An Analysis of OpenACC Programming Model: Image Processing Algorithms as a Case Study

Marko J. Mišić, *IEEE Member*, Darija D. Dašić, and Milo V. Tomašević

Abstract — **Graphics processing units and similar accelerators have been intensively used in general purpose computations for several years. In the last decade, GPU architecture and organization changed dramatically to support an ever-increasing demand for computing power. Along with changes in hardware, novel programming models have been proposed, such as NVIDIA's Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) by Khronos group. Although numerous commercial and scientific applications have been developed using these two models, they still impose a significant challenge for less experienced users. There are users from various scientific and engineering communities who would like to speed up their applications without the need to deeply understand a low-level programming model and underlying hardware. In 2011, OpenACC programming model was launched. Much like OpenMP for multicore processors, OpenACC is a high-level, directive-based programming model for manycore processors like GPUs. This paper presents an analysis of OpenACC programming model and its applicability in typical domains like image processing. Three, simple image processing algorithms have been implemented for execution on the GPU with OpenACC. The results were compared with their sequential counterparts, and results are briefly discussed.**

Keywords — **CUDA, graphics processing units, image processing, OpenACC, parallel programming.**

I. Introduction

In the past, graphics processing units (GPUs) have been primarily used as fixed-function processors in the domain of computer graphics. They were mostly used in rendering of 3D scenes and programmed using graphics-oriented application programming interfaces like OpenGL. Driven by the demand for high quality video and 3D effects, they have evolved to more flexible and highly programmable processors that are nowadays used for general purpose computations in various commercial and scientific domains. Modern GPUs consist of billion transistors, having more than ten times higher raw computing power and bandwidth than central processing units (CPUs) [1]. Although they can achieve a very high number of FLOPS, they are suitable only for a certain set of problems that show high regularity. GPUs have been traditionally programmed for non-graphics computations through low-level application programming interfaces, such as Compute Unified Device Architecture (CUDA) or Open Computing Language (OpenCL). Those programming models offer significant control over program execution and performance optimization, but they pose a problem for non-experts in the field of computer science. It is easy to write a correct program for GPU execution, but it is hard to optimize it for maximum performance. Following the OpenMP approach for multicore CPUs, a novel, directive-based OpenACC programming model has been developed [2]. In OpenACC model, programmer annotates regions of sequential code suitable for GPU execution with compiler directives. Compiler then generates a code for GPU execution.

Image processing is an important scientific field, as various algorithms are applied in domains like multimedia, medicine, telecommunications, robotics, etc. Those algorithms are suitable for GPU execution because of inherent data parallelism, thus we chose three of them to analyze this directive-based programming model for GPUs. In this paper we present a use case analysis of OpenACC programming model, as we applied it to parallelize three image processing algorithms of different complexities. First, we implemented them sequentially for the execution on the CPU. Those algorithms were adapted for parallelization with known optimization techniques and then OpenACC directives have been applied. Finally, we measured execution times of both sequential and parallel implementations and calculated the speedups.

The paper is organized as follows. In Section II, we give some details on the related work. Section III presents OpenACC programming model with a short overview of GPU architecture and a comparison with low-level, CUDA programming model. The fourth section discusses implemented algorithms and OpenACC directives used in implementation. The results of this analysis are presented in the fifth section, together with a discussion of achieved performance. A short conclusion and directives for future work are given in the final section.

Paper received March 14, 2014; revised May 5, 2014; accepted May 6, 2014. Date of publication July 31, 2014. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Jovan Đorđević.

This paper is a revised and expanded version of the paper presented at the 21th Telecommunications Forum TELFOR 2013.

This work has been partially funded by the Ministry of Education, Science and Technological Development of the Republic of Serbia (projects III44009 and TR32039). The authors gratefully acknowledge the financial support.

Marko J. Mišić is with the University of Belgrade, School of Electrical Engineering, Serbia (phone: 381-11-3218-392; e-mail: marko.misic@etf.bg.ac.rs).

Darija D. Dašić is with the University of Belgrade, School of Electrical Engineering, Serbia (e-mail: darijadasic@gmail.com).

Milo V. Tomašević is with the University of Belgrade, School of Electrical Engineering, Serbia (phone: 381-11-3218-302; e-mail: mvt@etf.bg.ac.rs).

