accULL: An OpenACC Implementation with CUDA and OpenCL Support

Ruymán Reyes, Iván López-Rodríguez, Juan J. Fumero, Francisco de Sande

Dept. de E. I. O. y Computación
Universidad de La Laguna, 38271–La Laguna, Spain
{rreyes,ilopezro,jfumeroa,fsande}@ull.es

Abstract. The irruption in the HPC scene of hardware accelerators, like GPUs, has made available unprecedented performance to developers. However, even expert developers may not be ready to exploit the new complex processor hierarchies. We need to find a way to leverage the programming effort in these devices at programming language level, otherwise, developers will spend most of their time focusing on device-specific code instead of implementing algorithmic enhancements. The recent advent of the OpenACC standard for heterogeneous computing represents an effort in this direction. 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 accULL, a novel implementation of the OpenACC standard, based on the combination of 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.

1 Introduction

The widespread use of graphics accelerators for general purpose computing has leveraged the entry cost of high performance computer systems. A modest commodity computer in combination with a graphic card constitutes a powerful tool which empowers users to solve problems with a significant size so far unavailable without the aid of large scale computers.

Despite the improvements achieved in the hardware field, there is still a lack of parallel problem solving environments that can help scientists to use easily and efficiently these hybrid architectures. From our point of view, at this moment, efforts have to be directed towards the development of high level abstractions of these heterogeneous environments. This will allow more users to take advantage of these architectures without the need of detailed hardware knowledge.

CUDA is the most mature and extended approach to GPU programming, although currently only supports Nvidia devices. Despite of being partially simple to build a code using this tool, achieving good performance rate usually requires a noticeable coding and optimisation effort.

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.

The presentation during Seattle SC2011 of the new OpenACC standard for heterogeneous computing [4] clearly represents a major effort in the aforementioned direction of leveraging the development effort. Following the OpenMP approach, in the OpenACC API, the programmer annotates its 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.

Prior to OpenACC, the PGI Accelerator model [6] proposed a high-level programming model for accelerators, such as GPUs, similar in design and scope to the widely-used OpenMP directive approach. Also, the CAPS HMPP [1] toolkit is a set is a set of compiler directives, tools and software runtime that supports parallel programming in C and Fortran. Both PGI and CAPS are founders of the OpenACC standard, and have recently announced versions of their tools compliant to the standard.

As a continuation of our recent years work [5], here we present a first release of our implementation of the OpenACC standard. We offer 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 addition, we present results using both CPU and GPU OpenCL platforms. User can select the desired platform using the appropriate environment variable, conforming to the standard.

The contributions of this work are manifold: (a) It represents one of the first non-commercial implementation of the OpenACC standard. (b) This is the first implementation, as far as we know, with support for both OpenCL and CUDA platforms. (c) We present a runtime suitable to be decoupled from our compiler and used together with a different compiler infrastructure. (d) We validate our approach using codes from widely available benchmarks and using both GPU and CPU devices.

The rest of the paper is organized as follows. Section 2 discusses the implementation of our compilation framework. In Section 3 we expose the key ideas behind our approach. We guide our explanations through the use of a code example. Also in Section 3 we present computational results for the guiding example and three additional well-known algorithms. Finally, Section 4 includes the conclusions we have been able to achieve so far and ideas about future work regarding this framework.

2 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. However, instead of generating a final binary file, the result of our compilation stage is a 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 [5] 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 a few lines of Python code. The framework is available as an open source tool [1]. 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 generated from the extracted block statements

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, by specification definition, requires a device specific pointer, and the `deviceptr` clauses, which are 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.

[1] `http://code.google.com/p/yacf/`

Table 1: Compliance with the OpenACC 1.1 standard (constructs)

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

Frangollo is divided into 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. Frangollo's API provide high level entry points to the runtime, independent from the destination platform. The compiler can emit these generic runtime calls, like `registerVar`, `createContext` or `launchKernel`, and Frangollo can handle the rest of the work, i.e, choosing the appropriate platform, load the kernel file, estimate the best grid configuration, copyin/out the result and even perform reductions over the selected variables.

`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 1 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 3. 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.

