Assessing the Performance of OpenACC Compilers

Lucas Grillo, Ruymán Reyes, and Francisco de Sande

Resumen—El estándar OpenACC es un importante avance en el mundo de la programación paralela. Los compiladores de OpenACC, al igual que los de OpenMP, son herramientas útiles para desarrollar aplicaciones paralelas sin necesidad de aprender una nueva forma de programación. Sin embargo, el estándar OpenACC es más reciente que OpenMP, por lo que la comunidad académica y de industria está buscando formas de evaluar su rendimiento y productividad. En este sentido, el presente artículo se centra en evaluar la performance de los compiladores de OpenACC disponibles comercialmente.

Palabras clave—OpenACC, Accelerators, GPGPU, CUDA, OpenCL, compiler, performance

I. INTRODUCTION

The OpenACC standard for heterogeneous computing [1] is quickly gaining momentum and receiving support from a broad range of industrial and research partners. Several different training sessions have been held in the past months, and the majority of HPC centers are now aware of the possibility of using OpenACC in their implementations. However, when facing a real-world project, the decision of using OpenACC rather than low-level programming models (namely CUDA or OpenCL) may not be clear. Initial low-quality implementations of the standard, and previous experiences with other programming models, could influence the choice of this standard in favor of a manual implementation. Also, the never-ending search for high performance at all cost makes HPC developers suspicious of this kind of semi-automatic approaches.

As in his day the introduction of OpenMP was a major boost to the popularization of shared memory HPC systems, the OpenACC standard lightens the coding effort required to develop heterogeneous parallel applications. Following the OpenMP approach, in the OpenACC API, the programmer annotates the sequential code with compiler directives, indicating those regions of code susceptible to be executed in the GPU.

Nowadays the number of OpenACC implementations is limited. Both the Portland Group (PGI) and CAPS, OpenACC endorsers, delivered their implementations in early 2012. Cray also has an OpenACC implementation available for their compilers, although it is not openly available. The authors of this work produced an OpenACC implementation in late 2011, accULL [2]. This is the first non-commercial implementation of the standard supporting both CUDA and OpenCL platforms and it is publicly available in the accULL project home page [3]. Our approach to the OpenACC implementation is based on two components: a source-to-source compiler and a portable runtime library. The compiler translates the annotated C+OpenACC source code into standard C code containing calls to the runtime API. The source-to-source translation injects a set of runtime calls within the serial code. Whenever these calls are issued, control is deferred to the runtime library, that will execute the code of the proper API call or whatever other code it might require (for example, to handle previous asynchronous operations). The runtime deals with two main issues of any OpenACC implementation: memory management and kernel execution.

The current status of the available OpenACC implementations may not be mature enough to compete with a hand-written implementation performed by a specialist. A key aspect for the gain in maturity and robustness by the OpenACC compilers is the availability of OpenACC annotated codes representative of real situations. Although there are some projects which are starting to include OpenACC into their codes, and some applications ported successfully, it is not yet as widely used and accepted as OpenMP is.

While these real applications become available, it is important for the development of the standard to have benchmarks that allow the GPGPU community to evaluate the compilers and the standard itself.

In previous works we have used the Rodinia benchmark suite [4], translating the OpenMP version of the Rodinia codes into OpenACC, to evaluate accULL. As far as we know, the recently published EPCC OpenACC benchmark suite [5] is the only benchmark available specific to OpenACC, and the evaluation of the OpenACC compilers available under this benchmark motivates this work. We present a comprehensive evaluation of existing OpenACC compilers (PGI, CAPS and accULL) from both industry and academia by the means of the EPCC OpenACC benchmark suite.

