Exploring Performance Improvement Opportunities in Directive-Based GPU Programming

Rokiatou Diarra, Alain Merigot, Bastien Vincke

Exploring performance improvement opportunities in directive-based GPU programming

Rokiatou DIARRA
SATIE, Univ. Paris-Sud
Univ. Paris-Saclay
94235 Cachan, France
rokiatou.diarra@u-psud.fr

Alain MERIGOT
SATIE, Univ. Paris-Sud
Univ. Paris-Saclay
94235 Cachan, France
alain.merigot@u-psud.fr

Bastien VINCKE
SATIE, Univ. Paris-Sud
Univ. Paris-Saclay
94235 Cachan, France
bastien.vincke@u-psud.fr

***Abstract*—GPUs offer an impressive computing power, but their effective exploitation remains an open problem. Kernel based programming models such as CUDA or OpenCL, allow a direct programming of the GPU architecture and can drive to excellent performance. However, these programming models require significant code changes, often tedious and error-prone, before getting an optimized program. Directive-based programming models (such as OpenMP and OpenACC) are now available for GPU and can offer good trade-off between performance, portability and development cost. In this paper, we do a comparative performance study between OpenACC, OpenMP 4.5 and CUDA, which is essential for facilitating parallel programming for GPUs. In order to find most significant performance issues, we port a suite of representative benchmarks and three computer vision applications to OpenACC, OpenMP and CUDA. Beyond runtime, we explore factors that influence performance, such as register counts, workload, grid and block sizes. The results of this work show that either OpenACC or OpenMP are good alternatives to kernel based programming models, provided some careful manual optimization is performed. Through the analysis of generated PTXs files, we discover that there is in general a systematic overhead in the kernel launch for OpenMP, but, for most applications, it is not a big issue provided the kernel has a sufficient workload.**

***Index Terms*—GPU, OpenMP, OpenACC, CUDA**

I. Introduction

Heterogeneous programming has become a reality with the omnipresence of accelerators, such as the graphics processing unit (GPU) and Xeon Phi, in current architectures. GPUS can achieve significant performance for certain categories of application (e.g.: computer vision algorithms, dense linear algebra). Nevertheless, achieving this performance is not only dependent on an important effort of programming and code tuning, but also a good knowledge of GPUS architecture. Thereby, porting sequential applications on these systems requires large efforts of rewriting. Kernel based languages such as CUDA or OpenCL are well suited to GPUS. They offer a number of features for performance optimization as the architecture is directly accessible to the user that makes possible to obtain excellent performance but adds complexities for application developers.

Directive-based programming models may therefore become an interesting solution. GPUS programming using directives is an alternative to kernel based language (CUDA and OpenCL). Because of their ease of use, directive-based programming models can offer a good trade-off between productivity, portability and performance. However, such programming strategies impose technical challenges on compiler optimizations, which could result in lower performance than with CUDA or OpenCL. With standards such as OpenACC and OpenMP $4.0/4.5$, programmers can easily implement an accelerated code by adding compiler directives in their sequential code and architecture management being left to the compiler. Each API has its own execution model, which is intentionally abstract to avoid coupling it to the specificities of any device.

Although many works [1]–[7] have been done on GPUS programming with OpenACC and OpenMP since their release, we shall demonstrate that, despite all the attention this topic has received, current compilers still need improvement in order to generate codes that can be equal to the performance of a hand-optimized CUDA code. We make the following contributions:

- The configuration of OpenMP and OpenACC directives affects the overall performance of the application. We test different combinations of directives to show their impact on performance. We show also that a good comprehension of OpenACC and OpenMP offloading model is necessary on behalf of the programmer in order to help compilers efficiently parallelize.
- Using various applications from the Rodinia benchmark suite and the field of computer vision, we do a comparative performance study between OpenACC, OpenMP 4.5 and CUDA, which is essential for facilitating parallel programming for accelerators.
- While kernel compute time is not sufficient to understand what is really happening at the architectural level when kernel running, we carry out a detailed analysis of the results with the help of performance counters and PTXs codes to identify execution models differences between OpenACC, OpenMP 4.5 and CUDA.
- Many embedded systems integrate GPUs. Embedded systems applications developers can use OpenMP or OpenACC to accelerate their codes. Currently, LLVM/Clang is the only open source compiler that has a support for NVIDIA GPU and ARM CPU. To the best of our

knowledge, this is the first work evaluating performance of LLVM/CLang implementation of OpenMP 4.5 on NVIDIA Tegra embedded platforms.