3 Evaluation

To evaluate our `accULL` OpenACC implementation, we have used codes from different benchmarks and tested them on the next four different platforms:

- `M1`: Desktop computer with an Intel(R) Core(TM) i7 930 processor (2.80 GHz), with 1MB of L2 cache, 8MB of L3 cache, shared by the four cores. The machine has 4 GB RAM and two GPU devices are attached:
  - `M1a`: Tesla C1060 with 240 multiprocessors, 3 GB memory
    - Bandwidth from host to device: 2.40 GB/s
    - Bandwidth from device to host: 2.29 GB/s
  - `M1b`: Tesla C2050 with 448 multiprocessors, 4 GB memory
    - Bandwidth from host to device: 2.35 GB/s
    - Bandwidth from device to host: 2.20 GB/s
- `M2`: One 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.
- `M3`: Laptop computer with one Intel(R) Core(TM) i3 CPU M 350, using Hyperthreading to enable four virtual processors, 3GB RAM, and an integrated Nvidia OPTIMUS graphic card.
- `M4`: A second cluster node. `M4` is a shared memory system, with 4 Intel(R) Xeon(R) E7 4850 CPU, with 2.50MB L2 cache and 24MB L3 cache (for all its 10 cores). 6GB of memory are available per core.

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 in 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 478.36 GFLOPs in double performance (77.76 GFLOPs from Tesla C1060 + 345.6 GFLOPs from Tesla C2060 + 55 GFLOPs from main processor). This kind of user might have some insight in programming and even in GPU computing, but she is not an expert. Starting with his 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.

`M3` represents a usual nowadays medium-end laptop computer. It uses reduced versions of desktop GPUs that support GPGPU computing. Laptop computers are not relevant in terms of HPC, however, `accULL` is suitable for other environments wherever GPU computing could be beneficial.

`M4` is a shared memory system that showcases an alternative case use of OpenCL. Nowadays shared memory machines 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. Our runtime detects whether the platform is a GPU or a CPU and uses the appropriate variable register and copy method, without requiring to know this parameter at compile time.

3.1 Molecular Dynamic simulation

```
1  int main(...) {
2    ...
3    // Initial energy calculation
4    compute(position, velocity, mass, force, &potential, &kinetic);
5    ...
6    // (S) Simulation
7    for (i = 0; i < NSTEPS; i++) {
8      compute(position, velocity, mass, force, &potential, &kinetic);
9      printf(..., potential, kinetic);
10     update (position, velocity, mass, force, &potential, &kinetic);
11   }
12   ...
13 }
14 void compute(...) {
15     // (C) Compute forces
16     for (...) {
17     }
18 }
19 void update(...) {
20     // (U) Update velocity/position
21     for (...)
22      for (...) {
23          ...
24     }
25 }
```

Listing 1.1: Sketch of MD simulation in OpenACC

Given positions, masses and velocities of `np` particles, the pseudo code shown in Listing 1.1 computes the energy of the system and the forces on each particle. The code is a C implementation of a simple Molecular Dynamics (MD) simulation. It employs an iterative numerical procedure to obtain an approximate solution whose accuracy is determined by the time step of the simulation. Particles are represented by three three-dimensional double precision matrices: Position, Velocity and Force (parameters). Rows of each matrix represent a particle, whereas columns represent a dimension. For example, the coordinate $\{3, 1\}$ contains the parameter value for the particle number three in dimension one.

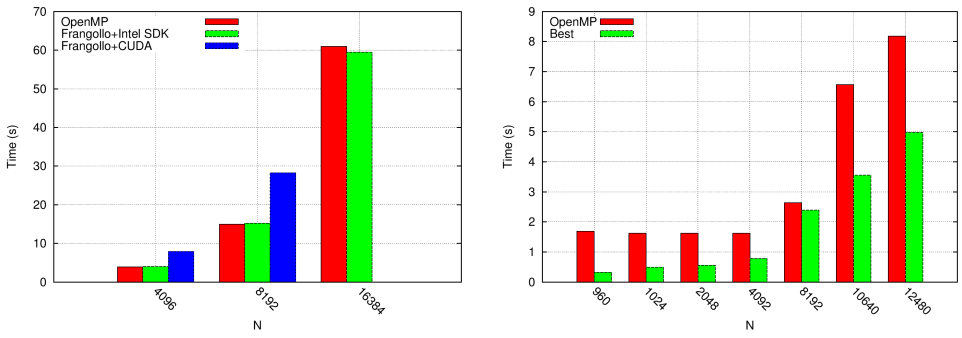
After an initial forces computation, on each simulation step, the algorithm performs two basic operations: *compute* (C) and *update* (U). C operation consists of several nested loops computing the forces for each position. An external loop iterates over all particles computing its forces in the current simulation step. This requires computing the distance among all other particles, hence accessing