The rest of the paper is organized as follows: section [1] presents the EPCC benchmark. Section [1] presents performance and productivity figures. In the last Section we provide conclusions and future lines of work. This paper is an updated version of our paper Performance Evaluation of OpenACC Compil-

1Dept. de Ingeniería Informática Universidad de La Laguna, 38271-La Laguna, Spain e-mail: {jlgrillo, rreyes, fssande}@ull.es

1accULL is an open source project: https://bitbucket.org/ruyman/accull
II. The EPCC OpenACC benchmark suite

The EPCC OpenACC benchmark suite\cite{bib:5} is comprised of a set of low-level operations designed to test raw performance of compilers and hardware and a set of kernel operations which mimic those commonly seen in scientific codes. The suite is designed to be used to gauge the performance of various operations, kernels and test applications and enables the evaluation of both different accelerator platforms and OpenACC implementations.

The benchmarks are divided into three sections: Level 0, Level 1 and Applications. Table I summarizes the number of lines and directives for each benchmark. Level 0 and Level 1 benchmark tests are grouped in two files (level0.c and level1.c), whereas a separate file is used for each application in Level 2. Some additional support files required for timing are also provided with the package.

<table>
<thead>
<tr>
<th>Level / File</th>
<th>Test</th>
<th>Lines</th>
<th>Pragmas</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td></td>
<td>189</td>
<td>2</td>
</tr>
<tr>
<td>common.c</td>
<td></td>
<td>218</td>
<td>2</td>
</tr>
</tbody>
</table>

| Level 0       | ContigH2D | 24  | 1       |
|               | ContigD2H | 24  | 1       |
|               | SlicedD2H | 36  | 1       |
|               | SlicedH2D | 74  | 3       |
|               | Kernels_if | 35  | 2       |
|               | Parallel_if | 35  | 2       |
|               | Parallel_private | 36  | 2       |
|               | Parallel_1stprivate | 36  | 2       |
|               | Kernels_combined | 31  | 3       |
|               | Parallel_combined | 31  | 3       |
|               | Update_Host | 52  | 11      |
|               | Kernels_Invocation | 47  | 4       |
|               | Parallel_Invocation | 47  | 4       |
|               | Parallel_Reduction | 40  | 3       |
|               | Kernels_Reduction | 42  | 5       |

| Level 1       | 2MM | 108 | 7 |
|               | 3MM | 119 | 7 |
|               | ATAX | 83 | 5 |
|               | BICC | 86 | 5 |
|               | MVT | 98 | 5 |
|               | SYRK | 80 | 5 |
|               | COV | 112 | 8 |
|               | COR | 145 | 10 |
|               | SYR2K | 88 | 5 |
|               | GESUMMV | 74 | 3 |
|               | GEMM | 76 | 4 |
|               | 2DCONV | 60 | 3 |
|               | 3DCONV | 83 | 3 |

| Applications  | himeno | 279 | 4 |
|               | 27-stencil | 246 | 8 |
|               | le_core | 906 | 6 |

A. Level0

Level 0 contains a set of micro-benchmarks designed to measure the speed of operation of different OpenACC directives.

ContigD2H, ContigH2D, SlicedD2H and SlicedH2D measure the time to copy contiguous and non-contiguous chunk of data to/from the device. Kernels If and Parallel If measure the difference in time between executing a simple loop on the host and on the device by way of the if clause. Parallel Private and Parallel 1stPrivate measure the overhead of privatizing or 1st-privatizing a chunk of data. Parallel Combined and Kernels Combined measure the difference (if any) between using separate or combined kernels directives. Parallel Invocation and Kernels Invocation measure the overhead of invoking a simple kernel. Update measures the cost of performing an update from device to host inside a data region. Parallel Reduction and Kernels Reduction measure the overhead of performing a reduction.

B. Level1

The Level 1 benchmarks are OpenACC ports of thirteen computational kernels (dense linear algebra dwarf \cite{bib:6}) extracted from the Polybench \cite{bib:7} and Polybench/GPU \cite{bib:8} kernels.

The codes 2MM, 3MM, ATAX, MVT, GESUMMV and GEMM –very similar among them– perform different matrix and vector operations. BICC is a sub-Kernel of the BiCGStab Linear Solver. SYRK and SYR2K perform symmetric rank-k and rank-2k operations respectively. COV and COR compute covariance and correlation, respectively. 2DCONV and 3DCONV perform 2D and 3D convolutions.

