A Pattern for Overlapping Communication and Computation with OpenMP* Target Directives

Jonas Hahnfeld[1(✉)], Tim Cramer[1], Michael Klemm[2], Christian Terboven[1], and Matthias S. Müller[1]

[1] Chair for High Performance Computing & IT Center, JARA–HPC, RWTH Aachen University, 52074 Aachen, Germany
{hahnfeld,cramer,terboven,mueller}@itc.rwth-aachen.de
[2] Intel Deutschland GmbH, 85622 Feldkirchen, Germany
michael.klemm@intel.com

Abstract. OpenMP* 4.0 introduced initial support for heterogeneous devices. OpenMP 4.5 improved programmability and added capabilities for asynchronous device kernel offload and data transfer management. However, the programmers are still burdened to optimize data transfer for improved performance and to deal with the limited amount of memory on the target device. This work presents a pipelining concept to efficiently overlap communication and computation using the OpenMP 4.5 `target` directives. Our evaluation of two key HPC kernels shows performance improvements of up to 24% and the ability to process data larger than device memory.

1 Introduction

Accelerators and coprocessors of different kinds continue to impact the HPC landscape: From the current Top500 list, a total of 97 systems are equipped with GPU devices from NVIDIA and AMD or the Intel® Xeon Phi™ coprocessor.

OpenMP* other strives to ease the burden for the programmer by providing a rich set of compiler directives complemented by API routines to control runtime behavior. OpenMP 4.0 introduced support for heterogeneous programming with the `target` construct family. It allows to transfer the control flow from a host thread to a thread on the target device and also provides means to direct the data flow between host and device. Being vendor-neutral, a target device in OpenMP may be a GPU, a coprocessor, or other heterogeneous devices like a DSP engine or an FPGA. OpenMP 4.5 addresses some shortcomings and added support for asynchronous offloading from the host to devices.

Nevertheless, achieving good application performance on heterogeneous clusters still puts a burden on the programmer, who, for instance, has to lay out data structures and compute kernels in appropriate ways. Today's predominant configuration is a host that is equipped with DDR memory and (multiple) devices equipped with memory of much smaller capacity yet much higher memory bandwidth. Effective slicing and management of the working set that is present on

B.R. de Supinski et al. (Eds.): IWOMP 2017, LNCS 10468, pp. 325–337, 2017.
DOI: 10.1007/978-3-319-65578-9_22

the device is crucial for achieving high application performance. Well-explored optimization techniques include the extension of device regions to enable data to reside on the device memory for reuse.

In this work, we emphasize on the technique of overlapping communication and computation to overcome bandwidth and latency bottlenecks in transfers between host and devices, for instance with the PCIe bus. We describe the realization of a pipeline concept based on features recently introduced with OpenMP 4.5 and present a performance evaluation with two devices per host. Applying our pattern may not only ease the implementation of complex applications exploiting accelerator devices, we will also show that it can improve performance via the overlapping and better use of the memory and bus capabilities. It can also enable the use of devices for problems that are larger than the device memory.

2 Related Work

The concept of overlapping communication and computation for parallel applications is widely spread. It is considered a key technique to obtain performance for architectures that rely on some form of message passing to transfer data to the computational units of the executing system. To best of our knowledge, we present a corresponding pipelining pattern to overlap communication with the offload target with computation for the first time. Another study [11] applies the concept to an Intel Xeon Phi coprocessor using the MPI programming model.

Several studies have shown that performance can be significantly increased by overlapping communication and computation (e.g., [3,8]). LibNBC by Hoefler et al. [8] is a portable library that provides support for non-blocking collective operations. It laid the foundation for similar concepts that have been introduced in the Message Passing Interface (MPI) version 3.0. Furthermore, their work references to further studies dealing with the overlapping of communication and computation in general. Extensions of MPI, such as the work of Aji et al. [1], address the problem of accessing GPU memory during MPI communication. In contrast to our work, these studies focus on applying non-blocking MPI primitives.

Beltran et al. [2] start multiple threads on each accelerator and achieve an overlap by efficiently scheduling them. Liu et al. [9] present double buffering for matrix multiplications and implement it with extensions to OpenMP. In contrast, we will use standard-compliant features from OpenMP 4.5 which will result in a reusable pattern across different accelerators and, thus, more portable code.

