OpenACC — First Experiences with Real-World Applications

Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey

JARA, RWTH Aachen University, Germany
Center for Computing and Communication
{wienke,springer,terboven,anmey}@rz.rwth-aachen.de

Abstract. Today's trend to use accelerators like GPGPUs in heterogeneous computer systems has entailed several low-level APIs for accelerator programming. However, programming these APIs is often tedious and therefore unproductive. To tackle this problem, recent approaches employ directive-based high-level programming for accelerators. In this work, we present our first experiences with OpenACC, an API consisting of compiler directives to offload loops and regions of C/C++ and Fortran code to accelerators. We compare the performance of OpenACC to PGI Accelerator and OpenCL for two real-world applications and evaluate programmability and productivity. We find that OpenACC offers a promising ratio of development effort to performance and that a directive-based approach to program accelerators is more efficient than low-level APIs, even if suboptimal performance is achieved.

1 Introduction

Due to a promising performance per watt ratio and an attractive price, HPC architectures prevailingly tend towards heterogeneous computer systems comprising general-purpose cores with attached accelerator devices. However, programming accelerators such as general-purpose graphic processing units (GPGPUs) with low-level APIs is difficult, may complicate the software design and usually couples 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.

Recent approaches promise to make the compiler responsible for many of the low-level programming tasks by offering a directive-based high-level API. As finding parts of an algorithm that can efficiently be executed on an accelerator device is still up to the programmer, these approaches do not simplify programming accelerators in general. However, they improve the development productivity and simplify code maintenance. Unifying the syntax of various directive-based approaches for accelerators with the intention to make it available across multiple vendors, a group of members of the OpenMP Language Committee published OpenACC in November 2011. OpenACC enables the offloading of loops and regions of C/C++ and Fortran code to accelerators and is initiated by the companies CAPS, CRAY, NVIDIA and PGI. In this work, we present our first experiences with OpenACC applied to two real-world simulation codes from the fields of engineering and medicine that can both benefit from GPU acceleration.

Our work comprehends a performance analysis of these codes, as well as an evaluation of programmability and productivity.

The paper is structured as follows: Section 2 covers related work and Sect. 3 gives an overview of OpenACC. In Sect. 4, the two real-world applications and the undertaking for their porting to OpenACC are explained. The measured performance values with OpenACC are evaluated in Sect. 5 and compared to the ones gained with OpenCL and PGI Accelerator. In Sect. 6, we discuss programmability and productivity aspects. Finally, we assess OpenACC's effort-performance ratio regarding our real-world applications in Sect. 7.

2 Related Work

The desire for general-purpose computations on GPUs caused the advance of new programming paradigms. Nowadays, the dominant GPU programming models are CUDA [13] and OpenCL [10]. Both empower the programmer to exploit performance from the accelerator by porting code to GPU kernel functions at a low level. CUDA is coupled to NVIDIA GPUs, while OpenCL as a standard is portable across vendors and targets different hardware architectures. As using low-level APIs often results in an unproductive development process due to repeatedly written code portions and error-prone programming [15], several directive-based approaches for accelerator computing have been proposed in which compilers undertake the implementation guided by hints from the user. OpenACC [7] is an industry standard for this directive-based accelerator programming that may contribute to the specification of OpenMP for accelerators [2].

The Portland Group provides the PGI Accelerator programming model [14] for C and Fortran which enables compiler-aided and directive-based work offloading to NVIDIA GPUs and specifies a broad range of features. It served as the foundation for the OpenACC specification. Furthermore, CAPS has previously established its hybrid multicore parallel programming (HMPP) environment [6] providing directives to declare codelets, which are functions suitable for hardware acceleration, and targeting a variety of accelerators. Intel also relies on directives to offload code to its Many Integrated Core (MIC) accelerator [11]. hiCUDA [9] defines a high-level abstraction of CUDA with kernel directives, data transfer clauses and function calls. While supporting CUDA-specific concepts, hiCUDA leaves more responsibilities to the programmer than OpenACC. OpenMPC [12] approaches the generation of CUDA code by translating OpenMP regions. Additional directives control CUDA-related parameters and the compiler may find an appropriate tuning configuration for the program. It is also restricted to NVIDIA GPUs due to the CUDA affiliation. The Barcelona Supercomputing Center has developed the StarSs programming model [1] which provides extensions to the OpenMP language to exploit several architectures, for instance GPUs. It focuses on OpenMP tasks and their distribution to different targets during runtime by forcing the programmer to specify all data input and output dependencies of a task.

