**accULL: An User-directed Approach to Heterogeneous Programming**

Ruymán Reyes, Ivan López, Juan J. Fumero and Francisco de Sande

*Dept. de EIO y Computación*  
*Universidad de La Laguna,*  
*38271–La Laguna, Spain*  
*Email: {reyes, ilopezro, jfumeroa, fsande}@ull.es*

**Abstract**—The world of HPC is undergoing rapid changes and computer architectures capable to achieve high performance have broadened. The irruption in the scene of computational accelerators, like GPUs, is increasing performance while maintaining low cost per GFLOP, thus expanding the popularity of HPC. However, it is still difficult to exploit the new complex processor hierarchies. To adapt the message passing model to program heterogeneous CPU+GPUs environments is not an easy task. Furthermore, message passing does not seem to be the best choice from the programmer point of view. Traditional shared memory approaches like OpenMP are interesting to ease the popularization of these platforms, but the fact is that GPU devices are connected to the CPU through a bus and have a separate memory space. We need to find a way to deal with this issue at programming language level, otherwise, developers will spend most of their time focusing on low-level code details instead of algorithmic enhancements. The recent advent of the OpenACC standard for heterogeneous computing represents an effort in the direction of leveraging the development effort. This initiative, combined with future releases of the OpenMP standard, will converge into a fully heterogeneous framework that will cope the programming requirements of future computer architectures. In this work we present preliminary results of accULL, a novel implementation of the OpenACC standard, based on a source-to-source compiler and a runtime library. To our knowledge, our approach is the first providing support for both OpenCL and CUDA platforms under this new standard.

**Keywords**—OpenACC, Accelerators, GPGPU, CUDA, OpenCL, OpenMP, compiler, productivity, code performance

I. INTRODUCTION

The publication in the last SC2011 of the OpenACC standard [1] for high-level programming of heterogeneous host+accelerator applications represents a milestone to increase the programmability of these architectures. It is easy to find scientific applications that are well suited to benefit from the kind of parallelism exploited by such architectures: image processing, particle physics or chemical simulations are just some of the areas of application of this technology. The combination of a conventional computer connected to one or more accelerators has become a system able to achieve levels of performance that until now were only possible with the help of large supercomputers.

From our point of view, the weakness of these hybrid architectures has been up to now its lack of programmability. The most widely used options so far have been CUDA and OpenCL. CUDA is the most mature and extended approach to GPU programming, being its lack of portability its main drawback. Despite of being partially simple to build a program in CUDA, it is hard to achieve a good performance rate, requiring huge coding and optimization efforts to obtain the maximum performance of the architecture. The OpenCL standard represents an effort to create a common programming interface for heterogeneous devices, which many manufacturers have joined. However, its programming model is not simple and its use is still limited.

Following the successful case of OpenMP, OpenACC proposes an approach based on compiler directives by which the programmer marks those parts of the sequential code (mainly loops) that can be offloaded from a host CPU to an attached accelerator. The simplicity of the OpenACC 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 natural continuation of our previous work in this field, [2], [3], in this paper we present our preliminary results of an implementation of the OpenACC standard that we have called accULL. We offer support for most of the OpenACC constructs, and we are able to run in both CUDA and OpenCL platforms. User can select the desired platform using the appropriate environment variable, conforming to the standard.

The most relevant contributions of this work are: (a) Ours is one of the first non-commercial implementations of the OpenACC standard. (b) To our knowledge, accULL is the first implementation with support for both OpenCL and CUDA platforms. (c) We present Frangollo, a runtime library suitable to be decoupled from our YaCF [3] compiler and combined with a different compiler infrastructure. (d) We validate our approach using codes from widely available benchmarks and using both GPU and CPU devices.

The remainder 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. The implementation of our compilation framework is presented in Section III. In Section IV we expose the key ideas behind our approach and present computational results for four well-known algorithms. Finally, Section V offers some concluding remarks and hints on future research.
II. RELATED WORK

An overwhelming variety of works are available in the field of increasing the programming productivity for hardware accelerators. This work tackles two main areas of research: compiler technology and memory management.

In the area of compiler technology, the Cetus project [4] from Purdue University 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 [5] 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. Our 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 Frangollo runtime.

