Comparative Analysis of OpenACC Compilers

Daniel Barba, Arturo Gonzalez-Escribano, and Diego R. Llanos [⋆]

Universidad de Valladolid, Departamento de Informatica, Valladolid, Spain
{daniel|arturo|diego}@infor.uva.es

Abstract. OpenACC has been on development for a few years now. The OpenACC 2.5 specification was recently made public and there are some initiatives for developing full implementations of the standard to make use of accelerator capabilities. There is much to be done yet, but currently, OpenACC for GPUs is reaching a good maturity level in various implementations of the standard, using CUDA and OpenCL as backends. Nvidia is investing in this project and they have released an OpenACC Toolkit, including the PGI Compiler. There are, however, more developments out there. In this work, we analyze different available OpenACC compilers that have been developed by companies or universities during the last years. We check their performance and maturity, keeping in mind that OpenACC is designed to be used without extensive knowledge about parallel programming. Our results show that the compilers are on their way to a reasonable maturity, presenting different strengths and weaknesses.

1 Introduction

OpenACC is an open standard which defines a collection of compiler directives or pragmas for execution of code blocks on accelerators like GPUs or Xeon Phi coprocessors. OpenACC aims to reduce both the required learning time and the parallelization of sequential code in a portable way [1]. OpenACC specification is currently on its 2.5 version [2], which has been released recently.

OpenACC was founded by Nvidia, CRAY, CAPS and PGI, but now there is a large list of consortium members, both from the industry and academy, including the Oak Ridge National Laboratory, the University of Houston, AMD, and the Edinburgh Parallel Computing Centre (EPCC), among others. The Corporate Officers are, at the time of writing this paper, from Nvidia, Oak Ridge National Laboratory, CRAY and AMD. Academic memberships are available to interested institutions.

There are several compilers supporting OpenACC. The PGI Compiler (from the Portland Group, which is a subsidiary of Nvidia for some time now) is being distributed as part of the Nvidia OpenACC Toolkit, under a free 90-day

[⋆] This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H5 network (TIN2014-53522-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS). Thanks also to Dr. Hector Ortega and Dr. Javier Fresno for their thoughtful comments, help, and encouragement.

trial license, a free annual academic license, or a commercial license. The PGI Compiler uses CUDA or OpenCL as backend. CAPS enterprise, a provider of software and services for the High Performance Computing community, also developed a compiler which supports CUDA and OpenCL. However, the company is no longer in business and its development is not available anymore. CRAY Inc. has its own OpenACC compiler, available with their computers. It is reported to be one of the most mature commercial compilers. However, they have not yet offered an academic or trial license for the purposes of studies like this one. Pathscale Inc., a compiler and multicore software developer, also made an OpenACC implementation on their ENZO compiler. Unfortunately, after private communications they were reluctant to allow us to use their compiler for this study.

There are also several academic attempts of developing an OpenACC compiler. In particular, OpenUH [3] developed at the University of Houston, and accULL [4] from Universidad de La Laguna (Spain). Both of them are available for free to anyone interested.

This work presents a study about the level of support of OpenACC in the available compilers, examining their strengths and weaknesses, and giving insights on their performance.

We analyze the level of maturity of each compiler, in terms of their *completeness in the support of the standard* and *robustness*, using each compiler's documentation to check what parts of the specification have been implemented. We check the support of OpenACC compiler directives with the help of a benchmark suite, developed by the Edinburgh Parallel Computing Centre, which will be described later.

Another aspect to test is the *relative performance* of the generated code. For this, we would want to run more complex applications and measure the differences between the performance of the executable code generated by each compiler. The ideal situation would be to test applications as close to real world problems as possible, avoiding synthetic code fragments. Since the use of OpenACC is not common yet, we have to rely on existing benchmarks. At this point, comparing the results from OpenACC code with CUDA or OpenCL direct implementations might seem appropriate, but porting the different benchmarks to these languages makes the result dependent on human interference, as the developer's ability for CUDA programming impacts on the performance.

