# OpenACC - A Step Towards Heterogeneous Computing

**Paul Springer**

*German Research School for Simulation Sciences GmbH*
*Laboratory for Parallel Programming*
*Paul.Springer@rwth-aachen.de*

*Supervisor: Sebastian Rinke (s.rinke@grs-sim.de)*

**Abstract:** With the fast growing number of heterogeneous supercomputers, consisting of massively parallel coprocessors attached to multi-core processors, it becomes increasingly important to program these heterogeneous systems in a productive manner. Programming these coprocessors through low-level APIs such as CUDA or OpenCL is often a tedious task and may result in poor productivity. OpenACC tries to overcome this drawback by allowing programmers to annotate C/C++ or Fortran code with directives which are then translated into accelerator-specific code by the compiler. This paper summarizes OpenACC's features, its limitations and possible future directions. Moreover, I will present two case studies where I evaluate OpenACC's performance and productivity in comparison to CUDA and OpenMP.

## 1 Introduction

During the recent years we have witnessed a paradigm shift from single-core CPUs to multi-core CPUs since it was not feasible to keep increasing the frequency of single-core CPUs due to power constraints. Alongside this development, we have seen that graphics processing units (GPUs) have continuously increased their floating point performance and features (e.g. ECC support, caches) which attracted the attention of the high-performance computing (HPC) community. In the beginning of general-purpose computing on graphics processing units (GPGPU) these devices had to be programmed through 3D graphics application programming interfaces (APIs) such as Direct3D and OpenGL which was a tedious and counterintuitive way of writing general-purpose applications for GPUs. In order to improve the programmability of NVIDIA GPUs, NVIDIA introduces the *Compute Unified Device Architecture* (CUDA) [1] in 2007 which was closely followed by the release of the *Open Computing Language* (OpenCL) [2] in 2008. While CUDA is restricted to NVIDIA GPUs, OpenCL is not. However, both APIs are relatively low-level and cherished the desire for an easy-to-use and portable programming model for various coprocessors which eventually led to the development of OpenACC.

With the release of the latest Top500 list [3] it is shown that the ongoing trend towards systems consisting of nodes with multi-core CPUs attached to an accelerator/coprocessor[1] continues. As of today the Top500 list already contains 62 such heterogeneous systems. Most notably is the worlds fastest supercomputer *Titan* at Oak Ridge National Laboratory which gains about 90 percent of its performance from NVIDIA's K20 GPU accelerator. One driving force behind this ongoing trend is the good power efficiency of today's coprocessors [4], since power consumption is an important objective for future architectures as it is likely to become the limiting factor for Exascale supercomputers.

In parallel to the increasing number of heterogeneous systems we also see a growing diversity of coprocessors (e.g. NVIDIA GPUs, AMD GPUs, Intel's *Many Integrated Core Architecture* (MIC) [5], Convey's FPGA solutions, Texas Instruments' DSPs) which makes it more and more important to

[1] I use the term *coprocessor* instead of *accelerator* because the word *accelerator* has a promise to it which might not be true for all applications.

write portable code. Another difficulty with respect to coprocessors is that they add yet an additional level of parallelism to current architectures which makes writing efficient coprocessor applications even more difficult than writing "ordinary" parallel MPI/OpenMP applications. Even though CUDA and OpenCL provide the means to program coprocessors, these APIs are relatively low-level and verbose which might limit productivity. The OpenACC API [6] follows a different approach as it enables the programmer to offload work to a coprocessor by adding compiler directives to C/C++ or Fortran code. Hence, the programmer does not have to write low-level code in order to utilize the massive parallelism of today's coprocessors. This approach aims at high productivity [7] and provides portability across different CPUs, operating systems and coprocessors by delegating the responsibility of generating correct coprocessor code to the compiler.

This paper is structured as follows: Section 2 shows related work. Section 3 introduces OpenACC. Section 4 presents the performance and productivity results of two case studies. Finally, I comment on possible future directions of OpenACC in Section 5 and conclude this paper in Section 6.

## 2 Related Work

In [7], Wienke et al. give two case studies evaluating the performance and productivity of OpenACC in comparison to hand-tuned OpenCL versions on the example of two real-world applications. Their OpenACC implementations achieved a performance of 40% and 80% with respect to their OpenCL counterparts. It is worth mentioning that these case studies used an early version of CRAY's OpenACC compiler and are likely to perform better as the OpenACC compiler continues to mature. Moreover, they conclude that OpenACC is favorable in terms of productivity.

Hart et al. [8] ported an existing Fortran code of a three dimensional Poisson solver to a GPU accelerated system. Their OpenACC version is about twice as fast as their hybrid MPI/OpenMP version running on a comparable CPU-only system. Despite the lack of a comparison between OpenACC and CUDA, their results are quite promising as they only required little code restructuring to port this application to a GPU-accelerated system.

Levesque et al. [9] give a case study porting a hybrid MPI/OpenMP implementation of a turbulent combustion solver to the Titan system hosted at Oak Ridge National Laboratory. Based on their MPI/OpenMP version they stated that "changing to the new OpenACC required only a trivial change in syntax" [9]. Moreover, comparing the performance of the OpenACC version running on the GPU-accelerated system against the MPI/OpenMP version running on a comparable CPU-only system shows that the OpenACC implementation performs ≈50% faster than the CPU-only version.

## 3 OpenACC

The OpenACC [6] API specifies compiler directives and runtime API functions which enable the programmer to offload work to one or more coprocessors. These high-level directives are then translated into coprocessor code by the compiler, which hides the complexity of low-level APIs such as CUDA or OpenCL. However, OpenACC is not fully automated and requires the programmer to identify regions of the code that should be offloaded to the coprocessor. Since its introduction by NVIDIA, PGI (The Portland Group), CAPS and CRAY in November 2011, OpenACC enjoys an increasing popularity within the HPC community as it mimics the OpenMP [10] syntax (see Listing 1) and allows programmers to incrementally migrate existing C/C++ or Fortran applications to coprocessors. Moreover, OpenACC is fully interoperable with MPI and OpenMP which makes it perfectly suited for large heterogeneous clusters.

```
#pragma acc directive-name[clause [[,]clause]...]
  { structured block }
```

**Listing 1:** OpenACC syntax for C/C++.

While portability and productivity are clearly the main goals of OpenACC, it also allows programmers to target different architectures without the need to fork the development process. Even though directive based programming seems to be a promising approach, OpenACC still has to prove that its performance is comparable to its low-level API counterparts (see Section 4.3).

