraction of a 2D kernel for the GPU. To illustrate both syntaxes, Listing 1 shows both spellings (lines 3 and 13), both of them supported by `accULL`. Table 2 shows the floating point performance of the different implementations.

One of the most important aspects of CUDA tuning is an appropriate thread and kernel block selection. OpenACC provides the `gang` and `worker` clauses to enable manual tuning of kernel dimensions. However, how these clauses map to each level of the GPU architecture is not constant across each implementation.

Figure 2 showcases the effect that varying the number of `gang`, `worker,` and `vector` has on the overall performance, and how this effect varies from one compiler implementation to another. The implementation of the aforementioned clauses in each compiler is not exactly the same.

In the CAPS HMPP implementation used, support for the `gang/worker` clauses enables a strip-mining transformation of the loop where the number of gangs is used as the number of blocks, and the number of workers as the number of threads. By default, the CAPS HMPP compiler swaps loops assuming that they are written in the usual C style (where the outermost loop iterates over the rows and the innermost iterates the columns). For this particular case, (where the outermost loop iterates over the columns and the innermost over the rows), swapping the loops is not useful and degrades performance. To avoid this loop interchanging, we swapped the gangs and workers clauses (i.e, workers clause will be in line 13 and gangs clause will be in line 15 in Listing 1). Default values for gangs and workers are 256 and 32, respectively, as no automatic detection is performed. Each thread inside the block will access non-contiguous memory positions, with a stride of 32 for the rows and a stride of 256 for the columns.

In `accULL`, we do not use strip-mining to generate the kernel. The kernel we run assumes that each thread in each block will execute a single iteration, and the runtime will adjust the number of threads and blocks dynamically so as to execute the appropriate number of iterations. This imposes an upper limit on iterations since the maximum number of threads and blocks depends on the GPU architecture. Figure 2 shows the performance achieved when varying the number of threads. As clauses gangs and workers are not implemented in `accULL`, we used the environment variable to manually set a number of threads, thus, forcing the runtime to compute the appropriate number of blocks.

To maximize performance, the runtime library favors cache over shared memory if the architecture supports it (as NVIDIA Fermi and future Kepler cards do). This enhances coalescent memory accesses for each thread.

The PGI compiler optimizes loop nests in-depth by using an advanced planner, described in detail in [11]. These loop optimizations require the loops to be fully parallelizable (i.e., each iteration completely independent from the rest); thus, if the compiler is not able to ensure this condition, it will not generate the GPU kernel. To force the compiler to generate the kernel, users can use the `independent` clause, which instructs the compiler not to check dependencies and assume that iterations are independent. Iterations from the loop in line 13 of Listing 1 correspond to the X-dimension of the kernel, whereas the iterations in line 15 are distributed in the Y-dimension. Innermost k-loop is unrolled.

PGI compiler supports different combinations of `worker`, `gang,` and `vector` in different nested loops. Each `acc loop` directive may contain combinations of `worker`, `gang,` and `vector` clauses.

When encountering a loop nest (like the one in the $\mathtt{M} \times \mathtt{M}$ code), iterations for each loop are spread across each dimension similarly to what `accULL` does. Depending on the memory access pattern of the loop nest body, loops might be interchanged. The PGI compiler output informs that a two-dimensional kernel is created with each loop in a different dimension. Information about how these loops transformations are performed is showed during the compilation. Performance figures shown in Fig. 2 were obtained equally distributing the gangs and vectors across each dimension (for example, if gangs were 256, then in PGI we used 16 for the gangs in each dimension).

It is advisable to use the PGI information command line option to show detailed information on the CUDA code generation. This information enable users to improve their parallelization by solving the performance bottlenecks indicated by the compiler. Sometimes the PGI compiler does not create the GPU kernel, but the compilation finishes properly. When we displayed the information at compile time, we realized that the code was not parallelized at all due to false dependency detection among iterations. We believe that it is important to provide developers with not only a proper set of directives, but also with profile and debugging tools that ease development.

Table 3 Speed-up relative to the native CUDA implementation ($t_{CUDA}/t_{OpenACC}$) for different instance sizes of `LUD`, `HS`, and `PF`

	Psize	PGI	`accULL`	HMPP
LUD	2048	0.035	0.036	0.048
	4096	0.035	0.036	0.049
HS	64	0.077	0.725	0.08
	512	0.261	0.681	0.248
	1024	0.483	0.671	0.449
PF	1000	0.553	0.584	0.398
	4000	0.215	0.314	0.183
	8000	0.202	0.286	0.155
	12000	0.229	0.319	0.158

4.2 Overall performance

The LU decomposition has many row-wise and column-wise interdependencies and requires significant optimization to yield good parallel performance. The speed-up relative to a CUDA native version for two different problem sizes is shown in Table 3. For these problem instances, the best OpenACC implementation performs below 4 % with respect to CUDA. In other experiments, we have observed that an OpenMP version with 4 threads performs better than its OpenACC counterpart. This is due to the transfer overheads, which do not compensate for the time saved computing in the GPU.

