Compiler Assisted Hybrid Implicit and Explicit GPU Memory Management under Unified Address Space

Lingda Li
Brookhaven National Laboratory
lli@bnl.gov

Barbara Chapman
Brookhaven National Laboratory
Stony Brook University
barbara.chapman@stonybrook.edu

ABSTRACT

To improve programmability and productivity, recent GPUs adopt a virtual memory address space shared with CPUs (e.g., NVIDIA's unified memory). Unified memory migrates the data management burden from programmers to system software and hardware, and enables GPUs to address datasets that exceed their memory capacity. Our experiments show that while the implicit data transfer of unified memory may bring better data movement efficiency, page fault overhead and data thrashing can erase its benefits. In this paper, we propose several user-transparent unified memory management schemes to 1) achieve adaptive implicit and explicit data transfer and 2) prevent data thrashing. Unlike previous approaches which mostly rely on the runtime and thus suffer from large overhead, we demonstrate the benefits of exploiting key information from compiler analyses, including data locality, access density, and target reuse distance, to accomplish our goal. We implement the proposed schemes to improve OpenMP GPU offloading performance. Our evaluation shows that our schemes improve the GPU performance and memory efficiency significantly.

KEYWORDS

Unified Memory Management, GPU, Compiler Analysis, Runtime, Implicit and Explicit Data Transfer, Reuse Distance, OpenMP

ACM Reference Format:
Lingda Li and Barbara Chapman. 2019. Compiler Assisted Hybrid Implicit and Explicit GPU Memory Management under Unified Address Space. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356141

1 INTRODUCTION

The next milestone in supercomputing, exascale systems will be reached by relying heavily on accelerators to meet performance and energy efficiency goals. The massive threading ability of GPUs, the most popular accelerator today, benefits applications in science and engineering with large amounts of parallelism, and also supports machine learning computations. However, accelerator-based nodes still suffer from poor programmability and thus introduce productivity concerns that are potentially a major obstacle to the efficient utilization of exascale systems.

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356141

One of the most profound programming challenges in this context is handling the necessary data management. Traditionally, the GPU can only operate on data in its own memory. This requires application developers to explicitly move data between GPU local memory and CPU system memory, and they must do so carefully to ensure correctness and efficiency. In recent years, the size of application datasets has grown steadily. Yet during this time, GPU memory capacity has not experienced a significant increase, due to power constraints. Thus many applications have datasets that exceed the GPU's memory capacity; to overcome this, manual reorganization of the computation may be needed.

To address these challenges, recent NVIDIA GPU architectures support *unified virtual memory*, where a unified memory address space is shared across different computing devices, including CPUs and GPUs [35, 40]. Unified memory offers two major benefits. First, a single memory space enables seamless data sharing between different computing devices. Programmers can assume a virtual memory system which they are already familiar with. The responsibility for data transfer between different devices is migrated from programmers' shoulders to the system software and hardware. Second, unified memory enables the GPU code to address datasets that are larger than its memory capacity. For such workloads, the CPU memory serves as backup storage, and data is implicitly moved into and out of the GPU memory on demand at page granularity.

We have analyzed the performance implications of unified memory. Our studies show that on the one hand, unified memory can avoid redundant data movement thanks to its on-demand, page-grained *implicit* transfer. Yet it may suffer from page fault overhead, and data thrashing for GPU-memory-oversubscribed workloads. *Explicit* data transfer techniques (e.g., explicit data copy, prefetching), on the other hand, provide the means to eliminate these overheads. Therefore, we may wish to exploit explicit data transfer techniques as well as the implicit mechanisms in order to achieve efficient, user-transparent memory management across CPUs and GPUs. Throughout, we refer to on demand data transfers initiated by the unified memory driver/OS as *implicit*, and data transfers arising from bulk data movement calls as *explicit*, no matter whether they are inserted by the application developer, the compiler, or its runtime system.

In this paper, we propose a set of hybrid implicit/explicit data transfer schemes to accomplish efficient GPU unified memory management, taking the needs of large datasets into account. In order to retain the programmability advantage of unified memory, we adopt a compiler-runtime collaborative approach which requires no effort from application developers. The compiler is responsible for analyzing high level data access behavior, while the runtime takes into account both dynamic execution status and the static

information provided by the compiler to optimize data mapping and movement. Unlike previous memory management proposals that rely heavily on the runtime or OS [6, 31, 36, 45], our schemes have much lower execution overhead because most of the analysis is performed at compile time.

We have implemented the proposed strategies in the LLVM infrastructure and used them to optimize OpenMP GPU offloading performance. OpenMP is a widely used model for on-node parallel programming [12]. It supports the offloading of computations to accelerators such as GPUs since version 4.0 [1]. The recently released OpenMP 5.0 supports unified memory [3]. In an era where new accelerators are emerging rapidly, portability across different hardware platforms is more important than ever. OpenMP provides significant portability benefits compared with native GPU programming models such as CUDA [33] due to its wide applicability and robust support. It is moreover easier to apply OpenMP than CUDA or OpenCL [41] to existing code, thanks to its directive-based approach. Besides, OpenMP has a broader user and developer community than OpenACC [2], and thus enjoys more vendor support. Nevertheless, our compiler-runtime collaborative approaches do not depend on any specific programming model and can be easily deployed to support other GPU programming models including CUDA and OpenACC.

Overall, we make the following contributions in this paper:

- We analyze the performance of OpenMP GPU offloading under unified memory. Our analysis reveals that implicit and explicit system-level data transfer techniques may both be needed to efficiently utilize unified memory. It also shows that data thrashing may hurt performance significantly for large workloads (Section 2.3).
- We propose a set of compiler-runtime collaborative schemes for the purpose of providing reliably efficient unified memory management (Section 3). To the best of our knowledge, this is the first work to optimize unified memory management via compiler analyses as well as in the runtime. Among these schemes, the target reuse distance based method is able to exploit fine grained data locality, and thus has the best performance and data movement efficiency overall. Our evaluation shows it achieves a geometric mean speedup of 7.6× across all benchmarks and an 18.7× speedup for oversubscribed workloads over the default unified memory. The speedup is 9.7× for the ECP proxy application `miniVite` (Section 5).
- We propose a set of compiler analyses for data locality, access density, and target reuse distance (Section 4), and demonstrate that they provide useful information to the unified memory management schemes proposed in Section 3.

2 BACKGROUND AND MOTIVATION

2.1 Unified Memory

Traditional GPU architectures do not permit the CPU and GPU to access each other's memory. To enable data communication, data transfers between their distinct memories need to be explicitly programmed at the application level. This method suffers from two major drawbacks: 1) a significant user effort is required to manage data allocations and transfers manually, especially when trying to achieve high performance, and 2) the GPU cannot handle datasets whose footprints exceed its memory capacity. Yet many of today's applications operate on huge datasets. To support their deployment on a GPU, significant code and algorithm changes are often required to partition data and computation effectively.

To simplify GPU programming as well as enable large dataset processing, NVIDIA introduced *unified memory* in the Kepler GPU architecture [34], and significantly enhanced it in the subsequent Pascal architecture [35]. Under unified memory, programmers are provided with a single virtual memory space encompassing both CPU and GPU memory, just as they are accustomed to on contemporary CPUs. They do not need to worry about the physical locations of data. The underlying system software and hardware manage the mapping between virtual and physical memory space automatically at the granularity of pages.

Unified memory thus relieves application developers of the burden of data management and enables several GPU data mapping options that were not available on previous GPU architectures. We leverage these mapping options in the schemes proposed in Section 3. The options are as follows.

1. Move data between different memories on demand (i.e., implicit transfer). This is the default policy employed by unified memory which is intended to leverage page level data locality. A page in CPU memory is automatically moved into GPU memory when the GPU requests it, and vice versa. If the target memory is full, an LRU-like policy is used to select pages to evict [40]. As a result, recent GPU-accessed pages are cached in the GPU memory, i.e., the GPU memory acts as a "cache" of the CPU memory.

