A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing

Sandra Wienke[1,2], Christian Terboven[1,2], James C. Beyer[3], and Matthias S. Müller[1,2]

[1] IT Center, RWTH Aachen University, 52074 Aachen, Germany
[2] JARA – High-Performance Computing, Schinkelstr. 2, 52062 Aachen, Germany
{wienke,terboven,mueller}@itc.rwth-aachen.de
[3] Cray Inc., 380 Jackson Street, Suite 210 St. Paul, MN, USA
beyerj@cray.com

Abstract. Nowadays, HPC systems frequently emerge as clusters of commodity processors with attached accelerators. Moving from tedious low-level accelerator programming to increased development productivity, the directive-based programming models OpenACC and OpenMP are promising candidates. While OpenACC was completed about two years ago, OpenMP just recently added support for accelerator programming. To assist developers in their decision-making which approach to take, we compare both models with respect to their programmability. Besides investigating their expressiveness by putting their constructs side by side, we focus on the evaluation of their power based on structured parallel programming patterns (aka algorithmic skeletons). These patterns describe the basic entities of parallel algorithms of which we cover the patterns *map*, *stencil*, *reduction*, *fork-join*, *superscalar sequence*, *nesting* and *geometric decomposition*. Architectural targets of this work are NVIDIA-type accelerators (GPUs) and specialties of Intel-type accelerators (Xeon Phis). Additionally, we assess the prospects of OpenACC and OpenMP concerning future development in soft- and hardware design.

Keywords: OpenACC, OpenMP 4, GPU, Xeon Phi, programmability, parallel patterns.

1 Introduction

Heterogeneity and specialized accelerating hardware add a further level of complexity to parallel programming. Although, accelerator programming with low-level APIs like CUDA or OpenCL opens up opportunities for performance tuning, it also challenges the software design or may lead to error-prone tasks or even hardware-specific implementations. By attempting to overcome these difficulties, directive-based models for accelerator programming gained more interest, lately. Up to now, the most prominent one is OpenACC [13] that was released as industry standard in November 2011 and incorporates two years of maturity now. While OpenMP [14] has been the de-facto standard for programming multi-core CPUs for over ten years, it also covers high-level accelerator programming since version 4.0 (July 2013). Having two well-promoted directive-based models for accelerators around, developers are currently wondering which programming model to chose. Emerging questions relate to the power of the programming paradigm, opportunities for performance and the long-term perspective of the usage of the programming model and its mapping to future hardware architectures.

In this paper, we discuss answers to most of these questions to assist developers in their decision-making process between OpenACC and OpenMP. We examine the programmability and potency of the two models by comparing both the available constructs side by side and the expressiveness taking a pattern-based approach, following the classification by McCool et al [12]. Covered patterns are *map*, *stencil*, *reduction*, *fork-join*, *superscalar sequence*, *nesting* and *geometric decomposition*. Implementations are illustrated for NVIDIA-type accelerators (GPGPUs) and occasionally for Intel-type accelerators (Intel Xeon Phis). From these, we derive similarities and differences in programmability. A comparison of performance measurements is currently not possible since OpenACC/ OpenMP implementations for the same device hardware do not exist.

The paper is structured as follows: Section 2 covers related work. In Section 3, we give an overview on available accelerator directives in OpenACC and OpenMP and show fundamental differences in their expressiveness. The pattern-based comparison is carried out in Section 4 and examines the fit for certain algorithmic tasks. Finally, we conclude our findings in Section 5 and discuss future perspectives of both models.

2 Related Work

Over the years, numerous approaches to characterize parallel algorithms have been undertaken. An early work [5] classifies algorithms into *skeletons*. A pattern language for parallel programming that uses *design patterns* and makes up four design spaces is defined by Mattson et al [11]. A famous categorization is given by Berkley's dwarfs (or motifs) [1] that characterize workloads for the evaluations of parallel architectures, for instance, dense/ sparse linear algebra, (un-)structured grids or n-body applications. We chose a lower level of abstraction by applying parallel patterns for structured programming defined by McCool et al [12]. While few works applied different categorizations of parallel algorithms to accelerator paradigms (e.g. [4]), we are the first to our knowledge that use the novel characterization by McCool et al. The relative low abstraction level and the applicability to scientific programming makes this characterization specifically suitable to compare parallel programming paradigms.

