Manage OpenMP GPU Data Environment Under Unified Address Space

Lingda Li[1(✉)], Hal Finkel[2], Martin Kong[1], and Barbara Chapman[1]

[1] Brookhaven National Laboratory, Upton, USA
{lli,mkong,bchapman}@bnl.gov
[2] Argonne National Laboratory, Lemont, USA
hfinkel@anl.gov

Abstract. OpenMP has supported the offload of computations to accelerators such as GPUs since version 4.0. A crucial aspect in OpenMP offloading is to manage the accelerator data environment. Currently, this has to be explicitly programmed by users, which is non-trival and often results in suboptimal performance. The unified memory feature available in recent GPU architectures introduces another option, implicit management. However, our experiments show that it incurs several performance issues, especially under GPU memory oversubscription. In this paper, we propose a compiler and runtime collaborative approach to manage OpenMP GPU data under unified memory. In our framework, the compiler performs data reuse analysis to assist runtime data management. The runtime combines static and dynamic information to make optimized data management decisions. We have implement the proposed technology in the LLVM framework. The evaluation shows our method can achieve significant performance improvement for OpenMP GPU offloading.

Keywords: Data management · Unified memory
OpenMP offloading · Compiler · Runtime · LLVM

1 Introduction

Today's computing systems rely on accelerators to achieve performance and energy efficiency goals. As the most popular accelerator nowadays, the massive threading ability of GPUs can especially benefit applications with large amounts of parallelism, such as scientific computing and machine learning. Therefore, GPU is and will remain a crucial component of supercomputing systems in the foreseeable future. For instance, in the next OLCF supercomputer, Summit, each node is equipped with 6 NVIDIA Volta GPUs while the number of CPUs remains 2 [2].

In order to leverage accelerators like GPUs, OpenMP 4.0 introduced the ability to offload computations to accelerators [3]. It is called device offloading. Compared to native GPU programming models such as CUDA [15] and OpenCL [19], using OpenMP for GPU programming has a shorter learning curve for users and is more performance portable. Compared to other directive based methods like OpenACC [1], OpenMP has a broader user community and better compiler support. Therefore, we expect the number of OpenMP+GPU users will continue to grow.

However, writing efficient GPU programs is still a non-trivial job with OpenMP. One of the biggest challenges in GPU programming is how to efficiently program GPU memory. Normally, CPU and GPU are attached with separate memory since they have different memory preferences. While CPU prefers low access latency, GPU performance is more sensitive to memory bandwidth compared with latency. Separate memory also helps reduce memory contention caused by sharing. Traditionally, CPU and GPU use separate memory spaces for their individual memory. As a result, they cannot access each other's memory, and data exchange has to be managed explicitly by programmers.

To ease the programming of GPU memory, a feature called *unified memory (UM)* is introduced in recent NVIDIA GPU architectures. Unified memory introduces a single memory space which covers both CPU and GPU memory. From programmers' perspective, they do not need to worry about the location of accessed data, and data is moved between CPU and GPU by the underlying system software automatically if necessary. The burden of programming data transfer is relieved.

The other major advantage of unified memory is that it enables running kernels with memory footprints larger than the GPU memory capacity. Without on demand page migration of unified memory, GPU offloading is possible only if the dataset fits into the GPU memory. While with it, part of data can reside in the CPU memory, and they will be fetched into the GPU memory when actually required at runtime. These advantages promote more usage of unified memory in future GPU programming.

Every story has two sides. As we will show, unified memory also brings many challenges along with its benefits. First of all, page fault overhead can be significant in cases when data transfers dominate in the execution. More importantly, although unified memory is able to address working sets that exceed the GPU memory capacity, significant data thrashing often happens in such scenarios. Programmers often have no clues about these issues. Therefore, we believe it is crucial to address the performance issues of unified memory for OpenMP offloading, and it would be preferable if the solution is transparent to programmers.

This paper makes the following contributions for this goal.