In all cases, these benchmarks are implemented using the acc parallel directive, and explore different combinations of explicit and implicit data regions.

C. Applications

The Applications level of the benchmark is composed by three real applications codes: (1) himeno, (2) 27stencil and (3) le_core. The three of them implement a structured grid pattern \cite{bib:6}, with increasing complexity.

(1) himeno \cite{bib:9} is a widely known benchmark used to evaluate the performance of incompressible fluid analysis code. The code takes measurements to proceed major loops in solving the Poisson’s equation solution using a point-Jacobi iteration method. It is widely used to measure memory bandwidth performance in computer systems.

(2) 27stencil. Partial differential equation solvers are employed by a large fraction of scientific applications. These applications are often implemented using iterative finite-difference techniques that sweep over a spatial grid, performing nearest neighbor computations called stencils \cite{bib:10}. These operations are then used to build solvers that range from simple Jacobi iterations to complex multigrid and adaptive mesh refinement methods. The 27-point 3D stencil code in the suite, includes the edge and corner points of a 3x3x3 cube surrounding the center grid point.

(3) le_core is a linear elasticity (LE) \cite{bib:11} code simulation. LE is used extensively in structural analy-
sis and engineering design and it studies how solid objects deform and become internally stressed due to prescribed loading conditions. The complexity of this code is noteworthy, and it contains many non-coalesced access to arrays of structures from within the computational kernels.

III. EVALUATION

A. Experiment Setup

We used two different NVIDIA GPUs as experimental platforms: a Tesla M2090 (Fermi) with 512 CUDA cores, 16 streaming processors with clock rate at 1.30 GHz and 5GB DRAM and a Tesla K20c (Kepler) with 2496 CUDA cores, 13 streaming processors with clock rate at 0.71 GHz and 4.7GB memory. The host system consists of one Intel Xeon E5-2660 CPU at 2.20 GHz with 64GB of memory. The performance of each compiler was evaluated by measuring the run time of the output CUDA programs generated by each compiler.

Figure 1 shows the bandwidth available for each card in our setup. Notice that, when transferring from the host to the device, the bandwidth is slightly higher with Fermi than with Kepler. This happens for other transfer directions for the smaller problem sizes. This is probably due to limitations of the host PCI Express bus and chipset of our configuration. In the following subsections, we will mention this effect whenever we believe it could affect the performance of an implementation.

![NVIDIA Host to Device Bandwidth (MB/s)](Image)

Fig. 1: Host/Device memory transfer bandwidth. Only small transfer sizes as those used in the micro-benchmarks are shown.

In the following subsections, whenever we mention a kernel grid it will be represented in the form (block.dimX, block.dimY, block.dimZ) × (grid.dimX, grid.dimY, grid.dimZ).

For CUDA compilation, NVIDIA Toolkit version 5.0 was used. For PGI OpenACC (PGI from now on), PGI compiler version 13.5-0 was used, and for CAPS OpenACC (CAPS from now on) and accULL versions 3.3.4 and 0.4 were used respectively.

B. Level 0 Results

The data in Table I show the output for each benchmark test and compiler on the two NVIDIA GPUs. For each test the Table shows the average test time in microseconds for 10 executions of the test using 64MB datasize.

<table>
<thead>
<tr>
<th>Test</th>
<th>Tesla M2090 (Fermi)</th>
<th>Tesla K20c (Kepler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGI</td>
<td>CPU</td>
<td>GPGPU</td>
</tr>
<tr>
<td>capsULL</td>
<td>-58.60 1469.71 -6950.76</td>
<td>580.84 462.97</td>
</tr>
<tr>
<td>capsULL</td>
<td>5073.96 2778.54 12747.34</td>
<td>5073.96 2778.54 12747.34</td>
</tr>
</tbody>
</table>

The codes in Level 0 are divided into two categories: those which measure the time spent in a particular directive/functionality and those measuring the difference between two different constructs. They are marked with a T and a D respectively in Table II. Listing II shows part of the Parallel_combined test, an example of a D-type test. In line 16 the code returns the difference of the execution time of the code sections between lines 2–7 and 9–15. The timing pattern shown in Listing II is common to all test in Level 0; each test measures two different implementations of a construct and it returns the difference. Therefore, the negative numbers in Table II correspond to situations where the second code section is slower than the first.