At the time of writing the three OpenACC compiler vendors PGI, CAPS and CRAY only support GPUs as target devices. More precisely, CAPS is currently the only vendor who supports both NVIDIA and AMD GPUs, while PGI and CRAY only support NVIDIA GPUs (see Section 5 for future direction). However, OpenACC is designed such that it can support a wide variety of coprocessors. This is possible because many current coprocessors share several architectural features [11]. Looking at NVIDIA GPUs, AMD GPUs and Intel's MIC architecture, all these coprocessors (1) can run asynchronously to the host (i.e. CPU), (2) have their own shared-memory space and (3) provide a large number of processing elements (PEs). These PEs run in parallel and can issue SIMD instructions. In terms of NVIDIA's Kepler architecture these PEs would map to streaming multiprocessors (SMXs) [12].

## 3.1 Execution Model

The OpenACC execution model introduces three execution units: *gang*, *worker* and *vector*. With respect to NVIDIA's GPUs it is reasonable to think of a gang as a threadblock, worker as a warp and vector as a thread [12]. Similar to CUDA and OpenCL, OpenACC does not offer an efficient way of synchronization between gangs. This preserves high performance and enables these APIs to transparently scale to arbitrary large numbers of PEs which is important for scaling of future architectures. Moreover, OpenACC offers the opportunity to run device calls (so-called *kernels*) asynchronously to the main program which runs on the host. The programmer is able to synchronize these calls through runtime API functions (see Section 3.4).

## 3.2 Memory Model

The OpenACC memory model assumes that the host and the coprocessor have separate shared-memory spaces, meaning that the host is typically not able to access the device memory directly (and vice versa). Prior to the execution of a kernel (1) memory needs to be allocated on the device and (2) the corresponding input data has to be transferred to the coprocessor. These transfers typically use direct memory access (DMA) [6] and are limited by the bandwidth of the PCIe bus, which is slower than the interconnect between the PEs and their on-device global memory. (3) After the execution of the kernel on the device, (4) the output data needs to be copied back to the host and finally (5) the allocated memory can be freed. Hence, it is good practice to choose computationally intensive loops for acceleration by the coprocessors since these loops typically compensate these additional data transfers.

Moreover, OpenACC uses a weak memory model that does not support memory coherence between operations executed by different PEs [6]. Programmers have to pay attention to this memory model since it might affect the correctness of the application.

Furthermore, some coprocessors support software managed caches (low-latency, high-bandwidth memory) which can be utilized by the compilers. Programmers can improve the utilization of these caches by specifying special directives.

| Environment Variable | Description |
|---|---|
| ACC_DEVICE_TYPE | Sets the desired coprocessor (e.g. NVIDIA). |
| ACC_DEVICE_NUM | Selects a coprocessor ID. |
| ACC_NOTIFY | Print additional information on each kernel launch. |
| PGI_ACC_TIME* | Print timings for each kernel and data transfers. |

**Table 1:** OpenACC environment variables. *only available for PGI compilers.

| #pragma acc kernels | #pragma acc parallel | #pragma acc loop | #pragma acc data |
|---|---|---|---|
| if() | if() | collapse(int) | if() |
| async([int]) | async([int]) | gang [(int)] | |
| copy(list) | copy(list) | worker[(int)] | copy(list) |
| copyin(list) | copyin(list) | vector[(int)] | copyin(list) |
| copyout(list) | copyout(list) | seq | copyout(list) |
| create(list) | create(list) | independent | create(list) |
| pcopy(list) | pcopy(list) | reduction(op:list) | pcopy(list) |
| pcopyin(list) | pcopyin(list) | | pcopyin(list) |
| pcopyout(list) | pcopyout(list) | | pcopyout(list) |
| pcreate(list) | pcreate(list) | | pcreate(list) |
| deviceptr(list) | deviceptr(list) | | |
| | num_gangs(int) | | |
| | num_workers(int) | | |
| | vector_length(int) | | |
| | private(list) | private(list) | |
| | firstprivate(list) | | |

**Table 2:** Clauses for some OpenACC directives. *[ ]* denotes optional parameters.

## 3.3 Basic Features

In the course of this section, I describe the most salient features of OpenACC, which are sufficient to implement a basic OpenACC application in C/C++. For more advanced OpenACC features please refer to Section 3.4.

OpenACC specifies environment variables (see Table 1), runtime functions (see Section 3.4) and many compiler directives (see Table 2), please refer to [6] for a full overview of supported directives.

There are conceptually two different ways to offload work to the coprocessor, namely the *parallel* region and the *kernels* region. The parallel region is similar to the OpenMP parallel region in the way that both create a parallel region that will be executed redundantly by the spawned gangs if no work-sharing construct (i.e. *#pragma acc loop*) is specified. The *loop* directive is the counterpart to OpenMP's *for* directive and it can be combined with the *kernels* or *parallel* directive (see Listing 2), much like *#pragma omp parallel for*. This combination is merely a shortcut for a parallel region directly followed by a loop construct. One important difference between OpenMP's *for* and OpenACC's *loop* is that there is no barrier at the end of each loop within a parallel region. This is not so surprising if you think of this parallel region as a single kernel [13] and recall that OpenACC does not provide synchronization between gangs. Another difference is that the current OpenACC specification does not allow a parallel or kernels region inside another parallel or kernels region.