- First, we analyze the performance of unified memory. The results reveal that its performance mainly depends on accessed data properties, including size, access density and reuse situation (Sect. 3).
- We design a compiler-runtime collaborative framework to optimize unified memory performance and implement it in Clang and LLVM OpenMP runtime [12]. The proposed method analyzes data object properties to find out proper optimization strategies, which are applied at runtime (Sect. 4).
- The experimental results demonstrate that our technique can improve unified memory performance significantly while having low overhead (Sect. 5).

2 Related Work

Since the introduction of device offloading in OpenMP 4.0, several compilers have adopted this feature. For instance, [5] describes how to implement this extension in the LLVM framework. Our optimization uses this work as the baseline.

There are several proposals to simplify and optimize the GPU memory management. CGCM [10] provides compiler and runtime support to automatize the GPU memory management for CUDA programs. Pai *et al.* propose a software coherence mechanism to reduce redundant data transfers between the CPU and GPU [17]. Zhao and Xie propose to leverage hybrid DRAM and NVM GPU memory systems and a data migration mechanism to reduce GPU power consumption [20]. These works aim at traditional GPU programs where data movement is managed explicitly by users, and do not consider unified memory.

Some recent research aims to study or improve the performance of unified memory. In the presence of heterogeneous memory, Agarwal *et al.* propose that the ratio of data allocation in each memory should be proportional to the memory bandwidth in order to achieve the highest total bandwidth [4]. The method we propose in this paper is orthogonal to this work.

Several research efforts have studied and optimized OpenMP device data management. Grinberg *et al.* introduce a method to use unified memory within the current OpenMP implementation [8]. Mishra *et al.* study the OpenMP offloading performance under unified memory [14]. Cui *et al.* propose a pipeline directive to break down OpenMP parallel loops and thus achieve device computation and communication overlapping [7]. Hahnfeld *et al.* propose to use existing OpenMP 4.5 directives for similar purposes [9]. Olivier *et al.* discuss double buffering for Intel Xeon Phi processors in OpenMP [16]. These methods are limited to cases where data access patterns are analyzable, and they also require programming efforts. In contract, our work is able to address unpredictable memory access pattern using unified memory, and does not require inputs from users.

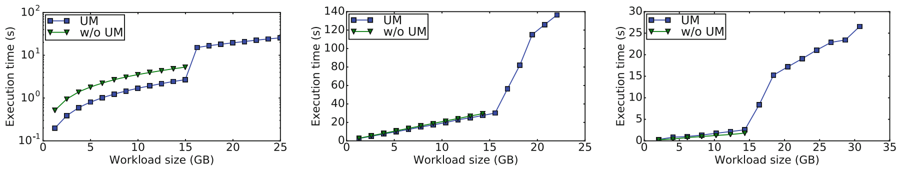
3 Unified Memory Analysis

As the first step, we compare the performance of unified memory with that of traditional GPU memory management approach, and analyze how it performs in different scenarios. Table 1 shows our benchmarks. We use the OpenMP offloading version of BFS, CFD, and SRAD from the Rodinia benchmark suite in our experiments [6,14]. For each benchmark, we generate inputs with various sizes to study the performance impact of workload sizes. The detailed experimental setup is described in Sect. 5.1. We modified the LLVM OpenMP runtime so that it supports the placement of data in unified memory.

Figure 1 illustrates the GPU execution time when data is placed in unified memory and transferred implicitly, versus when data is transferred by OpenMP runtime explicitly. The x axis represents the working set size and y axis represents the execution time. The measured execution time captures both the GPU

Table 1. Benchmarks.

Name	Domain	Description
Breadth first search (BFS)	Graph algorithms	Breadth first search traverses all the connected components in a graph
Computational fluid dynamics solver (CFD)	Fluid dynamics	The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow
Speckle reducing anisotropic diffusion (SRAD)	Image processing	SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs)



(a) BFS. (b) CFD. (c) SRAD.

Fig. 1. GPU performance under traditional approach and unified memory.

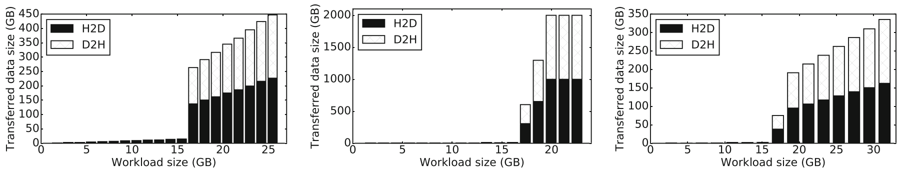
computation time and data transfer time between CPU and GPU. Note that the y axis of BFS is on the logarithmic scale due to the dramatic performance change in the presence of memory oversubscription. Our key observations are as follows.