Miki et al. [10] propose language extensions for OpenACC* other that overlap communication and computation for stencil-type kernels. Cui et al. [6] propose pipelining directives to extend OpenMP. Our work uses the existing directives of OpenMP 4.5 instead and does not restrict the pattern to stencil computations that require the presence and exchange of halo cells. It is generally applicable to any type of applications that allows for splitting computations into subcomputation to overlap communication and computation.

Some OpenACC compilers employ double-buffering strategies to speed-up the data transfer itself [4]. It works by pre-allocating buffers before starting a transfer to the target device and thus physical allocation of buffers and the corresponding data

transfers can be overlapped. Chen et al. [5] discuss different buffering schemes for DMA and their latencies. However, they do not investigate the overall improvement in runtime when overlapping DMA transfers and computation. Our approach is orthogonal to this, as we employ OpenMP pragmas to pipeline data transfers and computation at the application level. If an OpenMP implementation would offer such an underlying mechanism to improve low-level communication, our approach could transparently make use of it and automatically apply the low-level double buffering to further speed-up communication with the offload device.

3 OpenMP for Accelerators and Coprocessors

OpenMP's accelerator model is based on structured blocks with `target` directives to tag them for offload execution. A target region may be executed by OpenMP threads on a different device in a distinct data environment. By using `map` clauses a programmer can express which (non-scalar) variables have to be made available on a device. In OpenMP terms, this is called *mapped* from the host to the device, because the host may or may not share the memory with the device. In the case of devices with separate memory this typically involves copy operations. The `map` clause accepts, among others, the motion attributes `to`, `from`, and `tofrom` determining the point in time and the direction of copy operations.

The usage of `target data` regions allows to reduce data transfer in the case of multiple consecutive `target` regions using the same variables, as the data environment on a device is persistent for the whole duration of the `target data` region.

It is important to know that the `map` clause creates a fixed association between the host and the target device. In consequence, a re-mapping of a memory region on a device with an address on the host is not possible, if it was mapped before. In order to make the device data environment consistent between two `target` regions encountered in the same `target data` region, the `target update` construct can be used. The specified motion clause determines if the values from the host or from the target device have to be updated.

Listing 1.1. Dependencies with stand-alone directives for managing the device data environment.

```
1  double A[100];
2  #pragma omp target enter data nowait \
3      ①   map(to: A[0:100]) depend(out: A[0:100])
4
5  #pragma omp target map(to: A[0:100]) \
6          nowait depend(inout: A[0:100])
7  {
8    ③ // Computation that uses A[0:100]
9  }
10
11 #pragma omp target exit data nowait \
12     ⑤   map(release: A[0:100]) depend(in: A[0:100])
```

Optimizing the data transfers was hard to realize in OpenMP 4.0. With all data transfers defined as being synchronous, it was impossible to overlap the computation and communication. OpenMP 4.5 defined the `target` region to

be an implicit task, meaning it is executed as if it was surrounded by a `task` construct. The execution of a target task may be deferred if the `nowait` clause is added to the `target` construct to make the execution of the target code and the corresponding data transfers asynchronous.

In order to have an asynchronous data transfer without executing user code on the device, the stand-alone directive `target enter data` can be used to map data to a device. Correspondingly, the `target exit data` will unmap the specified variables from the device data environment and might copy back the values from the target device to the host. Both of these directives also generate a target task which might be deferred.

Finally, the `depend` clause can be added to all device directives to associate dependencies with the generated target task. It supports the same dependency types `in`, `out`, or `inout` introduced with the OpenMP tasking model. For the use case of asynchronous data transfer and kernel execution, this feature allows to defer the execution of a `target` region until the required data is transferred to or from a device and thus bring the data transfers and compute regions into a specific order as shown in Listing 1.1. After the `target enter data` directive ① has executed, the dependency ② is resolved and the computation ③ can start. As the mapping is already present on the device, this will not result in additional data transfer. The dependency ④ is satisfied when the kernel has finished and the `target exit data` operation ⑤ finally executes.

4 Pipelining Concept for Overlapping Communication

Depending on the available hardware, the mapping and/or the data transfer to or from a target device might be relatively slow compared to the available memory bandwidth on the host or the target device. Thus, the communication time of an application using a large amount of data might become a significant overhead factor and limit scalability and performance. To reduce this communication overhead we present a pipelining concept.

The main idea of the pipelining concept is to divide a single operation into smaller sub tasks. By interleaving these smaller sub tasks we can increase the throughput of a system, because different kinds of sub tasks can use different parts of the available hardware resources at the same time. In our case, a sub task belongs to one of the two kinds: computation or communication.