If the compiler encounters a *#pragma acc parallel* directive, it will try to convert the succeeding structured block into a single kernel that is executed by the coprocessor. In order to identify portions of the code that are suitable for acceleration and to spot dependencies which might prohibit parallelization, the compiler has to perform sophisticated analysis of the encapsulated code. Since OpenACC is prescriptive, much like OpenMP, programmers can override the compilers choices and

```
#pragma acc parallel loop
  for(i = 0; i < n; ++i)
    x[i] = alpha * x[i] + beta * y[i];
```

**Listing 2:** Parallel loop on the example of a vector addition.

```
19, Accelerator kernel generated
    19, CC 2.0 : 18 registers; 0 shared, 80 constant, 0 local memory bytes
    20, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
19, Generating present_or_copy(x[0:n])
    Generating present_or_copyin(y[0:n])
    Generating compute capability 2.0 binary
```

**Listing 3:** PGI compiler feedback related to Listing 2.

force parallelization which might result in incorrect code.

Through out this seminar paper, I use PGI's C compiler pgcc 12.9 and an NVIDIA Quadro 6000 GPU to compile and execute the shown code snippets. In order to tell the PGI compiler to target NVIDIA GPUs with compute capability 2.0 the *-ta=nvidia,cc20* compiler flag is required. Furthermore, if the *Minfo=accel* flag is specified, the compiler provides a detailed feedback (see Listing 3) which is a quite handy feature that helps programmers to detect possible performance and correctness problems.

Listing 3 shows the compiler feedback generated for Listing 2, it informs the programmer that an accelerator kernel has been generated[2] and that this loop has been work-shared across the gangs, each with a vector-length of 256 (i.e. in CUDA terms: across multiple threadblocks each consisting of 256 threads). Furthermore, *blockId.x* and *threadIdx.x* indicate that the loop has been is scheduled in x-dimension for both threadblocks and threads. The fact that the compiler is tuning the generated code for the target architecture is shown by the vector-length of 256. More precisely, work-sharing in OpenACC is done across gangs, hence, the compiler could have chosen to launch $n$ gangs with one CUDA thread each, which would have resulted in very poor performance. As it is evident from this scheduling strategy it is typically a good choice to have stride-1 accesses (i.e. accesses to consecutive elements in memory) in the inner-most loop and use vector parallelism for this loop such that the compiler can map consecutive elements to consecutive CUDA threads. We will see the performance implication of these choices in Section 4.

This small example already depicts some advantages of OpenACC over low-level APIs: (1) Data dependencies/transfers are automatically detected and the programmer does not have to explicitly deal with memory allocation, deallocation and transfers, even though this is possible (see Section 3.4). (2) The loop is automatically strip-mined into chunks of 256 and the compiler takes care of the boundary cases (i.e. if $n$ is not divisible by 256). (3) The compiler is able to choose the best scheduling strategy for the accelerated region based on it's analysis of the code and the targeting coprocessor. This is especially important for two reasons, first, it releases the programmer from the burden to specify these parameters, which typically requires some knowledge of the underlying architecture and second, it allows to target different architectures by merely recompiling the application. It is worthwhile mentioning that this does not mean that the same code can perform equally good on a variety of coprocessors. More precisely, in order to achieve high performance for a specific coprocessor typically some code refactoring is necessary as we will see in Section 4.1.

The *acc kernels* directive specifies a region that will be compiled into potentially many kernels which can have different scheduling policies (see Section 3.4) and will be executed in-order on the device. Typical candidates for these kernels are nested loops within the kernels region (see Listing 4).

[2]If you don't get this feedback, your parallel region has not been parallelized!

```
#pragma acc kernels
{
  for(i = 0; i < n; ++i)
    for(j = 0; j < n; ++j)
      x[i] = alpha * x[i] + beta * y[j];

  for(i = 0; i < n; ++i)
    x[i] = alpha * x[i] + beta * y[i];
}
```

**Listing 4:** Kernels region.

```
19, Generating present_or_copy(x[0:n])
    Generating present_or_copyin(y[0:n])
    Generating compute capability 2.0 binary
21, Loop is parallelizable
22, Complex loop carried dependence of '*(x)' prevents parallelization
    Loop carried dependence of '*(x)' prevents parallelization
    Loop carried backward dependence of '*(x)' prevents vectorization
    Inner sequential loop scheduled on accelerator
    Accelerator kernel generated
    21, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    22, CC 2.0 : 23 registers; 0 shared, 80 constant, 0 local memory bytes
25, Loop is parallelizable
    Accelerator kernel generated
    25, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
        CC 2.0 : 15 registers; 0 shared, 80 constant, 0 local memory bytes
```

**Listing 5:** PGI compiler feedback related to Listing 4.

Looking at Listing 4, the PGI compiler is not able to vectorize either loop without further assistance of the programmer. This was due to the fact that the compiler was not certain that the pointers $x$ and $y$ do not overlap (i.e. point at the same memory locations), which would prevent parallelization due to loop carried dependencies. There are different approaches to resolve this issue. One solution would be to add a *loop* directive with an *independent* clause to all independent loops (i.e. in front of both $i$ loops). Another common solution is to add the *restrict* keyword to the declaration of the $x$ pointer (i.e. *double * restrict x;*).

The compiler feedback for Listing 4 is shown in Listing 5, it indicates that the parallelization of the inner $j$ loop was not possible due to loop carried dependencies. Since this can have major performance implications, I show how to resolve these kinds of dependencies in Section 4. For the time being, the outer loop is distributed among gangs with a vector-length of 128. Furthermore, the compiler was smart enough to realize that the data is only required at the end of the kernels region hence there is no communication between host and device after completion of the first kernel.

For a more detailed discussion on the differences between the *parallel* and *kernels* directive, please refer to the well written article by Micheal Wolfe [13].

## 3.4 Advanced Features

This section introduces some of the more advanced OpenACC features which help to improve the performance of the application but are not necessary to write correct OpenACC code. The interested reader is referred to [6] for further information.

```c
#pragma acc parallel copy(x[0:n]), copyin(y[0:n])
    for(i = 0; i < n; ++i)
        x[i] = alpha * x[i] + beta * y[i];
```

**Listing 6:** Verbose parallel loop on the example of a vector addition.

Writing efficient coprocessor code is not trivial at all, however, there are some guidelines which generally apply to coprocessor programming [12]:

- Avoid communication between host and device.
- Keep the coprocessor busy.
- Reuse data to avoid memory bandwidth bottlenecks.