1. For working sets that fit into the GPU memory, unified memory outperforms when less data is actually required at runtime. While traditional approach needs all data to be present in the GPU memory before computation starts, unified memory only transfers the actually accessed data at runtime and thus may result in less data transfer. However, the data transfer bandwidth is lower under unified memory, because it incurs extra address translation and page fault processing overhead.

Here, we define the ratio of actually accessed data size and total data size as *access density*. The lower the density is, the less data is transferred under unified memory. When it is lower than a threshold (mostly depends on the hardware), the benefit brought by less data transfer outweighs the lower bandwidth disadvantage of unified memory. Therefore, unified memory outperforms in such cases. BFS belongs to this category. For other programs including CFD and SRAD, traditional approach outperforms.

In summary, for data with high density, we would like to explicitly transfer all data beforehand to reduce page fault overhead. Otherwise, we should let unified memory fetch data on demand at runtime.

2. Unified memory suffers from poor performance for oversubscribing workloads with data reuse. For working sets that are larger than the GPU memory size, unified memory is able to work correctly while traditional approach fails. Its performance is largely decided by data reuse for such workloads. When large amount of data gets reused, it is likely that reused data will thrash between CPU and GPU memory. While all 3 benchmarks exhibit various degree of data thrashing behavior, BFS incurs the largest performance loss.



(a) BFS. (b) CFD. (c) SRAD.

Fig. 2. Data transfer volume under unified memory. H2D and D2H represent traffic from CPU to GPU and that from GPU to CPU respectively.

Table 2. Unified memory performance summary and optimization strategies.

Data size	Reuse	Density	Performance	Optimization
≤GPU memory size	\	High	Slightly worse (page fault overhead)	Explicit data copy
		Low	Better (less data transfer)	None
>GPU memory size	High	\	Poor (data thrashing)	Data pinning
	Low		Good	None

GPU programs are more likely to suffer from data thrashing because of the following reason. In the GPU execution paradigm, different threads usually perform similar operations on different data items to exploit its massive threading and data parallel ability. As a result, a large volume of data gets accessed in a single OpenMP target region (i.e., kernel) call, which fills up the GPU memory and evicts old data out. Since data reuse usually happens across different GPU kernel invocations, soon-to-be-reused data is not likely to survive in the GPU memory under the default replacement algorithm, LRU [11,13,18].

Figure 2 shows the data transfer volume of both directions under different workloads. For oversubscribing workloads, the dramatic traffic increment of both directions demonstrates the existence of data thrashing. Data thrashing not only adversely affects performance, but also wastes a lot energy on redundant data transfer.

To avoid data thrashing, we propose to pin data in a certain memory to prevent harmful data movement. The pinned location, GPU or CPU memory, should be selected based on the overall locality of a data object. For instance, data with good locality should be pinned to GPU, and data with poor locality should be pinned to CPU instead.

3. Unified memory performs well for oversubscribing working sets with little data reuse. If little data reuse exists, the performance of unified memory does not show significant difference whether GPU memory is oversubscribed or not. Since there is no reuse, all data is brought to the GPU memory once and replacement decisions do not affect performance. On demand data fetching works well in this case.

Conclusion. Table 2 summarizes the performance of unified memory and corresponding optimization strategies in different scenarios. Data size, access density and data reuse (i.e., locality) play important roles in the performance of unified memory. Later we will introduce how we identify different scenarios and apply optimization strategies accordingly, in order to improve unified memory performance.

4 Unified Memory Management

In this section, we propose a compiler-runtime combined framework to optimize GPU unified memory management. The key idea of our framework is to analyze the properties of data objects in unified memory, and apply optimization according to the analysis results for each object. The data analysis is performed by both compiler and runtime in our framework, while the optimization is applied by runtime. We will introduce each part separately in the rest of this section.