Since the productive development is one main issue of directive-based models, we examine the programmability and productivity with OpenACC. Only few studies consider productivity aspects. While [3] provides a general overview of coding effort, [8] looks mainly at the effort from a computing center's point of view. In our previous work

[15], we concluded that the PGI Accelerator model has a good effort-performance ratio and [2] also approaches productivity with respect to a ratio estimation of performance to time effort. In this work, we do not only elaborate on productivity in general, but also specify the number of modified code lines as an indication of the development effort.

3 OpenACC Overview

The directive-based OpenACC API for C/C++ and Fortran delegates the responsibility for low-level GPU programming tasks to the compiler, while providing portability across operating systems, host CPUs and accelerator devices. Concerning accelerator types, up to now, the existing OpenACC implementations only support NVIDIA GPUs. In this section, we provide a brief overview of OpenACC, point out the most important use cases with respect to GPUs and map the terminology to the OpenCL nomenclature.

The OpenACC API assumes a host-directed execution model in which the main program runs on the host and compute-intensive regions are offloaded to an attached accelerator. The memory model is based on separate host and device memories which do not synchronize automatically. GPU devices implement a weak memory model that prevents coherence between operations on different compute units and enables coherence within the same compute unit only by using explicit synchronization.

The execution and data management is guided by the programmer using OpenACC directives. Some basic constructs are illustrated in Listing 1.1.

Listing 1.1. Brief use case of basic OpenACC constructs

```
//Initialization: |x[i]| < 1, i=0,...,size-1
#pragma acc data copy(x[0:size]) //Data movement to/from device
{   while(error > eps) {
        error = 0.0;
#pragma acc parallel present(x[0:size]) //Kernel execution
#pragma acc loop gang vector reduction(+:error) //Loop schedule
        for(int i=0; i<size; ++i) {
            x[i] *= x[i];
            error += fabs(x[i]);
}   } }
```

The most important directives are `parallel` and `kernels` which describe regions of code to be accelerated `async`hronously or synchronously. Here, we focus on the `parallel` directive due to some restrictions on the `kernels` construct in the recent implementation. A `parallel` region maps to an OpenCL kernel function which can be enqueued for execution on the device in an n-dimensional range of work-items. To improve the performance, it is possible to specify the so-called number of `gangs`, the number of `workers` or the `vector_length`. The terms `gang` and `vector` correspond to work-groups and work-items (usually) within a work-group, respectively, in OpenCL. A `worker` defines a certain union of work-items, i.e. a *warp* on CUDA architectures. Within `parallel` regions, a `loop` directive instructs the worksharing of a loop among the accelerator's workers. The programmer can insert additional clauses in `parallel`, `kernels` or `loop` directives to optimize or correct the implicit data management chosen by the compiler. Furthermore, the data movement can be decoupled from these compute regions by using an enclosing explicit `data` region. Corresponding data clauses can specify the kind and direction of data movement, e.g. `copyin` or `copyout`. It is also possible to `create` arrays only on the device, define device `private` data, tell the compiler that data is already `present` on the device or specify different kinds of `reductions`. Since the user has to manually manage the data coherence between host and device, the executive `update` directive can be applied to synchronize the separate memories of host and device. The GPU's memory hierarchy also often supports a low-latency local memory for which the compiler may optimize the program. To guide the compiler, the developer can apply the `cache` directive which specifies (sub-)arrays to be stored in this fast memory.

Besides directives, the OpenACC API provides runtime library calls and environments variables too. For instance, library calls can gather information about the device, initialize it or allocate data on the device.