Various directive-based paradigms fed into the current OpenACC and OpenMP standards. Some of these approaches (PGI Accelerator, hiCUDA, HMPP, OpenMPC, R-Stream) have been compared to OpenACC (CAPS, PGI, accULL) in [6,15,9]. Groundwork for OpenMP for accelerators [3] was done by our author Beyer (et al).

While few works deal with OpenMP for accelerators so far, much research has been carried out on OpenACC in the last years. However, most of it focuses on performance evaluations rather than on programmability—as we do. In [7], the authors compare the performance of Cray's, PGI's and HMPP's OpenACC implementation to a low-level CUDA version using two micro-benchmarks and one real-world code. Our previous work [17] covers performance results on two real-world applications comparing OpenCL with Cray's OpenACC and the PGI Accelerator Model. Performance investigations also cover different architectures such as Intel Xeon Phi and NVIDIA GPUs [16]. Some of these works [17,9,6] also include programmability aspects with respect to learning curve, code size, development effort or adaptability. For evaluating expressiveness, we follow a more general approach and exhibit a structured comparison by well-defined parallel patterns. With respect to the OpenMP accelerator model, only few investigations have been published yet at all. The research implementation *HOMP* is introduced in [10] for NVIDIA GPUs. The authors compare performance of HOMP to PGI's and HMPP's OpenACC versions. To the best of our knowledge, we provide the first comparison of programmability between OpenACC and OpenMP for accelerator programming (basing on Beyer's webinar [2]). Wolfe [19] makes rather skeptical comments on the extension of accelerator offload regions to OpenMP. We contribute our own view that bases on experiences in academia, industry and our work in the OpenMP/ OpenACC committees on the prospects of both standards in Section 5.

3 Overview on OpenACC and OpenMP for Accelerators

OpenMP has been the de-facto standard for shared-memory multi-core programming since about ten years. Additionally, the OpenMP language committee has been working on the integration of accelerator support since 2009, which resulted in the `target` construct as part of OpenMP 4.0. In between, the independent sub group of Cray, CAPS, PGI, and NVIDIA released their own industry standard as OpenACC in 2011. OpenMP aims to extend known concepts from multi-core programming to accelerators and allows heterogeneous programming with just one paradigm, while OpenACC was motivated by GPGPU users being tired of low-level APIs. OpenACC's specification 2.0 from June 2013 contains advances and feedback gathered from the last two years. Similarly, the OpenMP language committee is already working on improving the accelerator support for the next (minor) standard update.

Both models build on a host-directed execution model in which the host offloads data and compute-intensive loops to an accelerator (or as fallback to the host itself). An abstract machine model is presented in [20, p. 5]. Both models also exhibit a weak device memory model so that memory coherence between operations executed by different threads is not assured. The memory entities between host and device are presumed to be separate. However, the devices may share memory with the host [13, p. 9f.] [14, p. 17ff.]. OpenACC and OpeMP both contain constructs, clauses, runtime library routines and environment variables to control the workflow and express parallelism. A direct comparison of important features is given in Table 1.

4 Pattern-Based Comparison

Patterns are the basic structural entities of algorithms and represent common control flows and data organizations in applications. We apply these parallel patterns as defined by McCool et al to accelerator programming models and focus on these special accelerator features rather than on the base language characteristics of C/C++ or Fortran. By parallel patterns, we show concepts and differences of the programmability and potency of OpenACC and OpenMP.