II. Related work

OpenACC is a relatively new standard for heterogeneous computing, so related work is quite sparse. Several papers from the open literature describe different OpenACC implementations available, while there are few that present OpenACC in practice.

Reyes et al. published a series of papers about their OpenACC implementation, comparing it to existing directive-based programming models for GPUs. In [3], they present accULL – a freely available, open-source implementation of OpenACC. Their implementation combines YaCF compiler framework and Frangollo runtime to produce CUDA or OpenCL code from an annotated sequential code. Although native code implementations outperform those using accULL, authors believe that it is a good choice for non-experts as it takes much less development effort. A comparative study of directive-based programming models is given in [4] and [5]. Both papers address hiCUDA, PGI Accelerator, and OpenACC programming models. While paper [4] is more focused on performance aspects, paper [5] gives a wider overview and evolution of directive-based programming models, together with some advice on performance issues. A hybrid model built from OpenMP and OpenACC is described in [6]. It explores the possibility of multi-GPU execution, since OpenACC currently supports execution only on a single GPU. The authors proposed extensions to OpenACC standard to support multi-GPU execution, as their experiments with three different applications showed the effectiveness of such an approach. Most commonly used features of OpenACC are described in [7]. It deeply describes OpenACC *data* construct, and compares the use of the *parallel* and *kernels* constructs implemented in PGI Accelerator compiler.

First experiences with real-world applications using OpenACC are given in [8]. Authors ported two applications from fields of engineering (Bevel Gear Cutting simulation) and medicine (Neuromagnetic Inverse Problem solver) using OpenACC. They claim that OpenACC offers a promising ratio of development effort to performance and that the move to directive-based accelerator programming is essential for the further growth and acceptance of accelerator devices. An experience report on porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers is presented in [9]. Authors of the paper ported Himeno benchmark to run on the Cray XK6 hybrid supercomputer and compared the results with the corresponding CPU implementation. They suggest that the use of asynchronous algorithms could have huge potential impacts on the performance of applications executed on GPUs. A comparison between OpenACC, OpenCL, and CUDA in terms of performance, programmer productivity, and portability is shown in [10]. A Lagrangian-Eulerian explicit hydrodynamics mini-application has been used as an example, as authors find that OpenACC is an extremely viable programming model for accelerator devices, improving programmer productivity and achieving a better performance than OpenCL and CUDA.

III. OpenACC programming model

As it was mentioned before, GPUs have been traditionally programmed through low-level application programming interfaces like CUDA and OpenCL. CUDA programming model and API have been developed by NVIDIA and it could be considered as an extension to C programming language [11]. OpenCL is an open source, C-based programming language intended for execution on heterogeneous hardware. It supports both CPUs and GPUs, together with other types of accelerators like Intel Xeon Phi or FPGA devices.

Both CUDA and OpenCL require explicit programming using library calls and specially written compute kernels. CUDA is a vendor-specific technology, and applications written in CUDA could be executed only on NVIDIA GPUs, although frameworks from academic community like Ocelot [12] enable execution of programs written in CUDA on other platforms. On the other hand, OpenCL could be executed on different parallel platforms, but it has a rather complicated context management, making it hard to maintain portability and performance.

In both environments, parallel applications are usually written from scratch. An existing sequential code is usually heavily reengineered and used as a template for a parallel code. As pointed in [8], programming accelerators with low-level APIs is difficult, may complicate the software design and usually restricts the code to a device of a particular vendor. This leads to an unproductive development process with error prone programming tasks and highly hardware-specific implementations, which is not acceptable for large development projects with a long projected code lifetime.

Presented in 2011, OpenACC has a directive-based approach to programming accelerators. This standard has been developed by a group of hardware and compiler vendors like NVIDIA, PGI, Cray, and CAPS. It enables offloading different portions of C, C++, and Fortran code for execution on the GPUs. Much like OpenMP, it comprises different compiler directives and library calls that are used during compilation process to parallelize regions of sequential code and offload them for GPU execution.

There are many advantages of this, directive-based approach, and it has been already verified by the success of OpenMP standard for multicore CPUs as OpenMP is nowadays a *de facto* standard for programming shared memory systems. OpenACC directive could easily be applied to existing sequential code by annotating sources of parallelism like loops. Also, a good measure of syntax similarity between OpenMP and OpenACC enables easy adoption of the standard by a wider user community that already used OpenMP to speed up their applications on the CPU. In 2011, OpenMP did not have any kind of support for accelerators, and OpenMP language committee was still in discussion on the form of support that should be adopted with the new standard. OpenMP standard 4.0 brings support for accelerators, but it adopted a slightly different approach towards more manual control, and less freedom for the compiler in the parallelization process.