4 Applications

For our investigations on performance, productivity and programmability of OpenACC, we chose two real-world applications from the fields of engineering and medicine. In the following, we point out their contributions to their domains, describe implementation relevant features and explain how we carried out implementations with OpenACC.

4.1 Simulation of Bevel Gear Cutting

The engineering application *KegelSpan* [4] written in Fortran and developed by the Laboratory for Machine Tools and Production Engineering (WZL) at RWTH Aachen University is a 3D simulation software of the bevel gear cutting process and a leading-edge application in the automotive industry. It aims at minimizing the number of tool changes in the production process of bevel gears and contributing to a cost-efficient manufacturing by enabling a detailed tool load and wear analysis. The module under investigation computes the intersection of tool and gear. It contains one loop nest where outer and inner loop each iterates tens of thousands times (small dataset). The inner loop of the nest contains dependencies due to a minimum computation which is needed for the key value of chip thickness. The intersection module has to be executed repeatedly to optimize the manufacturing parameters.

We approached the *basic* implementation of the intersection module with OpenACC analogously to the one with PGI Accelerator in our previous work [15]: We distributed the work of the outer loop amongst all work-items and executed the inner loop serially. For that, we mainly applied a `parallel` region with numerous data clauses and a `loop` directive with `gang vector` schedule. Using the `vector_length` attribute, we optimized the number of work-items within a work-group. A second variant (*restruct*) comprises a restructured data format for an optimized, coalesced data access and serves for comparing performances rather than development efforts of the different programming models. In comparison to the basic version, just the adaption of OpenACC data clauses was necessary. Furthermore, we examined a more complex type of parallelism by distributing the outer loop to the work-groups and the inner loop to the work-items within a work-group. This approach required the addition of a minimum reduction, denoted by `reduction(min:varlist)`, to the inner loop due to the chip thickness computation. However, the position of the minimum, i.e. the array index, is needed as well. Since manual extraction of this index was tedious and delivered low performance, we omitted a detailed analysis. In OpenCL, we further leveraged the work-group's local memory (*locMem*) by storing intermediate data and input data that is needed multiple times in the fast software cache of the GPU (compare [15]). A similar approach was possible neither with PGI Accelerator nor with OpenACC, but the results gained with OpenCL show the potential of the caching technique in Sect. 5.1.

4.2 Neuromagnetic Inverse Problem

The second application comes from the field of medicine or more precisely magnetoencephalography (MEG). In MEG, the magnetic field induced by the current density inside the human brain is measured outside the head. To reconstruct the focal activity in the brain, the neuromagnetic inverse problem can be solved by means of a minimum p-norm solution. Since this unconstrained nonlinear optimization problem is challenging in terms of computational efficiency and accuracy effecting the convergence behavior [5], first- and second-order derivatives are computed by automatic differentiation (AD) in the software package of Bücker, Beucker and Rupp [5]. The software package is implemented primarily in MATLAB. To enable AD combined with parallel computing, the objective function of the optimization problem, as well as its first- and second-order derivatives are written in C. For our investigations, we concentrate on these three kernels which include the computations of matrix-vector products using a matrix of dimensions 128×512000. Additionally, the matrix can be divided into a big dense and a small sparse part. Each kernel contains a single loop or loop nest with summation reductions.