4.1 Map

The elementary map pattern is the foundation of numerous algorithms (e.g. Monte Carlo sampling) and other patterns. It represents a parallel version of a serial iterating loop of

Table 1. Comparison of constructs and clauses of OpenACC and OpenMP

OpenACC	OpenMP	Remark
parallel	target	offload of computational work to the device (synchronously)
parallel	teams, parallel	creation of in parallel running threads
kernels		compiler may find parallelism in associated block automatically
data	target data	structured data management between host & device
loop	distribute, do, for, simd	worksharing across the parallel units
host data		interoperability with low-level languages like CUDA
cache		move object closer to the execution units in the memory hierarchy
update	target update	data movement between host & device within data environment
declare	declare target	declaration of global, file static or extern objects used inside a parallel region
routine	declare target	declaration of functions called inside a parallel region
enter data		unstructured data management to the device
exit data		unstructured data management from the device
	tasks	creation of explicit tasks for task parallelism
async(*int*)	task depend	asynchronous execution with dependencies
wait		synchronization of streams
async wait		asynchronous waiting on a specific stream
parallel in parallel	parallel in parallel or team	nested parallelism on the device
tile		strip-mining of data collections
device_type		device-specific tuning of clauses
atomic	atomic	atomic operations
	sections, critical, barrier, master, single	non-iterative workshare, critical sections, synchronization, control flow for single thread

which all iterations of the body are independent and the number of iterations is known in advance. This pattern *maps* in parallel the different elements of the input data within the index space to an output collection using a so-called elemental function.

The elemental function f of the map example in Listings 1.1–1.4 describes a naive scaled matrix transpose: $B = p \cdot A^T$ with $p \in \mathbb{R}$, $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times n}$. OpenACC and OpenMP both support the map pattern. Listings 1.1 and 1.2 show implementations for NVIDIA-type accelerators that leverage the GPU's two levels of parallelism. While the `parallel` construct in OpenACC directly starts the parallel execution on the device, an additional `target` construct must be specified in OpenMP to differentiate between host and device execution. OpenMP is also verbose on the different hierarchies of parallelism: on a GPU, the `teams distribute` spreads the work of the outer loop in independent chunks onto the compute units (as defined in [8, p. 23f.]). Here, `teams` creates a parallel teams region; `distribute` indicates the workshare. It does not contain an implicit barrier at its end and must be closely nested in or combined with `teams`. Then, the `parallel for` distributes the work of the inner loop across the processing elements [8, p. 23f.] within a compute unit. With OpenACC, the `loop` directive is sufficient for worksharing, but should be extended by an efficient loop

Listing 1.1. Map with two levels of parallelism in OpenACC (GPU)

```
#pragma acc routine seq
 double f(double p, double aij) {
  return (p * aij);
 }

// [..]
#pragma acc parallel
#pragma acc loop gang
 for(i=0; i<n; i++) {
#pragma acc loop vector
  for(j=0; j<m; j++) {
   b[j][i] = f(5.0,a[i][j]);
 } }
```

Listing 1.2. Map with two levels of parallelism in OpenMP (GPU)

```
#pragma omp declare target
 double f(double p, double aij) {
  return (p * aij);
 }
#pragma omp end declare target
// [..]
#pragma omp target
#pragma omp teams distribute
 for(i=0; i<n; i++) {
#pragma omp parallel for
  for(j=0; j<m; j++) {
   b[j][i] = f(5.0,a[i][j]);
 } }
```

Listing 1.3. Map in OpenACC (Phi)

```

#pragma acc routine seq
 double f(double, double);

// [..]
#pragma acc parallel
#pragma acc loop gang vector
 for(i=0; i<n; i++) {
  for(j=0; j<m; j++) {
   b[j][i] = f(5.0,a[i][j]);
 } }
```

Listing 1.4. Map in OpenMP (Phi)

```
#pragma omp declare target
#pragma omp declare simd
  double f(double, double);
#pragma omp end declare
// [..]
#pragma omp target
#pragma omp parallel for simd
 for(i=0; i<n; i++) {
  for(j=0; j<m; j++) {
   b[j][i] = f(5.0,a[i][j]);
 } }
```

scheduling clause. Here, `loop gang` and `loop vector` equal the work distribution of the OpenMP example. Additionally, OpenACC provides the "magical" `kernels` directive that delegates the responsibility of finding parallelism to the compiler.