One of OpenACC's features to address these issues is the *data* region (see Table 2). It enables programmers to keep data on the coprocessor and avoid needless data transfers and reallocation. For code examples please refer to Section 4 where I make use of this feature in both of the case studies. In addition to the *data* region there are various data clauses (see Table 2) which can be used to specify data transfers explicitly. This is of special interest if the compiler makes poor decisions for data transfers or if it is not possible to determine the sizes of the data automatically. A more verbose version of Listing 2 where I explicitly list the data clauses is shown in Listing 6. The syntax for the arguments to the data clauses in C are: *pointer*[<*lowest index of subarray*>: <*number of subarray elements*>].

Another interesting feature of OpenACC is its interoperability with low-level APIs such as CUDA. Programmers can hand tune some kernels using CUDA and pass results to the OpenACC-generated kernels via the *deviceptr* clause. For instance, a hand-tuned CUDA kernel processes an array $x$, this array can be passed to another kernel by specifying *deviceptr(x)* which indicates that $x$ is already present on the coprocessor and does not require further data transfers. This feature also allows to utilize existing CUDA libraries in OpenACC applications.

A parallel or kernels region can execute asynchronously to the host if the *async([int])* clause is present. The optional argument can be used by runtime functions such as acc_async_test() and acc_async_wait(). Moreover, all asynchronous activities with the same optional parameter will be executed in-order on the device, this is comparable to a stream in CUDA.
These asynchronous features enable the programmer to utilize the host and the coprocessor at the same time. This feature is essential for OpenACC's multi-coprocessor support. Programmers can interact with multiple coprocessors via runtime functions like: *acc_get_device_num()*, *acc_set_device_num()*, *acc_set_device_type()*, *acc_get_device_type()*, *acc_get_num_devices()*. However, current implementations of PGI's OpenACC-enabled compilers are not able to use multiple coprocessors for the same accelerator region [14].

Another important feature is the possibility to explicitly set the scheduling policy. This allows the programmer to decide whether a loop should be scheduled among gangs, workers or vectors. Listing 7 shows a verbose OpenACC implementation of a matrix-vector multiply $y = Ax$. The ability to limit the number of gangs is essentially the same as requesting each gang to carry out more work. In terms of CUDA this means that each threadblock processes multiple entries of the outer loop which would require the programmer to distribute this loop manually. This feature can reduce the overhead involved in scheduling new gangs and might enable the gangs to reuse data. Even though this feature might improve performance for a special coprocessor it can limit portability and scalability for future architectures.

```
#pragma acc parallel num_gangs(128), vector_length(256)
#pragma acc loop gang
    for(i = 0; i < n; ++i){
        double tmp = 0.0;
#pragma acc loop vector reduction(+:tmp)
        for(j = 0; j < n; ++j)
            tmp += A[i * n + j] * x[j];
        y[i] = tmp;
    }
```

**Listing 7:** Verbose version of a matrix-vector multiplication with the number of gangs set to 128 and the vector-length fixed at 256.

# 4 Case Studies

This section presents two case studies - Molecular Dynamics in Section 4.1 and Conjugate Gradient Method in Section 4.2 - to evaluate the performance and productivity of OpenACC compared to CUDA and OpenMP. I describe their basic principles, the rough layout of the algorithms and list code examples if they expose special features or behaviours of OpenACC.

## 4.1 Molecular Dynamics Simulation

Molecular dynamics (MD) is an important method to deal with classical many-body systems. Its applications range over various length scales, e.g. the simulation of the universe, motion of galaxies, motion of planets and down to microscopic systems to describe the motion of gases or liquids.

Molecular dynamics deals with the dynamics of $N$ particles (e.g. planets, atoms) to simulate their motion and to measure observables such as energy and pressure. The motion of the particles is governed by Newton's equation of motion:

$$\vec{f_i} = \vec{a}_i m_i = -\nabla_i U(t) \;\; \forall i = 1, 2, ..., N \tag{1}$$

Where $\vec{a}_i \in \mathbb{R}^3$ is the acceleration and $m_i \in \mathbb{R}$ is the mass[3] of particle $i$. Moreover, $U(t)$ is the interaction potential at time $t$ and is described as follows:

$$U(t) = \frac{1}{2} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} U_{i,j}(\|\vec{r}_{i,j}\|) \tag{2}$$

Where $U_{i,j}$ is the interaction potential between two particles $i$ and $j$ and $\vec{r}_{i,j} \in \mathbb{R}^3$ represents the distance between them. The choice of the proper interaction potential depends on the underlying physical problem. I use the Lennard-Jones potential (see Equation (3))[4] for the simulation of a synthetic example.

$$U_{i,j}(\vec{r}_{i,j})^{LJ} = 4\left[\left(\frac{\sigma}{\|\vec{r}_{i,j}\|}\right)^{12} - \left(\frac{\sigma}{\|\vec{r}_{i,j}\|}\right)^{6}\right] \tag{3}$$

The most computationally intensive part of an MD simulation is the computation of the potential in order to compute the forces such that the particles can be integrated in time. The naive approach to do this, is to compute the sum in Equation (2) which results in a computational complexity of $\mathcal{O}(N^2)$. As we are usually interested in systems consisting of millions of particles this approach is not feasible and there are algorithms available which approximate this sum requiring a complexity of

[3] For simplicity we assume that all particles have the same mass.
[4] Used to describe the interaction between molecules of noble gases.

```
1 For i:= 1 to m Do:
2     t ← t + dt
3     compute_forces(r⃗, f⃗)
4     integrate(r⃗, f⃗, v⃗, dt)
5     If(i % 100 == 0)
6         save_to_file(r⃗)
7 EndDo
```

**Listing 8:** Overview of the main Molecular Dynamics routine. Where $\vec{r} = (\vec{r}_1, \vec{r}_2, ..., \vec{r}_N), \vec{f} = (\vec{f}_1, \vec{f}_2, ..., \vec{f}_N), \vec{v} = (\vec{v}_1, \vec{v}_2, ...\vec{v}_N)$ and $m$ denote the particle position, force, velocity and the maximum number of iterations, respectively.

```
1 #pragma acc kernels loop
2     For i:= 1 to N Do:
3         For j:= 1 to N Do:
4             r⃗_{i,j} ← r⃗_j − r⃗_i
5             f⃗_{i,j} ← compute_force(‖r⃗_{i,j}‖)
6             f⃗_i ← f⃗_i + ‖f⃗_{i,j}‖ r⃗_{i,j}
7         EndDo
8     EndDo
```

**Listing 9:** Naive OpenACC version for updating the forces $\vec{f}_i$ of each particle. Let $\vec{r}_i$ denote the position of particle $i$ and $N$ the total number of particles.