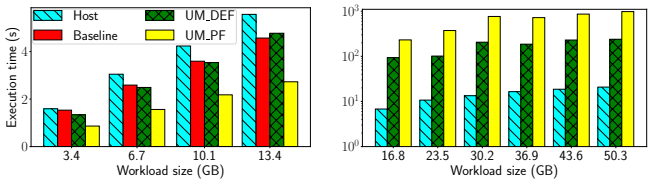
2. Prefetch data instead of fetching on demand (i.e., explicit transfer). Prefetching (with `cudaMemPrefetchAsync`) moves data into the GPU memory in advance of computation, and thus the associated memory accesses enjoy lower latency. As we will soon see in Section 2.3, prefetching data helps avoid expensive runtime page fault processing.

3. Assign the preferred mapping location of a virtual memory region to a certain place in physical memory. This prevents the system software from moving data once it is in the preferred location. Note that setting a preferred location does not mean data cannot be moved. It can still be replaced by other data when the preferred memory is fully occupied. It is also possible to have multiple copies of read-only data in different memories to facilitate read-only data sharing. This is achieved using the CUDA API `cudaMemAdvise`.

2.2 OpenMP GPU Offloading

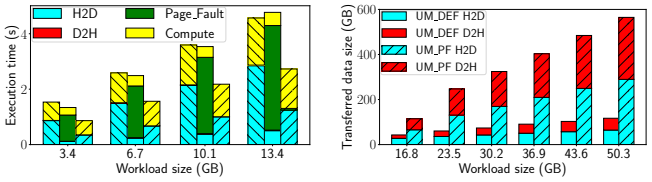
We focus on OpenMP GPU offloading in this paper. We expect that the rapidly improving maturity of OpenMP compiler support for offloading will lead to strong growth in its use for exploiting GPUs, especially in the HPC community. It is important to note that the problems uncovered here are generic to GPUs that support unified memory, and the proposed solutions are not limited to any specific programming model.

Listing 1 shows an example of OpenMP offloading code. The code region encapsulated with the `omp target` directive is executed on an accelerator (e.g., GPU). We call such a code region a *target region* or a *kernel*. The `teams distribute` directive instructs the compiler to distribute N inner loop iterations across all thread



(a) Fitted workloads. (b) Oversubscribed workloads.

Figure 1: Performance under different GPU memory management options. The x and y axes represent the workload size and execution time respectively. The y axis of (b) is on the logarithmic scale.



(a) Time breakdown for fitted workloads. (b) Traffic for oversubscribed workloads.

Figure 2: Execution time and interconnect traffic breakdown for different GPU memory management options. The bar hatched with '\' represents Baseline. The bar not hatched represents UM_DEF. The bar hatched with '/' represents UM_PF. H2D and D2H represent traffic from CPU to GPU and from GPU to CPU, respectively. Page_Fault represents the page fault processing time. Compute represents the remainder of the execution time. In reality, the computation partially overlaps with the other operations measured.