Closely related to the map pattern is the elemental function that is implemented as function call. OpenACC (2.0) supports function calls by the `routine` construct which needs the declaration of a parallelism level (`gang`, `worker`, `vector`, `seq`). A `seq` clause is used in the example to denote that the function does not express any parallelism itself, as it is already sufficiently exploited at the loop level. In turn, OpenMP has a more flexible way by denoting the `declare target` directive without specifying the parallelism. Thus, the function can be called from different contexts. Contrary, the absence of this hint might prevent some optimizations. In the following, we express the elemental function of the fundamental map pattern in formulas for better reading.

The same implementations will also work on an Intel Xeon Phi as OpenACC and OpenMP guarantee portability. However, performance portability may be implementation dependent. A more appropriate approach applies another level of parallelism (no hierarchy) and emphasizes vectorization (compare Listings 1.3 and 1.4). The mapping of work onto the threads on the Phi is employed by `loop gang` and `parallel for`. Vectorization is requested by `vector` and `simd` clauses, respectively.

Listing 1.5. Stencil in OpenACC (GPU)

```
 1 #pragma acc parallel
 2 #pragma acc loop tile(64,4) gang vector
 3  for(i=1; i<n-1; i++) {
 4   for(j=1; j<m-1; j++) {
 5
 6 #pragma acc cache(a[i-1:3][j-1:3])
 7
 8    anew[i][j] = (a[i-1][j] + a[i+1][j] +\
            a[i][j-1]+ a[i][j+1]) * 0.25;
 9   }
10  }
```

Listing 1.6. Stencil in OpenMP (GPU)

```
1 #pragma omp target
2 #pragma omp teams distribute collapse(2)
3   for(i=1; i<n-1; i+=64) {
4     for(j=1; j<m-1; j+=4) {
5 #pragma parallel for collapse(2)
6       for(k=i; k<min(n-1,i+64); k++){
7         for(l=j; l<min(m-1,j+4); l++){
8           anew[k][l] = (a[k-1][l] +        \
                 a[k+1][l] + a[k][l-1] +  \
                 a[k][l+1]) * 0.25;
9  } } } }
```

4.2 Stencil

The elemental function of the stencil pattern allows several input elements that can be accessed in a regular way, i.e. with fixed offsets. This structure of neighboring input elements enables data reuse and cache optimizations. To fit data into the software- or hardware-managed cache (especially) for multi-dimensional stencils, the 'layer condition' must be fulfilled. A common solution is the spatial blocking of data that is also known as strip-mining.

In Listings 1.5 and 1.6, a small part of a Jacobi solver for the Laplace equation is presented, omitting the matrix swap and the convergence iteration. The presented two-dimensional stencil can be `tile`d into blocks using OpenACC. The `tile` clause hides loop splitting and collapsing. This is illustrated in the OpenMP example since tiling must be explicitly expressed in OpenMP. Here, `distribute teams collapse(2)` combines the index space of the outer two loops for distribution to the compute units of a GPU and `parallel for collapse(2)` for distribution across the processing elements within the compute units. In addition to blocking, OpenACC provides the `cache`-ing capability (line 6) to let the developer specify that sub arrays should be fetched into the highest-level memory for data reuse. OpenMP does not yet support leveraging the on-chip caches explicitly. Summarizing, both models do not have built-in functions for stencils, but OpenACC provides some features for optimization.

4.3 Reduction

Another pattern that is often required in linear algebra is the reduction pattern. It combines every element of an input data set into a single element using a certain reduction operation (combiner function). For parallelization, the combiner function must be associative to support reordering of operations.

OpenACC and OpenMP allow directly to compute `reductions` with a clause. Reductions are supported at worksharing levels and parallel regions (`parallel`, `teams`). Listings 1.7 and 1.8 present an example for reductions on different levels of parallelism, i.e. a matrix vector multiply extended with a checksum computation: $\boldsymbol{b} = A \cdot \boldsymbol{x}$, checksum $= \sum_{i=1}^{n} b_i$ with $\boldsymbol{b} \in \mathbb{R}^n$, $\boldsymbol{x} \in \mathbb{R}^m$, $A \in \mathbb{R}^{n \times m}$ and $b_i \in \mathbb{R}$. The reduction value of the scalar `tmp` is already needed right after the inner loop to compute the checksum. For `vector` parallelism in OpenACC (line 5), it is necessary that the variable also appears in a `private` clause to get it updated right at the exit of the loop and