The time spent by the PGI compiler in the data region tests (ContigH2D, ContigD2H, SlicedD2H, SlicedH2D) is greater than in the other compilers in both architectures. According to the NVIDIA Visual Profiler, the PGI compiler allocates and then uses the cuMemHostRegister in order to use Page-lock (pinned) memory. Since using this mechanism the memory can be accessed directly by the device, it can be read or written with potentially much higher bandwidth. However, given the small data size used in these experiments, the additional time required of obtain pinned memory does not compensate the higher bandwidth. Neither accULL nor CAPS use page-locked memory in this case and hence they are faster. It is expected that with larger problem sizes the usage of pinned memory compensates the extra time invested in allocation. The fact that the transfers to/from the Fermi card are slightly faster for small problem sizes cause the large times for Kepler, and it is not a problem due to the implementation.
The results of the Kernels.If test seem to indicate that the code in the acc kernels is faster than the code without any OpenACC annotation. The profiler reveals that the if clause is not honored in the PGI implementation, and the code is running in the device, running much faster than the host code. The implementation of the if clause in the accULL compiler is such that the code executed if the value is zero is exactly the original – with no transformation applied – but it is affected by some runtime calls. This means that the host implementation (not affected by any runtime calls) is slightly faster. When using the parallel if combination, the execution times favor the non-annotated code. However, the profiling analysis reveals that there is a kernel executing on the GPU, but it takes long time to execute. This may be caused due to the lack of a loop clause in the code, which forces the inner loop to be run sequentially on each thread. In the kernels if variant, the compiler automatically parallelizes the loops within its scope. Neither in the parallel nor in the kernels case, the profiler shows a kernel running when using the CAPS compiler, in spite of the fact that the code of the annotated region is slightly faster than its pure host counterpart. Since the differences among them are not significant, we believe that generic optimizations applied to the annotated loop before the final code generation phase might be slightly improving the performance of the host code.

The Parallel_private and Parallel_1st_private tests measure the difference of using create/copyin clauses instead of private/firstprivate respectively. Theoretically, in both cases the second case should be faster, as it is possible to use private copies in registers for each thread. However, since the private element is an array of dynamic size in both cases, and CUDA does not support non-constant size arrays, both implementations are showing similar performance.

Kernels_combined and Parallel_combined do not show relevant differences. In accULL both constructs are represented using the same form in the intermediate representation, hence the differences are negligible. A similar situation may be happening in the other implementations.

The huge times for Kernels_Invocation and Parallel_Invocation with accULL are caused by calls to the cuModuleLoadDataEx function. This call is used to load the PTX file into the device. The current implementation of our StS compiler generates calls to load the kernel into memory immediately before the first time it is called in the main program. Once a kernel is loaded in the runtime, there is no need to load it again, thus, this load does not pose additional overhead. However, given the nature of this particular benchmark, where each kernel is different and the three of them are loaded consecutively, the times for loading the kernel into the runtime are hindering accULL performance.

The Parallel_Reduction and Kernels_Reduction tests show the effect of applying the reduction clause to a variable. The lower the number (i.e., more negative), the greater is the overhead of implementing the reduction clause. In the accULL compiler, since the implementation is based on a two-stage process (first an scalar expansion and then a reduction kernel execution), the overhead of this kernel execution does not compensate the benefit for small problem sizes. PGI and CAPS seem to use solutions combining in-kernel operations together with a final reduction in a separate kernel. The profiling output of the PGI execution reveals that it is executing a reduction kernel also for the second section of the code, probably because the PGI compiler detected and applied the reduction even when it was not annotated. In this case, the CAPS implementation does not detect the reduction automatically, thus the time difference shows the real overhead of the reduction operation. It is somewhat surprising to find that the overhead of these reductions is higher in CAPS and accULL when using Kepler than using Fermi. This seems to be caused by two issues: (1) The reduction kernel in both CAPS and accULL is slower in Kepler than in Fermi (3.431 µs in Fermi vs 5.632 µs in Kepler using CAPS) and (2) The slightly better host-to-device bandwidth measured with Fermi in our particular setup.