blocks, and the `parallel for` directive instructs the compiler to further distribute iterations belonging to a thread block across all its threads. In the absence of unified memory, data copying between CPU and GPU is specified using `map` clauses, where `to` means the *data object* should be copied from the CPU to the GPU, and `from` means data copy from GPU to CPU. Here, a data object means a certain memory region that is required for GPU execution. It could be a scalar variable, a data structure, or an array. In this simple vector addition example, A and B are copied to the GPU before execution of the target region's code, and C is copied out after the GPU finishes. With unified memory, the code can be further simplified by removing the `map` clauses, because the GPU and CPU directly share data.

```
// Without unified memory
#pragma omp target teams distribute parallel for \
    map(to:A[0:N],B[0:N]) map(from:C[0:N])
for (int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
// With unified memory
#pragma omp target teams distribute parallel for
for (int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

Listing 1: An OpenMP GPU offloading example of vector addition.

2.3 Case Study of UM Management Options

We have experimented with different GPU memory management options and compared their performance, in order to study the inherent performance problems and to help us design the desired unified memory management scheme that overcomes them. After porting `miniVite` [18] to use OpenMP GPU offloading, we studied its performance under different input datasets and use it here as an illustrative example. `miniVite` is a US Exascale Computing Project (ECP) proxy application that implements a single phase of the Louvain method for graph community detection. We conducted our experiments on the ORNL Summit system which is equipped with NVIDIA Tesla V100 GPUs [4]. The detailed experimental setup is described in Section 5.1. Section 5.2 will show the results of other benchmarks.

Figure 1 gives the application execution time under different GPU memory management options. We show the performance of GPU-memory-fitted (where the dataset fits into GPU memory) and GPU-memory-oversubscribed (where it does not) workloads separately, since they display very different characteristics. "Baseline" is a version of `miniVite` that represents the traditional programming approach, where GPU data is explicitly allocated and copied between devices at the application level. Note that Baseline inherently works for fitted workloads only. Host represents the case when data is pinned to the CPU memory and GPUs access them remotely. The behavior of two versions of unified memory-based code are given. UM_DEF is the default case where pages are migrated on demand (i.e., implicit transfer), while in UM_PF requested data is always prefetched to the GPU before kernel execution (i.e., explicit transfer). Our key observations are as follows.

1. Prefetching removes GPU page fault overhead effectively. UM_PF always outperforms UM_DEF for fitted workloads as shown in Figure 1(a). To understand why, Figure 2(a) shows the execution time breakdown reported by the NVIDIA GPU profiling tool nvprof. Most traffic is from CPU to GPU (H2D) for fitted workloads. We observe that UM_DEF spends most of the execution time on GPU page fault processing. On a GPU page fault, an exception is sent to the CPU to look up and update the page table. This process involves multiple CPU-side system software invocations, as well as CPU-GPU communication, and thus suffers from large overhead. As a result, the GPU does not have enough computation to hide the page fault latency. For example, the entire computation of Baseline is shorter than the page fault time of UM_DEF in Figure 2(a).

On the other hand, UM_PF does not suffer from the page fault overhead (no green portion for UM_PF in Figure 2(a)). This is because GPU page table entries have been appropriately updated during prefetching.

2. Unified memory helps avoid unnecessary data movement for irregular memory access patterns. Interestingly, although UM_DEF suffers from significant page fault overhead as described above, it still outperforms Baseline for small workloads. This is contrary to the common belief that explicit data movement (i.e., Baseline) always has better performance than implicit data movement (i.e., UM_DEF). This is because under user-specified explicit

data transfer (Baseline), whenever a memory region is mapped to the GPU/CPU, the entire region has to be moved, even if only a small fraction of it will be actually accessed. It is often impossible for the application developer to determine which portion of memory is required at runtime, e.g., under irregular memory access patterns. Therefore, Baseline may result in redundant data movement. In contrast, UM_DEF only migrates data that is actually requested, and thus potentially moves less data than Baseline.

Another interesting observation is the fact that although UM_PF prefetches the entire memory region, it incurs fewer data transfers than Baseline (i.e., UM_PF's H2D is shorter than Baseline's H2D in Figure 2(a)). This is the other case where the implicit transfer prevails: when a portion of the data transferred is already in the destination memory, implicit transfer will result in less data movement. Under UM_PF, although the entire memory region is prefetched on its very first prefetching operation, subsequent prefetching calls for the same memory region will not fetch data that is already in the GPU memory. As a result, UM_PF also has better data transfer efficiency than Baseline.

3. Data thrashing harms performance significantly for workloads that exceed the GPU memory capacity. Figure 1(b) shows that both UM_DEF and UM_PF suffer from severe performance degradation when the dataset is larger than the GPU memory. Their performance is orders of magnitude worse than that of Host. The reason for the poor performance of unified memory in this case is that data is transferred back and forth between CPU and GPU memory repeatedly (i.e., data thrashing takes place).

We do not give the breakdown of execution time for oversubscribed workloads, because almost all the time is used to address page faults. Instead, Figure 2(b) shows the data transfer volume between CPU and GPU. For both UM_DEF and UM_PF, there is a dramatic traffic increase in both directions. The traffic volume is much larger than the workload size, which indicates the existence of data thrashing.

Data thrashing happens because reused data items contend with each other for the limited GPU memory space. Such contention happens not only between data accessed by different kernels, but also among different data used by the same kernel. UM_PF shows more significant traffic growth, because even prefetched data objects contend with each other for memory when all data required by a kernel is larger than the GPU memory capacity. Data thrashing not only adversely affects performance, but also wastes energy because a lot of data is transferred redundantly.

In the GPU execution paradigm, where hundreds of thousands of threads run concurrently, different threads usually perform similar operations on different data items to exploit the massive parallelism of the GPU. As a result, a large volume of data is accessed in a short time, which fills up the GPU memory quickly and leads to the eviction of old data. When there is data reuse, it is very likely that reused data has already been evicted from the GPU memory under the default LRU like replacement algorithm before it is needed again [23, 28, 39]. Hence data thrashing is a widespread phenomenon on GPUs.

Note that V100 GPUs and CUDA 9.2 support access counters to avoid data thrashing [40], and this technology is enabled in our evaluation of UM_DEF. However, as shown in Figure 1(b), access counters cannot prevent GPU data thrashing. Rather, it is designed to alleviate thrashing between CPU and GPU.

4. Pinning data to the CPU memory achieves reasonable performance for all workloads. Host has reasonable performance for all workloads. For workloads that fit into the GPU memory, the average execution time of Host is 1.15× of that of Baseline, and 1.19× of that of UM_DEF. For workloads that exceed the GPU memory capacity, while UM_DEF and UM_PF suffer from data thrashing, Host keeps all data in the CPU memory and thus avoids data movement. Besides, Host benefits from the high CPU-GPU interconnect bandwidth of Summit (NVLink 2.0). Therefore, contrary to common sense, pinning data to the CPU memory is a viable choice in GPU data management, although it cannot achieve the best performance, as we will show later.

Conclusion. Based on these observations, we conclude that the keys to achieving efficient unified memory utilization are as follows: 1) retain the data transfer efficiency advantage of unified memory, 2) reduce page fault costs, and 3) avoid data thrashing for memory oversubscribed workloads.

The first 2 goals conflict with each other: reducing page fault costs requires prefetching as we show above, yet extensive prefetching may introduce unnecessary data transfer. The approach proposed in this paper seeks to moderate the use of prefetching.

Observation 4 shows that pinning data to the CPU memory can help achieve our third goal. However, it is not a good idea to place all data in the CPU memory, as it has lower bandwidth and larger latency. Therefore, the proposed GPU memory management policy should be aware of data locality and access patterns, so that data with better locality can be placed in the GPU memory, while harmful contention from data with poor locality needs to be prevented. The next section will introduce our strategies to achieve these goals.

3 UNIFIED MEMORY MANAGEMENT HEURISTICS

Section 2.3 has demonstrated that in order to achieve optimized unified memory performance, the GPU memory management scheme should 1) carefully choose either implicit or explicit data transfer, and 2) avoid data thrashing under oversubscribed workloads. Section 3.1 explains how we select implicit or explicit data transfer. To eliminate data thrashing, Section 3.2 and 3.3 introduce 2 schemes to address intra- and inter-kernel thrashing, respectively. Section 3.4 refines the latter by utilizing the target reuse distance to further exploit data locality. These data management decisions are made for individual *data objects*. E.g., the target region in Listing 1 involves 3 data objects: A, B, and C. Finally, Section 3.5 provides a theoretical discussion of the complexity of unified memory management.

3.1 Implicit vs. Explicit Transfer

The experiments in Section 2.3 demonstrate that implicit (on demand) data transfer triumphs when a small portion of the data is actually accessed, otherwise explicit data transfer (prefetching) is preferred. In order to choose from these methods for a given code region, we rely on a measure that assesses the fraction of a data object accessed which we call the *access density*. We introduce our density estimation method in Section 4.2.

When an OpenMP target region is encountered, for each data object involved, we will estimate its access density. If the density of an object is lower than a threshold, we select implicit transfer and do not do anything further for it. Otherwise, explicit transfer is selected and we apply prefetching to that object before the GPU starts execution of the target region. We have used an empirically selected value of 0.5 as the threshold in our experiments.

3.2 Addressing Intra-kernel Thrashing

Data thrashing can occur between different data in the same target region (or kernel), and between data in different target regions. In order to address intra-kernel data thrashing, we propose a *Regional Locality based Management (RLM)* scheme. This technique relies on an evaluation of the locality of references to data objects within a target region (which we call *regional* locality). We introduce our strategy for estimating the locality of data objects in Section 4.1. In this scheme, when an OpenMP target region is encountered, the locality of each data object within this region is computed and used to assign priorities to the data.

Objects with higher priorities will be placed in the GPU memory to enjoy fast and high bandwidth accesses. Other objects with lower priorities are placed in the CPU memory, so that they will not evict objects in the GPU memory.

Algorithm. When a target region is invoked, the runtime makes separate mapping decisions for every data object involved in the order of their regional locality values (from highest to lowest). For each data object, the algorithm goes through the following steps. ❶ Check the location of the currently accessed data object. If it is already in the GPU memory, no further action is needed. Otherwise, the runtime executes step 2. ❷ Check if there is enough available space in the GPU memory. If yes, the runtime moves the accessed data object from its current location into the GPU memory and finishes its processing. Otherwise, step 3 is required. ❸ Select the data object with the lowest priority in the GPU memory as the replacement candidate. Note that objects with higher priorities in the same kernel call cannot be replaced. The candidate is added to the replacement candidate list. ❹ Repeat step 3 until enough space can be released, or no suitable replacement candidates can be found. In the former case, data objects in the replacement candidate list are removed from the GPU memory, so that the GPU memory can accommodate the current object. In the latter case, the data object is placed in the CPU memory instead of moving it into the GPU memory. The replacement list is cleared without performing an actual replacement.