Listing 1.7. Reduction in OpenACC (GPU)

```
 1 #pragma acc parallel private(tmp)
 2 #pragma acc loop gang                            \
        reduction(+:checksum)
 3  for(i=0; i<n; i++) {
 4   tmp = 0;
 5 #pragma acc loop vector reduction(+:tmp)
 6   for(j=0; j<m; j++) {
 7    tmp += A[i][j] * x[j];
 8   }
 9   b[i] = tmp;
10   checksum += tmp;
11 }
```

Listing 1.8. Reduction in OpenMP (GPU)

```
 1 #pragma omp target
 2 #pragma omp teams distribute private(tmp)\
          reduction(+:checksum)
 3   for(i=0; i<n; i++) {
 4     tmp = 0;
 5 #pragma omp parallel for reduction(+:tmp)
 6     for(j=0; j<m; j++) {
 7       tmp += A[i][j] * x[j];
 8     }
 9     b[i] = tmp;
10     checksum += tmp;
11 }
```

Listing 1.9. Fork-join in OpenMP (Phi)

```
 1 #pragma omp declare target
 2  int fib(int n) {
 3   int x, y;
 4   if (n < 2) {return n;}
 5 #pragma omp task shared(x)
 6   x = fib(n - 1);
 7 #pragma omp task shared(y)
 8   y = fib(n - 2);
 9 #pragma omp taskwait
10   return (x+y);
11  }
12 #pragma omp end declare target
13 // [..]
14 #pragma omp target
15 #pragma omp parallel
16 #pragma omp single
17  result=fib(n);
```

Listing 1.10. Unstructured data lifetime in OpenACC (GPU)

```
 1 class CArray {
 2  public:
 3   CArray(int n) {
 4    a = new double[n];
 5 #pragma acc enter data create(a[0:n])
 6     }
 7   ~CArray() {
 8 #pragma acc exit data delete(a[0:n])
 9    delete(a);
10     }
11   void fillArray(int n) {
12 #pragma acc parallel loop
13     for(int i=0; i<n; i++) { a[i]=i; }
14   }
15  private:
16   double *a;
17 };
```

not only at the end of the parallel region. Correspondingly, the `checksum` variable is put properly in the reduction at the `gang` parallelism (line 2). For OpenMP, there are corresponding rules.

As an advantage, OpenMP supports user-defined reductions, especially useful on structured data types, which is currently not possible with OpenACC. Furthermore, the OpenMP `simd` construct also supports reduction operations.

4.4 Fork-Join

The fork-join pattern directs the workflow to be split (forked) into multiple parallel and independent flows and get merged (joined) later again. OpenACC and OpenMP both support `parallel` regions on the device that actually fork control into multiple threads and later return to a single master thread (compare Section 4.1). However, worksharing constructs in parallel regions only support data parallel execution across threads. OpenMP additionally provides task parallel execution on the device via `sections` or `tasks`. It enables parallel execution of instances with different computational work and efficient load balance. The fork-join pattern can also be applied for recursive algorithms such as divide-and-conquer. A simple recursive application is the

Listing 1.11. Superscalar sequence in OpenACC (GPU)

```




#pragma acc parallel loop async(1)

 // F = f(A)


#pragma acc parallel loop async(2)
 // G = g(B)
#pragma acc wait(1,2) async(3)


#pragma acc parallel loop async(3)
 // H = h(F,G)
#pragma acc wait(1)
 // S = s(F)
#pragma acc wait

```

Listing 1.12. Superscalar sequence in OpenMP (GPU)

```
#pragma omp parallel
#pragma omp single
{
#pragma omp task depend(inout:F)
#pragma omp target teams distribute \
     parallel for
 // F = f(A)
#pragma omp task depend(inout:G)
#pragma omp target teams distribute \
     parallel for
 // G = g(B)
#pragma omp task depend(in:F,G)      \
     depend(inout:H)
#pragma omp target teams distribute \
     parallel for
 // H = h(F,G)
#pragma omp task depend(in:F)
 // S = s(F)
#pragma omp taskwait
}
```

computation of Fibonacci numbers in Listing 1.9. This algorithm forks for each recursive call a new `task` and joins them by using the `taskwait` directive. This conceptual behavior can be approximated by host-directed nested parallel constructs in OpenACC.