$\mathcal{O}(Nlog(N))$ or even $\mathcal{O}(N)$ [15,16]. However, since this case study is concerned about the performance and productivity of OpenACC in comparison the CUDA and OpenMP, the naive $\mathcal{O}(N^2)$ approach is still a reasonable choice.

The remainder of this section lists three different OpenACC implementations of the $\mathcal{O}(N^2)$ MD algorithm. Listing 8 depicts the overall control flow of all three implementations. Henceforth, I omit non-relevant parameters and implementation details for simplicity reasons. Moreover, this MD implementation only ports the computation of the forces (i.e. Line 3) to the GPU. This means that each iteration requires data transfers between host and device. If we would be interested in a high-performance MD simulation we would definitely have to pay attention to this issue. However, the OpenACC implementation closely follows its CUDA counterpart which allows a fair performance comparison.

The *naive OpenACC* implementation of the *compute_force(...)* routine requires only a single *#pragma acc kernels loop* directive to offload the force calculation to the coprocessor (see Listing 9). However, the PGI compiler feedback indicates that only the outer $i$ loop is distributed among gangs and vectors while the parallelization of the inner $j$ loop was not possible due to loop-carried dependencies.

An *improved OpenACC* version which avoids these dependencies at the cost of additional reductions is shown in Listing 10. Profiling this *improved* versions reveals that the compiler chooses to allocate and deallocate the force and position arrays in every iteration. Adding a *#pragma acc data create(particle_force*$[0:N]$*, particle_pos*$[0:N]$*)* directive in front of Line 1 of Listing 8 creates a data region which keeps the arrays on the coprocessor and avoids these redundant (de)allocations. This data region requires additional *update* directives (see Listing 11) to update the particle positions on the device and the particle forces on the host, respectively.

```
1  #pragma acc kernels loop
2      For i:= 1 to N Do:
3          \vec{\tilde{f}} ← 0
4  #pragma acc loop reduction(+:\vec{\tilde{f}})
5          For j:= 1 to N Do:
6              \vec{r}_{i,j} ← \vec{r}_j − \vec{r}_i
7              \vec{f}_{i,j} ← compute_force(‖\vec{r}_{i,j}‖)
8              \vec{\tilde{f}} ← \vec{\tilde{f}} + ‖\vec{f}_{i,j}‖ \vec{r}_{i,j}
9          EndDo
10         \vec{f}_i ← \vec{\tilde{f}}
11     EndDo
```

**Listing 10:** Improved OpenACC version for updating the forces $\vec{f}_i$ of each particle. Let $\vec{r}_i$ denote the position of particle $i$ and $N$ the total number of particles.

```
1  #pragma acc update device(\vec{r}[0:N])
2  #pragma acc kernels loop present(\vec{r}[0:N], \vec{f}[0:N])
3      For i:= 1 to N Do:
4          \vec{\tilde{f}} ← 0
5  #pragma acc loop reduction(+:\vec{\tilde{f}})
6          For j:= 1 to N Do:
7              \vec{r}_{i,j} ← \vec{r}_j − \vec{r}_i
8              \vec{f}_{i,j} ← compute_force(‖\vec{r}_{i,j}‖)
9              \vec{\tilde{f}} ← \vec{\tilde{f}} + ‖\vec{f}_{i,j}‖ \vec{r}_{i,j}
10         EndDo
11         \vec{f}_i ← \vec{\tilde{f}}
12     EndDo
13 #pragma acc update host(\vec{f}[0:N])
```

**Listing 11:** Final OpenACC version for updating the forces $\vec{f}_i$ of each particle. Let $\vec{r}_i$ denote the position of particle $i$ and $N$ the total number of particles.

## 4.2 Conjugate Gradient Method

The Conjugate Gradient Method (CG) is one of the best known iterative solvers to solve Equation (4) for $x \in \mathbb{K}^n$ [17], where $A \in \mathbb{K}^{n \times n}$ is sparse, symmetric and positive definite (SPD) (i.e. has many zero entries and all of its eigenvalues are larger than 0) and $b \in \mathbb{K}^n$. These kind of problems frequently arise from the discretization of *partial differential equations* (PDEs) in physics.

$$Ax = b \tag{4}$$

The rough outline of the CG method is shown in Listing 12. Without going into algorithmic details, it starts from an approximate solution $x_0$ and improve its accuracy from iteration to iteration until the final result is given by $x_m$. Moreover, the implementation only requires one sparse matrix-vector multiplication and four $n$-dimensional vectors $x$, $p$, $Ap$ and $r$.

The overall runtime of a CG method is governed by the sparse-matrix vector multiplication in Line 4. In order to take advantage of the structure of $A$ (i.e. its sparsity)[5], I use the *Compressed Sparse Row*

[5] This algorithm does not exploit the symmetry of $A$.

```
1  r_0 ← b − A x_0
2  p_0 ← r_0
3  For j:= 1 to m Do:
4      α_j ← (r_j, r_j)/(A p_j, p_j)
5      x_{j+1} ← x_j + α_j p_j
6      r_{j+1} ← r_j − α_j A p_j
7      β_j ← (r_{j+1}, r_{j+1})/(r_j, r_j)
8      p_{j+1} ← r_{j+1} + β_j p_j
9  EndDo
```

**Listing 12:** Conjugate Gradient Method [17]. Where *(.)* denotes a scalar product and $m$ is the maximum dimension of the Krylov Subspace [17].

```
1  #pragma acc parallel vector_length(32)
2  #pragma acc loop gang
3  For i := 1 to n Do:
4      ỹ ← 0
5  #pragma acc loop vector reduction(+:ỹ)
6      For j := p_i to p_{i+1} Do:
7          ỹ ← ỹ + v_j x_{idx_j}
8      EndDo
9      y_i ← ỹ
10 EndDo
```

**Listing 13:** Matrix-vector multiplication using the CSR format. Let $n \in \mathbb{N}$ denote the number of rows of $A \in \mathbb{R}^{n \times n}, p \in \mathbb{N}^{n+1}, idx \in \mathbb{R}^{NNZ}, v \in \mathbb{R}^{NNZ}$ and $y, x \in \mathbb{R}^n$ with $NNZ$ being the number of non-zeros of $A$. Assume that all arrays are present on the coprocessor. Optional clause for increased performance.