The paper is organized as follows. Section II provides background information on CUDA, OpenMP and OpenACC. Section III and Section V presents an extensive performance evaluation and analysis. Section VI summarize concluding remarks.

II. Background

A. CUDA

CUDA is a parallel computing programming model that fully utilizes hardware architecture and software algorithms to accelerate various types of computation [6], [8]. In CUDA, the programmer writes device code in functions called **kernels**. A kernel will be executed by many GPU threads. Before launching the kernel on the GPU the user must specify the number of threads, by setting **grid** and **block** sizes, that will execute the kernel. A grid consists of multiple thread blocks and each block contains several threads. During kernel execution, threads have access to different types of memory on the GPU. Each type of memory has its properties such as access latency, address space, scope, and lifetime. Before launching kernels, data must be transferred to GPU memories. To obtain optimized code, the programmer must understand well not only GPU architecture, but also CUDA optimization strategies like memory-coalescing access, efficient usage of shared memory or tiling technology. Additionally, grid and block configurations, computing behaviors of each thread, and synchronization problems need to be carefully tuned [6], [9].

B. OpenMP

OpenMP is undoubtedly the most used standards for several years in the parallel programming for shared memory CPUs [10]. OpenMP 4.0 and 4.5 extended the OpenMP shared memory programming model with the introduction of device constructs to support accelerators. In order to offload a region of code into the device, OpenMP 4.0 and 4.5 uses the **target** construct to create a data environment on the device and then execute the code region on that device. Various directives are provided to express the levels of parallelism. The **teams** construct, creates a league of thread teams where the master thread of each team executes the region. The number of teams created and of threads participating in the contention group that each team initiates are implementation defined, but can be specified respectively by the **num_teams** and **thread_limit** clauses. The **distribute** construct specifies loops which are executed by the thread teams. OpenMP 4.0 provides the **target data** construct to handle data transfers and update data on both of the host and accelerators within the target data regions. The **use_device_ptr** clause allows to indicate that a list item is a device pointer already. The **target enter data** and **target exit data** constructs, newly added by OpenMP 4.5, allow programmers to specify that variables are respectively mapped/unmapped to/from a device data environment. Finally, OpenMP provides the declarative **declare target** construct that specifies that variables and functions are mapped to a device.

C. OpenACC

OpenACC is another specification focused on directive-based ways to program accelerators [11]. Unlike OpenMP, OpenACC standard is relatively new, the first version was released in 2011. OpenACC has fewer constructs, but most of them are analogous to those of OpenMP 4.0/4.5. The OpenACC **parallel** construct starts parallel execution on the current accelerator device by creating one or more gangs of workers. The number of gangs, workers and vector lanes are set by the compiler if no **num_gangs, num_workers and vector_length** are specified. The **kernels** construct defines a region of the program that is to be compiled into a sequence of kernels for execution on the current accelerator device. The **loop** construct specifies the type of parallelism for the associated loop. In addition to the **collapse** directive, the loop construct provides the **tile** clause which specifies that the implementation should partition a loop's iteration space into smaller chunks or blocks. Like OpenMP 4.0/4.5, OpenACC provides the **data** construct for data transfer management between host and accelerator. The **enter/exit data** or **declare** constructs and *deviceptr* clause work in the same way as their correspondent in OpenMP. OpenACC offers also the **cache** construct which allows to specify that this data should be fetched into the highest level of the cache. Listing 1 shows matrix addition kernel example in CUDA, OpenMP and OpenACC.

```
1  __global__
2  void addCuda(float* A, float* B, float* C) {
3    int x= blockDim.x * blockIdx.x + threadIdx.x;
4    int y= blockDim.y * blockIdx.y + threadIdx.y;
5    if (x >= 0 && x < W && y >= 0 && y < H)
6      C[x + y * W] = A[x + y * W] + B[x + y * W];
7  }
8  void addOpenMP(float* A, float* B, float* C) {
9    #pragma omp target teams is_device_ptr(A,B,C)
10   #pragma omp distribute
11   for(int i = 0; i < H; i++)
12     #pragma omp parallel for schedule(static ,1)
13     for(int j = 0; j < W; j++)
14       C[i * W + j] = A[i * W + j] + B[i * W + j];
15 }
16 void addOpenACC(float* A, float* B, float* C) {
17   #pragma acc kernels deviceptr(A,B,C)
18   for(int i=0; i<H; i++)
19     #pragma acc loop independent
20     for(int j=0; j<W; j++)
21       C[i * W + j] = A[i * W + j] + B[i * W + j];
22 }
```