Data Mapping Enforcement. As described in Section 2.1, the system software fetches data into the GPU memory on demand by default. Yet doing so could jeopardize the existing data mapping and evict data objects with higher priority. To avoid this, we enforce the mapping of data objects that are placed in the CPU memory by setting their preferred location to the CPU memory using `cudaMemAdvise`. In this way, on-demand data fetching is disabled, and the data location of these objects cannot be dynamically changed by the system software.

Runtime Book Keeping. To enable RLM and other data management schemes introduced later, some book keeping is necessary in the runtime system. For each data object related to GPU computation, the runtime supporting RLM must maintain a record of its current memory location, the replacement priority, and more.

3.3 Addressing Inter-kernel Thrashing

While RLM is able to eliminate intra-kernel data thrashing, it does not take into account other target regions, and thus may suffer from data thrashing across different target regions. To handle the inter-kernel case, we introduce the *Global Locality based Management (GLM)* scheme. GLM uses the locality of a data object in the entire program to assign it a priority. The global locality of a data object d is evaluated using the following formula:

$$GL_d = \sum_{i=1}^{n} (RF_i \times RL_{i,d}), \tag{1}$$

where n represents the number of target regions, RF_i represents the invocation frequency of the ith target region, and $RL_{i,d}$ represents the regional locality of object d in the ith target region. Section 4.1 introduces the methods used to estimate RF_i and $RL_{i,d}$. If d is not used in the ith target regions, $RL_{i,d}$ is set to 0.

Algorithm. The steps followed in GLM to make data mapping decisions are similar to those of RLM, except that the global locality of a data object is used to set its priority, instead of its regional locality. The usage of global locality ensures that the priority of a certain data object does not change across different target regions. The benefit of having consistent priorities is that it completely avoids thrashing: after the warm up phase in which the GPU memory is filled with objects with the highest global priorities, they will not be replaced in later execution.

Limitation. The major drawback of GLM is that when global locality is applied, dynamic execution details can no longer be taken into account. As a result, it then cannot exploit fine-grained locality optimization opportunities. For instance, some data objects may have poor locality globally, and thus be placed into the CPU memory by GLM. Yet they might have good locality in certain execution phases, during which it could be beneficial to move them into the GPU memory temporarily.

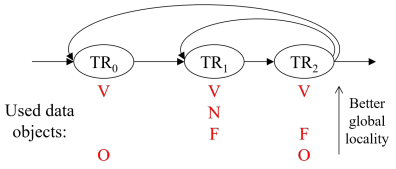


Figure 3: Target region control flow of `cfd`.

Figure 3 shows the target region relationship in the `cfd` benchmark [10], where GLM results in suboptimal data mapping decisions. In the figure, each node represents a target region, and edges represent control flow paths. There are two nested loops in this example. For each target region, the data objects that it accesses are listed in red below it, ordered with respect to their global locality. Those with highest global locality are placed on top, thus, $GL_V > GL_N > GL_F > GL_O$. Assuming the GPU memory can retain 2 objects, GLM will place V and N into GPU memory

throughout the entire code's execution. However, F is used in both TR_1 and TR_2 while N is only used in TR_1 in the inner loop. It is more beneficial to keep V and F in the GPU memory instead of V and N when the control flow enters the inner loop. Neither GLM nor RLM are able to suitably exploit the locality of F in this scenario.

3.4 Target Reuse Distance based Management

To address the limitations of schemes above and better account for data locality at the finer granularity, we propose the *Target Reuse Distance based Management (RDM)* scheme. RDM exploits the reuse distance of data objects at the OpenMP target region level for data management.

Target Reuse Distance Definition. When a target region is invoked, we may be able to make superior mapping decisions if we can take the reuse distance of an accessed data object, which represents how soon it will be reused in the future, into account. Traditionally, the reuse distance is defined as the number of different memory locations accessed between two consecutive accesses to the same memory location [14]. In our work, instead, the *target reuse distance* is measured in terms of the number of target regions encountered between two adjacent uses of the same data object, in order to simplify compiler analysis. Note that the target reuse distance is almost equivalent to the traditional reuse distance in the context of this paper. In the rest of this paper, reuse distance represents target reuse distance unless stated otherwise.

RDM is inspired by Belady's OPT algorithm [8], as will be discussed in Section 3.5. Its key idea is to preferentially keep data objects that will be reused sooner (i.e., have shorter reuse distance) in GPU memory. In Figure 3, the reuse distance of O in TR_0 is 2, since its next reuse is in TR_2. The reuse distance of F in TR_2 is 1, because in most cases, it is reused by TR_1 in the next iteration of the inner loop. Section 4.3 introduces the compiler analysis that we have developed to estimate reuse distance.

Next Reuse Time Predication. Given its reuse distance, the next reuse time of a data object can be predicted at a target region invocation. To achieve this, first, a global timer *target region counter (TRC)* is employed by the runtime to keep track of the number of target regions that it has encountered so far, as the representation of the current "time". It is initialized to 0 at the beginning of execution, and increased by 1 whenever a target region is encountered.

Upon encountering a target region, the reuse distance estimated by the compiler is passed to the RDM runtime along with all data objects involved. Then the next reuse time of a data object is computed as its reuse distance plus TRC. The runtime will keep track of the predicted next reuse time of every data object, which will be used to make data placement/replacement decisions.

Algorithm. When a target region is invoked, the runtime prioritizes the mapping of data objects with smaller reuse distance. Data mapping decisions are made for every data object following these steps. ❶ Check whether the current data object is already in GPU memory. If not, the 2nd step is executed. ❷ Check whether there is enough free space in GPU memory to accommodate the current data object. If not, the 3rd step is executed to try to release memory space. ❸ From all data objects in the GPU memory, select the one that is predicted to be reused in the furthest future (i.e., has the largest next reuse time) as the replacement candidate. The predicted next reuse time of the current object is compared with that of the candidate. Based on the result of the comparison, 1) if the current object has a larger next reuse time, we do not replace the candidate since it is expected to be reused sooner. The current data object is not moved to the GPU memory in this case. 2) If their next reuse time is equal, the global locality of data objects is used as a tie breaker to choose whether the candidate should be replaced. 3) If the candidate is expected to be reused at a later time in future, the replacement is appropriate. Qualified candidates are inserted into the replacement candidate list. ❹ Repeat step 3 until enough memory space is identified that can be released, or no further qualified replacement candidate can be found. If there is sufficient space for replacement, the data objects in the replacement candidate list are removed from the GPU memory. Otherwise, the current data object is kept in the CPU memory.

RDM achieves optimized data locality exploitation through adopting target reuse distance. For the example in Figure 3, RDM will keep F instead of N when the control flow reaches the inner loop, because F has a shorter reuse distance. Thus it is able to exploit the locality of F, unlike RLM and GLM.

Partial Mapping. We observe that many data objects in large-scale computations are large. Mapping data at the object granularity often underutilizes GPU memory space (by several GBs typically). To reduce such wastage, we implement a mechanism, *partial mapping*, that allows part of a data object to be mapped to the GPU memory while the remainder is mapped to CPU memory. When partial mapping is enabled, the data object whose priority immediately follows those of data objects in the GPU memory is partially mapped to both GPU and CPU memory. Currently, we do not distinguish different portions within a data object for simplicity: the first portion of a partially mapped object, beginning from the lowest address, is placed in the GPU memory, while the rest is placed in the CPU memory. Exploration of such options is beyond the scope of this paper.

3.5 Theoretical Analysis

Under unified memory, the GPU memory can be viewed as a cache for the CPU memory, therefore unified memory management resembles cache management to a certain degree. In 1966, Belady described the optimal replacement policy for caches with standard structures (e.g., direct mapped, set associative, etc.) if the future memory access pattern is known in advance [8]. The central idea of Belady's OPT is to evict the cache line that will be reused in the furthest future (i.e., with the largest reuse distance) on replacement. It has been widely used to study the theoretical performance boundary of cache replacement algorithms.

However, the replacement/management of GPU memory is much more complicated than that of standard caches for the following reasons. 1) While cache lines always have identical sizes, the data object size varies a lot. To accommodate one data object, it is often necessary to evict multiple others. 2) The transfer overhead also varies across different data objects based on their access patterns, which OPT does not take into account. 3) A tie break mechanism is required when two data objects have the same next reuse time, while OPT does not include such a mechanism because it is impossible for 2 cache lines to have the same next reuse time. It has been proved

that the replacement/management problem in the presence of such complex memory systems is *NP-complete* [9]. Therefore, Belady's OPT is not the optimal algorithm for unified memory management.

In this paper, we have described 3 heuristic management schemes. As Section 5.2 will show, they can achieve significant performance improvement while still being practical. Among them, RDM is directly inspired by Belady's OPT algorithm. It utilizes compile-time analysis to estimate the target reuse distance of data objects and thus predicts their next reuse time.

4 UNIFIED MEMORY ANALYSIS

This section will introduce the compiler analysis technology needed to support the schemes proposed in Section 3.

4.1 Data Locality Analysis

All 3 unified management schemes proposed in Section 3 need locality information for data objects as input. First, we will explain how to identify data objects that need locality analysis. Then, we introduce how to analyze the regional locality of a data object. Last, we describe the methodology used to evaluate global locality.

Data Object Identification. Our targets are all data objects that exist in the unified memory space. Such objects can be allocated using CUDA memory allocation APIs (e.g., `cudaMallocManaged`), or OpenMP memory allocation APIs (e.g., `omp_target_alloc`). Note that we modify the implementation of `omp_target_alloc` so that it is capable of unified memory allocation. We employ a *GPU object table (GOT)* to track these objects. GOT also records other useful information related to such objects, such as locality per target region and global locality.

We also need to capture the target regions that each data object is used in. This is done by tracking the data flow of allocated pointers. Whenever the pointer related to an object is passed as an argument on a target offloading call, we conclude that the corresponding target region is a user of that data object. The names of OpenMP offloading functions start with `__tgt_target`.

Data Locality in a Single Target Region. The locality of a data object in one target region, a.k.a, regional locality, is the key information required by RLM (Section 3.2). In the LLVM IR, all memory objects are accessed through load and store instructions. Therefore we can evaluate the regional locality of a data object by estimating the execution frequency of all related load and store instructions in a certain target region. The existing LLVM pass `BlockFrequencyInfo` can help achieve this purpose. It statically analyzes the execution frequency of every basic block based on the branch taken probability. We enhance this pass to improve its accuracy by leveraging GPU specific information, for instance, each loop iteration is executed by exactly one GPU thread. The execution frequency of a load/store instruction is equal to that of its parent basic block. Through pointer chasing that starts from the argument of a target region, all memory access instructions related to an object can be found. By accumulating the execution frequency of these instructions, the regional locality of a data object is calculated.

Data Locality in the Entire Program. As introduced in Section 3.3 and 3.4, GLM and RDM need global locality information to manage GPU memory. In Equation 1 which is used for global locality calculation, we already know how to estimate the regional locality *RL*. Here, we introduce a method to evaluate the invocation frequency of target regions, *RF*.