The experiments conducted in this work were carried out using several benchmarks. First, the EPCC benchmark suite which contains a group of 13 kernels ported to OpenACC, called "Level 1" benchmarks. This benchmark suite also contains three real applications called "Himeno", "27stencil", and "le_core" [5]. We have also used the Pathscale port of the Rodinia benchmark [6]. We also wanted to use the OpenACC Validation Testsuite [7] developed by the University of Houston, but at this moment that tool is only available for OpenACC members.

Our conclusion is that the different compilers are on their way to a reasonable maturity. However, there is a number of features not fully implemented yet by some of the compilers.

The rest of this paper is organized as follows. Section 2 describes the selected compilers. Section 3 shows the characteristics of the benchmark suites chosen, enumerating some problems encountered when compiling them with the compilers selected. Section 4 contains the result of our analysis, in terms of completeness of the OpenACC features supported, robustness of compiler implementations, and relative performance of the generated code. Finally, Section 5 concludes our paper.

2 Available Compilers

In the introduction we mentioned several compilers. In this section we describe with more detail the compilers we were able to obtain and use for this study, and we will discuss their installation particularities on our Linux based platform.

2.1 PGI Compiler

The PGI Compiler [8] is being developed by The Portland Group, being owned by Nvidia. This compiler is widely used in webinars, workshops, and conferences.

The PGI Compiler is, at the time of writing this paper, available for download as part of the OpenACC Toolkit from Nvidia. This toolkit includes a 90-day free trial, the possibility of acquiring an academic license for a whole year, or buying a commercial license.

2.2 accULL

The accULL [4] compiler developed by Universidad of La Laguna (Spain) is an open source initiative. accULL consists on a structure of two layers containing YaCF [9] (Yet another Compiler Framework) and Frangollo [10], a runtime library. YaCF acts as a source-to-source translator while Frangollo works as an interface providing the most common operations found in accelerators.

2.3 OpenUH

The OpenUH [3] compiler, developed by the University of Houston (USA) is another open source initiative. It makes use of Open64, a discontinued open-source optimizing compiler.

3 Benchmark Description

This section describes the different benchmarks used in our work, enumerating the main characteristics that make them interesting for this study, and any issue detected during their compilation with the three compilers studied.

3.1 EPCC OpenACC Benchmarks

This benchmark suite [11] has been developed by the Edinburgh Parallel Computing Centre (EPCC). The benchmarks are divided in three categories: "Level 0", "Level 1" and "Applications". The compiled program launches all the benchmarks in the suite sequentially. By default, the number of repetitions is ten, and the result is the average for each benchmark. Time is measured in microseconds in double precision, using the OpenMP function `omp_get_wtime()`. We describe briefly the benchmarks included bellow:

Level 0 Level 0 includes a collection of small benchmarks that execute single host and accelerator operations, such as memory transfers.

Level 1 Level 1 benchmarks [12] consist on a series of BLAS-type kernels. They are based on Polybench [13] and Polybench/GPU kernels. They measure the performance of executing those codes. These benchmarks are run on the CPU first in order to have results to compare those obtained on the GPU.

A brief description of the different issues found while running this suite for these compilers follows.

OpenUH: The following benchmarks cannot be compiled due to unsupported pragmas present in their code:

kernels_if: Problems using #pragma kernels if(0).
parallel_private: Problems declaring params as private.
parallel_firstprivate: Problems declaring params as firstprivate.
le_core: Problems with non scalar pointers.
himeno: Problems with non scalar pointers.

accULL: There is a problem related to a function pointer in the host program. The compiler, during the source to source translation modifies the syntax of the function pointer. A `double (*test)(void)` is converted to a `double *(test(void))`. This was solved by manually correcting this change in the intermediate C code generated, re-compiling the object file and copying it to the main directory to link all the object files again. No warning from any of the pragmas was detected so we were able to run all the benchmarks.