Version	Time transfer in	Time transfer out	Kernel Time	Total time	% Speedup
Naive Approach	$< 0.02s$	$0.0127834s$	$5.69122s$	$5.791388s$	-
Using a data clause	$< 0.02s$	$0.0121639s$	$5.63023s$	$5.729317s$	1%
Splitting C loops	$< 0.02s$	$0.0120155s$	$3.87633s$	4.046456	30.1%

Table 2: Time per phase and speedup for each incremental optimization over the naive implementation, as measured in `M3` using the Intel OpenCL SDK over the CPU. In this situation, using the data clause does not represent and important performance benefit, due to the fact that (1) Frangollo OpenCL implementation uses the native pointer whenever possible and (2) Intel OpenCL features lower initialization time than GPU approaches

the `position` matrix, and computes the total potential and kinetic energy of the system, which requires access to the `velocity` matrix. In terms of the data access pattern, the code is highly un-coalesced, requiring several non-contiguous loads to compute each particle. In addition, it features several costly double precision operations (*sqrt*, *sin* and *cos*) which traditionally perform badly on GPU devices. The U operation is simply a for loop that runs over the particles, updating their positions, velocities and accelerations. C is more compute–intensive than U.

A naive porting of this code using OpenACC directives would consist into adding the `kernels loop` construct to the top of the outermost loops in both routines (before C and U in Listing 1.1), and writing the appropriate copy clauses to indicate variable directionality related to the loop.



(a) MD in `M3` **(b)** MD in `M4`

Fig. 1: (a) Performance comparison of the (best) OpenACC implementation vs. OpenMP in `M3`. OpenCL uses Intel OpenCL SDK over the CPU. CUDA version uses the CUDA 4.0 driver using the laptop's GPU. The largest problem size could not be executed in GPU due to the lack of graphics memory. It is worth to note the flexibility of the runtime, and how it is able to match OpenMP performance despite of the runtime overhead. (b) Performance comparison of the (best) OpenACC implementation with OpenMP in `M4`. OpenCL uses Intel OpenCL SDK over the CPU. OpenMP implementation was provided by GCC version 4.4.5.

In this case, our compiler would extract the kernel from the loops and inject the appropriate runtime calls. Each time these functions are executed, memory transfers between host and GPU will take place. Transfer time between host and GPU could represent a significant percentage of the total time. Developers should take into account that the outstanding performance achieved by accelerator devices can be easily hidden by an excessive memory transfer time. Usage of profiling tools is highly recommended to detect bottlenecks.

OpenACC features a `data` directive that enables to create a data region where the information of the indicated variables is transferred into the GPU, and back to the host at the end of the data region. `accULL` creates a context at this point, and the directionality information provided through the copy clauses is used to register the variables in the runtime. In this case, we precede the `S` loop with the aforementioned `data` construct, indicating that the parameters Force, Position, Velocity and Acceleration can be transferred into the device at this point. From now on, all references inside a kernel to these variables will not require a memory transfer from the host, as they are all already stored on the device. When entering kernels inside `C` and `U` functions, the runtime will not create a new context, but it creates a new scope level within the existent context. With this method, we ensure that variables are registered once and directionality from higher scopes is preserved. However, new variables might be added to these nested constructs. Existence and directionality of these variables inside the device is restricted to the scope of the current scope. In the MD code example, we require the variables `pot` and `kin` to be transferred in/out between iterations in order to show the appropriate information to the user. However, as both `pot` and `kin` are registered within an inner scope, whenever these inner scopes are exhausted, variables are transferred back from the device to the host.

`accULL` enables users to perform incremental parallelization over GPU devices with minor effort. Traditional GPU performance tools can be used with the resulting codes. For example, in the MD code, the Nvidia profiler shows that more than 80% of the time is devoted to the `C` kernel. As stated before, this kernel is highly compute-intensive, as it features un-coalesced memory accesses and costly/non parallel floating point operations. One possible solution is to split this kernel into several smaller ones, to increase coalescence. This could be considered counterintuitive in traditional CPU programming, where processor features large caches, but in GPUs it is a good idea. In order to rewrite this kernel into smaller ones, a CUDA developer would require a considerable effort, as she would be forced to write additional kernel calls, memory transfers, etc.. In OpenACC, the programmer only needs to split the sequential code and put the appropriate directives on the new loops. The compiler will extract the required kernels.

`accULL` provide the means to execute our codes on several platforms, not only restricted to GPU devices, with a single source code. Performance figures for the low-end system `M3` are shown in Figure 1a. On the other hand, Figure 1b showcases the benefit of using an OpenCL implementation using an high-end shared memory multiprocessor (`M4`).