The Mercurium [6] 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.

Some attempts to clarify the situation of the separate address spaces between host and device have been made by several authors. The aforementioned OmpSs [7] is a programming model that provides a framework to develop parallel applications for cluster environments with heterogeneous architectures. Based on OpenMP and StarSs, it offers a set of compiler directives that can be used to annotate a sequential code. Additional features have been added to support the use of accelerators like GPUs. However, as far as we know, it is up to the user to provide optimized versions for each platform, no automatic kernel extracting/generation is done.

The hiCuda project [8] defined a directive-based language for programming only NVIDIA GPUs, and attempts to ease GPU development. They also designed and implemented a source-to-source compiler for its language. However, their annotations are low-level and still preserve the CUDA programming model.

With respect to the memory-management area, the GMAC project designed and implemented the ADSM (Asymmetric Distributed Memory Model) [9] on top of nodes with one or more GPU devices. ADSM is a data-centric programming model for heterogeneous systems. In a similar way to Frangollo, it offers programmers a shared address space between general purpose CPUs and accelerators. However, in ADSM CPUs can access data hosted by accelerators, whereas our model allows programmers to use CPU data inside accelerators. Their API is similar to CUDA and does not offer a high-level annotation approach similar to OpenACC.

The PGI Accelerator model [10] is a high-level programming model for accelerators, such as GPUs, similar in design and scope to the widely-used OpenMP directive approach. This compiler features a Planner analysis module to map loop parallelism into the different parallelism levels available in accelerator devices. Its support is currently limited to NVIDIA GPU devices. PGI has recently announced that their compilers will support also the OpenACC standard.

The CAPS HMPP [11] 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 will be run in a hardware accelerator. These codelets can either be handwritten for a specific architecture or be generated by some code generator. CAPS HMPP is also part of the OpenACC standard and they have announced an upcoming release of their compiler with support for this standard.

III. THE IMPLEMENTATION

Our approach is a two-layer based implementation 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. However, instead of generating a final binary file, 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, while the runtime (Frangollo) has been designed from scratch. We have named accULL to the combination of our compiler driver and the runtime.

YaCF translates the annotated C+OpenACC source code into a C code with calls to the Frangollo API. The YaCF compiler framework [3] 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 in just a few lines of Python code. The framework is available as an open source tool [12]. On top of the YaCF infrastructure, we have built a set of Python modules, capable of extracting the kernel code from the annotated source and replace it with the appropriate runtime calls. Both OpenCL and CUDA kernels are extracted.

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 Frangollo runtime, who 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 major issues of any OpenACC implementation: memory management and kernel execution.

It is important to take into account that nowadays compute accelerator devices do not share the host processor address space. Therefore, it is critical to transparently handle the existence of several instances of an user variable on different devices. To address this situation, our runtime uses a base pointer address detection mechanism to match each host variable to its device counterpart. Using this mechanism, we are able to track accesses to variables across interprocedural calls. The only exception to this behaviour is the implementation of the acc host construct, which, following the standard, requires a device specific pointer, and the deviceptr clause, which is not currently implemented.

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

Frangollo is divided into separate pluggable components. A common component serves as an abstract interface to all other 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.

YaCF supports most of the syntactic constructs in the OpenAcc 1.1 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. Table I describes some of the constructs implemented in accULL.

Being an initial release, our approach allows translating to CUDA/OpenCL a comprehensive set of codes properly annotated, as we show in Section IV. Nevertheless, at this point, we do not aim to create a full commercial implementation of the standard, but a research tool to demonstrate its potential.

### IV. Evaluation

To evaluate our accULL OpenACC implementation, we have annotated four different applications and tested them on the next different platforms:

- **M1**: Desktop computer with an Intel Core i7 930 processor (2.80 GHz), with 1MB of L2 cache, 8MB of L3 cache, shared by the four cores. The system has 4 GB RAM and two GPU devices attached, a Tesla C1060 with 3GB memory (M1a) and a Tesla C2050 with 4GB memory (M1b). Accelerator platform is CUDA 4.0.
- **M2**: A cluster node consisting on two quad core Intel Xeon E5410 (2.25GHz) processors, 24 GB memory and an attached Fermi C2050 card with 448 multiprocessors and 4 GB memory. Accelerator platform is CUDA 4.0.
- **M3**: A second cluster node. M3 is a shared memory system, with 4 Intel Xeon E7 4850 CPU, with 2.50MB L2 cache and 24MB L3 cache (for all its 10 cores).