C. Level 1 Results

The codes in Level 1 exhibit a greater complexity than those in Level 0, hence the execution times are higher. Increasing the problem size in steps of 4MB, we have executed all the tests in the Level 1 benchmark with problem sizes ranging from 16MB to 64MB. For each compiler, Figures 2, 3, 4 and 5 show the difference of the execution time with respect to the average of the three compilers. Values greater than zero mean that the implementation performed better than the average. Values lesser than zero mean the opposite. Figures 2 and 3 correspond to problems size of 16MB while Figures 4 and 5 are for 64MB instances.

Table II shows for each compiler, card and input size 16MB and 64MB, the number of tests with execution time below the average (above the horizontal line in Figures 2, 3, 4 and 5). Considering all the experiments performed, we observe that the CAPS
compiler exhibits the best performance in average in the different kernels, followed by the accULL implementation in some cases.

PGI and CAPS compilers perform a powerful analysis to determine the best scheduling clauses combination. The fact that in the accULL implementation the values used for these directives are set to a constant unless the user specify manually the values greatly degrades performance.

Under some circumstances, both commercial implementations are able to exploit the shared memory of the device. This requires powerful transformations and analysis that are still not available in the accULL implementation, which focus the optimizations and the runtime choices to favor cache access. Profiling output reveals that both CAPS and PGI compilers favor the usage of shared memory over cache, requesting 48Kb on both the Kepler and the Fermi. However, accULL requests 16Kb, favoring the usage of the caches. If we compare the performance of each compiler in the Fermi and Kepler architectures, we observe that the gap between accULL and the commercial compilers is closing. The more advanced and complex becomes the accelerator –particularly with the expansion of the cache hierarchy– the less critical the optimization of the memory access patterns becomes.

As a detailed example of the behaviour of the compilers with the tests in Level 1, Figures 6 and 7 show the execution time for the 2MM and SYRK tests respectively when increasing the input size from 4MB to 64MB. For these tests, the code generated by the accULL compiler exhibits the best behaviour in both Fermi and Kepler cards.

With respect to the analysis of the results in Level 1 benchmark and due to space contrains, we will limit our analysis to the 2MM and 3MM tests. The remaining tests in this level are affected by similar considerations to those presented here. The 2MM and 3MM tests perform two and three matrix multipli-
Fig. 6: Execution time for the 2MM test for different problem sizes. (Y-axis uses log scale).

Fig. 7: Execution time for the SYRK test for different problem sizes. (Y-axis uses log scale).

Fig. 8: Execution time for the 27stencil benchmark for different problem sizes (Y-axis uses log scale).

Fig. 9: Execution time for the himeno benchmark for different problem sizes (Y-axis uses log scale).

ditions respectively. Each one generates a separate kernel in the three compilers, as expected by the use of the parallel clause. However, the multiplications in 2MM are implemented using a three-dimensional loop nest with each dimension mapped to a kernel grid, whereas the 3MM only uses two dimensions of the grid, performing the inner one sequentially in the kernel. This forces 2MM to create a reduction operation in the inner loop. The execution time for the kernels in 2MM is larger than in 3MM, but surprisingly not only because of the reduction operation, but also the kernel itself requires more execution time with all compilers.

Each compiler selects a different grid and block size: CAPS (32, 8, 1) × (192, 1, 1) in all kernels and both benchmarks, PGI (256, 1, 1) × (161, 1, 1) in 2MM and (256, 1, 1) × (136, 1, 1) in 3MM. For accULL, the limitations of the parallel implementation always create a grid of (16, 16, 1) × (16, 16, 1) for two dimensional nested loops.