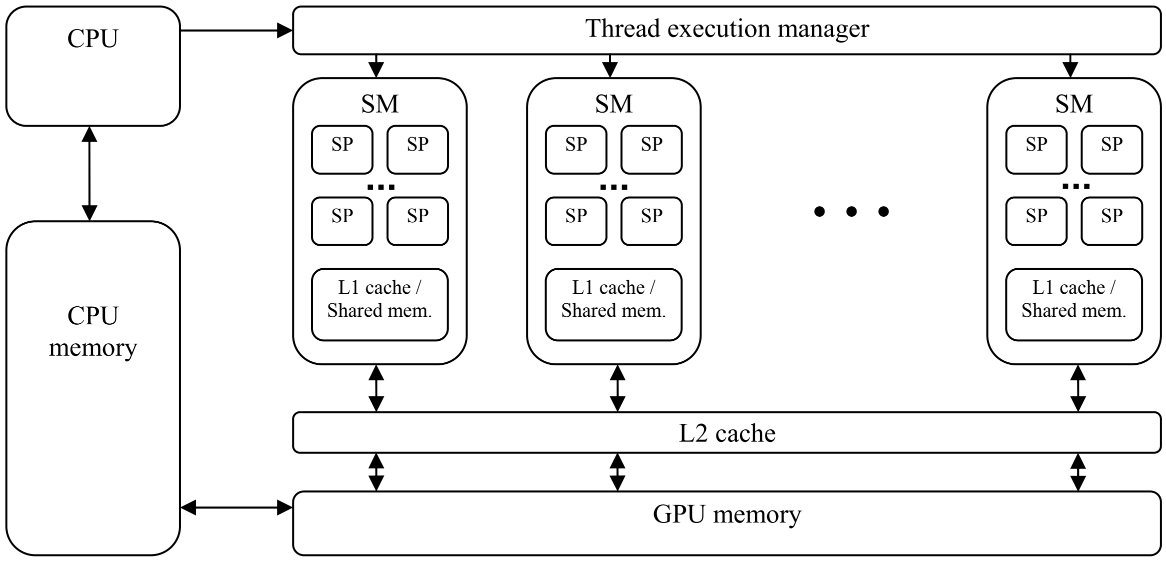

Fig. 1. Typical GPU architecture.

First compilers that support OpenMP 4.0 are expected in 2014.

A. GPU architecture and programming model

GPUs have a specific, manycore architecture, thus code execution differs considerably from the execution on the CPU. In this paper, we will mostly focus on the NVIDIA CUDA architecture, as it is the most mature platform in the domain of general-purpose computation on graphics processing units. Moreover, we used NVIDIA GPUs to evaluate algorithms implemented using OpenACC. With smaller differences, code is executed in the same way using AMD GPUs or similar.

Typically, CUDA programs are executed in co-processing mode. Sequential parts are executed on the CPU, while compute-intensive parts are offloaded to the GPU for parallel execution, as a special function called kernels. A kernel is executed by a large number of lightweight threads that run on streaming multiprocessors (SMs). Threads are organized in blocks executed on a single multiprocessor, and kernel execution is organized as a grid of thread blocks. Every streaming multiprocessor is a SIMD processor, consisting of a number of scalar processors (up to 192), depending on the generation of the graphics hardware. The number of threads per block varies from 64 to 1024. Thread blocks executed on the same SM share available resources, such as registers and shared memory. A typical GPU architecture is shown in Fig. 1.

Threads in the same block can synchronize their execution using a barrier. Due to execution scalability reasons, threads from different thread blocks cannot cooperate, since they may or may not execute on the same SM. Global synchronization is achieved only through repeated kernel calls, pointing to the one of the significant disadvantages of the CUDA programming model that strongly affects the algorithm design.

To support fast, parallel execution, GPUs provide a specific memory architecture as simultaneous execution of many threads consumes significant bandwidth to access data. Global, DRAM memory accesses are slow, so threads can utilize other smaller memories in the hierarchy to speed up the execution. Those memories differ in speed, capacity, and some other characteristics. Each thread has an exclusive access to given (allocated) registers and local (private) memory. Threads in a block could share data through a, user-managed, per-block shared memory. Also, threads can access constant and texture memories. It is worth mentioning that CPU and GPU use different, physically separated memory spaces, so explicit transfers of data between CPU and GPU are needed.