<table>
<thead>
<tr>
<th>Construct</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernels</td>
<td>Implemented</td>
<td>Kernels for OpenCL and CUDA are generated for each loop inside the scope</td>
</tr>
<tr>
<td>loop</td>
<td>Implemented</td>
<td>Indicates a potential accelerator kernel. Some restrictions apply (e.g., no external definitions)</td>
</tr>
<tr>
<td>kernels loop</td>
<td>Implemented</td>
<td>A kernel will be extracted. Dependency analysis is used to check and allocate RO variables if possible.</td>
</tr>
<tr>
<td>parallel</td>
<td>Not implemented</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>Implemented</td>
<td>Mixing host and device clauses in the same construct does not work, they must be separated</td>
</tr>
<tr>
<td>copy, copyin, copyout,...</td>
<td>Implemented</td>
<td>Runtime dynamically handles memory transfers</td>
</tr>
<tr>
<td>pcopy, pcopyin, pcopyout,...</td>
<td>Implemented</td>
<td>Runtime dynamically handles memory transfers when required</td>
</tr>
<tr>
<td>async</td>
<td>Not implemented</td>
<td>-</td>
</tr>
<tr>
<td>host</td>
<td>Partially implemented</td>
<td>Optional clause to name a particular acc region or loop and refer it from an external optimization file at compile time.</td>
</tr>
<tr>
<td>name</td>
<td>Not in standard</td>
<td>-</td>
</tr>
</tbody>
</table>

Table I: Compliance with the OpenACC 1.1 standard (constructs)
6GB of memory are available per core. In this case, the accelerator platform is Intel OpenCL SDK 1.5, which runs on the CPU.

With platforms M1a / M1b we mimic the usual scenario of an OpenACC developer: A slightly experienced user interested in improving the performance a scientific code can purchase a new GPU card and plug it into her desktop computer. It is a relatively cheap platform as opposed to a multinode cluster and could achieve a combined peak theoretical performance of 478.36 GFLOPs in double performance. This kind of user might have some insight in programming and even in GPU computing, but she is not an expert. Starting with her own serial code and using an OpenACC compliant compiler, this user will take advantage of the GPUs without investing excessive time in low level programming.

M2 is a node of a common multinode cluster. Nowadays clusters are composed by multicore processors and GPU devices, thus it is possible to take advantage of OpenACC in these platforms. Moreover, our implementation integrates seamlessly with MPI programs, and can be used to take advantage of the attached GPU devices without additional effort. These kind of compute nodes have higher acquisition and maintenance costs than a desktop computer like M1.

M3 is a shared memory system that showcases an alternative use of OpenCL. Nowadays shared memory systems feature several CPUs with several cores on each. These cores also contain vector processing units that require particular compiler support (or a deep understanding of these technologies) to unleash their potential. There are implementations of OpenCL, like the Intel OpenCL SDK or the AMD APP SDK, targeting these shared memory machines. Writing algorithms in OpenCL is not an effortless task, but it allows a better mapping of hardware resources and improve thread scheduling. Using CPU-targeted OpenCL platforms along with OpenACC represents an interesting alternative to traditional OpenMP programming that we will explore in different examples. Shared memory systems with a large number of cores are still expensive in comparison to desktop machines.

Although a detailed investigation of performance per Euro or investment return is out of the scope of this contribution, when analyzing performance data, it is important to notice that, in terms of acquisition costs, M1 is two times cheaper than M2, and M3 is three times more expensive than M2. Giving the present economic climate, it is worth to take into account the performance delivered by any investment. This is also important not only in choosing a hardware platform, but also when acquiring tools. Although at the time of writing we could not access a commercial OpenACC implementation to compare the results of our implementation, it is worth to note that the expected costs of these tools could exceed the cost of M1 platform.

We have made our best effort to compare our codes with a native CUDA or OpenCL implementation whenever possible.