The MATLAB program calls the kernels about thousand times during the optimization process (simple configuration). For simplification, we established a C framework that mimics the original call hierarchy: Implemented with an explicit `data` region, the matrix is copied to the accelerator once per optimization run and then each kernel is called thousand times. The operands needed for the computations are copied into or `updated` on the device and the results are transferred to the host in between and at the end of the explicit data region. Porting the three kernels with OpenACC to the GPU, we first implemented a *basic* approach again. That means that only the outer loops run in parallel and the inner loops are executed serially. Each kernel consists of two or more `parallel` regions to distinguish at least between matrix-vector multiplications, vector-vector operations and summation reductions. Additionally, the implementations of first- and second-order-derivative evaluations require resolving race conditions. Therefore, intermediate values are stored in auxiliary arrays and the access to these arrays is globally synchronized by creating an additional parallel region. A loop interchange is applied as well. The corresponding basic PGI Accelerator implementation looks similarly. In a second variant (*l2par*), we added a level of parallelism to the OpenACC kernels which distributes the outer loops to the `gangs` and runs the inner loops in `vector` mode. This approach needed the usage of reduction clauses on outer and inner loops. A similar approach was not possible with PGI Accelerator due to a limitation in its implementation. The third approach aims at leveraging the cache of the GPU and improving the data access pattern by chunking the matrix-vector multiplications into blocks of size `vector_length`. Each block is mapped to a work-group, whereas the work-items within a work-group execute the multiplications row-wise. With PGI Accelerator, the *blocked* version was implemented analogously except that only one level of parallelism could be applied to the loop nests that do not access the matrix. The OpenCL version also employs a blocked matrix-vector multiplication. Furthermore, it is highly optimized with respect to the usage of device local memory and host pinned memory, asynchronous data transfer and kernel execution, the specification of constant values as preprocessed macros or loop unrolling. Loop unrolling is also applied to all PGI and OpenACC versions. The OpenACC compiler automatically unrolls loops within accelerated code regions or can be guided by the `#pragma unroll(size)`. Although the PGI Accelerator API includes an `unroll` clause, here, it is ignored by the compiler. Therefore, we had to unroll most of the loops manually up to a level of 32 to achieve comparable performance. `cache` and `async` optimizations could not yet successfully applied with both directive-based models.

5 Performance Evaluation

In this section, we present performance results of the OpenACC implementations of both applications and compare them to implementations made with OpenCL[1] and PGI Accelerator. Since tool support across all three programming models is limited, our evaluation also contains assumptions of result explanations. This analysis is based on NVIDIA's Visual Profiler. All reported runtimes include the accelerator's setup time, data transfers between host and device, kernel execution times and the overhead introduced by the need of manual management (OpenCL) or adapting the data structure. Each runtime is the minimum value of five program runs.

For all measurements, we used an NVIDIA Tesla C2050 GPU with ECC enabled and CUDA toolkit 4.0. The host system[2] on which the OpenACC results were gathered consists of one AMD Magny-Cours 12-core processor and runs SUSE Enterprise Server 11. There, we use the Cray 8.1.0 compiler[3] with the optimization flag *-O3*. For the results gained with OpenCL and PGI Accelerator, we worked on an Intel Westmere 4-core host processor and Scientific Linux 6.1. The OpenCL and PGI Accelerator implementations are compiled with the Intel 12.1.2 compiler and the PGI 12.3 compiler (*-ta=nvidia,4.0,cc20,fastmath(,nofma)*), respectively.

[1] OpenCL and CUDA performance results are comparable in almost all our application versions.

[2] Since this machine is an experimental system from Cray, performance should be better on the Cray XK6 product.

[3] At time of developing, Cray provided the first OpenACC release. The used Cray compiler comprises an early implementation of OpenACC which does not contain all features of OpenACC yet. Furthermore, the performance is likely to increase with future releases. First partly implementations by PGI and CAPS were released in March and May 2012, respectively, and will be subject to further investigations.

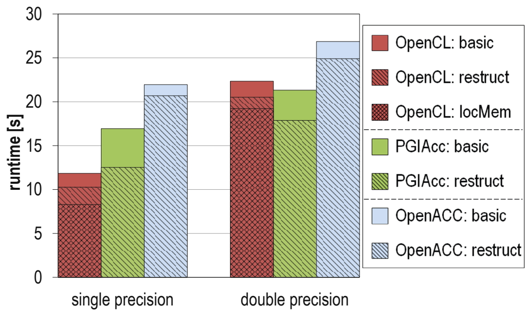


Fig. 1. Runtimes of OpenCL, PGI Accelerator and OpenACC (engineering application)