4.1 Static Analysis

The compiler identifies unified memory data objects and performs static analysis on them. The proposed compiler analysis includes 3 stages: data allocation analysis, GPU data usage analysis, and data access frequency analysis. All analysis is performed on the LLVM IR level. We briefly describe them as follows.

Data Allocation. As the first step, we identify all data objects in unified memory space and record them. Such objects can be allocated through CUDA APIs (e.g., `cudaMallocManaged()`), and OpenMP memory allocation APIs (e.g., `omp_target_alloc()`). Note that we modified the implementation of `omp_target_alloc()` in the LLVM OpenMP runtime to support unified memory allocation. The compiler employs a *GPU object table (GOT)* to keep records of detected unified memory objects and their information obtained in the following steps.

GPU Usage. Then, we would like to find where these data objects are used in GPU execution (i.e., which OpenMP offloading regions) for later analysis. We implement a pass to check all usage of a unified memory object's allocated memory space. When a related address of the object is passed to an OpenMP target launching function (e.g., `__tgt_target()`), we identify one instance of its GPU usage and record this OpenMP target region in the corresponding GOT entry.

Data Access Frequency. At last, we design a compile-time analysis pass to help understand data access frequency within target regions, which will be used to estimate *data reuse* and *access density* that is critical for unified memory optimization as shown in Sect. 3. First, for a certain data object in GOT, we would like to calculate the access frequency for every OpenMP target region that uses it, namely *local access frequency (LAF)*. The existing LLVM pass `BlockFrequencyInfo` can help achieve this purpose. This pass takes the taken probability of branch instructions as input, to derive the execution frequency of every basic block. It achieves so with static information. Using the information provided by `BlockFrequencyInfo`, we get the execution frequency of each memory access instruction. Then we accumulate the frequency of all memory instructions within a target region that relate to a data object to get its LAF.

We would also like to get the *global access frequency (GAF)* of each data object, which represents the overall GPU access frequency across the whole program. For this purpose, we implement an inter-function/module analysis pass to build a global call graph that includes both CPU and GPU functions. In this graph, we calculate the call frequency of each parent and child function pair, using the results from `BlockFrequencyInfo`. Then we estimate the overall execution frequency of each GPU function by traversing all leaf nodes in the built call graph. Combining the execution frequency and LAFs of all GPU functions, the GAF of a unified memory object is calculated.

Data Reuse. Since the desired optimization only needs a relative not absolute data reuse results, i.e., it is good enough to tell object A gets better reuse than object B, the *data reuse* of a data object is derived from its GAF directly. We rank all unified memory objects based on their GAFs, and use the ranking number to represent the data reuse. Assuming object A has the highest GAF and B has the lowest GAF, the ranking (i.e., data reuse) of A and B will be 1 and n respectively, where n is the total number of objects. We modify the OpenMP runtime interface, so that for every unified memory argument, both data reuse and LAF information is passed to the OpenMP runtime on offloading. LAFs will be used by the runtime to estimate access density as will be introduced in Sect. 4.2.

Completely compiler-based data access analysis has the drawback of low accuracy and non-awareness of dynamic execution pattern. The latter is critical for GPU execution since a code fragment can be executed by millions to billions of threads. For data reuse, a relative result is good enough and thus we use static analysis results for simplicity. For data size and access density, we leave a significant part of analysis to the runtime, as will be introduced in Sect. 4.2.

Overhead. The proposed analysis utilizes results from existing LLVM passes and does not need to be performed recursively. Compared to dozens of default analysis and optimization passes in Clang, its time overhead is negligible. In our experiments, we do not observe notable compile time change when the proposed analysis is enabled.

4.2 Runtime Analysis

Our runtime analysis utilizes runtime information to help the compiler finalize data analysis results. Particularly, data size and access density are estimated combining both runtime and compile-time information.

Data Size. The size of data objects depends on input in many cases and thus it is natural for runtime to get this information. In the OpenMP offloading runtime interface, the size of arguments is passed along with arguments themselves, and it is thus free for runtime to get data size. One problem is the existing data size is measured in bytes, while we would also like the number of elements for the access density estimation introduced below. Luckily the element size can be easily obtained by the compiler through type checking, and we pass it to the runtime so that the number of elements can be computed by dividing the total size by the element size.