3.2 Rodinia OpenACC

Rodinia is a benchmark suite for heterogeneous computing [14,15]. It includes applications and kernels for multicore CPU or GPU applications.

There is an effort to port existing Rodinia benchmarks for OpenACC. Pathscale [6] is working on this. We have tested their Rodinia version committed to GitHub on April 25, 2014. Most of the suite works with PGI, but OpenUH and accULL have many problems to compile most of the tests. We have been able to successfully compile the following benchmarks contained in the suite with two or more compilers: gaussian, nw, lud, cfd, hotspot, pathfinder, and srad2.

4 Evaluation

In this section we analyze the OpenACC compilers, using both documentation and experimentation. We use each compiler's documentation to check the *completeness of OpenACC features supported*. Then we use the EPCC benchmarks to check both *robustness* and *relative performance*. Finally we check *thread-block size sensibility*, measuring the impact on performance of different geometries.

4.1 Experimental Setup

We used a Nvidia GTX Titan Black to run the experiments. This GPU contains 2880 CUDA cores with a clock rate of 980Mhz and 15 SMs. It has 6GB of RAM, and Compute Capability 3.5. The host is a Xeon E5-2690v3 with 12 cores at a clock rate of 1.9GHz, and 64GB in four 12GB modules.

The PGI compiler is the one contained in the Nvidia OpenACC Toolkit, version 15.7-0, published in Jul 13, 2015. We used OpenUH version 3.1.0 (published in November 4, 2015), based on Open64 version 5.0 and using GCC 4.2.0, prebuilt, downloaded from the High Performance Computing Tools group website [16]. accULL is version 0.4alpha (published in November 28, 2013), downloaded from Universidad de La Laguna's research group "Computación de Altas Prestaciones" [17].

4.2 Completeness of OpenACC Features Supported

From each compiler documentation we get some insight on the completeness of the OpenACC features supported. From this information, we can conclude that the OpenACC standard is not fully implemented yet by any of the available compilers. There is work to be done, but the three compilers are at a respectable maturity level.

4.3 Robustness and Pragma Implementation

The EPCC Benchmark suite contains several benchmarks for testing OpenACC directives. These benchmarks are contained in the "Level 0" group, which has been described in the previous section. Table 1 contains the results obtained for the three compilers. In this section we enumerate the problems with each benchmark and we explain the results obtained, including the overhead of the different pragma implementations.

Except for *Update_host*, *Kernels_Invoc.*, and *Parallel_Invoc.*, the time shown is the difference between executing and not executing each pragma. When the overhead is zero (or the pragma is not implemented), the times are very similar, with minimal stochastic variation. These variations may produce a very small negative result when calculating the difference. When differences in time are on the order of tens of microseconds (positive or negative), it can be assumed that there is no difference in time between the different versions tested in that benchmark.

Table 1: EPCC level 0: directive's overhead (in μsec), 1 MB dataset

EPCC L0	**PGI**	**OpenUH**	**accULL**
Kernels_if	-37.50	Fail	4.54
Parallel_if	-30.76	-0.48	1237.02
Parallel_private	-21.94	Fail	51.09
Parallel_1stpriv	Fail	Fail	-213.83
Kernels_comb.	-1.67	-108.43	-127.17
Parallel_comb.	-0.05	-2.74	33.38
Update_host	478.63	373.22	548.77
Kernels_Invoc.	Fail	12.76	2398.20
Parallel_Invoc.	31.81	13.47	1377.88
Parallel_reduct.	-14.85	-164.41	-2168.12
Kernels_reduct.	-8.49	-172.31	-2009.11

PGI There was a problem with the "Kernels_Invocation" benchmark: It returned an incorrect result. The code was not being parallelized and the pragmas were ignored because it wasn't specifically stated that the iterations were independent. This could be solved adding the keyword `restrict` to the pointer or the clause independent to the pragma.

The "kernels_if" and "parallel_if" results are very similar, and in both cases the results indicate that the code with the pragma is slightly faster than the one without it, even though both are being run on the host. In [5] it was stated that this could be because of optimizations done by the compiler while or after processing the pragmas.