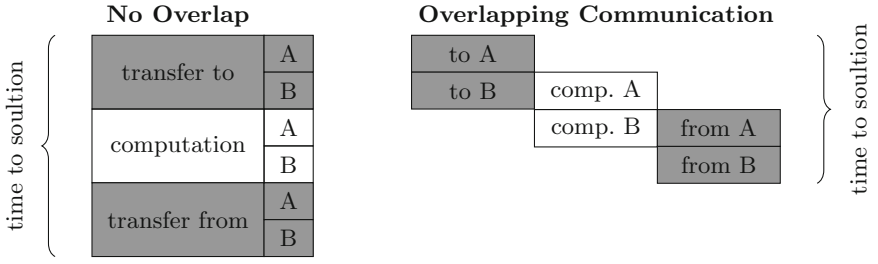

Fig. 1. Pipelining concept for overlapping computation and communication.

Figure 1 exemplifies the concept. The white boxes represent the computation and the gray boxes depict transfers. The example assumes that the computation can be split into two parts. The data required for each computation is transferred to a device and the result is transferred back to the host afterwards.

On the left hand side of Fig. 1, communication and computation are not overlapped, but are executed in order. On the right hand side, each of the two sub tasks are put into the pipeline stages. Thus, the computation on a target device does not have to wait until the complete data block is transferred to the device. As can be seen, this leads to an improved utilization of both the host and the offload device.

We refer to the first *transfer to* at the beginning as *wind-up phase* and to the last *transfer from* as the *wind-down phase*. In both of these phases, no computation can happen as the pipeline has to be filled with data transfers or the system has to wait until all in-flight data transfers have been completed. It can be seen that the time to solution using the pipelining concept decreases significantly in this case.

4.1 Performance Projection of Pipelining Pattern

To estimate the potential gain of overlapping communication and computation using the pipelining pattern, we conduct a very simple, yet effective performance projection. We will also use this simple performance model in Sect. 5 to assess the measured performance.

The total runtime t_{exec} of an offloaded kernel consists of

$$t_{exec} = t_{comp} + t_{comm}, \tag{1}$$

where t_{comp} is the time for the computation on the device and t_{comm} the communication time to transfer control and data. The latter can be predicted for a given amount of data d by

$$t_{comm} = \frac{d}{B} + t_{overhead}. \tag{2}$$

This assumes that d is sufficiently large so that the transfer of d saturates the maximum bandwidth B available. $t_{overhead}$ is the time that the runtime needs for preparational tasks. Depending on the data size d or the runtime implementation this overhead time may be significant for the overall communication time t_{comm}, as will be discussed below. In some cases, $t_{overhead}$ may also depend on data size d.

Based on these characteristics, the maximum optimization o_{max} is given as

$$o_{max} = \frac{min(t_{comp}, t_{comm})}{t_{exec}}. \tag{3}$$

Thus, pipelining transfers and computation works best in cases where the communication and computation time are equally balanced. The optimization potential approximates to a performance increase of up to $o_{max} = 0.5$, not taking the wind-up and wind-down phases into account.

Typically, the available memory of target devices like the Intel Xeon Phi coprocessor or GPUs is significantly smaller than the memory on the host. With the pipelining concept, a device kernel can use more memory than available on the device by transferring the necessary data chunk-wise and free any memory chunk as soon as the partial result was transferred back to the host. This forms a second promising application scenario for the concept in addition to the first one, namely the speedup.

4.2 Implementation with OpenMP

The `map` clause in OpenMP creates fixed associations between device and host. It is not possible to map a specific memory region from the host to a buffer on the device which was previously associated with a different address on the host.

There are multiple possibilities to overcome this limitation: First one could copy the needed data to a temporary buffer which is then transferred to the device using a `target update`. As a second option, we can allocate a single buffer for the whole array in a `target data` region. In a `target update`, we can then specify the corresponding start index to transfer the needed part of the array. Lastly, we can create a new buffer for each block of the array that has to be transferred. Here, OpenMP 4.5 offers the above mentioned stand-alone directives for mapping: `target enter data` and `target exit data`.

While the first solution would surely work, it doesn't promise to give the best performance due to the extra copy on the host. The second alternative fails to allocate the buffer if it exceeds the device memory. Creating a new buffer for each block solves this problem as memory can be freed on `target exit data` after the computation has finished. This allows to process more memory than available on the device at one moment.