```
1  void computeAllBB() {
2    int PreBB2TR[][] = INF; // The distance from the beginning of a
         basic block to a target region.
3    int PosBB2TR[][] = INF; // The distance from the end of a basic
         block to a target region.
4    bool IsVisit[]; // Identify whether a basic block is visited in
         this iteration.
5    do {
6      IsVisit[] = false;
7      computeBB(MainEntryBB); // Start from the entry basic block of
           the main function.
8    } while (!PosBB2TR.isConverged());
9  }
10
11 void computeBB(BasicBlock BB) {
12   if (IsVisit[BB]) // This block has been visited.
13     return;
14   IsVisit[BB] = true;
15   foreach TR in Program.targetRegions()
16     PosBB2TR[BB][TR] = 0;
17   foreach SBB in BB.successors() {
18     computeBB(SBB); // Visit successors.
19     // Update PosBB2TR.
20     foreach TR in Program.targetRegions()
21       PosBB2TR[BB][TR] += Probability(from BB to SBB) * PreBB2TR[
           SBB][TR];
22   }
23   // Update PreBB2TR.
24   foreach TR in Program.targetRegions() {
25     if (TR.isChildOf(BB)) // TR is in this BB.
26       PreBB2TR[BB][TR] = BB.targetRegionNumBefore(TR); // Number
           of target regions before TR in BB.
27     else
28       PreBB2TR[BB][TR] = PosBB2TR[BB][TR] + BB.targetRegionNum();
29   }
30 }
```

Listing 2: The algorithm to calculate the basis block to target region distance.

We adopt a method similar to that used to get the execution frequency of loads/stores, to estimate the execution frequency of target region invocation instructions (i.e., *RF*). A major difference is that RF_i is the execution frequency of the ith target region per program invocation. For instance, assume the `main` function calls the function `foo`, and the latter one includes a target region TR. We need to get the execution frequency of TR for one `main` invocation (i.e., one execution of the program). To achieve this, we evaluate the function invocation frequency recursively. For this example, it first estimates the execution frequency of `foo` in `main`, $F_{foo,main}$. Then the global invocation frequency of TR is computed as $F_{foo,main} \times F_{TR,foo}$, where $F_{TR,foo}$ is the execution frequency of TR in `foo`. This method can also be used in the presence of recursive functions, e.g., when `foo` calls itself. It is easy to prove that the invocation frequency of recursive functions will eventually converge under this algorithm. Once *RF* and *RL* are both available, the global locality of a data object is computed using Equation 1.

4.2 Access Density Analysis

Access density information is key to making the implicit/explicit data transfer selection. Formally, access density is defined as the number of actually accessed bytes divided by the total object size. The total object size can be easily obtained at runtime. However, obtaining the number of actually accessed bytes generally requires expensive runtime technology such as profiling, especially under

irregular memory access patterns. Fortunately, OpenMP target regions follow a regular pattern, which makes a simple compiler-runtime collaborative solution possible.

A target region is typically composed of a single loop, e.g., Listing 1. In most cases, the loop body does not include nested loops, which makes it possible for the compiler to analyze the data amount accessed in a single iteration (B_{iter}) in a relatively accurate manner. B_{iter} is calculated as $\sum_{i=1}^{n}(F_i \times S_i)$, where n is the number of memory access instructions, F_i is the execution frequency of the ith instruction, and S_i is its access granularity (e.g., 8 bytes for a `double` load). We already show how to estimate F_i in Section 4.1. In addition, the pre-existing OpenMP runtime already has the loop count (LC) information, and therefore the total number of bytes accessed across all iterations can be estimated using $B_{iter} \times LC$. Then, the access density is calculated as $B_{iter} \times LC / Total\_Size$.

Note that this solution does not consider data reuse (i.e., multiple accesses to the same memory location), and thus the result density may be larger than the actual value. Section 5.2 will analyze the efficiency of this algorithm.

4.3 Target Reuse Distance Analysis

RDM requires the target reuse distance information to manage unified memory. Our target reuse distance calculation algorithm includes 3 steps.

Basic Block to Target Region Distance Calculation. As the first step, we calculate the distance from every basic block to every target region. We define the distance from basic block BB to target region TR (`PosBB2TR` in Listing 2) as the number of target regions encountered between BB and TR during execution (i.e., it is a dynamic rather than static concept.). Note that the definition is consistent with that of target reuse distance in Section 3.4.

Listing 2 shows the pseudo code of the basic block to target region distance calculation algorithm. It resembles classic data flow analysis. The algorithm outputs two arrays, `PreBB2TR` and `PosBB2TR`, which represent the distances from the beginning and end of a basic block, respectively, to a target region. The basic idea of this algorithm is to derive the distance from a basic block to a target region based on those of its successor blocks, which is implemented in the `computeBB` function. To make the algorithm converge, `computeAllBB` employs a do-while loop to run `computeBB` on the entry basic block of the `main` function multiple times. Note that this algorithm also works for inter-procedural (i.e., inter function) reuse distance analysis, by extending the control flow graph at function call sites.

Region to Region Distance Calculation. With the basic block to target region distance, we are able to calculate the distance between target region pairs. Assume the source and destination target regions are TR_s and TR_d respectively, and the parent basic block of TR_s is BB_s. The distance from TR_s to TR_d is computed as `PosBB2TR[`BB_s`][`TR_d`]` + the number of target regions in BB_s after TR_s. In Figure 3, the distance between TR_1 and TR_2 is 1, while the distance from TR_2 to TR_0 is a large number because of the inner loop.

Target Reuse Distance Calculation. Finally, we derive the reuse distance of a data object based on the distance between target regions. For a specific data object D at the target region TR, we find all other target regions that also use D. The minimal distance from TR to these target regions is the reuse distance of D at TR.

5 EVALUATION

5.1 Experimental Methodology

To evaluate the performance of the proposed unified memory management schemes, we performed a set of experiments on the Summit system at Oak Ridge National Laboratory [4], which is the current number one on the TOP500 list [42]. Each Summit node is equipped with 2 POWER9 CPUs and 6 Tesla V100 GPUs. They are connected through NVLink 2.0, which provides up to 300GB/s IO bandwidth per GPU. The Tesla V100 GPU has 84 SMs and is equipped with 16GB HBM2 memory. It supplies a local memory bandwidth of 900GB/s.

We have implemented our schemes in the latest LLVM 9.0 framework [25], including its C/C++ front end Clang, and the OpenMP runtime which is mainly contributed by Intel and IBM. The latest LLVM includes OpenMP GPU offloading support [7]. To enable offloading for NVIDIA GPUs, along with `-fopenmp`, we pass the flag `-fopenmp-targets=nvptx64-nvidia-cuda` to Clang. We use the Linux kernel 4.14.0 and the CUDA 9.2.148 on Summit.

In addition to `miniVite`, we also rewrote `backprop`, `bfs`, `cfd`, `hotspot`, `kmeans`, `nn`, and `srad` from the Rodinia benchmark suite to use OpenMP GPU offloading [10, 32]. The performance of our optimized OpenMP offloading codes matches that of the corresponding CUDA versions. These benchmarks and `miniVite` are used in our evaluation. For each benchmark, we generated inputs of various sizes to assess the performance impact of workload sizes. The default optimization level in the Rodinia benchmark suite is used during compilation. Each experiment was run 3 times and the average result is reported.