Listing 1. Matrix addition kernel example

III. Previous Work

In recent years, much work has been done on directive-based programing models for accelerators. Par4All [12] is an open source directive-based programing that allows to target parallel architectures including GPUS. PPCG [13] is a source-to-source compiler based on polyhedral compilation techniques. HMPP [14] is another high-level directive-based

language and source-to-source compiler that can generate CUDA and OpenCL code. Since the release of the OpenACC and OpenMP 4.0 standards, many works have been done to compare their performance with that of CUDA. Hoshino et al. [1] have studied the performance aspects of OpenACC and found that in general OpenACC is approximately 50% slower than CUDA, but for some applications, it can reach up to 98% with careful manual optimizations. Winke et al. [15] compared OpenMP 4.0 and OpenACC, predicting that OpenMP 4.0 would likely achieve best adoption in the long-term because it is such a prominent standard. Martineau et al. evaluated in [3] OpenMP 4.0 effectiveness as a heterogeneous parallel programming model and found that OpenMP 4.0 can achieve good performance while decreasing development cost. They also analyzed, in [5], the Clang OpenMP 4.5 current implementation on an NVIDIA Kepler GPU and suggested some potential solutions that can improve Clang code generation performance. Graham et al. [16] explored the performance portability of directives provided by OpenMP 4 and OpenACC to program various types of node architectures. They concluded that due to the slightly different interpretations of the OpenMP 4 specification, it is crucial to understand how the specific compiler being used implements a particular feature on different platforms. Hayashi et al. [4] evaluated and analyzed OpenMP $4.x$ on an IBM POWER8 + NVIDIA Tesla K80 platform. They found that the OpenMP generated codes are in some cases faster, in some cases slower than straightforward CUDA implementations written without complicated hand-tuning. The work presented in this paper brings, to the state of art, a clarification about the impacts of directives configuration on performance, variations in generated PTX and the additional costs related to the launch of kernel and thread management.

IV. Experimental methodology

In this section, we present the experiments carried out to perform our performance comparison between CUDA, OpenACC and OpenMP.

A. Applications

As part of this research it was necessary to port a number of applications that would be used in the performance analysis. For what purpose, applications: Back Propagation (BP), Breadth First Search (BFS), Heartwall Tracking (HW), HotSpot (HS), LavaMD (LMD), LU Decomposition (LUD), Needleman - Wunsch (NW) and Speckle Reducing Anisotropic Diffusion (SRAD) come from Rodinia[1] benchmark suite. We chose the Rodinia benchmark because it contains several classes of applications, and this variety allowed us to show the impacts of our different combinations of directives on performance. Detailed description of Rodinia benchmark is provided in [17]. We also used three applications typically used in preprocessing steps of complex computer vision algorithms: Canny filter (Canny), Harris Corner Detector (HCD) and Horn & Schunck (H&S) method of estimating optical flow.