4.5 Superscalar Sequence

The superscalar sequence pattern describes the parallelization of the serial sequence which executes an ordered list of tasks. In a superscalar sequence, the specific order can be lifted by parallel execution as long as all data dependencies are satisfied. On multi-core processors with attached accelerators, the superscalar sequence can also be interpreted as heterogeneous or hybrid parallelization for the combination of host and device using asynchronous execution.

To denote data dependencies, OpenACC follows a streaming concept that is known from CUDA programming. As seen in Listing 1.11, the streams are expressed by `async` clauses that take a positive integer as stream label. Data that contains dependencies must be put into the same stream (same integer) for sequential ordering. Tasks that can be executed in parallel should be in different streams. The `wait` construct and clause help with the synchronization across different streams. For OpenMP, the tasking model can be applied with the extension of data dependency capabilities. Tasks that do not `depend` on each other can be employed in parallel. In Listing 1.12, all tasks (except the last one listed) start a target region for execution on the device. Other than in OpenACC, the OpenMP host thread that picks up the scheduled task has to wait until the task has been completed to return to the thread pool to execute further tasks.

4.6 Nesting

The nesting pattern is a compositional pattern for creating hierarchies. They are needed for a modular code structure and the incorporation of libraries. Here, we look

Listing 1.13. Update in OpenACC (GPU)

```
 1 void stencilOnAcc(double **a, double **\
        anew, int n, int m) {
 2 #pragma acc parallel present              \
        (a[1:n-2][0:m], anew[1:n-2][0:m])
 3 #pragma acc loop
 4   // stencil computation
 5 }
 6 // [..]
 7 #pragma acc data create(Anew[0:n][0:m]) \
        copyin(A[0:n][0:m]) if(test)
 8 {
 9
10  while (iter < iter_max) {
11    stencilOnAcc(A,Anew,n,m);
12 #pragma acc update host(Anew[1:n-2][0:m])
13
14    swapOnHost(A,Anew,n,m);
15 #pragma acc update device(A[1:n-2][0:m])
16
17    iter++;
18 } }
```

Listing 1.14. Update in OpenMP (GPU)

```
 1 void stencilOnAcc(double **a, double **   \
        anew, int n, int m) {
 2 #pragma omp target map(tofrom:              \
        a[1:n-2][0:m], anew[1:n-2][0:m])
 3 #pragma omp teams distribute parallel for
 4   // stencil computation
 5 }
 6 // [..]
 7 #pragma omp target data                    \
        map(alloc:Anew[0:n][0:m])            \
        map(to:A[0:n][0:m]) if(test)
 8 {
 9  while (iter < iter_max) {
10    stencilOnAcc(A,Anew,n,m);
11 #pragma omp target update                  \
        from(Anew[1:n-2][0:m])
12    swapOnHost(A,Anew,n,m);
13 #pragma omp target update                   \
        to(A[1:n-2][0:m])
14    iter++;
15 } }
```

especially at nested parallelism. In OpenACC, a modular composition can be explored by the ability of nesting `parallel` regions or `kernels` into each other. For OpenMP, some restrictions are imposed to with `target` and `teams` constructs: both are not allowed to be nested in themselves. Only `parallel` directives can be applied inside of `target/teams/parallel` directives.

4.7 Parallel Update

While various parallel data management patterns are defined by McCool et al, no specific pattern displays the data relationship between a host and an accelerator. Therefore, we extend the parallel patterns by defining a *parallel update* pattern. The parallel update pattern does not have a pendant in serial execution, as it exposes capabilities to synchronize data between host and device.