5.2 Result Analysis

Figure 4 illustrates the performance of various GPU memory management schemes. They include Baseline (the traditional approach where data transfer is explicitly specified through OpenMP `map` clauses), UM (the default unified memory), Host (where data is always placed in the CPU memory), RLM, GLM, and RDM (the schemes proposed in this paper). Besides, we also implement a fully runtime-based GPU data management scheme, LRU, which leverages the Least Recently Used algorithm [13], as a reference. LRU keeps the most recently used data objects in the GPU memory and other objects in the CPU memory, so it is immune to intra-kernel data thrashing. Besides, LRU transfers data explicitly.

For workloads that fit into the GPU memory (i.e., to the left of the vertical line), all three proposed schemes perform similarly because no replacement is involved. For workloads that exceed the GPU memory capacity (i.e., to the right of the vertical line), when Baseline cannot work anymore, UM incurs significant data thrashing except in the case of `nn`. Therefore, the logarithmic scale is used for the y axis except for results with `nn`.

Data Transfer Efficiency for Fitted Workloads. For workloads that fit into the GPU memory (to the left of the vertical line), we observe that RLM, GLM, and RDM, which are able to adaptively choose from implicit or explicit transfer, achieve better performance

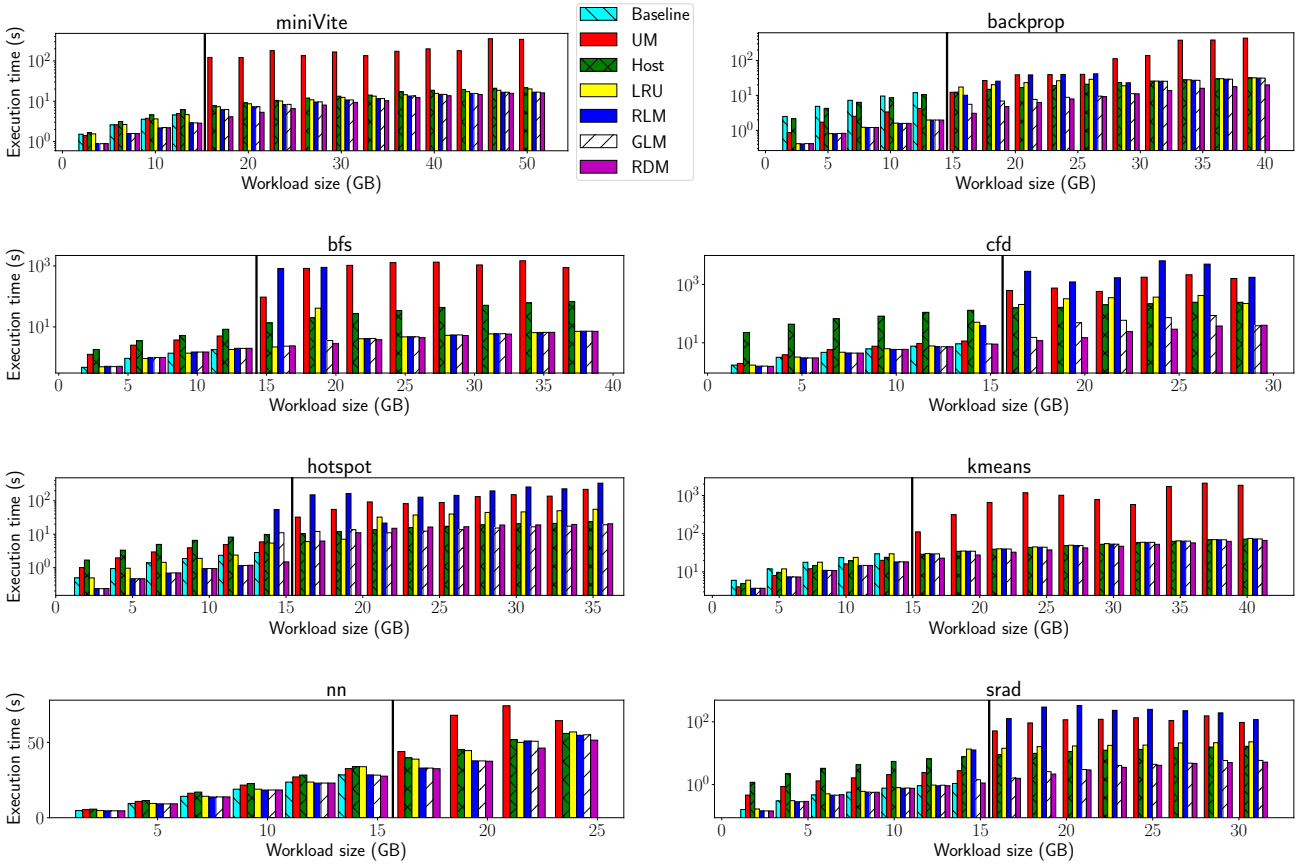


Figure 4: Performance under various schemes. The x axis represents the workload size, and workloads to the right of the vertical black line exceed the GPU memory capacity. The y axis represents the execution time and is on the *logarithmic* scale except that of `nn`. Note that the performance difference is greater than it appears to be because of the use of logarithmic scale.

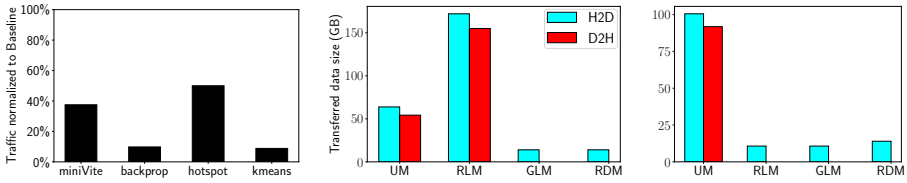
than Baseline, UM, and LRU for `miniVite`, `backprop`, `hotspot`, and `kmeans`. Using RDM as an example, by a geometric mean, the execution time of Baseline is reduced by 58.3% across these benchmarks for fitted workloads (2.40× speedup). Figure 5(a) illustrates the data volume transferred under RDM, which is normalized to that of Baseline. It shows that the CPU-GPU traffic is significantly reduced under RDM, thanks to implicit data transfer. Roughly 90% of traffic for `backprop` and `kmeans` is eliminated, and thus they enjoy substantial performance improvement from our schemes. The results demonstrate the ability of the hybrid implicit/explicit method to achieve efficient data movement for fitted workloads where the traditional approach is feasible.

Our schemes have on par performance with Baseline for other benchmarks. For these benchmarks, explicit transfer always has the best performance. Our methods cannot further reduce traffic.

Data Thrashing Prevention. For workloads that oversubscribe the GPU memory (to the right of the vertical line), we observe that other schemes outperform UM significantly. The geometric mean speedup of Host, LRU, RLM, GLM, and RDM over UM is 8.2×, 8.5×, 3.7×, 15.7×, and 18.7×, respectively. While Host, GLM, and RDM can effectively prevent the data thrashing that UM suffers from, RLM and LRU still incur inter-kernel data thrashing for some workloads. As discussed in Section 3.3, RLM is not aware of inter-kernel data access patterns. On the other hand, LRU has been shown to be insufficient to prevent data thrashing [23, 28, 39]. As a result, it achieves an overall speedup similar to that of Host.

As an example, Figure 5(b) and 5(c) illustrate the data transfer volume for 2 oversubscribed workloads of `bfs`. UM incurs a large number of data transfers in both directions, because of data thrashing. RLM prevents this for the 31GB workload, but experiences even more traffic for the 19GB workload. There are 2 interleaving target regions in `bfs`. For this 19GB workload, RLM fetches a large data object into the GPU memory at one target region but evicts it at the other target region to release space for other objects. As a result, RLM moves this object back and forth repeatedly, incurring many redundant data transfers, and leading to the poor performance of RLM for this workload as shown in Figure 4.

GLM and RDM are able to completely remove data thrashing for both workloads, as indicated by their 0 D2H traffic. RDM has slightly more H2D traffic when compared with GLM for the 31GB workload because of the use of partial mapping. Recall that these schemes



(a) Fitted workloads. (b) `bfs` 19GB. (c) `bfs` 31GB.

Figure 5: Data transfer efficiency. (a) Normalized RDM traffic for fitted workloads. (b) (c) Traffic for 2 oversubscribed workloads of `bfs`. H2D and D2H represent data transfer from CPU to GPU and that from GPU to CPU respectively.

place data objects with poor locality (or larger reuse distance) into the CPU memory, and prevent them from being moved into the GPU memory on demand. As a result, data objects with better locality in the GPU memory are not evicted.

RDM vs. GLM. RDM outperforms GLM for benchmarks with complex target region interactions. Taking the 19GB workload of `cfd` as an example, when encountering the most time consuming target region TR_1, as shown in Figure 3, N has better global locality but larger reuse distance. On the other hand, O that is already in the GPU memory has poor global locality but will be used in the next target region TR_2. In this scenario, RDM keeps O in the GPU memory, while GLM incorrectly chooses to retain N and evict O out of the GPU memory. As discussed in Section 3.4, GLM cannot adapt to the changing memory access behavior because it uses a fixed global locality during the entire execution. Instead, RDM is able to adapt to dynamic execution behavior and thus exploit fine grained data locality. For future applications with increasing amounts of target regions, RDM is more desirable. Since RDM has the best performance, we focus on it in the rest of this section.

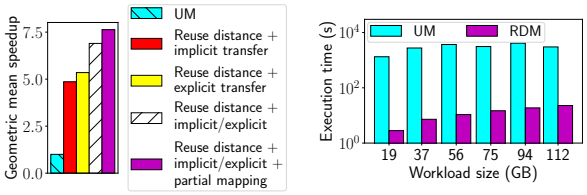


Figure 6: Contribution of different RDM components.

Figure 7: Scalability with `bfs` workload size.

Performance Contribution Breakdown. Figure 6 shows the geometric mean speedup against UM of different RDM components across all benchmarks' workloads. First, we observe that exploiting reuse distance alone brings significant performance benefits because it successfully exploits data locality and avoids data thrashing. Second, combining implicit and explicit data transfer outperforms the cases when only one of them is used, by a large margin. The results demonstrate that it is important to leverage both implicit and explicit transfer in unified memory management, as discussed in Section 2.3. Lastly, partial mapping brings an additional 0.7× improvement. Putting all components together, RDM achieves an overall speedup of 7.6×.

Access Density Analysis. Recall that the choice between implicit and explicit transfer is decided by access density analysis results. For most benchmarks studied in this paper, we find our access density analysis method gives the desired results, except `bfs`.

The largest data structure of `bfs` is used to store the graph edges. Depending on the connectivity of a graph, a large fraction of edges may not get accessed, and thus it may be benefitial to transfer this data structure implicitly. Our algorithm estimates this data structure to have a large access density, because there is an inner loop where each thread traverses all edges connnected to a vertex. Therefore, explicit transfer is used for it. When we change the percentage of edges that are actually accessed from 0% to 100%, we find that the execution time when transferring edges implicitly varies between 29% and 303% of that using explicit transfer. This demonstrates that more accurate access density estimation methods can exploit such performance opportunities, and we plan to explore them in the future. As this example shows, more accurate methods will need runtime knowledge since the access density depends on the input graph. Therefore, such methods will incur more runtime overhead compared with our static method, while our current solution is simple and incurs very little overhead.