The complex row-wise and column-wise interdependencies of the original algorithm are difficult for the compiler to identify, and the resulting kernel is not as efficient as it could be. Depending on PGI compile-time information, some references to the matrix are cached, improving performance. `accULL` does not cache anything at all, so its performance is severely degraded.

For small problem sizes, there is no benefit from offloading the code to GPU, though the advantage of doing so increases with problem size.

HotSpot (`HS`) is a thermal simulation tool used for estimating processor temperature based on an architectural floor plan and simulated power measurements. The main routine of `HS` contains two nested loops that run for a predefined number of iterations. The first loop computes the actual temperature of each position inside the chip, while the second one just updates the data with the information computed from the current iteration.

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. Loops are annotated using the `loop` directive. The compiler can then create a unique data region, where the information is copied to the device before the iteration steps are performed and then transferred back to the host when finished.

In the `accULL` implementation, it is not necessary to inline the subroutine. 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 to handle their device counterparts properly.

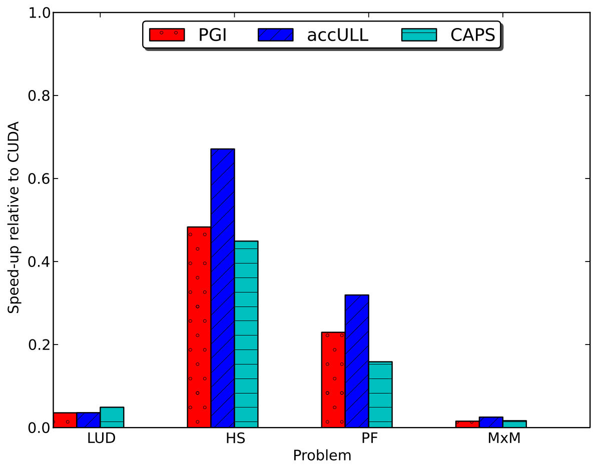

Fig. 3 Speed-up relative to CUDA ($t_{CUDA}/t_{OpenACC}$) for the largest problem instance

Although in this case it is easy to inline the routine, there might be other scenarios where the usage of `data` directives that are lexically distant from the point where the kernel is used can be beneficial from a productivity standpoint. As expected, the native CUDA version is the best for all problem sizes, and Table 3 shows the speed-up relative to CUDA for each implementation.

PathFinder (`PF`) uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, and where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, with each node picking a neighboring node in the previous row that has the smallest accumulated weight and adding its own weight to the sum.

Speed-ups relative to CUDA are shown in Table 3 for four different `PF` problem sizes. The usage of the `independent` directive allowed us to force the GPU code generation. In this case, the quality of the compiler generated kernel is not good, and more effort is needed to enhance the usage of shared memory and caches.

Figure 3 summarizes the speed-up (relative to the corresponding CUDA native counterpart) for all four codes implemented in OpenACC using the largest problem instance. The CUDA native implementation of each code from the Rodinia Benchmark was used as reference. For the `M` $\times$ `M` code, which is not in the Rodinia suite, we decided to use the CUBLAS `dgemm` implementation as a reference. Our aim is to agree a theoretical limit to the quality of the generated code. The closer the performance of the compiler-generated code gets to the performance of a hand-written optimized version, the better is the quality of the compiler.

While for the `M` $\times$ `M` and `LUD` cases all the implementations are below 5 % of the CUDA speed-up, the `accULL` implementation of `HS` reaches 67 % of that. Although it is not possible to match the performance of the native CUDA implementation, OpenACC can be used to provide a major performance boost with a low development effort. These results represent an early evaluation of the first implementations of the OpenACC standard and, therefore, they can be considered a clue about what we can get in the near future when compilers gain in maturity.

5 Conclusions and future work

In this paper, we have evaluated three different implementations of the OpenACC standard. The PGI compiler featuring OpenACC support shows the current status of the implementation by the industry. PGI focus on stability and correctness rather than on implementing frills. Although it covers a large subset of the OpenACC standard, not all the specification is implemented and some features are missing. The CAPS HMPP compiler covers an important subset of the OpenACC standard, offering support for loop partitioning clauses, but the performance of the current release has to be improved. The ability to see the generated kernel and to manually modify it, similar to our `accULL` implementation, allows advanced developers to further improve the kernels. It is also independent from the host compiler, thus making it possible to combine it with GNU, Intel, or other compilers, whereas PGI is an integrated solution for the entire platform. Although it is true that it is possible to see the intermediate code generated by the PGI compiler, it is hard to read as it has been heavily optimized, and barely resembles the original source.