The "parallel_private" benchmark shows that the creation of private variables for each thread running the loop is slightly faster than the allocation of device memory.

"Kernels_combined" shows a very small difference of time between writing two pragmas instead of a combined one, the former being slightly faster than the latter although the difference is almost negligible. The same occurs for the "parallel_combined" benchmark, the difference being smaller in this case.

Finally "Parallel_reduction" and "Kernels_reduction" show that PGI has very little overhead for the reduction clause. In [5] it is stated that the PGI compiler does the reduction even if it is not annotated. This could explain the very small difference in both benchmarks.

OpenUH We got some errors during compilation of the "Kernels_if", "Parallel_private" and "Parallel_1stprivate" benchmarks so they are ignored in this analysis. However, the "Parallel_if" directive is supported and the difference between using the pragma to run code on the host or running it directly is almost negligible.

"Kernels_combined" shows an overhead of the combined pragma versus the separated version. However, this is not the case for the "Parallel_combined" benchmark, where the difference is much smaller. The invocation of kernels and parallel directives are very similar. And for both of them, the reduction adds a similar overhead. This might be related to OpenUH assuming loops to be independent inside kernels regions.

accULL No errors were shown while compiling or running the benchmarks with accULL. There is a big difference between the two versions contained in the "Kernels_if" and the "Parallel_if" benchmarks, where the kernels directive version has a very small overhead compared to the non-annotated code. This overhead is very large in the parallel directive version. This is explained by the accULL developers in [5] where they say that the absence of a loop clause in the parallel directive is causing the loop to be executed sequentially in each thread. Therefore, this clause is not correctly supported, as we understand from the OpenACC Specification that the loop should be executed only on the host.

Robustness Summary The overall results indicate that some of the clauses are not implemented yet, but the three compilers are in their way to a reasonable maturity level and, since the most used directives are working, they can actually be used for code parallelization using OpenACC.

4.4 Relative Performance of Generated Code

In this section we analyze the performance of the generated code describing the impact of pragmas overhead in accULL. Performance measurement is divided into *data movement*, where we analyze the results of the data movement benchmarks in Level 0 of EPCC OpenACC Benchmark Suite, and *execution performance*, using Level 1 and Application Level of EPCC OpenACC Benchmark Suite, and selected benchmarks from Rodinia.

Effect of Pragmas Overhead in accULL Some results from the Level 0 of the EPCC Benchmark Suite show a performance impact introduced by some clauses and directives in the accULL generated code.

Kernels and Parallel invocations in accULL have a higher overhead than other compilers. This is due to the runtime calls and it is specially noticeable in the `reduction` clause. These overheads accumulation does not have a significant impact for complex kernels, or launching the same kernel over and over again. However, this could be a problem when running simple kernels or many different small kernels. This is the main reason behind the overall results showing a worse performance of the accULL compiler in this analysis.

Data Movement Data movement performance can be measured in four benchmarks from the Level 0 of the EPCC OpenACC benchmark suite. We have launched 10 repetitions of those benchmarks with datasizes of 1 kB, 1 MB, 10 MB, and 1 GB. The results can be seen in Tables 2, 3, 4, and 5.

Table 2: EPCC data movement results (in μsec), 1 kB dataset. White cells highlight the best results. *norm.* is the normalized result using PGI as reference.

Data Mvmnt	PGI		OpenUH		accULL	
10reps, 1kB	time	norm.	time	norm.	time	norm.
ContigH2D	30.827	1.0	322.699	10.47	338.218	10.97
ContigD2H	14.686	1.0	323.319	22.01	343.919	23.42
SlicedH2D	12.087	1.0	310.897	25.72	315.914	26.13
SlicedD2H	14.948	1.0	324.010	21.67	327.714	21.92
			GeoMean	18.93	GeoMean	19.58

Table 3: EPCC data movement results (in μsec), 1 MB dataset. White cells highlight the best results. *norm.* is the normalized result using PGI as reference.