Both programming models support basic parallel update methods like data clauses, data regions and update constructs. The usage of these patterns is illustrated in Listings 1.13 and 1.14 that show a simplified iterative Jacobi solver with stencil computations. Data movement is controlled by data clauses next to the `parallel`, `kernels` or `target` construct, which take a variable list and a map type determining the data transfer direction or creation/deletion. Basic OpenACC map types are `create`, `copy`, `copyin` and `copyout`. OpenMP provides `alloc`, `tofrom`, `to` and `from`, respectively, in combination with the `map` clause. The variable list must only denote arrays or pointers, as scalar variables are transfered automatically. Statically-allocated arrays can be automatically recognized and moved by the compiler. Thus, we did not have to specify them in previous examples. In contrast, the size of dynamically-allocated memory must be manually denoted in the form of array sections or sub arrays (e.g. line 12) representing rectangular or contiguous memory (depending on the construct, base programming language and vendor implementation). Besides data clauses, OpenACC and OpenMP also support data regions (`data`, `target data`) which decouple the data movement from computational regions. The same data map types apply. Hitherto un-

Listing 1.15. Partition in OpenACC (GPU)

```
1 // determine idDev, stIdx & #rows per dev
2 acc_set_device_num(idDev, \
       acc_device_nvidia);
3 #pragma acc parallel loop copy(x[stIdx:\
       rows][0:n],y[stIdx:rows][0:n])
4   // y = a * x (on distributed rows)
```

Listing 1.16. Partition in OpenMP (GPU)

```
1 // determine idDev, stIdx & #rows per dev
2
3 #pragma omp target device(idDev) map(x[\
       stIdx:rows][0:n],y[stIdx:rows][0:n])
4 #pragma omp teams distribute parallel for
5  // y = a * x (on distributed rows)
```

mentioned are the present checks employed for data on the device. OpenACC provides `present_or_copy` (and similar) clauses that test the existence of data on the device and moves the data if necessary. The OpenMP runtime implies this check for all data transfers. OpenACC also allows to explicitly express that a variable is and must be already `present` in a given data context (see line 2). If the variable is not accessible on the device (`if(test)` evaluates to false), the runtime will throw an error. OpenMP applications must specify the `map` clause with inclusive present check. Thus, the program continues executing. The `update` directive allows solely data movement between host and device and can be used flexibly within the corresponding data environment.

Additionally, both models enable an automatic deep copy of flat objects to the device, i.e. structs and classes with static member types. With OpenACC's data API, a manual deep copy of pointer structures is further possible, but tedious. The concept is called unstructured data lifetime and can also be expressed by directives. In Listing 1.10, `enter data create` allocates memory on the device as soon as the constructor of class `CArray` is called. Respectively, `enter data delete` in the destructor destroys the data on the device. Copy clauses are also possible. An OpenMP counterpart does not exist, momentarily.

4.8 Geometric Decomposition

Data reorganization is a necessary pattern for many algorithms. The geometric decomposition pattern divides the data collections into sub domains. For parallel execution on independent data sections, the sub domains should be non-overlapping at best. If this is the case and the sub domains are uniform in size, we call this the partition pattern.

The partition pattern is illustrated by a vector scaling in Listings 1.15 and 1.16: $\boldsymbol{y} = \alpha \cdot \boldsymbol{x}$ with $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n, \alpha \in \mathbb{R}$. Using OpenACC or OpenMP, data subdivision can take place between host and accelerator or between multiple accelerators. The decomposition between host and device can be employed using asynchronous call capabilities covered in Section 4.5. The distribution across multiple accelerators of the same type can be applied by API calls (OpenACC: line 2) or clauses (OpenMP: line 3), respectively, specifying a certain device ID.

The distribution across multiple accelerators of different types (e.g. GPU, Xeon Phi) is only supported by OpenACC. OpenACC provides API calls for setting the current device type (`acc_set_device_type(`*`type`*`)`) and additionally `device_type` clauses that enable device-specific clause tuning for computational work.

5 Conclusion

In the context of structured parallel patterns, we compared the power of OpenACC and OpenMP for accelerators. A summary table is provided in [18]. We conclude that

OpenACC is one step ahead of OpenMP, momentarily. Although OpenACC does not directly support the fork-join pattern, it provides more features concerning the remaining patterns. Contrary, the OpenMP model provides more general concepts such as sections and task parallelism today. Thus, if developers want to start directive-based programming on GPGPUs now, we recommend to use OpenACC. A port to OpenMP 4.0 can be easily carried out, any time, if only features from OpenACC 1.0 were used. Similarly to OpenACC, OpenMP aims to quickly add missing functionality.