Version	Time transfer in	Time transfer out	Kernel time	Total time	% Speedup
Naive Approach	$0.02524s$	$0.016229s$	$1.03017s$	$3.747910s$	-
Using a data clause	$0.01133s$	$0.016193s$	$1.02849s$	$1.433504s$	61%
Splitting C loops	$> 0.01s$	$0.016176s$	$0.23832s$	$0.434439s$	88.4%

Table 3: Time per phase and speedup for each incremental optimization over the naive implementation, as measured in `M2` using the Nvidia CUDA platform over the GPU. The cost of the CUDA calls, context initialization and memory transfers were noticeable in this case, thus using the data clause improved performance.

Tables 2 and 3 show detailed timing information for transfers, kernel and total time obtained using Frangollo's internal tracing module. In both Tables, the problem size was 4096 particles and 20 iteration steps were performed. Results were validated against the sequential implementation.

Users can turn this tracing feature when building the runtime and produce these statistics through an internal Frangollo call. In addition to this simple profiler module, we have experimental support for the Extrae tracing library, that enables us to perform detailed performance analysis with the Paraver [3] tool.

3.2 Mandelbrot computation set

Listing 1.2 shows an implementation using a Monte-Carlo method to compute the area of the Mandelbrot set using OpenACC.

When creating the parameters for the kernel launch, `YaCF` indicates to the runtime that the `numoutside` parameter requires a reduction operation and expands the scalar variable to a vector. This vector stores a private copy of the variable in each thread. Later, both CUDA and OpenCL components of the runtime, using a separated and optimized kernel, perform the reduction operation. The reduction operation is not performed during kernel execution, but later on when the variable is transferred back to the device, or if the variable were required by another kernel.

```
1  #pragma acc kernels loop reduction(+:numoutside) private(i,j) copyin(
       npoints, c[npoints]) copy(numoutside)
2    for(i = 0; i < npoints; i++) {
3      z.creal = c[i].creal; z.cimag = c[i].cimag;
4      for (j = 0; j < MAXITER; j++) {
5        ....
6        if ( z is outside set ) {
7          numoutside++;
8          break;
9        }
10     } /* for j */
11   } /* for i */
```

Listing 1.2: The Mandelbrot set computation in OpenACC

Another issue that has a large impact on the performance of the CUDA code is the number of threads per block, particularly in the presence of irregular computations. Figure 2 shows the variability of the execution time while changing the number of threads. This variability reflects the significance of a proper launch

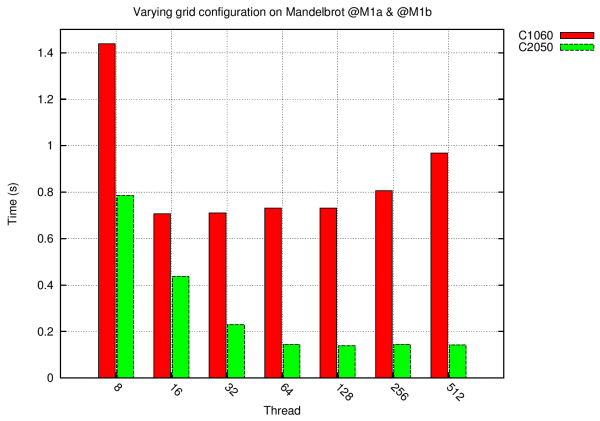


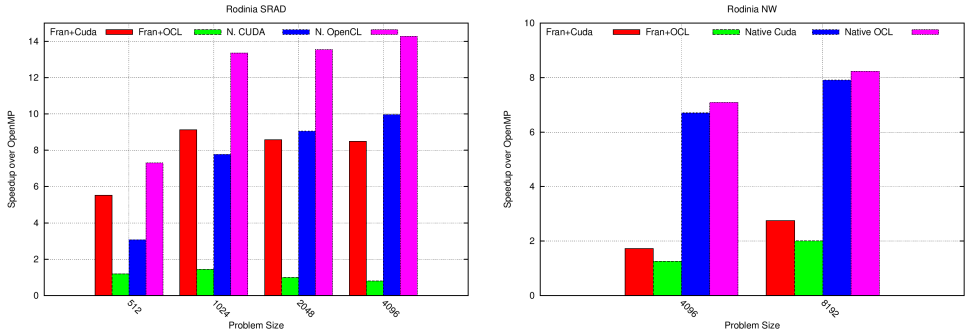
Fig. 2: Execution time of the implementation greatly varies in terms of the number of threads, using $N = 32768$ points. Besides, the optimal number of threads varies from Tesla C1060 to Tesla C2050.

grid configuration. Our compiler extracts compute intensity information from the kernel (i.e, relation between floating point operations and memory accesses) and passes the information to the runtime through the `launchkernel` API call, together with additional information about the loop, like boundaries, number of iterations, etc. This information is used to guess a first estimation to the number of threads per block. In case the user wants to influence this choice, an environment variable which varies the relation between floating point and memory operations is available.