Although it is not difficult to write a correct code for GPU execution, achieving maximum performance is a daunting task. Performance can vary greatly depending on the resource constraints of the particular device architecture, and it is much on the developer to exploit all parallelism available [1]. For those reasons, new high-level programming models, like OpenACC, have been developed, to allow easier programming and hide architecture details from the developer, while still maintaining a good measure of performance.

B. OpenACC programming model

As mentioned before, OpenACC is a set of compiler directives, library functions, and environment variables that enables offloading regions of code for execution on the GPU (device). OpenACC is compatible with a number of CPUs and GPUs, as well as all major operating systems. The standard supports annotating of C, C++, and Fortran codes, thus allowing easy parallelization of existing applications written in those programming languages [2]. For example, all OpenACC directives for C/C++ can be identified from the string *#pragma acc* just like an OpenMP directive can be identified from *#pragma omp*.

Like other directive-based programming models, OpenACC offers implicit parallelization, as all initialization tasks like memory transfers and execution configuration are transparent to the programmer. Those tasks are typically done by the compiler and runtime environment, and the primary role of the programmer is to specify and annotate those regions of code that are suitable for execution on the device. Those regions of code are translated into compute kernels by the compiler. Parallelization is not fully automated, but assisted, as it is up to the programmer to define execution context – data involved, memory transfers that should occur, dependencies, which will be used by the compiler in the parallelization process.

C. OpenACC execution model

OpenACC execution model is similar to CUDA execution model, as CPU (host) manages the execution of compute kernels on the GPU. To address different device architectures, OpenACC execution model offers three levels of parallelism: *gang*, *worker* and *vector*. Depending on the device capabilities, compiler decides how to map those constructs to the underlying hardware and what is the best mapping for the problem [13]. Execution model assumes that device will contain multiple processing elements (PE) that run in parallel and can efficiently perform vector-like operations. For NVIDIA GPUs, PE is a streaming multiprocessor, *gang* represents a block of threads, *worker* is effectively a warp of threads, and *vector* is a form of CUDA thread.

OpenACC uses two types of directives to create a parallel region – *parallel* construct is used to annotate *work-sharing* loops, while *kernel* directive enables assembling of kernels that contain more than one loop [7]. OpenACC *parallel* directive is closely related to the same OpenMP directive and allows more explicit, user-defined parallelism. Only one kernel will be generated and executed at the launch time with the fixed number of gangs, and workers in the gang. On the other hand, *kernels* construct is more flexible, as each loop in the kernel can differ in the number of gangs and workers it utilizes. A programmer can specify the level of parallelism in *parallel* and *kernel* directives using clauses. Still, it is recommended to leave these optimizations to the compiler in order to avoid possible problems and performance cliffs on different architectures.

OpenACC does not support synchronization or data sharing between gangs. Both CUDA and OpenCL programming languages make these same assumptions in order to maintain execution scalability on an arbitrarily large numbers of PEs. These limitations mean that OpenACC programmers must map the parallelism in their code in such a way that data is only shared among workers within the same gang [13].

D. OpenACC memory model

In heterogeneous CPU/GPU architectures, host and device memory spaces are usually physically separated, so data transfers should occur between them. This need is obvious in low-level programming models like CUDA and OpenCL, although CUDA 6.0 brought unified memory support which enables applications to access CPU and GPU memory without the need to manually copy data from one to the other [11]. There are similar efforts from AMD with its Fusion series of Accelerated Programming Units (APUs). Still, the need to transfer the data remains, and OpenACC compiler does that task implicitly based on the directives inserted in the code like *copyin*, *copyout*, etc.

Multiple memory spaces raise some performance related issues. Capacity of the device memory is limited, so large data sets should be constantly streamed to the device. Because of that, memory bandwidth is a limiting factor for the performance of the parallelized code. On the other hand, contemporary GPUs are non-coherent processors, and no kind of memory coherency is supported between PEs. Even memory coherency in one PE is not guaranteed, except if operations are synchronized with a barrier. A good compiler is able to detect these problems, but it is still possible to generate incorrect compute kernels that will produce wrong results.