Assessing the long-term perspective of both models, the question is whether they will co-exist, converge or diverge. Based on our work in the OpenACC and OpenMP accelerator committees, we assume that they will continue to live independently because of business interests. However, if users advocate for a certain model, vendors cannot neglect their need. While accelerator capabilities of OpenMP, that target a broad user base, might always lag behind OpenACC's, OpenMP might have the advantage in the long term: It is widely excepted in the user community and supported by numerous vendors for broad portability. Additionally, it provides a unified model for programming accelerators and CPUs. On the other hand, the effort for a complete OpenMP 4.0 implementation is significant, possibly preventing full support of all OpenMP concepts in offload regions.

A further uncertainty is the development of future architectures. It is likely that accelerators will get closer to the host processor and/ or might share the same memory. Not forgetting, Amdahl's law still holds. Then, the offload model of OpenACC and OpenMP might lose importance and hosts with large-scaling capabilities might be superior. While in principle OpenACC can also be compiled for the host, OpenMP is already well-known for a productive usage on CPUs. Thus, OpenMP might take the lead with its non-offload parallel features then.

At the end, developers look out for one productive parallel programming model that also delivers performance. While directive-based models might deliver lower performance than low-level approaches, performance differences between equivalent approaches in OpenACC and OpenMP are not expected, if compared on the same target architecture. The only dissimilarity might occur if the general concept differs, for instance, as with asynchronous streams and asynchronous tasks. Unfortunately, we could not investigate performance measurements so far since current OpenACC implementations only exist for GPUs and an OpenMP 4.0 implementation with device offloading capabilities is only existent for Intel's Xeon Phi. These performance examinations are left for future work. Further investigations will also cover a pattern-based comparison between low-level and directive-based accelerator models.

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183 (2006)
2. Beyer, J.C.: OpenACC 2.0 vs OpenMP 4.0 Programming Comparison. GTC Express Webinars, ID GTCE058 (2013)
3. 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)

4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 44–54 (2009)
5. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1991)
6. Ghosh, S., Liao, T., Calandra, H., Chapman, B.: Experiences with OpenMP, PGI, HMPP and OpenACC Directives on ISO/TTI Kernels. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 691–700 (2012)
7. Hoshino, T., Maruyama, N., Matsuoka, S., Takaki, R.: CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In: 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 136–143 (2013)
8. Khronos OpenCL Working Group: The OpenCL Specification, v2.0 (2014)
9. Lee, S., Vetter, J.S.: Early Evaluation of Directive-based GPU Programming Models for Productive Exascale Computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 23:1–23:11. IEEE Computer Society Press, Los Alamitos (2012)
10. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early Experiences with the OpenMP Accelerator Model. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 84–98. Springer, Heidelberg (2013)
11. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional (2004)
12. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation, 1st edn. Morgan Kaufmann (2012)
13. OpenACC-Standard.org: The OpenACC Application Programming Interface, v2.0 (2013)
14. OpenMP ARB: OpenMP Application Program Interface, v. 4.0 (2013)
15. Reyes, R., Lopez, I., Fumero, J., De Sande, F.: Directive-based Programming for GPUs: A Comparative Study. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 410–417 (2012)
16. Wang, Y., Qin, Q., See, S.C.W., Lin, J.: Performance Portability Evaluation for OpenACC on Intel Knights Corner and Nvidia Kepler. HPC China (2013)
17. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC – First Experiences with Real-World Applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012)
18. Wienke, S., Terboven, C., Beyer, J.C., Müller, M.S.: A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing, slides (2014), `https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/WienkeEtAl_OpenACC-OpenMP-PatternComparison.pdf`
19. Wolfe, M.: Compilers and More: Accelerated Programming. HPC Wire (2013)
20. Wolfe, M.: Programming Heterogeneous X64+GPU Systems Using OpenACC. IEEE Comupter Society Webinar (2013)