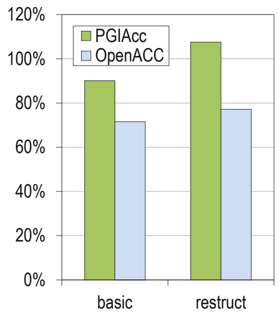


Fig. 2. Percentages of OpenCL performance in double precision

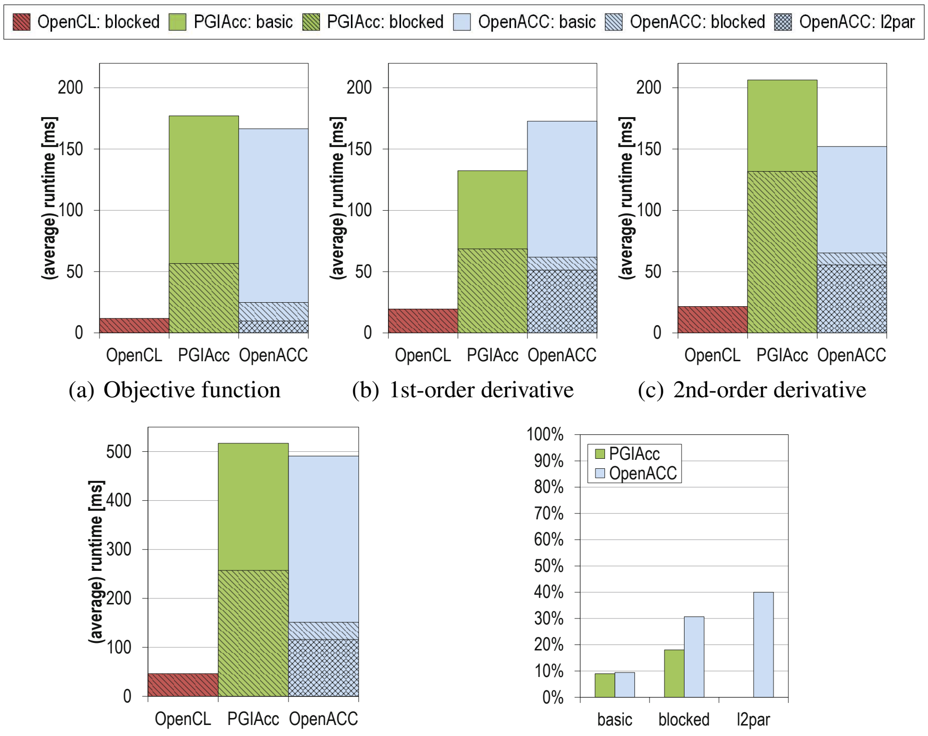
5.1 Simulation of Bevel Gear Cutting

The results for the engineering application can be found in Fig. 1. Each color represents one programming model, whereas the patterns correspond to different versions as mentioned in Sect. 4. For single and double precision, the absolute difference between PGI Accelerator and OpenACC runtimes of the *basic* and the *restructured* version remain approximately the same. A significant fraction of this difference is introduced by a great amount of L2 read misses in the OpenACC programs which extends the one of PGI Accelerator by a factor of 1.5. These L2 read misses must be resolved by long-latency global memory accesses that hurt the performance. The PGI compiler feedback shows that constant memory is used, whereas the OpenACC compiler does not elaborate on this. We assume that the constant cache is ignored by the current OpenACC compiler which would lead to the high number of L2 read misses. Additionally, the best-effort PGI Accelerator versions using double precision omit the generation of FMA/MAD instructions. However, neither the OpenCL nor the OpenACC compiler provide an easy way to disable FMA operations without losing other optimizations to verify this result. Figure 2 illustrates the double precision results in percentages. Here, the runtime of the best-effort OpenCL implementation serves as reference. With the *restruct* version, PGI Accelerator outperforms OpenCL, and OpenACC achieves a considerable fraction of approximately 80 % of OpenCL.

Furthermore, the OpenCL bars in Fig. 1 show that the usage of local memory was beneficial for this application. The PGI Accelerator API provides a cache clause for the loop schedule, but the compiler takes it as a hint instead of a rule. In this case, the compiler apparently cannot apply the hint successfully. In OpenACC, the cache directive is not yet fully implemented in the Cray compiler. Given Cray's ongoing implementation work, we hope to examine OpenACC's cache capabilities in the near future.