A. Blocked Matrix Multiplication

Matrix multiplication (MxM) is a basic kernel frequently used to showcase the peak performance of GPU computing. In this Section, we focus on a blocked matrix multiplication algorithm, similar to that used in the widely-known BLAS routines [13]. The recommended way to implement a matrix product in a source code is to use a BLAS implementation tuned for the machine. Our choice of the blocked MxM has been made bearing in mind the clarity of explanation rather than a recommendation for implementation.

Listing 1 shows a potential OpenACC implementation of the blocked MxM. 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 will translate 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 (that iterates inside a matrix block). Despite of accULL issuing a kernel call for each matrix block, it does not transfer any data for each call as the whole matrix has been already transferred to the device when the region was created.

```
Listing 1: Sketch of MxM in OpenACC
1 #pragma acc kernels name("mkm") copy(a[M*N])
   copyin(b[L*M],c[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   for (k = 0; k < M; k++)
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    /* Iterate inside a block */
12    #pragma acc loop collapse(2) private(i, j, k)
13    for (i=i; i < min(L, ii+tile_size); i++)
14    for (j=j; j < min(N, jj+tile_size); j++)
15    for (k=k; k < min(M, kk+tile_size); k++)
16    a[i*L+j] += (b[i*L+k] * c[k*N+j]);
17  }
18 }
```

YaCF features a loop optimization module, which uses data dependency analysis to enable different optimizations. In this example, the index expression for the array access inside the loop a[i*L+j] is independent from the innermost loop. As it its shown in Listing 2, the loop analysis module can detect this situation and replace the array access with a private variable. When running on GPU devices, this variable is mapped to a register, thus, greatly leveraging the number
of memory accesses and increasing performance. Figure 1 shows the impact in the kernel performance of this loop invariant optimization.

Listing 2: Extracting the array access

```c
1 tmp = a[i * L + j];
2 for (k = kk; k < min(m, kk+tile_size); k++)
3 tmp += (b[i * L + k] * c[k * m + j]);
4 a[i * L + j] = tmp;
```

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 feeded with the compute intensity information extracted by the YaCF driver. The OpenCL component delegates the local and global partitionings to the underlying platform implementation.

In Figure 2 we present performance figures using the Intel OpenCL implementation in M3, and compare them with its OpenMP counterpart. It seems that, in some cases and depending on the problem size, OpenCL outperforms the OpenMP implementation. This make us believe that using Frangollo with CPU implementations is an approach worth exploring.

B. Rodinia

The Rodinia Benchmark suite [14] 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. To demonstrate the completeness of our approach, in this Section we present performance figures for three benchmarks taken from this suite. Rodinia does not currently feature OpenACC versions of their codes, but it is easy to produce an OpenACC implementation from the OpenMP version.

HotSpot (HS) is a thermal simulation tool [15] 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 series 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.

The main routine in HS, shown in Listing 3 contains two nested loops that runs 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.

Listing 3: Sketch of HS

```c
1 for (i = 0; i < num_iterations; i++)
2 { /* Compute current temperatures */
3   for (r = 0; r < row; r++)
4     for (c = 0; c < col; c++) {
5       ...
6     }
7   /* Update */
8   for (r = 0; r < row; r++)
9     for (c = 0; c < col; c++) {
10       ...
11     }
12 }
```

It is worth noting that, although OpenACC and OpenMP have great similarities, a major conceptual difference between them exists: data regions are decoupled from parallel regions. A developer used to work with OpenMP
could attempt to parallelize both loops separately, using a parallel loop construct for each one. 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.

Taking advantage of the OpenACC semantics it is easy to avoid such an unnecessary amount of data transfers. The kernels directive defines a data region containing a set of loops that will be executed on the accelerator device. Loops are annotated using the loop construct. In this case, the kernels directive can be written outside the outermost loop, while the inner loops are marked with the loop directive. This allows accULL to create a unique data region, where the information is copied to the device before performing the iteration steps, and transferred back to the host when finished. Computations are performed only in the device and unnecessary transfers are avoided.