E. OpenACC version 2.0 enhancements

In 2013, a new version of OpenACC standard was revealed. It brings several improvements to the programming model [14]. Function calls are now permitted within compute regions. Early compilers used to inline function calls if possible, which proved to be complicated in some applications. The new standard allows unstructured data regions which helps annotating C++ class code. Like OpenMP, OpenACC 2.0 supports nested parallelism and dynamic parallelism which enables GPU to launch another level of parallelism dynamically. Also, a number of improvements have been added to OpenACC API, together with new directives like *tile* and *atomic* that allow better performance optimizations [15].

IV. Implemented algorithms

To show potentials of OpenACC programming model, we chose to parallelize three image processing algorithms: image thresholding, image rescaling, and simple Gaussian blur. Algorithms are first implemented sequentially for execution on the CPU, and then parallelized for the GPU execution using OpenACC directives. Sequential implementations were chosen in such a way as to be suitable for parallelization on the GPU. Test images were represented with three separate arrays that contain respective RGB components.

A. Image thresholding

Image thresholding is the simplest method of image segmentation which is used to create binary images from grayscale or color images depending on a given threshold value. Every pixel in the input image is first converted to a grayscale value, and then compared with the threshold value. If a grayscale value is above the threshold, then it is assigned with maximum intensity, otherwise it is assigned with zero (or inverse, if needed).

Image thresholding is a good example of an embarrassingly parallel class of algorithms, as almost no cooperation and synchronization between threads is needed to implement them. We used gang and vector level of parallelism and OpenACC *kernels* directive to implement it. The number of vectors (threads) was set to 512, while the number of gangs was not explicitly defined. Instead, OpenACC compiler defined it depending on the dimensions of the input image.

B. *Image rescaling*

Image rescaling is the process of resizing an image. Rescaling is usually a non-trivial process, as it involves quality issues like sharpness, smoothness, etc. For our purposes, we chose the bilinear interpolation found in [16]. It uses four pixels from original image to produce a pixel in the rescaled image. The algorithm maps a pixel of the rescaled image to the original image, and computes its value as a weighted sum of four neighboring pixels chosen as a 2x2 matrix. If image is enlarged, each pixel from the original image gets its place in the resized image, while new, additional pixels are filled with bilinear interpolation. Similarly, if image size is reduced, new pixels are filled with bilinear interpolation, but some pixels from the original image participate only partially. Bilinear interpolation is done separately for all three RGB components.

Implementation of bilinear interpolation requires access to a 2x2 matrix of original image for each output pixel, making a total of 12 memory reads (four for every RGB component). As pointed earlier, memory accesses are expensive on the GPU, so we preferred bilinear interpolation over bicubic interpolation which we also took into consideration. On the other hand, bilinear interpolation is a bit slower than the nearest neighbor interpolation, but offers much better results in terms of image quality. The algorithm is implemented with OpenACC *parallel* construct, as it offers better control over code execution on the GPU. The algorithm consists of only one loop, and *parallel* directive uses clauses to explicitly specify the number of gangs and vectors depending on the input image sizes.

C. *Blur*

The most complex algorithm that we included in our analysis is image blurring. We chose it because of its computational complexity, as computing of each resulting pixel requires the values of several neighbors in both dimensions. In this work, we used Gaussian blur method described in [16] to calculate the values of convolution matrix (convolution kernel) used in actual blurring. Implemented algorithm can accept different radiuses for convolution kernel and calculates its values.

To improve performance on the GPU, sequential algorithm is restructured to use a convolution array instead of a convolution matrix. The array is applied two times to the original image, first horizontally, and then vertically. It produces the same effect like using convolution matrix, but computational complexity is much lower. The same effect of these operations is guaranteed by the properties of the Gaussian function used for convolution [16].

Since two independent loops were used for horizontal and vertical passes with a convolution array, we used *kernels* directive to parallelize the code. Each pass was implemented with two nested loops. The iterations of outer loop are independent and distributed between different gangs, while every vector (thread) executes the iterations of inner loop sequentially, as they are dependent.