[1]https://www.cs.virginia.edu/ skadron/wiki/rodinia/index.php/Downloads

B. Hardware platforms & Compilers

For our experiments, performance data were collected on an NVIDIA GPU Quadro M2000M hosted in an Intel I7 CPU and an NVIDIA Tegra X2 GPU. LLVM/Clang[2] (4.0) being the only free compiler with OpenMP 4.5 support for NVIDIA GPUs, it is the latter that was used to compile our OpenMP versions. For OpenACC, we used the PGI $17.10.0$ compiler and CUDA Driver/Runtime version is 9.1. On all platforms, we used Ubuntu 16.04.

C. Approach

Since our goal is to explore performance improvement opportunities in directive-based GPU programming, we have studied several aspects that can impact on OpenACC and OpenMP achievable performance. For that purpose, our study will take place in two steps:

1) Although the use of directive-based methods does not necessarily require a knowledge of GPU architecture, it is nevertheless useful to understand the offloading mechanism used by OpenACC and OpenMP in order to better exploit them. Indeed, the quality of the PTX code generated by the OpenACC and OpenMP compilers is dependent on how the programmer configures the directives. So, as a first step, we will analyze in detail the impact of directives configurations on performance. These configurations include: using directives separately on multiple lines or combining them into one, loops collapsing and grid and block sizes setting. Indeed, it is well known that using the optimal grid and block sizes helps to improve GPU code performance. Since default grid and block sizes chosen by OpenACC and OpenMP compilers are not always the right ones, the developer must often set theses sizes using the clauses dedicated for this purpose.
2) For the second step, we will study the performance evolution according to the workload. This will allow us to see how the additional cost related to the preparation and launch of the kernel as well as thread management varies depending on the body of the kernel.

All performance data were collected with NVIDIA profiler *nvprof*. In our experiment, we used Rodinia CUDA implementations for BP, BFS, HW, HS, HS3D, LMD, LUD, NW and SRAD and ours for Canny, HCD and H&S. In the following, we will take the performance of CUDA versions as a baseline.

V. Experiments Results and Analysis

In this section, we will discuss the results obtained from our experiences.

A. Evaluation of directives configuration impacts on performance

Although accelerator directive based programming model does not exhibit the same level of complexity as CUDA, it is necessary to understand how OpenACC and OpenMP 4.5

[2]https://github.com/clang-ykt

map the data as well as the distribution of the computation on the device in order to achieve good performance. With directive-based programming methods, the first opportunity to improve performance is how to combine directives. Figure 1 shows examples of directives configuration in OpenMP and OpenACC. In order for comparisons to be as fair as possible, all memory allocations are made with CUDA APIs in all versions. Thus, data transfers between the CPU and the GPU are managed in the same way as in CUDA versions. In all that follows, the CUDA versions of our applications are optimized using texture memory, constant memory or even shared memory. All optimizations made in OpenMP and OpenACC rely mainly on the use of directives and clauses. It should be noted that unlike OpenACC, whose **cache** directive allows the developer to tell the compiler that this data needs to be put in shared memory, OpenMP does not offer any directive for this service.

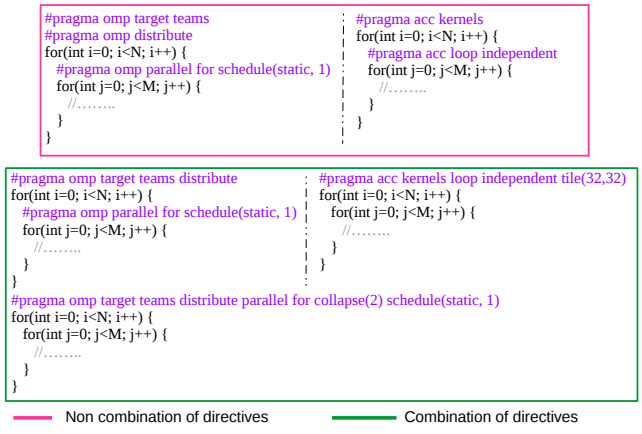


Fig. 1. Examples of directives configuration

In all following graphs, **nonComb** corresponds to the uncombined configuration of directives as we can see an example, in the red box in figure 1, **Comb** means combined configuration of directives (green box in figure 1) and **SettingGrid&Block** corresponds to the case of manual configuration of grid and block sizes in addition to directives combination. It should be noted that, directives configuration only has an impact on application performance (runtime, executed instruction count, register used per thread, etc.) and not on the accuracy of the result.