(CSR) format [17] that only stores the non-zero entries. Even though I don't list implementation details for this algorithm, I want to point out that all required vector-vector and matrix-vector operations have been ported to the GPU and that the OpenACC implementation follows its CUDA counterpart as closely as possible. For example, it uses the *data* region before Line 3 (see Listing 12) that moves all the data (i.e. $A$, $x_0$ and $b$) to the GPU and only requires a single data transfer from device to host (i.e. $x_m$) at the end of the algorithm after Line 9.

Listing 13 shows the outline of the implemented matrix-vector multiplication using the CSR format, please refer to [17] for more detailed information of the CSR format. To port this matrix-vector multiplication to the coprocessor I used three OpenACC directives. (1) *acc parallel* generates an accelerator region. (2) *acc loop gang* tells the compiler to schedule the outer $i$ loop across gangs. (3) *acc loop vector* instructs the compiler to use vector parallelism for the inner $j$ loop. These directives are sufficient to offload the matrix-vector multiplication to the coprocessor. However, the compiler chooses to use a vector-length of 256 for the $i$ loop which results in poor performance (see Section 4.3). The additional *vector_length(32)* clause modifies this schedule and sets the vector-length to 32 per gang.

## 4.3 Performance Results

This section outlines the performance of the different versions of the two presented case studies. The OpenMP versions are run on two Intel Xeon E5-2700 processor each having 8 cores with *Simultaneous multithreading* (SMT) support which allows to run two threads per core. The CUDA and OpenACC versions are executed on two Intel Xeon X5650 processors with 6 cores attached to an NVIDIA Quadro

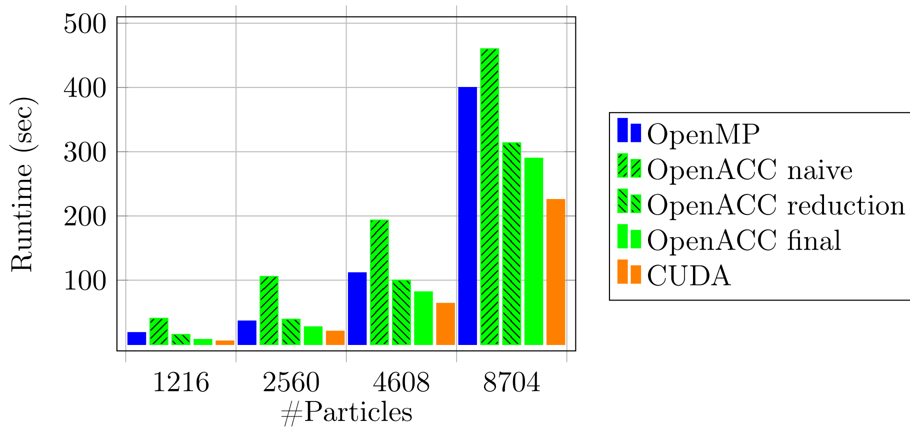


**Figure 1:** Runtime of the MD code for different problem sizes over 10.000 iterations. All calculations are run in double precision.

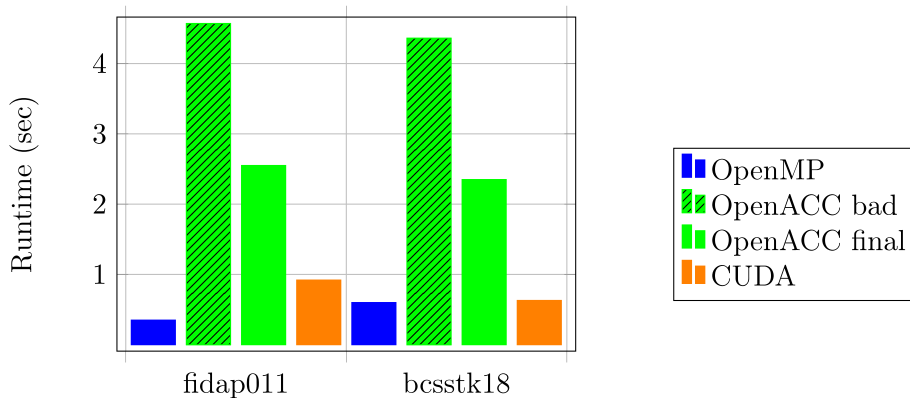


**Figure 2:** Runtime of the CG code for two sparse SPD matrices. All calculations are run in double precision.

6000 GPU.

In order to provide a fair comparison I have tuned each version equally well. More precisely, the OpenMP version of the MD case extensively uses the AVX capabilities of the CPU and is run with 32 threads at these 16 cores and it achieves an almost linear speedup of 14.7× over its serial counterpart. Moreover, the OpenACC versions are designed such that they are as similar to the CUDA versions as possible.

Figure 1 provides several interesting facts: (1) Each OpenACC version improves over its predecessor. (2) The final OpenACC version is roughly 40% faster than the OpenMP version. (3) The OpenACC version achieves roughly 80% of the performance of the CUDA version which is a reasonable performance for a directive-based approach. Even though it is not evident from Figure 1, I like to point out that even the *naive* OpenACC version yields a significant speedup (i.e. $\approx 6.7\times$) over the single-threaded OpenMP Version.