On the academia side, `accULL` is a research-oriented implementation that prioritizes flexible design above overall stability. It aims to provide researchers and early adopters a platform for exploring the potential of directive-based programming. Although not explored in this paper, our runtime library also supports OpenCL platforms. `accULL` can also be used to generate code for ATI GPU cards or CPU OpenCL implementations.

Adding new directives or features to the `accULL` runtime is relatively easy. Nevertheless, it is obvious that the stability and correctness cannot match those of its industrial counterparts.

Directive-based GPU programming eases code maintenance and improves portability. The source code is clean from nonstandard CUDA language extensions and incremental parallelization eases debugging. Although the speed-up achieved does not match those obtained with a native CUDA implementation, the programming effort required is significatively lower than manually writing CUDA kernels. The availability of orphaned directives and the possibility of interacting with subroutines written in the native language of the accelerator greatly enhances the capabilities of this language enabling an incremental parallelization approach. The OpenMP committee claims that in the future it will integrate accelerators into its programming model, and there are several proposals to do that. However, until all the parties involved agree on a definitive standard for accelerators, the need for better tools for programming accelerators can be satisfied with OpenACC. Members of the OpenACC committee also belong to the OpenMP committee and, therefore, feedback among them is ensured.

At the programmer level, it is worth noting that, although OpenACC and OpenMP have similarities, a major conceptual difference between them exists: *Data management is independent from parallelism.* A developer used to working with OpenMP could attempt to parallelize loops separately by using an `acc parallel loop` construct for each one, similar to how she would use an OpenMP `omp parallel for`. This would force the compiler to create separate data regions for each loop, meaning the data would be copied into and out of the device when entering and exiting the loops. Since these loops are frequently executed in an iterative statement, data would be transferred repeatedly into and out of the device.

Ongoing research is exploring the usage of remote accelerators directly from the runtime through the use of MPI or other technologies. Hybrid MPI and OpenACC codes are also being used. Some optimizations are being implemented into the runtime to improve its performance when used in combination with MPI. Combining OpenACC and OpenMP directives is also in our plans for the near future. Implementing the source-to-source transformations into a different compiler is also worth exploring, and would notably increase the maturity of the `accULL` project.

Acknowledgements 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, ACIISI (contract PI2008/285).

References

1. Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2001) An updated set of basic linear algebra subprograms (BLAS). ACM Trans Math Softw 28(2):135–151
2. Bodin F, Bihan S (2009) Heterogeneous multicore parallel programming for graphics processing units. Sci Program 17(4):325–336. http://dl.acm.org/citation.cfm?id=1662626.1662632
3. Che S, Sheaffer JW, Boyer M, Szafaryn LG, Wang L, Skadron K (2010) 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), IISWC '10. IEEE Computer Society, Washington, pp 1–11
4. Faulk S, Porter A, Gustafson J, Tichy W, Johnson P, Votta L (2004) Measuring high performance computing productivity. Int J High Perform Comput Appl 18(4):459–473
5. Lusk E, Yelick K (2007) Languages for high-productivity computing: the DARPA HPCS language project. Parallel Process Lett 17(1):89–102
6. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. Queue 6(2):40–53
7. OpenACC: OpenACC directives for accelerators (2011). http://www.openacc-standard.org (Online; Last accessed October 2012)
8. Reyes R, López-Rodríguez I, Fumero JJ, de Sande F (2012) accULL: an OpenACC implementation with CUDA and OpenCL support. In: Kaklamanis C, Papatheodorou TS, Spirakis PG (eds) Euro-Par 2012 parallel processing—18th international conference, Euro-Par 2012, Rhodes Island, Greece, August 27–31, 2012. Lecture notes in computer science, vol 7484. Springer, Rhodes Island, pp 871–882
9. Reyes R, de Sande F (2012) `accULL` project home page. http://cap.pcg.ull.es/accULL (Online; Last accessed November 2012)
10. Reyes R, de Sande F (2012) Optimization strategies in different CUDA architectures using llCoMP. Microprocess Microsyst 36(2):78–87
11. Wolfe M (2010) Implementing the PGI Accelerator model. In: Proceedings of the 3rd workshop on general-purpose computation on graphics processing units, GPGPU '10. ACM, New York, pp 43–50