Figure 2 shows speedup factors relative to the baseline CUDA versions, overall execution time (including API calls and data transfer), on an NVIDIA Quadro M2000M and Tegra X2 GPUs. Overall, for both OpenACC and OpenMP, SettingGrid&Block versions are faster than nonComb and Comb versions. With OpenACC, as we cans see on sub figure 2a, directives combination (plus loops collapsing or tiling when they are ready for it) improves applications compute time by a reduction of $1.269\times$, on average, with respect to uncombined versions. Typically, an OpenACC optimized version (here SettingGrid&Block versions) requires $1.556\times$ hand-optimized CUDA runtime. With OpenMP, as we can see in sub figures 2a and 2b, directives combination allows a reduction in application execution time by a factor of $1.456\times$ on average, with respect to uncombined version, on both Quadro M2000M and Tegra X2 GPUs. Typically, an OpenMP optimized version requires $2.114\times$ hand-optimized CUDA runtime. However, these factors are slightly higher for HW and LMD due to the fact that their CUDA versions extensively use the shared memory while in PGI compiler, current version, the *cache* directive does not work well when there are many arrays to put in shared memory and OpenMP has nos directives to put data in shared memory.

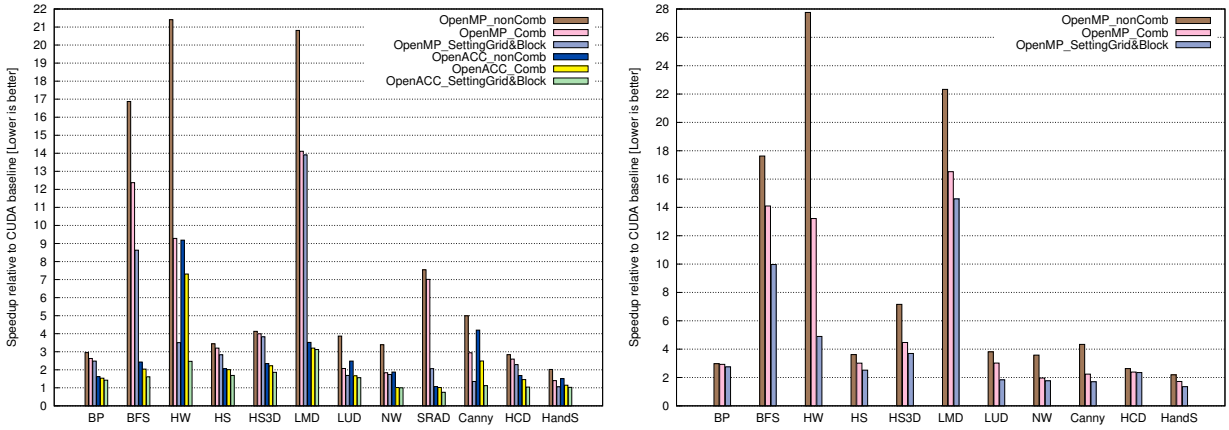
Directives combination helps, OpenACC and OpenMP compilers, to generate efficient code by influencing on the numbers of registers used per thread, the grid and block sizes, the number of generated instructions in PTX file, as well as the number of executed instructions by the threads. Inspecting generated PTXs, we have found that, for both OpenACC and OpenMP, the difference between uncombined and combined versions lies in how the parallelization is made. Table I shows the grid and block sizes and the number of registers used per thread automatically chosen by OpenACC/OpenMP compilers. We can see that directives combination reduces the number of registers used per thread in general. This reduction is more important in the case of OpenMP due to the fact that unlike OpenACC and CUDA, OpenMP uses more registers. For grid and block sizes, OpenACC and OpenMP compilers tend to use the same configurations in general.