Figure 2 shows the dependencies that have to be specified when working with the stand-alone directives. The first of these dependencies are based on the data usage: First, a specific block of data has to be allocated and transferred to the device. Second, the computation on the device can be done. Finally, the used data can be freed again.

Moreover, it has to be ensured that there are at most two buffers allocated at the same time. Hence, we need an explicit dependency between, for example, *exit #0* and *enter #2*. If this connection was omitted, there would be no limitation on how many *enter* tasks can start. This would be problematic because all *enter* tasks could run before *exit #0* frees the first part of the data, possibly exceeding the device memory. There are also dependencies between each *enter* and each *compute*. That is to avoid oversubscription which would negatively impact performance.

Listing 1.2 shows the code snippet with the OpenMP directives and their required dependencies. Each block of data is allocated on and transferred to the device with a `target enter data`. The computation is afterwards done in a `target` construct. After the computation has finished, `target exit data` will free the data on the device.

We specify the dependencies for data usage with the corresponding array section also given in the `map` clause. For mutual exclusion of the *enter* and

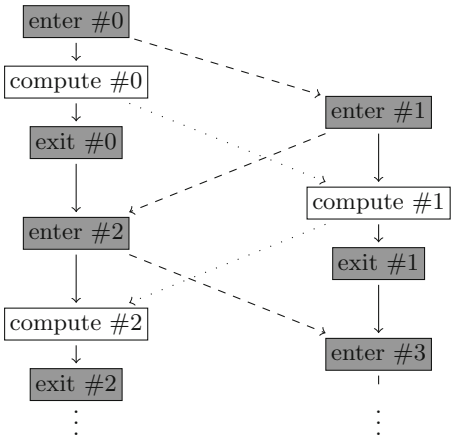

Fig. 2. Graph of the dependencies between target tasks. Continuous lines visualize dependencies based on the data usage, while dashed and dotted ones stand for the mutual exclusion of *enter* and *compute* tasks, respectively.

compute tasks one can use two `int` variables. These dummy variables are used to make the OpenMP implementation aware of the dependencies, but are not used in the code apart from their presence in the `depend` clause.

For simplicity, the code snippet only shows the default case in the middle of the loop iteration space, but not the wind-up and wind-down phase. In the first iteration of the loop with `block = 0`, we do not depend on the previous block having exited by omitting the dependence on `A[(block - 1) * LEN:LEN]`. Additionally, when the end of the iteration space is reached, there is no next block to transfer and therefore no *enter* task.

Listing 1.2. Declaring task dependencies with OpenMP for pipelining concept of multiple blocks with length *LEN* each. Special cases for `target enter data` are omitted for better readability.

```
1  double A[BLOCKS * LEN];
2  int enter, compute;
3
4  #pragma omp target enter data nowait map(to: A[0:LEN]) \
5          depend(out: enter) depend(out: A[0:LEN])
6  for (int block = 0; block < BLOCKS; block++) {
7      #pragma omp target enter data nowait depend(inout: enter) \
8              map(to: A[(block + 1) * LEN:LEN]) \
9              depend(out: A[(block + 1) * LEN:LEN]) \
10             depend(in: A[(block - 1) * LEN:LEN])
11     #pragma omp target nowait depend(inout: compute) \
12             map(to: A[block * LEN:LEN]) \
13             depend(inout: A[block * LEN:LEN])
14     {
15         // do computation here
16     }
17     #pragma omp target exit data nowait map(release: A[block*LEN:LEN])
           \
18             depend(inout: A[block * LEN:LEN])
19 }
```

Instead of creating a new buffer for each memory transfer, it would also be possible to use the *device memory routines* introduced with OpenMP 4.5. However, these routines are not available for Fortran and are only defined for C and C++. In addition, they do not support task dependencies and would thus have to be wrapped in regular OpenMP tasks to model proper synchronization between the different stages of the pipeline. They also require a developer to manage the buffers explicitly and free them manually. Thus, the usage of the stand-alone directives is more convenient and more productive compared to the usage of the *device memory routines*. For these reasons, we concentrate on the investigation of the directives in the following.