5.2 Neuromagnetic Inverse Problem

For the neuromagnetic inverse problem, we measured the runtime of the objective function evaluation, the first- and second-order-derivative computations and the whole program. The latter contains an equal number of calls to the three functional units as



(a) Objective function (b) 1st-order derivative (c) 2nd-order derivative

(d) Overall runtimes (e) Percentages of OpenCL performance

Fig. 3. Comparison of OpenCL, PGI Accelerator and OpenACC in double precision (medical application)

approximation of the actual optimization process. All runtimes represent an average over 1000 iterations including data transfers. Computations are done in double precision only due to the high impact of numerical inaccuracy.

Figure 3 illustrates the results of the various implemented versions as described in Sect. 4. The colors refer to the programming model and the bar pattern to the implementation variants. In all cases, the *basic* versions have the longest runtimes and are outperformed by the corresponding *blocked* versions. Although the blocking algorithm in OpenCL, PGI Accelerator and OpenACC is almost the same, the OpenCL versions contain much less read requests to the L2 cache and the device global memory. This is mainly due to the coalesced prefetching of matrix and vectors to the on-chip local memory and the consequent, cache-optimized computations that we implemented with OpenCL. Contrary, both directive-based models lack either of an implementation or a successful recognition of the caching clause. The runtime difference is intensified by an additional global synchronization in the directive-based models. With OpenACC, a performance improvement is surprisingly gained by using the algorithmic-simpler version *l2par* with two levels of parallelism. For the objective function evaluation (see Fig. 3(a)), the OpenACC performance even catches up with the OpenCL performance,

Table 1. Number of modified source code lines in host/kernel code with respect to the serial versions of the engineering application ($\sim$ 150 kernel code lines) and medical application ($\sim$ 100 kernel code lines)

	engineering			medical		
	basic	restruct	locMem	basic	blocked	l2par
OpenCL	106/35	183/39	183/58	-	330/300	-
PGI Acc	0/3	84/14	-	9/121	12/109	-
OpenACC	0/3	84/14	-	9/31	12/85	9/37

both containing an equal number of global memory accesses. In contrast, the runtimes of evaluating first- and second-order derivatives (Figs. 3(b) and 3(c)) are about 2.5 times higher than the ones from OpenCL. Here, OpenACC's number of global memory reads and writes exceeds the ones of OpenCL, possibly due to less local-memory usage. In general, the OpenCL implementations may also perform better due to asynchronous data transfer and kernel execution, extensive loop unrolling, usage of pinned memory or specification of constant values as preprocessed macros. Figure 3(e) presents the overall performance in percentages of the OpenCL version. The best-effort PGI Accelerator and OpenACC versions achieve about 20 % and 40 %, respectively.

Comparing OpenACC to PGI Accelerator, in most cases, the OpenACC versions run faster. We can see by profiling that OpenACC calls CUDA's internal 32-bit alignment routine. An optimal alignment may be the main reason for less memory accesses and the better performance. Furthermore, loop unrolling was essential for improving performance. While the OpenACC compiler automatically adapts the unroll level nicely with `#pragma unroll(size)`, the PGI compiler ignores any `unroll` clause so that a manual and assumingly suboptimal loop unrolling must be applied. PGI's only performance gain (see Fig. 3(b)) results from a better cache access pattern evoked by the reuse of the *pow* function.

6 Programmability and Productivity

Good programmability is the foundation of a productive development process. In this section, we examine OpenACC's ease-of-use in comparison with PGI's Accelerator model and OpenCL. We discuss OpenACC's capabilities, restrictions and our suggestions for improvement to decrease development effort.

Both applications show that the current OpenACC implementation allows successful porting to GPUs in a productive way. Table 1 lists the number of modified source code lines for each kernel and programming model. These values indicate how much time a programmer spent to port the application to GPUs and how maintainable the ported code is for further development.