By inspecting all PTXs generated by LLVM/Clang for all applications used in our experiment, we found what a third of instructions are used to compute parameters of functions such as *__kmpc_spmd_kernel_init*, *__kmpc_for_static_fini*, etc. These functions, systematically called in each kernel, are used by OpenMP in order to prepare the launch of the kernel on GPU, team creation, threads management and synchronization. So, it seems that the number of these functions called is higher in the case of non combination of directives, which contributes to degrade performance by increasing the number of registers used per thread for example. Thus, it is important to combine directives, in OpenMP, as much as possible in order to minimize the additional cost associated with these functions.

In summary directives must be combined and loops fused as much as possible. Indeed, we found that the *collapse* clause improves performance. With OpenACC, a significant improvement of performance can be achieved with the *tile* clause. Additionally, setting block and grid sizes contribute to improve application performance.

B. Performance evolution with workload

Code generation quality and cost, global memory access pattern and workload are factors which can affect OpenACC and OpenMP performance. In this section, we gradually increase in the loops the number of executed instructions. In order to prevent the compiler from using temporal variables, we use different memory address. In OpenMP and OpenACC



(a) Quadro M2000M

(b) Tegra X2 GPU

Fig. 2. Directives configuration impact on performance

TABLE I
REGISTERS, GRID & BLOCK CONFIGURATION GENERATED BY OPENMP/OPENACC COMPILERS ON QUADRO GPU: KN = KERNEL N; RG = REGISTERS/THREAD; THREADS/BLOCK = BLK; BLOCKS/GRID = GD

	CUDA	OpenMP_nonComb	OpenMP_Comb	OpenACC_nonComb	OpenACC_Comb
BP	K1[Rg:10], K2[Reg:20] Gd:4096, Blk:256	K1[Rg:47], K2[Reg:48] Gd:16, Blk:128	K1[Rg:47,Gd:16], Blk:128 K2[Rg:32,Gd:8193]	K1[Rg:22,Gd:16], Blk:128 K2[Rg:27,Gd:2049]	K1[Rg:22,Gd:16], Blk:128 K2[Rg:19,Gd:8193]
BFS	K1[Rg:18], K2[Rg:12] Gd:512,Blk:2048	K1[Rg:48,Gd:128] K2[Rg:40,Gd:1], Blk:128	K1[Rg:21], K2[Rg:21] Gd:8191,Blk:128	K1[Rg:22], K2[Rg:16] Gd:8191,Blk:128	K1[Rg:22], K2[Rg:16] Gd:8191,Blk:128
HS	Rg:32,Gd:1849,Blk:256	Rg:166,Gd:1024,Blk:128	Rg:72,Gd:1024,Blk:128	Rg:54,Gd:1024,Blk:32	Rg:48,Gd:1024,Blk:128
HS3D	Rg:32,Gd:1024,Blk:256	Rg:208,Gd:128,Blk:128	Rg:46,Gd:16384,Blk:128	Rg:42,Gd:16384,Blk:128	Rg:30,Gd:8192,Blk:128
LMD	Rg:40,Gd:128,Blk:1000	Rg:90,Gd:128,Blk:128	Rg:49,Gd:128,Blk:8	Rg:58,Gd:128,Blk:81	Rg:58,Gd:128,Blk:1000
NW	K1[Rg:40], K2[Reg:32] Gd:512, Blk:16	K1[Rg:55], K2[Reg:56] Gd:128, Blk:128	K1[Rg:32,Gd:512], Blk:128 K2[Rg:32,Gd:511]	K1[Rg:49,Gd:512], Blk:32 K2[Rg:50,Gd:511]	K1[Rg:49,Gd:512], Blk:32 K2[Rg:50,Gd:511]
SRAD	K1[Rg:20], K2[Reg:23] Gd:16384, Blk:256	K1[Rg:101], K2[Reg:56] Gd:32766, Blk:128	K1[Rg:40], K2[Rg:38] Gd:2048, Blk:128	K1[Rg:49], K2[Rg:40] Gd:33280, Blk:128	K1[Rg:28], K2[Rg:29] Gd:2048, Blk:128