A closer look to the first loop shows that it features a set of if/else statements to check whether if a particular point is in the halo or not. Using if/else inside accelerator kernels is discouraged, as it serializes the execution of the warp (set of threads executed simultaneously by the multiprocessor accelerator) [16]. When a warp runs a kernel containing if/else statements, each condition is evaluated by all threads simultaneously. However, only those threads matching the condition will execute the corresponding statement, while the remaining threads are waiting.

This behaviour represents a major performance bottleneck, as the condition is checked for each iteration. YaCF features a divergence-avoider transformation, that replaces a

Figure 3: Performance comparisons of SRAD and NW in M1b and M3 platforms. (a, b, c) show speedup over the OpenMP implementation while (d) presents efficiency relative to the Intel OpenMP implementation.
statement like that shown in Listing 4 with a less divergent operation as that shown in Listing 5.

Listing 4: Divergent if/else statements

```c
if (condition1) {
    delta = complex_expression_1
} else if (condition2) {
    delta = complex_expression_2
} else {
    delta = complex_expression_4
}
```

It still features a divergent set of condition checking, but threads are diverging less time. Although it might be considered as counterintuitive, the cost of repeating the computation in all threads is, depending on the complexity of the complex_expression_1..4), less than the cost of waiting for each thread to compute its private value. Performance figures applying this optimization to the first loop are shown in Figure 4. In (a) accULL show speedup close to native implementations when using GPU devices. In Figure 4 (b) best results were obtained using the Intel OpenMP implementation, and mapping processors to physical cores.

Listing 5: Leveraging thread divergence

```c
delta__1 = complex_expression_1
delta__2 = complex_expression_2
delta__3 = complex_expression_3
delta = (condition1) ? delta__1:
    (condition2) ? delta__2:
    delta__3;
```

The performance of the less-divergent implementation of HS is better in CPU devices, but it does not perform well in GPU devices. The set of complex_expression_1..N from this particular piece of code requires accessing to several memory positions, producing uncoalesced accesses to global memory. When running on CPU devices memory bandwidth is not as critical as in GPU devices, and the less-divergent version of the code benefits from better autovectorization through the OpenCL platform compiler. However, in GPUs lower memory bandwidth do not compensate the less divergent code.

Several other codes have been ported to OpenACC. In this work, we present results for the SRAD and NW benchmarks, whose results for M1 and M3 platforms are exposed in Figure 3. In (a) Native OpenCL outperforms the remaining implementations, potentially due to the usage of pinned memory. accULL running with CUDA performs closer to the native version. The OpenCL runtime requires more tuning effort for this particular situation. In (b) Using OpenACC directives and generating code to the OpenCL runtime on CPU we extract more performance than using OpenMP. In (c), speedup over OpenMP for native versions is higher than our generated code. However, we obtain approximately a 2x speedup. In (d), the best results were obtained using Intel OpenMP implementation with appropriate processor binding options. Although accULL does not match the OpenMP performance, we are not far from that of a native OpenCL.

V. CONCLUSIONS AND FUTURE WORK

As we demonstrate in Section IV, the current status of the accULL implementation meets the requirements of a non-expert developer, and will improve the time to solution by decreasing the overall development effort. There are several
implementations of the OpenACC standard. However, they are commercial solutions and, to our knowledge, do not currently feature support for OpenCL platforms. Our compiler implementation of the OpenACC standard can be used as a fast-prototyping tool to explore optimizations and alternative runtime environments. Frangollo 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.

We believe that accULL is a good choice for non-expert users to exploit GPUs in HPC. The results we have shown in this work represent a clear improvement in the way of increase programmability of heterogeneous architectures. These preliminary results make us believe that our approach is worth to be explored more deeply.

Work in progress within the framework of the accULL project includes integration with a commercial compiler, taking advantage of pre-existing auto-vectorization support and improvement of the support for memory allocation. We have work in progress to implement two dimensional arrays as cuadrMatrix or OCLImages to improve non-contiguous memory access. Also we are exploring different alternatives for integration with MPI. The OpenCL component is able to share pointers with MPI buffers, and we would like to use the new features of the latest CUDA release, like GPU Direct, in the same direction. We believe that there are still plenty of opportunities to improve performance from this point, as this work settles the foundations of a dynamic and detachable compiler+runtime infrastructure.

ACKNOWLEDGMENTS

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