Access Density. Density is calculated as the actual accessed element number divided by total element number. We already discussed how to get the total element number. The difficult part is how to estimate the number of actually accessed elements.

Fortunately, the regular code structure of OpenMP target regions makes a simple solution to calculate the number of accessed elements possible. In the common scenario which covers more than 90% of offloading regions, an outer `for` loop contains all work of an offloading region. Its iterations are distributed across all GPU threads. The loop body is usually short and has simple control flow. In such scenarios, the accessed element number in a single iteration is easy to estimate thanks to the simple loop body. The total accessed number mainly depends on how many iterations are executed.

We design a compiler-runtime combined scheme to compute total accessed element number and thus access density. The runtime is responsible to obtain the number of outer loop iterations, while the compiler estimates the number of accessed elements in a single iteration, using LAF. If we assume memory accesses distribute evenly across different elements in a data object, which is quite reasonable for GPU, the number of accessed elements in a loop body is equal to its LAF. By multiplying LAF and loop iteration number together at runtime, we get an estimation of the total accessed element number in a object. Then the access density can be computed as min(accessed element number/total element number, 100%).

Discussion. The limitation of our density estimation method is that it assumes a unified access distribution and does not distinguish elements within a data object. For instance, if a small fraction of elements receive all data accesses in an object, the access density estimated using the proposed method will be larger than the actual value.

In order to get more accurate analysis results, methods such as profiling and instrumentation can potentially be used. However, unlike our proposed method which puts little burden on compiler and runtime, these methods suffer from significant compiling/runtime overhead and often need help from programmers. Applying them will also add significant complexity to the implementation. As Sect. 5 will show, the proposed compiler-runtime combined analysis can already achieve significant performance improvement.

4.3 Runtime Management

This subsection will describe how we manage unified memory objects based on the above analysis results, namely data size, data reuse and access density. The runtime makes two key decisions for each object: (1) where it should be mapped, GPU or CPU memory, and (2) how it should be transferred if it is mapped to GPU, explicitly or implicitly (i.e., data transfer is performed on demand). Table 2 has listed our optimization strategies.

Data Mapping. When encountering an OpenMP target call, the runtime will follow these steps to map the arguments in unified memory before execution starts. After all properties are collected (i.e., size, reuse and density) as described earlier, all unified memory objects involved in this call are ranked based on their reuse. Then we select the proper mapping strategy for each object using the reuse order (from large to small), so that data with better locality has higher priority to be mapped to the GPU memory. If there is enough space in the GPU memory, the current object is mapped to the GPU memory. Otherwise, it is mapped to that of CPU to prevent data thrashing. In this case, we use the CUDA API `cudaMemAdvise` to pin data into the CPU memory.

Data Transfer. If an object is mapped to the GPU memory, we further decide how it should be transferred based on its density. Objects with small density (<0.6, an empirical number obtained based on experimental results) should be transferred implicitly to reduce data transfer volume, otherwise explicit transfer is used. For explicit transfer, a GPU memory object with the same size is allocated using `cudaMalloc` for GPU usage, and data transfer primitive `cudaMemcpy` is used to synchronize original and new copies. This is the default policy followed by the current OpenMP implementation. In the case of implicit transfer, we simply pass the original object to GPU kernels, and let the unified memory driver handle on demand data transfer during execution.

Book Keeping. To implement the method described above, several book keeping needs to be done at runtime. To keep track of GPU memory, the runtime uses two 64-bit counters for each GPU. They record the size of GPU memory objects that are transferred explicitly and implicitly, respectively. We can calculate the free GPU memory size with these two counters.

The proposed runtime also maintains a table to keep records of active data objects in the GPU memory. For each object, it records the size, data reuse, mapping place (GPU or CPU), and transfer mechanism (explicit or implicit).