4.3 Applying the Concept for Multiple Target Devices

A natural desire is to extend the above approach to also cover multiple devices and to extend the pipelining concept such that it can overlap communication and computation across these devices. The `target` constructs support the `device` clause to specify the device a `target` construct shall use at runtime. Thus, a simple mechanism to start multiple concurrent `target` regions, e.g., by iterating over all available devices is sufficient. Managing the corresponding device data environment works in a similar way by using the stand-alone directives or `target update` as discussed above. To ensure that all of operations have finished, the `taskwait` construct is suitable.

Based on this scheme, we can apply our concept and specify dependencies between tasks as described above. That way, we can for example allow unrelated tasks to execute in parallel and overlap computation with a data transfer or exchange that is only needed in the next step of the algorithm.

5 Evaluation

To show the applicability of our approach, we evaluated the concept with the `dgemm` kernel. Therefore, we used the implementation given in the Intel® Math Kernel Library which delivers a good performance on Intel architectures. For the evaluation of the presented pipelining concept for multiple target devices, we use a sparse Conjugate Gradients (CG) [7] method as a representative real-world compute kernel.

All presented kernels were measured on a 2-socket Intel® Xeon® E5-2650 system (codename "SandyBridge"), which is clocked at 2.00 GHz and has 16 physical cores in sum. The system includes two Intel Xeon Phi 5110p coprocessors with 8 GB of main memory and 60 cores (clocked at 1.053 GHz) each, connected via PCIe Gen2 with 16 lanes. In our setup, we measured approximately 6.7 GB/s with `target update` constructs between device and host. For all kernels, we used version 17.0 of the Intel compiler that already implements all required OpenMP 4.5 features. We present the minimum runtime of 10 repetitions as this will indicate the best performance that the system can deliver.

5.1 Matrix-Matrix-Multiplication

This section will show how the pipelining concept can be used to compute a problem whose memory requirements exceed the device memory. For this, we use a matrix-matrix-multiplication $A \cdot B = C$, where $A, B, C \in \mathbb{R}^{n \times n}$. The size of each matrix is 24576^2 `double` elements, which requires $3 \cdot 4.83$ GB ≈ 14.5 GB in total. We transpose the second matrix so that we can use rows instead of columns for the sub tasks of the multiplication. This results in contiguous storage in memory which is a requirement for the `map` clause.

Since the size of the matrices exceeds the available memory on the device, A and C need to be split into N and B into M parts that can be transferred separately. For the calculation of $A \cdot B^T = C$, the rows of A can be reused for multiple blocks of B and the result is stored in the corresponding parts of C. To minimize data movement the parts of A and C are transferred in `target data` regions. Furthermore, we apply the pipelining concept to B to hide the latency. In theory, we should also be able to apply the pipelining concept to A and C. Unfortunately, this is currently not possible due to some issues in the Intel compiler.

To minimize the data transfers, M has to be as small as possible because matrix B has to be transferred multiple times. For our test, case we chose $M = N = 4$ (i.e., four blocks for each matrix), which has shown to perform best. In theory choosing $N = 2$ fits into the device memory and thus should be beneficial in term of performance. However, this results in stability issues on the device. B could be split into more parts but that does not result in a lower runtime.

In total, the maximum memory usage will be $\frac{4 \cdot 4.83 \text{ GB}}{4} = 4.83$ GB on the device, because we need to store two parts of B simultaneously. For the transfer, we expect $(2 + 4) \cdot 4.83\,\text{GB}$ as B has to be transferred 4 times. In addition, measurements show that the coprocessor needs approximately 1.35 s to allocate each matrix. Based on (2), this sums up to

$$t_{comm} = \frac{(2+4) \cdot 4.83\,\text{GB}}{6.7\,\text{GB/s}} + (2+4) \cdot 1.35\,\text{s} \approx 4.33\,\text{s} + 8.1\,\text{s} = 12.43\,\text{s}.$$

With the measured runtime of $t_{exec} = 68.38\,\text{s}$ on the device, this leaves

$$t_{comp} = 68.38\,\text{s} - 12.43\,\text{s} = 55.95\,\text{s}$$

for the computation.

Based on (3), we should hence be able to obtain a maximum optimization of

$$o_{max} = \frac{min(55.95\,\text{s}, 4.33\,\text{s} + 8.1\,\text{s})}{68.38\,\text{s}} \approx 18.2\,\%.$$

However, as we can currently only apply our concept to B and not yet to A and C we are not able to save more than $4 \cdot (0.72\,\text{s} + 1.35\,\text{s}) = 8.28\,\text{s}$ which would mean an optimization of

$$o_{max} = \frac{8.28\,\text{s}}{68.38\,\text{s}} \approx 12.1\%.$$

Table 1. Minimum runtime on host and device of 10 repetitions with `dgemm`.