Data Mvmnt	PGI		OpenUH		accULL	
10reps, 1MB	time	norm.	time	norm.	time	norm.
ContigH2D	484.347	1.0	950.789	1.96	727.839	1.50
ContigD2H	461.936	1.0	632.691	1.37	792.761	1.72
SlicedH2D	17.094	1.0	267.982	15.68	274.462	16.06
SlicedD2H	36.335	1.0	254.702	7.01	285.685	7.86
			GeoMean	4.14	GeoMean	4.24

Table 4: EPCC data movement results (in μsec), 10 MB dataset. White cells highlight the best results. *norm.* is the normalized result using PGI as reference.

Data Mvmnt	PGI		OpenUH		accULL	
10reps, 10MB	time	norm.	time	norm.	time	norm.
ContigH2D	4141.402	1.0	6887.984	1.66	3354.666	0.81
ContigD2H	5876.088	1.0	2043.747	0.35	4396.052	0.74
SlicedH2D	27.322	1.0	404.214	14.79	427.203	15.64
SlicedD2H	48.017	1.0	269.818	5.62	280.203	5.84
			GeoMean	2.64	GeoMean	2.72

Table 5: EPCC data movement results (in μsec), 1 GB dataset. White cells highlight the best results. *norm.* is the normalized result using PGI as reference.

Data Mvmnt	PGI		OpenUH		accULL	
10reps, 1GB	time	norm.	time	norm.	time	norm.
ContigH2D	32310.009	1.0	788945.913	24.42	296340.991	9.17
ContigD2H	55179.119	1.0	553282.976	10.03	347280.359	6.29
SlicedH2D	400.066	1.0	535.011	1.34	533.943	1.34
SlicedD2H	158.071	1.0	2818.100	17.83	4294.407	27.17
			GeoMean	8.75	GeoMean	6.76

In [5] it was stated that PGI used pinned memory and that it was causing issues in smaller datasets. It seems that PGI has solved this issue since then and, looking at the documentation, it is now possible to specify the type of memory access we want with a compilation flag. When using large datasets, OpenUH and accULL do not show the expected results according to the evolution shown in tables 2, 3, and 4. We guess that this is related to the usage of pinned memory by the PGI Compiler, allowing it to obtain better results when datasets are big enough.

Execution Performance, EPCC Benchmarks In this section we will analyze the performance of the code generated by the PGI, OpenUH, and accULL compilers with the benchmarks contained in the EPCC Level 1 and Application level. We use three different datasets: 1kB, 1MB, and 10MB. This choice is based on the fact that bigger datasets result in an out of memory error due to how the benchmarks try to allocate memory on the device. We suspect the memory allocation is being done in each thread inside the generated kernels, using more memory than expected. In summary, PGI code obtains better results in almost every benchmark. However, the differences shorten when using bigger datasets.

Table 6: EPCC execution results (in μsec), 1 kB dataset. White cells highlight the best results.

Exec. time	PGI		OpenUH		accULL	
10reps, 1kB	time	norm.	time	norm.	time	norm.
2MM	99.087	1.0	522.304	5.27	2799.229	28.25
3MM	80.204	1.0	380.683	4.75	3799.048	47.37
ATAX	58.103	1.0	327.110	5.63	2564.702	44.14
BICG	72.408	1.0	350.380	4.84	2628.499	36.30
MVT	80.037	1.0	354.743	4.43	2665.299	33.30
SYRK	68.426	1.0	289.512	4.23	2394.803	35.00
COV	87.261	1.0	314.617	3.61	3795.372	43.49
COR	104.976	1.0	337.362	3.21	5208.668	49.62
SYR2K	73.290	1.0	317.574	4.33	2469.765	33.70
GESUMMV	65.613	1.0	312.996	4.77	1500.021	22.86
GEMM	49.710	1.0	323.725	6.51	1237.473	24.89
2DCONV	46.444	1.0	286.174	6.16	1207.528	26.00
3DCONV	45.514	1.0	285.792	6.28	1202.494	26.42
27S	335.884	1.0	432.801	1.29	3273.728	9.75
LE2D	6842374	1.0	*	*	*	*
HIMENO	547939	1.0	*	*	*	*
			GeoMean	4.39	GeoMean	24.38