Figure 2 shows the runtime of the implemented CG method for two different matrices. The *bad* OpenACC version denotes almost the same version as the *final* OpenACC version (see Listing 13) but without the additional *vector_length(32)* clause. These results require further explanations:

(1) The *bad* OpenACC version performs only at $\approx 55\%$ of the *final* version. This is due to the compiler's choice to use a vector-length of 256, which can be considerably larger than the number of non-zeros in a single row of the matrix[6] (i.e. many cores are idle).

[6] The compiler does not know the (avg) number of non-zeros per row at compile time.

(2) The *final* OpenACC implementation only achieves 36% and 27% of the CUDA performance for the *fidap011* and *bcsstk18* matrices, respectively. These results are not really satisfactory and need further analysis. Profiling both versions (i.e. OpenACC final and CUDA) with NVIDIA's Visual Profiler shows hardly any differences between the two implementations. More precisely, (a) both versions spent roughly 95% of their runtime for the sparse matrix-vector multiplication. (b) They both use equally many threadblocks, threads and registers. Given their similarities it is hard to say why the performance of the OpenACC version is so poor in comparison to its CUDA counterpart. It could be due to a performance issue of the current PGI compiler 12.9 and might get fixed for upcoming versions. A more elaborate investigation of this performance issue is left as future work.

(3) The OpenMP implementation trails the CUDA version for both matrices. This is actually not so surprising, since the CUDA version is not fully implemented on the GPU (i.e. it requires global synchronizations between all threadblocks) and does not utilize any asynchronous computation. Moreover, the CUDA version might outperform the OpenMP versions for different sparsity patterns and matrix sizes. A detailed performance analysis of the CG method on GPUs is presented in [18].

## 4.4 Productivity Results

At the time of writing **debugging support** was very limited. Even though well-known debuggers such as RogueWave's *Totalview* [19] and Allinea's *DDT* [20] support OpenACC for the Cray CCE 8.0 compiler, they lack full support for PGI and CAPS compilers (i.e. they don't allow to debug the accelerated regions). A viable workaround is to resort to debugging the logic of the application by compiling the application without OpenACC support.

One major problem I have encountered is the **lack of C++ support** of the current PGI compiler which forced me to port the existing C++ OpenMP code to C before I could start to parallelize it via OpenACC.

**Function calls** within accelerator regions are not yet supported by current compilers. However, they do not prohibit parallelization if the compiler is able to inline them automatically. Even though this has not been a problem with respect to the two case studies shown in this paper, it might pose major problems for larger projects where it is not feasible to inline these function calls manually.

A particular productive feature of OpenACC is its capability to deal with **"boundary conditions"**, which means that the compiler takes care of padding arrays or introducing if-statements in the kernel to avoid out-of-bounds accesses, if necessary. Moreover, memory **(de)allocation and data transfers** are very easy to use and save programming time.

Yet another very useful feature of the PGI compiler is its ability to detect data dependencies automatically and issue data transfers or create **reduction** operations, if necessary. This is a neat approach that can help to make coprocessor programming more straightforward.

While **multi-GPU support** is available in OpenACC it is not as automated and intuitive as one might expect from a directive-based paradigm. It would be desirable[7] to specify how many and which coprocessors should be used for accelerating a single accelerated region and let the OpenACC API distribute the work among the coprocessors accordingly. However, until now the programmer has this responsibility, which can be a laborious task.

Since it is hard to measure productivity, I list the added and modified lines of source code for each case study and version in Figure 3. As it is evident from Figure 3, CUDA requires the most refactoring of the existing serial code which is mainly due to the explicit data (de)allocation, data movement

[7] in addition to the current multi-GPU features

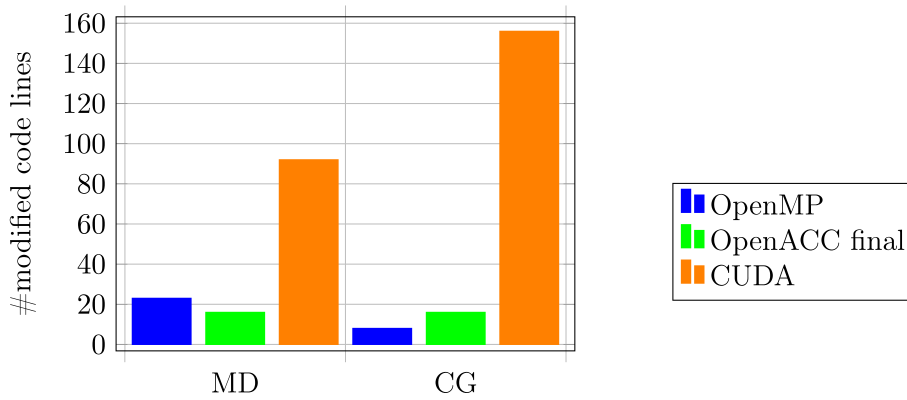


**Figure 3:** Number of added and modified lines of source code for each case study and paradigm with respect to the serial version.

and restructuring of the existing kernels. However, since I implemented the different versions in the order (1) OpenMP, (2) OpenACC and (3) CUDA, it is hard to say which paradigm took most of the development time. More precisely, before I began writing the OpenACC version I already had a highly tuned OpenMP version running which made the development of the OpenACC version much easier. The same holds for OpenACC and CUDA, where the implementations were straightforward because I only had to translated the OpenACC directives into correct CUDA code. With respect to the MD case study, I would say that both OpenMP and OpenACC required the same programming effort because the OpenMP version had to deal with manual inlining, alignment and blocking in order to take advantage of the vectorization capabilities of the processor. However, based on the CG case study it is fair to say that OpenMP required less programming effort than OpenACC or CUDA. Even though CUDA was the most time consuming approach, the code changes from the OpenACC versions to the CUDA versions were trivial. Hence, I would recommend to implement an OpenACC version first and translate this version into a CUDA version that can be fine-tuned for the given architecture if the performance results are not satisfactory.

# 5 Future Directions

We will see more heterogeneous systems in the near future. While Cray is currently occupying the 1$^{st}$ place of the Top500 with its current supercomputer XK7, Cray's upcoming supercomputer XC30[8] will eventually take advantage of different coprocessors (e.g. Intel Xeon Phi, Convey FPGAs) [21]. Another indicator for the continuing trend towards heterogeneous computing is the *Spampede* supercomputer at the Texas Advanced Computing Center, that will gain most of its performance through Intel Xeon Phi coprocessors.