The profiler reveals then that, despite having a three dimensionally nested loop annotated, the PGI compiler only runs one dimension, whereas the CAPS compiler uses two.

None of the compilers is able to detect the dependencies between kernels and running them concurrently on the GPU (which is possible on Kepler and, to a minor extent, on Fermi).

**D. Applications Results**

Figures 8, 9 and 10 show the execution time for the Level 2 applications.

The 27stencil code is implemented using a single data region containing two parallel regions, the first one computing the stencil and the second one updating the values for the next iteration. The fastest compiler for this benchmark is CAPS, closely followed by PGI. As expected, the most expensive kernel is the stencil, with 81.7% of the kernel time with the PGI compiler and 88.8% with CAPS. accULL invests 93.3% of the time in the computational kernel. PGI uses a grid of (256, 1, 1) × (38, 1, 1) whereas CAPS uses (32, 8, 1) × (192, 1, 1). accULL uses (16, 16, 1) × (16, 16, 1) due to the aforementioned implementation limitation. The computational kernel generated by accULL is the slowest, potentially due to the sub-optimal grid selection.

Figures 8 and 9 show the execution time for the 27stencil benchmark for different problem sizes (Y-axis uses log scale).

The second update kernel is completely memory bound, as the only task that performs is an array copy. Ideally the compiler could detect this update operation and implement it using a device-to-device copy, or simply by swapping device pointers. However, it is difficult to detect these kind of situations and no compiler seems to implement this optimization so far. The PGI and accULL implementations only transfer memory in before the first kernel and out after the second one, whilst the CAPS implementation produces some additional transfers between the two kernels within the region and across different iterations, probably (due the reduced size of the transfers) for kernel parameters. Despite this fact, the performance of the kernels generated by the CAPS implementation compensates these transfers, and produces the fastest code in both platforms.

The default source code for the himeno test in the
EPCC benchmark is hardwired to use the small problem size, but for the sake of completeness, we explored the three predefined problem sizes available. Unfortunately, only the PGI compiler was able to build the largest problem size. Most of the time is spent in the main computational kernel, 92% of the time for accULL and 82% for the other compilers. However, this benchmark performance is determined by memory transfers.

The performance of the accULL kernels is poor, achieving only a theoretical occupancy of 25% vs more than 75% on the other cases. This is, again, caused by the poor parallel implementation. The kernel grid selected by PGI in this case is \((256, 1, 1)\) whereas CAPS uses \((32, 8, 1)\) \((192, 1, 1)\). The 2D shape of the CAPS kernel in this case produces faster kernels \((24.28\text{ms vs } 28.99\text{ms with PGI})\). Despite of accULL using a 2D grid, the performance is poor \((846.11\text{ms})\) due to the suboptimal grid-shape choice. The reduction kernels of the different compilers feature similar execution times (i.e. total kernel time in Kepler for 10 repetitions: \(181.70\mu\text{s CAPS, } 163.07\mu\text{s, accULL and } 159.17\mu\text{s PGI})\).

The source code of le.core contains two OpenACC kernel regions, each one in a separate subroutine (step X and step Y). Both of them exhibit similar characteristics, and are called once each per integration time step. The default number of time steps is 400. Since all the computation resides on the accelerator, an OpenACC data region is used to ensure the copy of the data to the device before the time step loop. Each kernel region uses a present_or_copy clause to avoid unnecessary memory transfers to/from host.