For datasets of 1kB the results can be seen in Table 6. Benchmarks that fail to execute with a specific compiler are shown with an asterisk in the table. PGI code shows a very good performance, followed by the OpenUH code, which also behaves quite good. accULL is showing slightly worse results because it is paying a high price in overhead for loading kernels in memory for the first time while this operation is not required for subsequent kernel calls, or it is less noticeable on complex kernels where computation is more time consuming. This situation makes it very hard to compete with other compilers for such small and simple problems. This issue is less noticeable with bigger datasets.

Table 7: EPCC execution results (in μsec), 1 MB dataset. White cells highlight the best results.

Exec. time	PGI		OpenUH		accULL	
10reps, 1MB	time	norm.	time	norm.	time	norm.
2MM	2305.698	1.0	3467.703	1.50	4002.412	1.74
3MM	705.409	1.0	1453.137	2.06	5265.778	7.46
ATAX	484.204	1.0	1222.420	2.52	4212.914	8.70
BICG	502.849	1.0	1256.871	2.50	4229.466	8.41
MVT	538.135	1.0	*	*	4355.322	8.09
SYRK	1374.769	1.0	2543.616	1.85	4000.674	2.91
COV	3681.660	1.0	4251.957	1.15	23969.443	6.51
COR	3863.096	1.0	4318.953	1.12	25732.814	6.66
SYR2K	1968.789	1.0	2532.029	1.29	4586.741	2.37
GESUMMV	406.623	1.0	1195.669	2.94	2709.591	6.66
GEMM	1041.651	1.0	23642.850	22.70	3595.218	3.45
2DCONV	1637.363	1.0	1912.236	1.17	2991.542	1.83
3DCONV	9388.137	1.0	9670.520	1.03	10058.497	1.07
27S	2179.599	1.0	2224.064	1.02	8342.865	3.83
LE2D	6861089	1.0	*	*	*	*
HIMENO	540513	1.0	*	*	*	*
			GeoMean	1.92	GeoMean	4.11

When using the default dataset size of 1MB, we can see in Table 7 that the differences between PGI and the rest are smaller than when a dataset of 1kB was used. The increment of time due to the increment of dataset size for these benchmarks is more noticeable for the PGI compiler and, to a lesser extent, for OpenUH. accULL results are very similar to the results obtained with the first dataset of 1kB. Notice the results obtained for the GEMM benchmark with OpenUH, which is probably being executed sequentially.

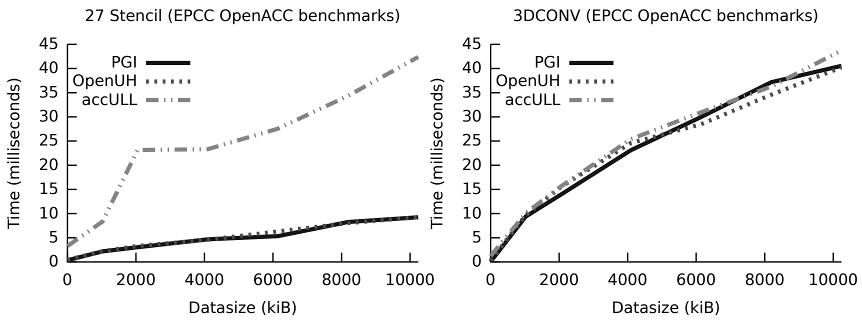
For datasets of 10 MB, the results can be seen in Table 8. Some benchmarks, for example “2MM” and “GEMM”, need unexpected amounts of time. Running the accULL generated code for 2MM requires one third of the time required by PGI and OpenUH codes. The GEMM benchmark shows a huge execution time, probably for the reason described in the previous paragraph.