V. Experimental results

After the described algorithms were implemented, they were carefully evaluated with different data sets. Execution times were measured for both sequential and OpenACC implementations and the results are presented in the following figures. Implementations were evaluated using Intel Core i3 M 350 2.27 GHz processor with 4GB RAM. Graphics card installed on the system was NVIDIA GeForce GT 335M with 72 CUDA cores. Algorithms were implemented, translated, and executed on Windows 7 64-bit platform using PGI Accelerator compiler, version 12.8 [17]. The execution time of sequential implementations was measured using high-resolution, Windows API timers. OpenACC implementations were timed using built-in options of PGI compiler which uses precise GPU timers. As input datasets, we used BMP images of different sizes, from several kilobytes up to tens of megabytes.

Fig. 2. shows the observed speedup on the GPU compared to the CPU for the image thresholding algorithm. OpenACC implementation achieves 30 to 50 times speedup, as this algorithm is embarrassingly parallel and suitable for execution on manycore architectures. This is comparable to the speedup of CUDA implementation we described in [18].

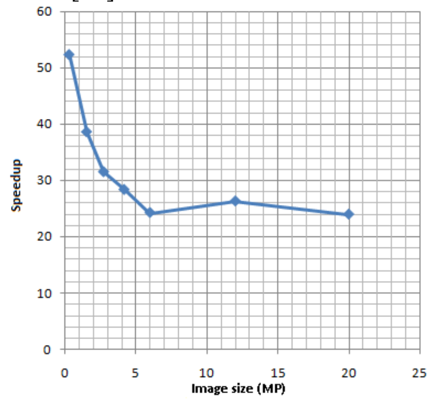

Fig. 2. Observed speedup of threshold algorithm.

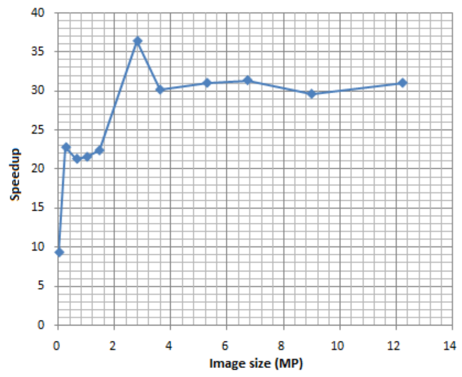

Fig. 3. Observed speedup of rescaling algorithm.

The results of image rescaling algorithm are shown in Fig. 3. The size of input image is not significant for this algorithm, but the size of output, rescaled image. The observed speedup of 30x is constant for larger images, while some performance drop is observed for smaller images, mostly because of memory transfers and parallel overheads imposed by OpenACC.

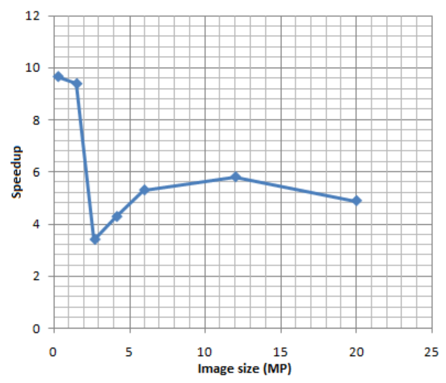

Fig. 4. Observed speedup of blurring algorithm.

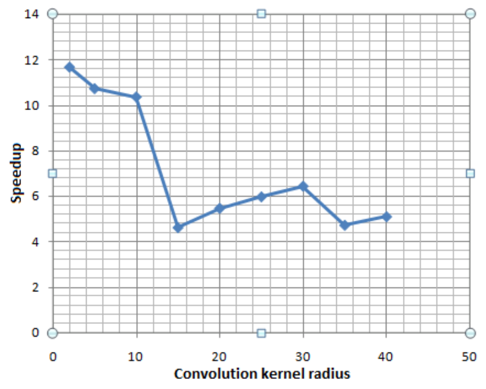

Fig. 5. Performance of blurring algorithm with constant image size and different radiuses of convolution kernel.

Blurring algorithm was the most complex that we implemented, and observed speedups are smaller, mostly due to the high computational intensity. Still, GPU algorithm performs up to 5x better, and keeps a constant speedup for larger images. Fig. 4. presents timing results for different image sizes, and a constant radius of convolution kernel, while Fig. 5. shows the performance for a constant image size, and different radiuses of convolution kernel. These results suggest that the radius of convolution kernel should not exceed 10, as larger radiuses impose around 50% drop in performance. This is understandable, as larger radiuses require more work for one worker (thread) and as a consequence more memory accesses.