Scalability with Workload Size. To demonstrate the scalability of RDM, Figure 7 shows the performance results of larger datasets for `bfs` (up to 112GB). For workloads that exceed 112GB, UM cannot complete execution within the 2 hour (7200 seconds) job limit of Summit. Therefore we do not report the results for such workloads. RDM shows good scalability overall. It achieves a geometric mean of 269× speedup across these workloads, and a 132× speedup for the largest 112GB workload. Other benchmarks show similar scalability. We do not show their results due to space limitations.

Overhead. There are 2 kinds of overheads introduced by the heuristic schemes that we have developed, 1) static analysis overhead, and 2) runtime overhead to make data management decisions.

For compiler analyses, the overhead in time to perform data locality and density analyses is negligible compared with dozens of default analysis and optimization passes in Clang. The reuse distance analysis is more expensive since it employs a recursive algorithm. In our evaluation, we found that it required at most a few seconds when compiling our benchmarks. However, we only incur these compile time overheads once in a program's life cycle.

Our runtime support is integrated seamlessly into the existing LLVM OpenMP device offloading runtime. All the requisite data management algorithms are computationally inexpensive. As an

example, for the 9GB workload of `bfs`, the runtime overhead of RDM takes 0.9% of the total GPU computing time (which excludes the data transfer and page fault time). All performance results reported in this paper have included the runtime overhead.

The runtime also consumes extra memory for the purpose of book keeping. For instance, RDM requires a 64-bit counter to keep the global time, a 64-bit counter per object to keep the predicted next reuse time, and an 8-bit counter per object to record the global locality. Overall, the extra memory consumption is negligible.

6 RELATED WORK

A variety of research has explored ways to optimize GPU memory management in traditional contexts where data movement has to be managed explicitly. CGCM [22] provides compiler and runtime support to automate GPU memory management for CUDA programs. It also optimizes CPU-GPU communications for iterative kernel invocations. DyManD [21] is a runtime system that supports automatic CPU-GPU memory management for recursive data structures. GMAC [17] proposes a programming model to simplify and optimize GPU data management. It requires application changes. Pai *et al.* introduce a software coherence mechanism to reduce redundant data transfers between the CPU and GPU [36]. Similarly, SemCache [6] leverages a cache coherence-like protocol to optimize communication between CPU and GPU. To improve the GPU memory utilization of deep neural network training, Superneurons [43] exploits tensor level data access patterns.

Our work is distinguished from these approaches in the following ways. 1) Prior approaches always tranfer data explicitly, while we also exploit implicit transfer enabled by unified memory. For the same reason, they do not introduce access density analysis. 2) These efforts assume that the data always fits into the GPU memory and do not consider oversubscribed workloads. As a result, they do not need to distinguish importance across different data objects, while we introduce data locality and reuse distance analyses for better management of oversubscribed workloads. 3) Previous methods adopt a heavy-runtime, light-compiler scheme, where the runtime takes most responsibilities and the compiler helps with instrumentation, scheduling, etc. On the other hand, we use a light-runtime, heavy-compiler scheme instead, which employs a simple and low overhead runtime that heavily relies on compiler analyses.

Some recent research aims to reduce the virtual-to-physical address translation overhead for accelerators in the presence of unified memory. Liu *et al.* propose to concentrate the address randomness into the bank and channel index bits during GPU virtual-to-physical address mapping, in order to achieve balanced page distribution across banks and channels [30]. The proposed address mapping is configured in hardware statically. DVM [20] reduces the address translation overhead by allocating memory space with identical virtual and physical addresses in most cases. GPU specific MMU designs have also been proposed to adapt to the massive threading nature of GPU [37, 38, 48]. Koukos *et al.* propose a simplified memory model to reduce cache coherence overhead [24]. ETC [26] implements hardware extensions including eager eviction, SM throttling, and memory compression, to address data thrashing under GPU memory oversubscription. Ganguly *et al.* propose a hardware prefetcher aware pre-eviction strategy, in order to deal with memory oversubscription [16]. These research focuses on hardware and OS technology, while this paper investigates unified memory from the compiler and runtime perspective. They can be used together with our schemes to further improve unified memory performance.

Some research investigates data management in the presence of a heterogeneous GPU memory system (i.e., a GPU has more than one kind of memory attached). Agarwal *et al.* demonstrate that the ratio of data allocation in each memory should be proportional to their bandwidth in order to achieve the maximal combined bandwidth [5]. DRAGON allows the GPU to directly access large capacity NVM through loads/stores [31]. It leverages the unified memory mechanism to trigger on-the-fly data transfer between GPU memory and NVM. Zhao and Xie propose a data migration mechanism to reduce GPU power consumption in heterogeneous GPU memory systems [47]. In the context of CPU memory systems, Tahoe is a runtime system to utilize the task and profiling information for hybrid DRAM and NVM memory management [45]. [15, 29, 44, 46] also propose runtime or OS technology for heterogeneous CPU memory management. These studies ignore the impact of implicit data transfer in unified memory. Besides, they incur significant runtime overhead for profiling and data tracking, while our schemes utilize static compiler analysis information to avoid such runtime overhead.

Since the introduction of device offloading in OpenMP 4.0, several research efforts have aimed to optimize OpenMP device data management via language extensions and/or implementation techniques. Cui *et al.* propose a pipeline directive to break down OpenMP parallel loops and thus achieve device computation and communication overlapping [11]. Grinberg *et al.* introduce a method to use unified memory within the current OpenMP implementation [19]. These methods require non-trivial programming effort. In contrast, our work does not need user input or any change to the API. Our previous work studies the OpenMP offloading performance under unified memory [32], and conducts some preliminary explorations of compiler-based data locality and access density analyses [27]. Building upon them, this paper proposes a set of systematic unified memory management policies, including the use of target reuse distance.

Comparison with Previous Research. To the best of our knowledge, the following contributions of this paper are unique: 1) combining implicit and explicit data transfer for unified memory, 2) eliminating data thrashing so that GPU achieves reasonable performance for oversubscribed workloads, 3) compiler-based target reuse distance analysis, and 4) compiler and runtime collaborative access density analysis. Besides, existing methods rely on either the runtime, OS, or hardware for data management, and neglect the impact that the compiler analysis can have in this context.