The performance bottleneck of this code is the complexity of the resulting device kernels. Although the code implements a 5-point stencil operation, there are many boundary conditions and expensive floating point operations. In addition, several conditional operations increase the divergence of the threads, decreasing the number of instructions that can be delivered in parallel. The PGI compiler handles the present_or_copy clause perfectly, only transferring data once in each direction. Most of the time is spent on kernel execution. The PGI compiler is the fastest, closely followed by CAPS. In this case, we used the kernels directive for the accULL compiler to force a proper selection of the scheduling clause, avoiding large execution times (twice than other compilers). This change made commercial implementations run slightly slower, thus in Figure \ref{fig:benchmark} we present results for the parallel directive for them. In this case, the fastest implementation is CAPS when using the Kepler card and PGI when using the Fermi. When using the Kepler card, PGI uses a one-dimensional grid shaped \((256, 1, 1)\) \((2000, 1, 1)\), using 63 registers per thread. CAPS uses a grid of \((32, 1, 1)\) \((192, 1, 1)\) with 73 registers per thread. accULL uses a 2D grid of \((8, 8)\) \((250, 250, 1)\) with 32 register per thread. It is interesting to observe how the performance is not related in this case to the occupancy of the device. Both PGI and accULL achieve higher theoretical occupancy \((\approx 50\%)\) whereas CAPS only achieves \(\approx 25\%\). However, CAPS invest only 5.27s executing the step Y and 5.16s for step X, whereas PGI needs 5.58s and 6.00s respectively. accULL produces the slowest kernels in this case, with a total of 7.34 and 7.39s respectively.

The implementations behave slightly differently when using the Fermi card. Both PGI and CAPS invest \(\approx 60\%\) of the time on step Y. Since in this two implementations the kernel grid is configured at compile time, it is the same than in the previous case. However, the CAPS kernels are much slower than their PGI counterpart \((5.52\text{s and } 10.53\text{s for PGI and CAPS respectively})\). The theoretical occupancy of the CAPS kernels is pretty low, around 16.7\%, whereas for PGI it is around 33.3\%. Despite of accULL selecting the kernel grid during runtime, the selected one is the same than in the previous case, which is suboptimal. The occupancy is similar to the PGI implementation. An in-depth analysis of the kernel performance in the accULL case reveals potential optimization improvements, particularly in the divergence-reduction side of the kernel generation, but it is outside of the scope of this paper.

### E. Native versus OpenACC performance

Although the existing EPCC OpenACC benchmarks thoroughly explore the performance of the various directives available in the standard, from our point of view this performance comparison should be completed with a comparative of native (CUDA) and OpenACC performance.

For this purpose we have implemented a simple vector addition code in CUDA and OpenACC. The code measures the time spent in the device-specific
part for CUDA and the data+compute region in OpenACC.

The execution time of a kernel for a vector addition should be similar across the different implementations. Aiming to extract the total overhead added by the OpenACC implementations from the overall program execution, we also run the programs under the `time` Unix command. Our findings are shown in Figure 11. The whole bar represents the total execution time. The blue portion of the CUDA bar represents the time spent in the device-specific part of the code. In the OpenACC implementations, blue and red portions represent the time spent in the `acc_init` and data+compute regions respectively. The green portion represents the time spent outside the regions. The native implementation is included here as a reference: no OpenACC implementation should require more time than its native counterpart, as this include all the CUDA calls required.

Ideally, the time spent in the data+compute region for the OpenACC implementations should be the minimum required to copy the data and execute the kernel, and the rest of the initialization should be performed outside the region. This is the case of the accULL and PGI implementations, whose time spent in the region is negligible in comparison with the total execution time. However, the CAPS compiler apparently performs the initialization inside the region, when the first directive appears, hence the time spent in the region is the same as the time the CUDA spent in device-specific code. In accULL, the initialization is performed before the `acc_init` call, when the program is loaded. The time spent in the region and in the `acc_init` is negligible, thus, the whole bar is green. The accULL implementation is slower than the rest due to some bookkeeping code executed before and after the program finishes, which detects the platform and creates the devices. This time is costant and should not pose significant overhead on large programs. However, we will work to reduce this time to match the performance of the other implementations.

IV. Conclusions and future work

Using the EPCC OpenACC benchmark suite, we have made an evaluation of three OpenACC compilers. As far as we know, this is the first usage of the benchmark to this end. Through using the NVIDIA Visual Profiler we have analyzed the reasons behind the results obtained on each of the benchmark levels for each compiler.