versions, directives are combined, but we did not set grid and block sizes. Thus, compilers have the freedom to choose grid and block sizes. CUDA versions are not optimized. In this experiment, arrays sizes are 4096×4096 and data type is *float*.

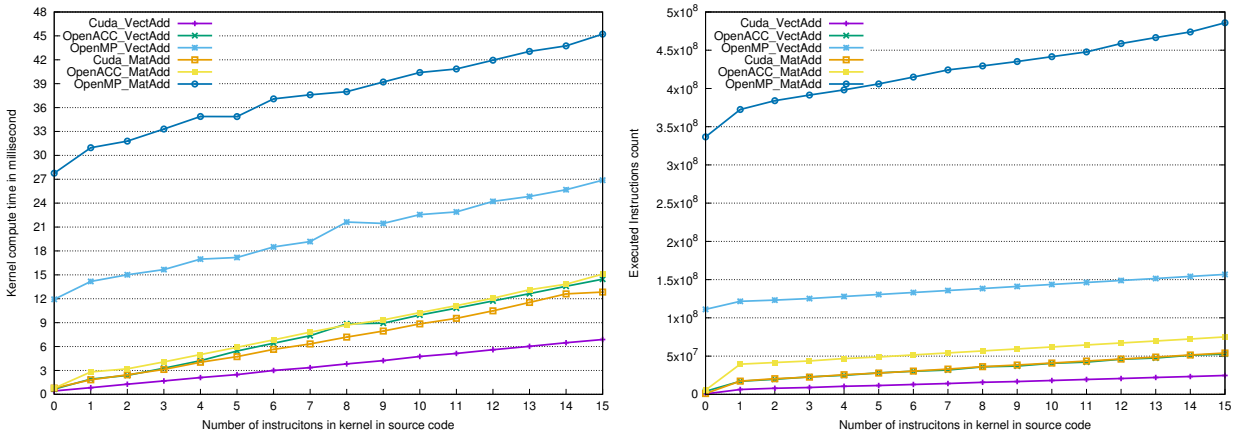
Figure 3 shows matrix and vector additions kernels compute time and executed instructions count evolution with the workload. On sub figures 3a and 3b, we found that global memory access pattern does not significantly affect performance for OpenACC.

The variation in computing time, according to the number of instructions in kernel (in source code), is relatively stable with an average slope of $878\mu s$ for both OpenACC, OpenMP and Cuda. However, the most important thing to see in figure 3 is that there is in general a systematic overhead in the kernel launch for OpenMP. As said in V-A and as can be seen in the figure 3b, OpenMP versions execute 2 to $8\times$ instructions more than CUDA versions and 2 to $6\times$ more than OpenACC versions. This shows that the current implementation of OpenMP in LLVM/Clang still needs improvement in order to generate better PTX code.

VI. CONCLUSION

This paper did a comparative study between OpenACC, OpenMP and CUDA. For that, we made investigations on applications taken from the Rodinia benchmark suite, three computer vision applications and synthetic programs in order to find most significant performance issues. We tested different configurations of OpenMP and OpenACC directives and evaluate their impact on overall application performance. We show that directives must be combined and loops must be collapsed as much as possible and, additionally, setting block and grid sizes contribute to improved application performance. It happens from our experiments that, provided some proper tuning of compilation directives is performed, either OpenACC or openMP can be good alternatives to direct CUDA programming. Kernel code generation is similar to the CUDA compiler. However, there is in general a systematic overhead in the kernel launch for OpenMP, but, for most applications, it's not a big issue provided the kernel has a sufficient workload.