For the CUDA component, a second estimator, which attempts to maximize the occupancy rate of the multiprocessors, is used. Information extracted from the PTX feeds this estimator. Using this two-tier system, we can guess a suitable number of threads for the target platform without user intervention. Current implementation features an environment variable which enables user to force a particular threads per block number and disables the thread estimation.

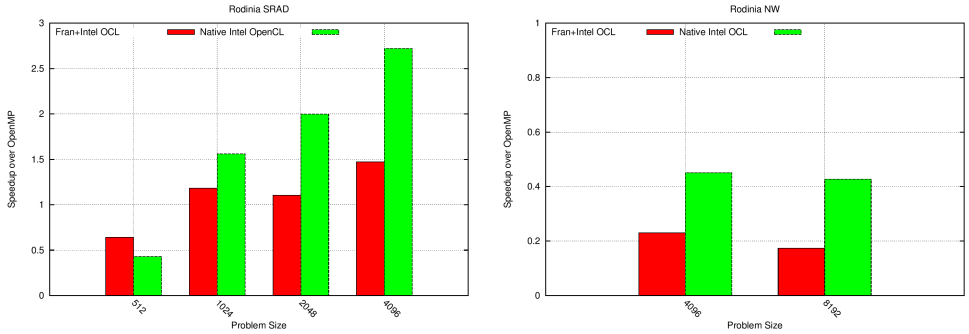
3.3 Rodinia Benchmarks

In order to complete our computational experience, in this Section we present performance comparisons for two benchmarks taken from the Rodinia suite [2]. Rodinia comprise compute-heavy applications meant to be run in the massively parallel environment of a GPU, and cover a wide range of applications. From this suite we have selected SRAD and NW for our experiments, and we present results for `M1` and `M4` platforms in Figures 3 and 4.



(a) SRAD **(b)** NW

Fig. 3: Performance comparison of `accULL` using `M1b` versus native implementation, showing the speedup against OpenMP. Although native implementation clearly outperforms both OpenMP and `accULL` implementations, the coding effort of OpenACC is lower than the required to write both CUDA and OpenCL implementations.



(a) SRAD **(b)** NW

Fig. 4: Performance comparison using `M4` of `accULL` versus native implementation, showing the speedup against OpenMP gcc implementation. It seems that using Intel OpenCL we are capable of extracting more performance from the shared memory machine. Native implementations of OpenCL also outperforms OpenMP.

4 Conclusions and future work

As we demonstrate in Section 3, 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.

Our runtime library can be fully detached from the compiler environment and used together with a 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 Frangollo 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 autovectorization support, and improvement of the support for memory allocation. We have work in progress to implement two dimensional arrays as cudaMatrix or OCLImages to improve non-contiguous memory access. Also we are exploring several possibilities of integration with MPI. The OpenCL component is capable of sharing pointers with MPI buffers, and we would like to use the new features 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.

References

1. Bihan, F.B.S.: Heterogeneous multicore parallel programming for graphics processing units. Sci. Program. 17, 325–336 (December 2009)
2. Che, S., Sheaffer, J.W., Boyer, M., Szafaryn, L.G., Wang, L., Skadron, K.: A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In: Proceedings of the IEEE International Symposium on Workload Characterization. pp. 1–11. IISWC '10, IEEE Computer Society, Washington, DC, USA (2010)
3. Giménez, J., Labarta, J., Pegenaute, F.X., Wen, H.F., Klepacki, D., Chung, I.H., Cong, G., Voigtländer, F., Mohr, B.: Guided performance analysis combining profile and trace tools. In: Proceedings of the 2010 conference on Parallel processing. pp. 513–521. Euro-Par 2010, Springer-Verlag, Berlin, Heidelberg (2011)
4. OpenACC directives for accelerators (2011), http://www.openacc-standard.org [Online; Last accessed October-2012]
5. R. Reyes, de Sande, F.: Optimization strategies in different CUDA architectures using `llCoMP`. Microprocessors and Microsystems - Embedded Hardware Design 36(2), 78–87 (Mar 2012)
6. Wolfe, M.: Implementing the PGI accelerator model. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 43–50. GPGPU '10, ACM, New York, NY, USA (2010)