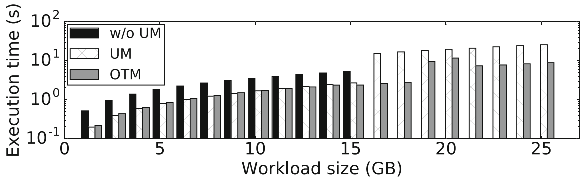
Overhead. The proposed runtime is integrated seamlessly within the existing LLVM OpenMP target offloading runtime. We do not introduce any expensive operation into it. In our experiments, we find that there is virtually no difference for the runtime execution time with or without our modification. Besides, all performance results in Sect. 5 include the runtime overhead, if there is any.

5 Evaluation

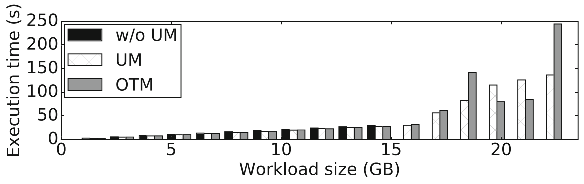
5.1 Experimental Methodology

To evaluate the performance of our benchmarks, we use the OLCF SummitDev, which is the prototype machine of Summit. Each SummitDev node is equipped with 2 POWER8 CPUs and 4 Tesla P100 GPUs. They are connected through NVLink 1.0, which provides up to 160 GB/s IO bandwidth per GPU. The Tesla P100 GPU has 56 SMs and is equiped with 16 GB HBM2 memory. It supplies a local memory bandwidth of 732 GB/s.

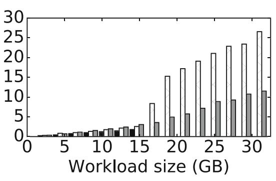
We use the up-to-date Clang [5] that supports OpenMP GPU offloading to compile benchmarks. To enable offloading for NVIDIA GPUs, we pass the flag `-fopenmp-targets=nvptx64-nvidia-cuda` along with `-fopenmp` to Clang. All benchmarks are compiled under the O2 optimization level. The Linux kernel version is 3.10.0 and the CUDA version is 9.0.69 on SummitDev.



(a) BFS.



(b) CFD.



(c) SRAD.

Fig. 3. Performance of various schemes.

5.2 Performance Results

Figure 3 illustrates the GPU performance under unified memory (UM), traditional approach (w/o UM), and our compiler-runtime collaborative OpenMP Target data Management framework (OTM). Again, note that the execution time (y axis) of BFS is on the logarithmic scale. While OTM helps BFS and SRAD achieve significant performance improvement, it fails to do so on some CFD workloads. Detailed analysis is as follows.

BFS. BFS receives the most performance gain from our approach. Under fitting workloads, OTM outperforms w/o UM by 113% on average and has similar performance compared to UM. Under oversubscribing workloads, OTM achieves a dramatic average speedup of 3.37× compared with UM.

There is a large amount of data reuse existing in BFS, because the same vertex and neighbor vertices are likely to be accessed in multiple iterations. However, the traversal happens in an irregular order, and thus it is difficult to optimize its performance using traditional methods. With OTM, the data structure that is used to store edges of each vertex, which is less frequently accessed but has the largest size, is often pinned to the CPU memory. This prevents it from thrashing other more important data in the GPU memory, so that data locality can be exploited within the GPU memory.

Note that there is a performance drop for OTM at the workload of 20 GB. This is because at this point, OTM decides to pin several small data objects into the CPU memory instead of a larger one, due to the GPU memory capacity limitation. As a result, having multiple objects in the CPU memory collectively has a larger impact on performance. When the workload is larger, once again, a large data object is pinned to the CPU memory. We will address this issue by enabling finer grained data mapping control in the near future, to further improve performance.

CFD. On average, OTM and UM has similar performance across all workloads. On some workloads, OTM is outperformed by UM. The reason that OTM fails to improve CFD performance, is that OTM currently does not handle complex scenarios well. Compared with BFS and SRAD, CFD has more complex data structures and control flow. Multiple target regions interleave with each other in multiple ways, and different regions use different sets of data objects as well as share some of them. Under some workloads, we find that for every target region, OTM pins some of its data objects into the CPU memory, which slows down all target region execution. The smarter choice here is to keep all data required by some target regions in the GPU memory to accelerate their execution, while have mixed data location for other kernels. We will develop technology to solve this problem soon.