The 27 stencil application is the only benchmark in the application level that compiles and runs successfully with all the compilers. It is a representative application of stencil codes that uses a three-dimensional neighbour synchronization pattern. Thus, it is a good representative of a well-known class of applications.

Table 8: EPCC execution results (in msec), 10 MB dataset. White cells highlight the best results.

Exec. time	PGI		OpenUH		accULL	
10reps, 10MB	time	norm.	time	norm.	time	norm.
2MM	64.407	1.0	64.336	0.99	20.476	0.32
3MM	11.610	1.0	21.113	1.82	30.009	2.58
ATAX	4.345	1.0	7.415	1.71	7.659	1.76
BICG	4.385	1.0	7.397	1.69	7.689	1.75
MVT	4.406	1.0	*	*	7.945	1.80
SYRK	26.537	1.0	57.242	2.16	42.834	1.61
COV	117.757	1.0	134.047	1.14	230.241	1.96
COR	120.612	1.0	122.814	1.02	223.836	1.86
SYR2K	23.450	1.0	27.939	1.16	30.062	1.28
GESUMMV	3.567	1.0	7.191	2.02	6.297	1.77
GEMM	17.788	1.0	239.713	13.48	48.567	2.73
2DCONV	7.848	1.0	7.725	0.98	8.687	1.11
3DCONV	40.551	1.0	40.235	0.99	43.729	1.08
27S	9.243	1.0	9.254	1.00	42.389	4.59
LE2D	6863	1.0	*	*	*	*
HIMENO	528	1.0	*	*	*	*
			GeoMean	1.59	GeoMean	1.63

In Fig. 1a we show the execution times obtained for this application in the chosen platform. The results show that one of the compilers cannot produce an efficient implementation. However, as it is shown in Fig. 1b, all compilers can derive similar solutions for simpler 3-dimensional stencil codes, as the 3DCONV benchmark of level one.



(a) 27Stencil (b) 3DCONV

Fig. 1: Execution results (in msec).

Execution Performance, Rodinia Regarding Rodinia, We should remark that the compilation presented some problems and there was a very limited amount of compiled benchmarks to choose from. The performance results can be seen in Tables 9 and 10. Only the Gaussian benchmark reports results for execution time including and not including memory transfers. Benchmarks that fail to compile with specific compilers are shown with an asterisk in the table.

Table 9: Rodinia execution time results including memory transfers. Total time (in msec). White cells highlight the best results.

Exec. time 3 reps	PGI	OpenUH	accULL
gaussian	2440.206	52.491	15422.944
nw	2640.497	652.180	322.101
lud	3803.756	1723.576	*
cfd	2677.387	0.846	*
hotspot	2386.325	53.219	*
pathfinder	5137.865	34.738	*
srad2	2488.895	692.063	*

Table 10: Rodinia execution results. Kernel time (in msec) not including memory transfers. White cells highlight the best results.

Kernel time 3 reps	PGI	OpenUH	accULL
Gaussian	57.345	36.092	15415.992

Here we can see how the code produced by PGI is not behaving as we would expect from the results obtained from the EPCC benchmark suite. PGI code is taking a lot of time in data transfer operations, while OpenUH code is not expending so much time in those operations. We discovered that the EPCC Benchmark Suite runs a function which contains an OpenACC pragma in order to make sure that the accelerator device is awake when the real benchmarks are run. This does not happen in Rodinia where the first OpenACC section is the data movement pragma. PGI generated code needs more than two seconds to set up the accelerator device, whereas OpenUH and accULL don't need that time. Beside this, accULL code is having some trouble running Gaussian as the time is not being spent in data transfer operations, but inside the generated kernels. We suspect that they might be executing sequential code in the host, due to a fail of the compiler or execution run-time to properly use the GPU. However, accULL obtains the best results in Needle-Wunsch.