Even though GPU programming has been around for quite some time, the OpenMP Architecture Review Board (ARB) just released a technical report [22] that introduces directives to support coprocessor programming. However, as of now, this is just a technical report, so there might be some changes to it before it will be incorporated into the OpenMP 4.0 standard somewhere next year.

It is still not 100% clear whether OpenMP 4.0 will eventually substitute OpenACC. However, PGI's senior compiler engineer Michael Wolfe said that there is hope that OpenMP and OpenACC will converge [23], if it will be possible to implement the OpenMP standard in an efficient way. For now, it looks like that both OpenMP and OpenACC are going to exist simultaneously [24].

[8]that is expected to scale beyond 100 petaflops [21]

A strong indicator for the continuing development of OpenACC are the proposed additions for OpenACC 2.0 which were recently released [25]. These additions enrich the features of the current OpenACC specification. More precisely, OpenACC 2.0 is going to support: (1) Function calls via the *routine* directive that can compile functions for the coprocessor (i.e. no more inlining or code refactoring required). (2) Nested parallelism (i.e. use of kernels or parallel directive inside an accelerator region). (3) The *tile* clause for the *loop* directive will allow programmers to take advantage of 2D/3D partitioning of tightly nested loops, that can yield an increased cache utilization. For further information on OpenACC 2.0, please refer to [25].

Moreover, PGI announced several interesting developments for the upcoming months. (1) Their OpenACC-enabled compilers will support NVIDIA's most recent Tesla K20 GPU accelerator by the end of this year [26]. (2) Beta support for Intel's Xeon Phi coprocessor is planed for the first half of 2013 [26]. (3) Support for AMD accelerators and a full implementation of OpenACC 2.0 is expected for mid 2013 [26].

## 6 Conclusion

All in all, it is fair to say that OpenACC can make GPU programming more convenient because it hides the complexity of the low-level APIs. However, if high performance is the main objective the programmer has to know the target architecture in order to specifically tune the OpenACC kernels for that coprocessor, which in turn limits portability. Based on the presented case studies, I found OpenACC to be a very productive tool that might lead to a new development process for coprocessor code. More precisely, first implementing the OpenACC version and then using this version to develop the low-level version could be a reasonable approach for future coprocessor programming. Moreover, we have seen that the OpenACC versions are not able to deliver the same performance as their CUDA counterparts (i.e. $\approx 80\%$ for MD and only $\approx 30\%$ for CG case study). However, this might change in the next few months as all OpenACC compilers continue to mature.

## References


[1] NVIDIA. CUDA 5.0. http://docs.nvidia.com/cuda/index.html, November 2012. Version 5.0.

[2] Khronos Group. The OpenCL Spezification. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, November 2012. Version 1.2.

[3] Top 500. Supercomputer Site. http://www.Top500.org/, November 2012.

[4] Green 500. http://www.Green500.org/, November 2012.

[5] Intel. Many Integrated Core Architecture. http://software.intel.com/en-us/mic-developer, November 2012.

[6] http://www.openacc-standard.org/. OpenACC Specification, November 2012. Version 1.0.

[7] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC-First Experiences with Real-World Applications. *Euro-Par 2012 Parallel Processing*, pages 859–870, 2012.

[8] A Hart, R Ansaloni, and A Gray. Porting and Scaling OpenACC Applications on Massively-Parallel, GPU-Accelerated Supercomputers. *The European Physical Journal Special Topics*, 210(1):5–16, September 2012.

[9] John M Levesque, Ramanan Sankaran, and Ray Grout. *Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-Petaflops and Beyond.* IEEE Computer Society Press, November 2012.

[10] OpenMP. OpenMP Application Program Interface, November 2012. Version 4.0.

[11] Michael Wolfe. The Heterogeneous Programming Jungle. http://www.hpcwire.com/, March 2012.

[12] Rob Farber. The OpenACC Execution Model. http://www.drdobbs.com, August 2012.

[13] Michael Wolfe. OpenACC Kernels and Parallel Constructs . http://www.pgroup.com/lit/articles/insider/v4n2a1.htm, August 2012.

[14] The Portland Group. PGI Accelerator FAQ. http://www.pgroup.com/resources/accel.htm, Dezember 2012.

[15] P. Gibbon and G. Sutmann. Long-Range Interactions in Many-Particle Simulation. *Quantum Simulations of Many-Body Systems: From Theory to Algorithm. Eds. J. Grotendorst, D. Marx and A. Muramatsu. NIC-series*, 10:467–506, 2002.

[16] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, and K. Schulten. Accelerating Molecular Modeling Applications with Graphics Processors. *Journal of computational chemistry*, 28(16):2618–2640, 2007.

[17] Y. Saad and Y. Saad. *Iterative Methods for Sparse Linear Systems*, volume 620. PWS publishing company Boston, 1996.

[18] M. VERSCHOOR and AC JALBA. Analysis and Performance Estimation of the Conjugate Gradient Method on Multiple GPUs. *Parallel Computing*, 2012.

[19] RogueWave. Totalview. http://www.roguewave.com/technologies/openacc.aspx, Dezember 2012.

[20] Allinea. DDT User Guide. http://content.allinea.com/downloads/userguide.pdf, Dezember 2012. Version 3.2.1.

[21] Michael Feldman. Cray Launches Cascade, Embraces Intel-Based Supercomputing. http://www.hpcwire.com/, November 2012.

[22] http://www.openmp.org/. Technical Report on Directives for Attached Accelerators, November 2012.

[23] Michael Wolfe. PGI's Michael Wolfe on OpenACC Directives for GPUs. http://www.insidehpc.com/, November 2012.

[24] Michael Feldman. OpenMP Takes To Accelerated Computing. http://www.hpcwire.com/, November 2012.

[25] http://www.openacc standard.org/. Proposed Additions for OpenACC 2.0, Dezember 2012. Version 2.0.

[26] The Portland Group. PGI News. http://www.pgroup.com/about/news.htm, Dezember 2012.