SRAD. OTM helps SRAD achieve an average speedup of 2.55× across all oversubscribing workloads compared with UM. Since the data reuse in SRAD is limited compared with that in BFS, the speedup of OTM is more moderate.

For smaller workloads, the performance of OTM is similar to that of UM while lower than the traditional approach by 30.7% on average. By taking a closer look, we find that OTM transfers some highly reused data objects using unified memory's on demand fetching, while they should be transferred explicitly. This is because loop nests exist in some target regions, which confuses the proposed compiler analysis. More accurate analysis methods can be used to alleviate this problem. Luckily selecting the incorrect data transfer manner does not impose a large performance penalty. The data mapping location has much more significant performance impact, in which OTM makes optimized decisions for all 3 benchmarks.

In all benchmarks, the data transfer between CPU and GPU is reduced significantly for large workloads under OTM. Since there are no existing tools that can extract data transfer volume when data pinning is applied, we do not compare the data transfer of different methods.

6 Conclusion

In this paper, we develop a compiler-runtime collaborative technology to improve OpenMP GPU data management under unified memory. There are several future directions worth exploring besides what we have mentioned in Sect. 5. First, application experts may wish to provide data locality hints directly rather than relying on compiler analysis. We plan to explore new OpenMP directives/clauses for this purpose. Second, ideas presented in this paper are not limited to unified memory but also applicable to more generic scenarios. We plan to further develop our techniques to have a generic optimized OpenMP GPU data management framework.

Acknowledgement. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

1. OpenACC. http://www.openacc.org
2. Summit. https://www.olcf.ornl.gov/summit
3. OpenMP 4.0 specifications (2013). http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
4. Agarwal, N., Nellans, D., Stephenson, M., O'Connor, M., Keckler, S.W.: Page placement strategies for GPUs within heterogeneous memory systems. In: ASPLOS 2015, pp. 607–618. ACM, New York (2015)
5. Antao, S.F., et al.: Offloading support for OpenMP in Clang and LLVM. In: LLVM-HPC 2016, pp. 1–11. IEEE Press, Piscataway (2016)
6. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 44–54. IEEE (2009)
7. Cui, X., Scogland, T.R.W., de Supinski, B.R., Feng, W.C.: Directive-based partitioning and pipelining for graphics processing units. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 575–584, May 2017

8. Grinberg, L., Bertolli, C., Haque, R.: Hands on with OpenMP4.5 and unified memory: developing applications for IBM's hybrid CPU + GPU systems (Part I). In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 3–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_1
9. Hahnfeld, J., Cramer, T., Klemm, M., Terboven, C., Müller, M.S.: A pattern for overlapping communication and computation with OpenMP* target directives. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 325–337. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_22
10. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. In: PLDI 2011, pp. 142–151. ACM, New York (2011)
11. Jaleel, A., Theobald, K.B., Steely, Jr., S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: ISCA 2010, pp. 60–71. ACM, New York (2010)
12. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004, p. 75. IEEE Computer Society, Washington, DC (2004)
13. Li, L., Tong, D., Xie, Z., Lu, J., Cheng, X.: Optimal bypass monitor for high performance last-level caches. In: PACT 2012, pp. 315–324. ACM, New York (2012)
14. Mishra, A., Li, L., Kong, M., Finkel, H., Chapman, B.: Benchmarking and evaluating unified memory for OpenMP GPU offloading. In: LLVM-HPC 2017, pp. 6:1–6:10. ACM, New York (2017)
15. NVIDIA: Compute unified device architecture programming guide (2007)
16. Olivier, S.L., Hammond, S.D., Duran, A.: Double buffering for MCDRAM on second generation Intel® Xeon Phi^{TM} processors with OpenMP. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 311–324. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_21
17. Pai, S., Govindarajan, R., Thazhuthaveetil, M.J.: Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In: PACT 2012, pp. 33–42. ACM, New York (2012)
18. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. In: ISCA 2007, pp. 381–391. ACM, New York (2007)
19. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
20. Zhao, J., Xie, Y.: Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration. In: ICCAD 2012, pp. 81–87. ACM, New York (2012)