The main features of OpenACC like data movement and loop acceleration can intuitively be applied, just as with PGI Accelerator. An inexperienced user can rely on an automatic loop scheduling, whereas more experienced users can tune it by using different levels of parallelism or by specifying the sizes of parallel chunks. Here, OpenACC is even capable of explicitly managing *warps*, while OpenCL and PGI Accelerator do not provide this degree of freedom. On the other hand, OpenACC is restricted to a single gang or vector dimension, whereas OpenCL and PGI Accelerator offer multiple dimensions to intuitively map e.g. a matrix to a two dimensional working space.

Synchronization. We find the automatic synchronization between work-items at any level of parallelism that is entirely within a work-group convenient. However, to synchronize between loops that are located within the same parallel region and whose work is distributed with different schedules, the user has to manually synchronize the data by splitting the parallel region. An additional executable directive that acts like a global barrier might be useful to circumvent manual splitting of parallel regions. An use-case of this directive would be the synchronization of GPU-local data after its initialization. [9] provides the `singular` directive for the purpose of compact initialization.

Reduction. In OpenCL, a manual implementation of the reduction operation often leads to poor performance. In contrast, the PGI compiler automatically recognizes a reduction (sometimes adding auxiliary variables is needed) and it creates well-optimized machine code for it. In OpenACC, the user has to specify the `reduction`, but the freedom to choose any typical kind of reduction (e.g. `min` reduction) and any scalar variable to reduce without adding intermediate variables. Here, the addition of user-defined reductions to the OpenACC standard would further improve the development. In our case study, the collection of the minimum along with its (array) index would be an appropriate application.

Function Calls. A major restriction of both directive-based models is that non-inlined function calls in accelerated code are not supported at the moment. Function calls in our investigated applications could be resolved by explicitly inling them. However, this is not bearable for bigger software packages. PGI Accelerator also does not support this feature currently, but in general, it seems possible to integrate it in high-level APIs [9].

Atomics and Critical Region. OpenMP-like atomics and critical regions for OpenACC would help to avoid race conditions. For instance, detecting a critical region, the compiler could undertake the needed addition of an auxiliary array, the split of parallel regions and the interchange of loops. The compiler could apply the most efficient implementation and prevent programming errors.

Asynchronous Data Transfers. Asynchronous data copies would be beneficial for OpenACC, not only regarding `update` directives, but also explicit `data` regions.

Multiple GPUs. To use multiple GPGPUs on a single host, the programmer must explicitly specify the particular device for work-offloading and manually manage data synchronization between the devices and the host using a low-level API. This distribution of workload and data could also be left to the compiler and runtime in future.

At the time of writing, several features of OpenACC were not yet fully implemented in the Cray compiler, but look promising. The `kernels` construct (combined with guided `loop` execution) will hopefully improve the productivity of accelerating code regions containing multiple loops. Aiming at increasing performance, the `async` clause of `parallel`, `kernels` and `update` constructs will be applied to the investigated medical application. Employing multiple command queues, it will enable the start of separate parallel regions simultaneously to independently compute matrix-vector multiplications and vector-vector operations and the overlap of kernel execution and data movement. In OpenCL, the usage of the GPU's software cache improved the performance of both software packages. The application of OpenACC's `cache` clause may also lead to further acceleration in the future. Moreover, we will examine the combination of OpenACC and MATLAB. With OpenCL, the communication to MATLAB was employed by exchanging the OpenCL context. However, at the moment, it is unclear whether a realization with OpenACC is possible.

7 Conclusion

In the context of two real-world applications, we examined the performance, programmability and productivity of OpenACC in comparison to OpenCL and PGI Accelerator.