We show that it is important to understand well most features of OpenACC and OpenMP in order to get competitive optimized codes with hand-optimized CUDA. We found that OpenACC optimized code typically requires $1.269\times$ hand-



(a) Kernels execution time

(b) Executed instructions

Fig. 3. Performance evolution with workload of matrix and vector addition kernels on Quadro M2000M GPU

optimized CUDA runtime which is consistent with factors found by Hoshino et al. in [1]. On the other hand, OpenMP requires in general $2.114\times$ hand-optimized CUDA runtime while Martineau et al. found in [3] a factor of $2.2\times$ with CCE implementation of OpenMP 4.0. However, it should be noted that current compilers (especially true for LLVM/Clang) still need some improvements to increase the competitiveness of OpenMP and OpenACC against CUDA. OpenACC and OpenMP standards also have to evolve to give more possibility (e.g.: access to constant and texture memories) to developers.

REFERENCES

[1] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, jun 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6546071/

[2] V. G. V. Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, "Early experiences writing performance portable openmp 4 codes." [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap161.pdf

[3] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating openmp 4.0s effectiveness as a heterogeneous parallel programming model," *IEEE International Parallel and Distributed Processing Symposium Workshops*, may 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7529889/?arnumber=7529889

[4] A. Hayashi, J. Shirako, E. Tiotto, R. Ho, and V. Sarkar, "Exploring compiler optimization opportunities for the openmp 4.x accelerator model on a power8+gpu platform," *Third Workshop on Accelerator Programming Using Directives*, nov 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7836582/

[5] M. Martineau, S. McIntosh-Smith, C. Bertolli, A. C. Jacob, S. F. Antao, A. Eichenberger, G.-T. Bercea, T. Chen, T. Jin, K. OBrien, G. Rokos, H. Sung, and Z. Sura, "Performance analysis and optimization of clangs openmp 4.5 gpu support," *7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, nov 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7836414/

[6] X. Li, P.-C. Shih, J. Overbey, C. Seals, and A. Lim, "Comparing programmer productivity in openacc and cuda: An empirical investigation," *International Journal of Computer Science, Engineering and Applications (IJCSEA)*, vol. 6, no. 5, oct 2016. [Online]. Available: http://aircconline.com/ijcsea/V6N5/6516ijcsea01.pdf

[7] K. Ikeda, F. Ino, and K. Hagihara, "An openacc optimizer for accelerating histogram computation on a gpu," *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, feb 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7439170/

[8] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 22, no. 1, jan 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5445082/

[9] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP'08 Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=1345220

[10] O. A. R. Board, "Openmp application programming interface," http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[11] OpenACC-Standard, "The openacc application programming interface," http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, 2015.

[12] M. AMINI, "Source-to-source automatic program transformations for gpu-like hardware accelerators," Ph.D. dissertation, cole nationale suprieure des mines de Paris, 2012. [Online]. Available: http://www.cri.ensmp.fr/classement/doc/A-506.pdf

[13] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers*, vol. 9, no. 4, jan 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2400713

[14] A. Sa-Garriga, D. Castells-Rufas, and J. Carrabina, "Omp2hmpp: Hmpp source code generation from programs with pragma extensions," in *HIP3ES Workshop, Vienna, January, 21st 2014*, 2014. [Online]. Available: https://arxiv.org/abs/1407.6932

[15] S. Wienke, C. Terboven, J. C. Beyer, and M. Mller, "A pattern-based comparison of openacc and openmp for accelerator computing," in *Euro-Par2014 Parallel Processing Workshops*. Springer, 2014, pp. 812–823.

[16] M. G. Lopez, V. V. Larrea, W. Joubert, O. Hernandez, A. Haidar, S. Tomov, and J. Dongarra, "Towards achieving performance portability using directives for accelerators," *Third Workshop on Accelerator Programming Using Directives*, nov 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7836577/

[17] 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 *IEEE International Symposium on Workload Characterization*, oct 2009.