Device	Time
Host device	125.83 s
Target device	68.38 s
w/pipelining concept	61.54 s

Table 1 lists the minimum runtimes on the host and target device. It can be seen that the matrix-matrix-multiplication on the target device (68.38 s) is significantly faster than the host (125.83 s) despite having to transfer the data. Using the pipelining concept yields another improvement of approximately 10% resulting in a runtime of 61.54 s. This is slightly below the estimation, because the model does not account for additional overhead introduced by the pipelining. However, it shows the applicability of the approach.

5.2 Conjugate Gradients Method on Multiple Target Devices

For the evaluation of the pipelining concept on multiple target devices, we implemented a Conjugate Gradients (CG) method. This compute kernel represents a popular and widely used iterative algorithm to approximate the solution for a sparse linear equation system. The computation is dominated by a sparse matrix-vector multiplication (SpMV). In general, the data transfer time for the execution of such a method is low compared to the compute time on a target device, because of the iterative nature of the CG algorithm. However, the amount of memory of a target device is typically small compared to the amount of memory of the host. In order to overcome the size limitation, our implementation of the CG solver can use multiple Intel Xeon Phi coprocessors by distributing the data.

We use a symmetric matrix with a regular sparsity pattern of five non-zero elements per row (except for the first and last few rows). Similar patterns emerge from PDEs with regular discretization. Thus, the decomposition does not require any complex partition algorithms for an adequate load balancing on the target devices. The matrix contains 80 million rows (about 400 million non-zero elements), which results in a memory footprint of about 4.8 GB in a compressed row storage (CRS) format. In addition to the right-hand side vector, the solution vector and temporary vectors (640 MB each) are required by the algorithm. This memory footprint exceeds the memory capacity of a single Xeon Phi coprocessor as used in our setup.

To decompose the data for two devices, we divide the matrix and each vector into two partitions. For all vector-vector operations, local results can be computed. Thus, for the complete solving process no additional data of the matrix needs to be exchanged between the two devices. However, the (partial) matrix-vector multiplication requires the complete intermediate result on each device for each single iteration in order to compute the corresponding (partial) output. Therefore, we need to exchange half a vector from each device in every solving step.

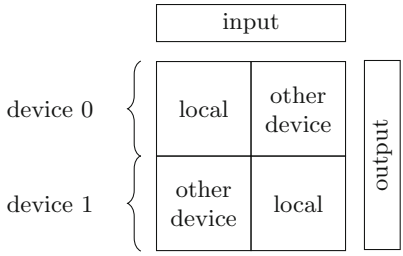

Fig. 3. Partitioning of the matrix, marking parts that are multiplied with local parts of the input vector and blocks that need data from the other device.

Nevertheless, the partial local computation of the SpMV result can be started directly, because the corresponding half of the vector is already present on the device. This enables us the apply our pipelining concept to overlap the transfer of the intermediate result from the other device with the computation that only needs the local part of the vector. After the transfer has completed, the computation can then be finalized with the data received from the other device.

To keep the computation efficient, we already partition the matrix in CRS format on the host: We create four sub-matrices and put each value into the corresponding block as shown in Fig. 3. Thus, it is not necessary to determine which part can be computed with the local part of the right-hand side vector in each iteration.

For the evaluation, we use two different versions of our CG solver: one baseline version that does not overlap the computation and communication, and one improved version that does. In the baseline version, each iteration spends roughly 250 ms for the matrix-vector multiplication which includes exchanging the input vector between the two devices. It first transfers the two parts to the host and then back to the other target device. This can be done concurrently for both devices and hence we assume a communication time t'_{comm} for each iteration based on (2):

$$t'_{comm} = 2 \cdot \frac{\frac{640}{2}\,\mathrm{MB}}{6.7\,\mathrm{GB/s}} \approx 96\,\mathrm{ms}.$$

The remaining time is spent for the computation which amounts to

$$t'_{comp} = 250\,\mathrm{ms} - 96\,\mathrm{ms} = 154\,\mathrm{ms}.$$

Based on these expected timings and (3), the upper bound for the optimization is determined by

$$o_{max} = \frac{min(154\,\mathrm{ms}, 96\,\mathrm{ms})}{250\,\mathrm{ms}} \approx 38.4\%.$$

In summary, the presented pipelining concept reduces the computation time of the dominating matrix-vector multiplication by roughly 32%, from 254 s to 173 s.