Relative Performance Summary Results indicate a better performance for the code generated by PGI for the simple codes in the EPCC benchmarks, but not for the Rodinia applications. OpenUH generated code is not affected by any noticeable overhead and its performance is very close to PGI code. In order to analyze accULL code results it is important to take into account the overhead produced by kernel loading operations. If a bigger input set was used, results could be much better, but due to limitations on benchmark implementation this was not possible at this time.

5 Conclusions

During this work, we have realized that both the OpenACC standard and its compiler implementations are in their way to a reasonable maturity level. Although many details are still not completely developed, the efforts to arrive at a solid implementation are promising. Nvidia and PGI are devoting many resources to this project, and this results in a very competitive and solid compiler. However, open-source alternatives are also on a good position. OpenUH and accULL, being academic implementations, are also very interesting and show a huge amount of work done by their creators.

Regarding completeness of OpenACC features, according to each compiler's documentation, we find that none of them fully support the standard. This was expected as all of them are still not totally mature.

Speaking about robustness and pragma implementation, PGI shows the best behaviour, as the errors detected were related to implementation issues of the benchmark codes instead of a compiler problem. Compared to the others, the overhead of the implementation is smaller and it even includes some optimizations when processing pragmas that are going to be executed in the host.

Finally, the performance comparison we have made shows better results for PGI, but the other alternatives have also shown their strengths. It would also be interesting to run this performance analysis on several machines and different GPUs in order to also observe the differences of the execution of the generated code in different hardware, and this is part of our future work. The lack of OpenACC benchmark suites makes it very difficult to try different problems or datasizes. Our work shows that there is a need for real application codes annotated with OpenACC pragmas to test the actual potential of the current compiler implementation, as many articles before [5,18,19] have stated. This is part of our current and future work.

References

1. OpenACC-standard.org, "About OpenACC."
2. OpenACC-Standard.org, "The OpenACC application programming interface version 2.5," oct 2015.
3. X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, "Compiling a high-level directive-based programming model for GPGPUs," in *Languages and Compilers for Parallel Computing*, pp. 105–120, Springer, 2014.
4. R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "accULL: an OpenACC implementation with CUDA and OpenCL support," in *Euro-Par 2012 Parallel Processing*, pp. 871–882, Springer, 2012.

5. L. Grillo, F. de Sande, and R. Reyes, "Performance evaluation of OpenACC compilers," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pp. 656–663, Feb 2014.
6. Pathscale, "Rodinia benchmark suite 2.1 with OpenACC port." `https://github.com/pathscale/rodinia`, apr 2014.
7. C. Wang, R. Xu, S. Chandrasekaran, B. Chapman, and O. Hernandez, "A validation testsuite for OpenACC 1.0," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pp. 1407–1416, May 2014.
8. PGI, "Pgi accelerator compilers with OpenACC directives." `https://www.pgroup.com/resources/accel.htm`, nov 2015.
9. U. de La Laguna, "YaCF." `https://bitbucket.org/ruyman/llcomp`, nov 2015.
10. U. de La Laguna, "Frangollo." `https://bitbucket.org/ruyman/frangollo`, nov 2015.
11. EPCC, "Epcc OpenACC benchmark suite." `https://github.com/EPCCed/epcc-openacc-benchmarks`, sep 2013.
12. S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Autotuning a high-level language targeted to GPU codes," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, IEEE, 2012.
13. L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.
14. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. (IISWC), 2009 IEEE International Symposium on*, pp. 44–54, IEEE, 2009.
15. S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pp. 1–11, IEEE, 2010.
16. U. of Houston, "Open-source UH compiler." `http://web.cs.uh.edu/~openuh/download/`, nov 2015.
17. U. de La Laguna, "accULL." `http://cap.pcg.ull.es/es/accULL`, nov 2015.
18. S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*, pp. 859–870, Springer, 2012.
19. A. Hart, R. Ansaloni, and A. Gray, "Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 5–16, 2012.