With OpenACC, we find that the performance of the moderately complex kernel of the simulation software for bevel gear cutting is about 80 % of the best-effort OpenCL performance regarding double precision computations, although the implementation of OpenACC that we used is still incomplete. This result matches the expectations of the performance of a directive-based programming model. In contrast, the OpenACC performance of the more complex medical program is only approximately 40 % of the best-effort OpenCL implementation. Although this value is rather distressing at first sight, we still believe that OpenACC is a promising approach: We assume that the loss of performance is mainly due to the current lack of the ability to leverage the local memory of the GPU intensified by our manually implemented global synchronization to prevent race conditions. These trade-offs may be eliminated by the ongoing implementation work on OpenACC or by introducing additional directives. Moreover, it must be taken into account that the highly optimized OpenCL code is quite verbose, hard to read and requires 630 modified code lines, whereas the best OpenACC version only needed 46 modified lines of code. Basing on these numbers, the OpenACC's ratio of development effort to performance is encouraging.

Thus, in terms of programmability and productivity, the OpenACC API is generally convincing. It can be intuitively applied if the programmer has certain knowledge of the accelerator's hardware architecture. If adopting features like device function calls, user-defined reductions or critical regions, the programming efficiency may be further improved while simultaneously reducing sources of errors.

In our view, the move to directive-based accelerator programming is essential for the further growth and acceptance of accelerator devices. To this end, OpenACC is an important step as it standardizes a directive-based API for accelerators for the first time. It is intended to integrate the feedback and the lessons learned from OpenACC into the OpenMP specification. The inclusion of corresponding functionality into the OpenMP standard will be technically demanding though and may slow down further development of the standard. However, from the user's point of view, OpenMP for accelerators is certainly promising.

References

1. Ayguadé, E., Badia, R., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzàlez, M., Igual, F., Jiménez-González, D., Labarta, J., Martinell, L., Martorell, X., Mayo, R., Pérez, J., Planas, J., Quintana-Ortí, E.: Extending OpenMP to Survive the Heterogeneous Multi-Core Era. International Journal of Parallel Programming 38, 440–459 (2010), doi:10.1007/s10766-010-0135-4
2. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for Accelerators. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 108–121. Springer, Heidelberg (2011)
3. Bordawekar, R., Bondhugula, U., Rao, R.: Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. Technical report, IBM Res. Division (2010)
4. Brecher, C., Gorgels, C., Hardjosuwito, A.: Simulation based Tool Wear Analysis in Bevel Gear Cutting. In: International Conference on Gears, Düsseldorf. VDI-Berichte, vol. 2108.2, pp. 1381–1384. VDI Verlag (2010)
5. Bücker, M., Beucker, R., Rupp, A.: Parallel Minimum p-Norm Solution of the Neuromagnetic Inverse Problem for Realistic Signals Using Exact Hessian-Vector Products. SIAM Journal on Scientific Computing 30(6), 2905–2921 (2008)
6. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A Hybrid Multi-core Parallel Programming Evironment. In: First Workshop on General Purpose Processing on Graphics Processing Units (2007)
7. CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. The OpenACC Application Programming Interface, v1.0 (November 2011)
8. Hacker, H., Trinitis, C., Weidendorfer, J., Brehm, M.: Considering GPGPU for HPC Centers: Is It Worth the Effort? In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) Facing the Multicore-Challenge. LNCS, vol. 6310, pp. 118–130. Springer, Heidelberg (2010)
9. Han, T.D., Abdelrahman, T.S.: hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems 22(1), 78–90 (2011)
10. Khronos OpenCL Working Group. The OpenCL Specification, v1.1.44 (2011)
11. Koesterke, L., Boisseau, J., Cazes, J., Milfeld, K., Stanzione, D.: Early Experiences with the Intel Many Integrated Cores Accelerated Computing Technology. In: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery, TG 2011, pp. 21:1–21:8. ACM, New York (2011)
12. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In: 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11 (November 2010)
13. NVIDIA. CUDA C Programming Guide, v4.0 (2011)
14. The Portland Group. PGI Accelerator Programming Model for Fortran & C, v1.3 (2010)
15. Wienke, S., Plotnikov, D., an Mey, D., Bischof, C., Hardjosuwito, A., Gorgels, C., Brecher, C.: Simulation of bevel gear cutting with GPGPUs-performance and productivity. Computer Science - Research and Development 26, 165–174 (2011), doi:10.1007/s00450-011-0158-0