As in the previous section, the improvement is again lower than the estimated maximum without additionally introduced overhead. Since our concept is only applicable on this most time-consuming kernel, the overall improvement for the total application is lower (about 24%).

6 Conclusion

We have shown how communication and computation can be overlapped when using OpenMP 4.5 `target` directives for a contemporary coprocessor. Besides simplifying programmability, the use of pipelining schemes can improve application performance by effectively hiding communication latencies between the host and the offload devices. It also provides an effective means to offload kernels that require more memory than is available on the device. Our pipelining scheme is portable and increases programmer productivity.

We have evaluated our implementation with two important kernels in HPC, matrix-matrix multiplication, and a sparse Conjugate Gradients solver. Our benchmarks show that overlapping communication and computation effectively reduces the runtime of these kernels by up to 24% for the CG solver. This achievement corresponds to a simple back-of-an-envelope performance model we have presented. The speed-up encourages a deeper evaluation of the profitability of our pattern with different codes.

As future work we plan to investigate the feasibility and profitability of the pattern on current GPUs with, for instance, the OpenACC programming model. We will also perform a performance comparison of the high-level OpenMP or OpenACC implementation with direct low-level implementations like the Intel Coprocessor Offload Infrastructure (COI) or NVIDIA CUDA. This will also include the evaluation how the presented CG will profit from faster interconnects such as NVLink introduced with NVIDIA Pascal.

Acknowledgment. Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant Number 01IH13008A (ELP). Simulations were performed with computing resources granted by JARA-HPC from RWTH Aachen University under project jara0001.

References

1. Aji, A.M., Panwar, L.S., Ji, F., Murthy, K., Chabbi, M., Balaji, P., Bisset, K.R., Dinan, J.S., Feng, W.C., Mellor-Crummey, J., Ma, X., Thakur, R.S.: MPI-ACC: accelerator-aware MPI for scientific applications. IEEE Trans. Parallel Distrib. Syst. **27**(5), 1401–1414 (2016)
2. Beltran, V., Carrera, D., Torres, J., Ayguadé, E.: CellMT: A cooperative multithreading library for the Cell/B.E. In: 2009 International Conference on High Performance Computing (HiPC), pp. 245–253, December 2009
3. Brightwell, R., Riesen, R., Underwood, K.D.: Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. Int. J. High Perform. Comput. Appl. **19**(2), 103–117 (2005). http://hpc.sagepub.com/content/19/2/103.abstract
4. Castelló, A., Peña, A.J., Mayo, R., Balaji, P., Quintana-Ortí, E.S.: Exploring the suitability of remote GPGPU virtualization for the OpenACC programming model using rCUDA. In: Proceedings of the 2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, pp. 92–95 (2015). http://dx.doi.org/10.1109/CLUSTER.2015.23
5. Chen, T., Sura, Z., O'Brien, K., O'Brien, J.K.: Optimizing the Use of Static Buffers for DMA on a CELL Chip. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 314–329. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72521-3_23
6. Cui, X., Scogland, T.R., de Supinski, B.R., Feng, W.C.: Directive-based pipelining extension for OpenMP. In: Proceedings of the 2016 IEEE International Conference on Cluster Computing, pp. 481–484 (2016)
7. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. J. Res. Natl. Bur. Stand. **49**(6), 409–436 (1952)
8. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and performance analysis of non-blocking collective operations for MPI. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, pp. 52:1–52:10. ACM, New York (2007). http://doi.acm.org/10.1145/1362622.1362692
9. Liu, F., Chaudhary, V.: Extending OpenMP for heterogeneous chip multiprocessors. In: 2003 International Conference on Parallel Processing, Proceedings, pp. 161–168, October 2003
10. Miki, N., Ino, F., Hagihara, K.: An extension of OpenACC directives for out-of-core stencil computation with temporal blocking. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives, WACCPD 2016, pp. 36–45. IEEE Press, Piscataway (2016)
11. Si, M., Ishikawa, Y., Tatagi, M.: Direct MPI library for Intel Xeon Phi coprocessors. In: 2013 IEEE International Parallel and Distributed Processing Symposium Workshop and PhD Forum (IPDPSW), pp. 816–824. IEEE (2013)