The EPCC suite is a decent collection of micro and macro benchmark codes to evaluate the quality of OpenACC implementations. However, the evaluated computational kernels are too focused on linear-algebra style codes or stencil applications. There is a lack of other dwarfs in this collection, in particular, unstructured grids or map reduce, which may be of interest for the OpenACC community. The Level 1 set of codes could benefit from a stencil-like kernel to evaluate compiler transformations without the complexity of the codes in the Applications section of the benchmark. Furthermore, the set of codes in the Applications level would be more complete with a Linear-algebra style application. Other comparisons with native code are interesting, such as the one we explored in this paper. Comparing the performance of OpenACC GEMM with CUBLAS (maybe by the means of the `host acc` directive) would be also of interest. Finally, adding more directives to the Level 0 would complete the evaluation of the implementation.

We were particularly interested in the evaluation of accULL through the implementation of third party codes. The results we have obtained are satisfactory: the performance of accULL, despite of being an openly-available project, in some cases outperforms the results of other approaches. When our results are poorer than those of the commercial compilers, the gap is not huge. It is worth to mention that being an academic and research oriented implementation, accULL is not aimed to compete with commercial implementations of the standard. We believe that accULL represents a good opportunity to be used as a research tool. Our runtime can be fully detached and used with a different compiler environment. Also, our approach produces legible code, which enables developers to optimize kernels by hand after the code has been generated. Although not explored in this contribution, accULL also supports OpenCL platforms, and accULL can also be used to generate code for ATI GPU cards or CPU OpenCL implementations.

Along the text we have mentioned some issues that would improve the performance of the accULL compiler. Particularly, the implementation of the `parallel` directive, or the reduction of thread divergence in the generated kernels are two aspects to be enhanced.

Apart from improving the accULL performance, as future work we would like to add other OpenACC compliant compilers to the performance comparison, to enlarge the experience with additional benchmarks, particularly using the Rodinia and NAS par-

![Fig. 11: Comparison of native vs. OpenACC execution time for the vector addition code in the Kepler GPU.](image)
allel benchmarks. We also believe that to widespread
the use of OpenACC it is mandatory to evaluate real
life codes under this standard.

Acknowledgments

The authors want to thank N. Johnson from
EPCC, Yann Mevel from CAPS and the anonymous
reviewers for their insightful comments and sugges-
tions. This work has been partially supported by the
EU (FEDER) and the Spanish MEC (Plan Nacional
de I+D+I, contract TIN2011-24598).

Referencias

[3] Ruymán Reyes and F. de Sande, “accULL project home-
page,” 2012.
[4] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron,
“A characterization of the Rodinia benchmark suite with
comparison to contemporary CMP workloads,” in Pro-
cedings of the IEEE International Symposium on Work-
load Characterization (IISWC’10), Washington, DC,
USA, 2010, IISWC ’10, pp. 1–11, IEEE Computer So-
ciety.
2013.
Keutzer, David A. Patterson, William Lester Plishker,
John Shalf, Samuel Webb Williams, and Katherine A.
Yelick, “The landscape of parallel computing research:
A view from Berkeley,” Tech. Rep. UCB/EECS-2006-
163, EECS Department, University of California, Berke-
[7] Louis-Noël Pouchet, “PolyBench/C the polyhedral
benchmark suite,”.
[8] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula,
and J. Cauzas, “Auto-tuning a high-level language tar-
geted to GPU codes,” in Proceedings of Innovative Par-
allel Computing (InPar ’12), 2012.
2003.
[10] Kaushik Datta, Samuel Williams, Vasily Volkov,
Jonathan Carter, Leonid Oliker, John Shalf, and Katherine
Yelick, “Auto-tuning the 27-point stencil for mul-
ticore,” in In Proc. iWAPT2009: The Fourth Inter-
national Workshop on Automatic Performance Tuning,
2009.
ty, Birkhäuser, Boston, 2002.