VI. Conclusion

In this paper we have presented our experience with OpenACC, new, directive-based programming model for GPUs. We used OpenACC to parallelize three sequential implementations of image processing algorithms. Our experience shows that OpenACC presents a good parallelization alternative to traditional, low-level GPU programming models, especially when legacy, sequential code exists. Observed speedups are significant, while development effort and learning curve is much less compared to manual parallelization efforts.

There are several directions for future work. Implemented algorithms could be tested with different OpenACC compilers, both commercial and open-source ones. Performance differences are expected, as compilers typically use different optimization methods. Also, it is worth comparing the development effort and performance of low-level, fine-tuned CUDA or OpenCL implementations to OpenACC implementations. Our early work [18] with CUDA implementation of image thresholding shows a comparable performance to OpenACC implementation, but for more complex applications, Reyes et al. [3] suggest that much work should be done to achieve a similar performance to native implementations. Therefore, it would be interesting to use OpenACC to parallelize more complicated image processing algorithms or scientific applications. Some results from related work show potential benefits from such an approach, with significantly lower development effort.

References

[1] D. B. Kirk, W. M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2010.

[2] *The OpenACC Application Programming Interface*, version 1.0 OpenACC Consortium, 2011.

[3] R. Reyes, I. López-Rodríguez, J. Fumero, and F. de Sande, "accULL: An OpenACC Implementation with CUDA and OpenCL Support", *Proc. of Euro-Par 2012 Parallel Processing*, Springer Berlin Heidelberg, 2012., pp. 871-882

[4] R. Reyes, I. López, J. Fumero, and F. de Sande, "A Comparative Study of OpenACC Implementations", *Jornadas Sarteco*, Elx, Spain, September, 2012.

[5] R. Reyes, I. López, J. Fumero, and F. de Sande, "Directive-based Programming for GPUs: A comparative Study", *Proc. of IEEE 14th Int. Conf. on High Performance Computing and Communication & 2012 IEEE 9th Int. Conf. on Embedded Software and Systems*, 2012. pp. 410-417

[6] R. Xu, S. Chandrasekaran, B. Chapman, "Exploring Programming Multi-GPUs Using OpenMP and OpenACC-Based Hybrid Model", *Proc. of 27th IEEE Int. Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE, 2013.

[7] B. Lebacki, M. Wolfe, D. Miles, "The PGI Fortran and C99 OpenACC Compilers,", *Cray User Group*, 2012.

[8] S. Wienke, P. Springer, C. Terboven, D. an Mey, "OpenACC - First Experiences with Real-world Applications", *Proc. of Euro-Par 2012 Parallel Processing*, Springer Berlin Heidelberg, 2012., pp. 859-870

[9] A. Hart, R. Ansaloni, A. Gray. "Porting and Scaling OpenACC Applications on Massively-parallel, GPU-accelerated Supercomputers", *The European Physical Journal Special Topics* 210, no. 1, 2012., pp. 5-16

[10] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, S. A. Jarvis, "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA", *Proc. of the 2012 SC Companion: High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012., pp. 465-471

[11] *NVIDIA CUDA C Programming Guide*, version 6.0, NVIDIA Corporaton, 2014.

[12] G. F. Diamos, A. R. Kerr, S. Yalamanchili, N. Clark, "Ocelot: a Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems", *Proc. of the 19th Int. conf. on Parallel Architectures and Compilation Techniques*, ACM, 2010., pp. 353-364

[13] R. Farber, "The OpenACC Execution Model", *Dr. Dobb's Journal*, August 2012., http://www.drdobbs.com/parallel/the-openacc-execution-model/240006334#

[14] J. Larkin, "7 Powerful New Features in OpenACC 2.0", 2014., http://devblogs.nvidia.com/parallelforall/7-powerful-new-features-openacc-2-0/

[15] *The OpenACC Application Programming Interface*, version 2.0 OpenACC Consortium, 2013.

[16] D. Phillips, *Image Processing in C: Analyzing and Enhancing Digital Images*, RandD Publications, 1994.

[17] *PGI Compiler User's Guide*, The Portland Group Inc. and STMicroelectronics Inc., 2012.

[18] M. Mišić, M. Tomašević, "Performance Analysis of the Memory Hierarchy on CUDA Graphics Processing Units", *Proc. of 56th ETRAN Conference*, ETRAN Society, 2012. (in Serbian).