7 CONCLUSION

We have developed and implemented compiler-runtime collaborative strategies to improve OpenMP GPU data management under unified memory. The proposed schemes enable the runtime to adaptively choose implicit or explicit data transfer and to leverage compiler analysis information, in order to avoid performance problems associated with unified memory. While these approaches are

applied for single GPU and single stream benchmarks in this paper, we plan to extend them for multiple GPU and stream environments. It will require our analyses and runtime systems to work in a per GPU and per stream manner. We have measured a geometric mean performance improvement of 7.6× on benchmarks and a miniapp as a result of these techniques. We have implemented them in the LLVM framework for general distribution.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviews for their insightful comments. We would also like to thank Hal Finkel, Hashim Sharif, and Martin Kong for their helpful discussions in the early phase of this work. 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. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] 2013. OpenMP 4.0 Specifications. (2013). http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
[2] 2017. The OpenACC Application Programming Interface. (2017). https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final-changes.pdf
[3] 2018. OpenMP Application Programming Interface Version 5.0. (2018). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
[4] 2018. Summit. https://www.olcf.ornl.gov/summit
[5] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *ASPLOS '15*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2694344.2694381
[6] Nabeel AlSaber and Milind Kulkarni. 2013. SemCache: Semantics-aware Caching for Efficient GPU Offloading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2464996.2465021
[7] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading Support for OpenMP in Clang and LLVM. In *LLVM-HPC '16*. IEEE Press, Piscataway, NJ, USA, 1–11. https://doi.org/10.1109/LLVM-HPC.2016.6
[8] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101. https://doi.org/10.1147/sj.52.0078
[9] M. Brehob, S. Wagner, E. Torng, and R. Enbody. 2004. Optimal replacement is NP-hard for nonstandard caches. *IEEE Trans. Comput.* 53, 1 (Jan 2004), 73–76. https://doi.org/10.1109/TC.2004.1255792
[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 44–54.
[11] X. Cui, T. R. W. Scogland, B. R. d. Supinski, and W. c. Feng. 2017. Directive-Based Partitioning and Pipelining for Graphics Processing Units. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 575–584. https://doi.org/10.1109/IPDPS.2017.96
[12] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (Jan 1998), 46–55. https://doi.org/10.1109/99.660313
[13] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (May 1968), 323–333. https://doi.org/10.1145/363095.363141
[14] Chen Ding and Yutao Zhong. 2003. Predicting Whole-program Locality Through Reuse Distance Analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 245–257. https://doi.org/10.1145/781131.781159
[15] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344
[16] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay Between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/3307650.3322224
[17] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 347–358. https://doi.org/10.1145/1736020.1736059
[18] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin. 2018. Distributed Louvain Algorithm for Graph Community Detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 885–895. https://doi.org/10.1109/IPDPS.2018.00098
[19] Leopold Grinberg, Carlo Bertolli, and Riyaz Haque. 2017. Hands on with OpenMP4.5 and unified memory: developing applications for IBM's hybrid CPU+GPU systems. In *International Workshop on OpenMP*. Springer, 3–16.
[20] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194
[21] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. 2012. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/2259016.2259038
[22] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU Communication Management and Optimization. In *PLDI '11*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/1993498.1993516
[23] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *ISCA '10*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971
[24] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead. *ACM Trans. Archit. Code Optim.* 13, 1, Article 1 (March 2016), 22 pages. https://doi.org/10.1145/2889488
[25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673
[26] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 49–63. https://doi.org/10.1145/3297858.3304044
[27] Lingda Li, Hal Finkel, Martin Kong, and Barbara Chapman. 2018. Manage OpenMP GPU Data Environment Under Unified Address Space. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 69–81.
[28] Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu Cheng. 2012. Optimal Bypass Monitor for High Performance Last-level Caches. In *PACT '12*. ACM, New York, NY, USA, 315–324. https://doi.org/10.1145/2370816.2370862
[29] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 369–383. https://doi.org/10.1145/2872362.2872401
[30] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo, and L. Eeckhout. 2018. Get Out of the Valley: Power-Efficient Address Mapping for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 166–179. https://doi.org/10.1109/ISCA.2018.00024
[31] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. 2018. DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 32, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291699
[32] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. 2017. Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading. In *LLVM-HPC '17*. ACM, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3148173.3148184
[33] NVIDIA. 2007. Compute unified device architecture programming guide. (2007).
[34] NVIDIA. 2012. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. (2012).

[35] NVIDIA. 2016. NVIDIA Tesla P100 - The Most Advanced Data Center Accelerator Ever Built. (2016).

[36] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2012. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme. In *PACT '12*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/2370816.2370824

[37] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS '14*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[38] J. Power, M. D. Hill, and D. A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *HPCA '14*. 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[39] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *ISCA '07*. ACM, New York, NY, USA, 381–391. https://doi.org/10.1145/1250662.1250709

[40] Nikolay Sakharnykh. 2018. Everything You Need to Know About Unified Memory. In *GTC '18*.

[41] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[42] TOP500 News Team. 2018. US Regains TOP500 Crown with Summit Supercomputer, Sierra Grabs Number Three Spot. (2018).

[43] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 41–53. https://doi.org/10.1145/3178487.3178491

[44] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime Data Managementon Non-volatile Memory-based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 58, 14 pages. https://doi.org/10.1145/3126908.3126923

[45] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime Data Management on Non-volatile Memory-based Heterogeneous Memory for Task-parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 31, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291698

[46] Seongdae Yu, Seongbeom Park, and Woongki Baek. 2017. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 18, 10 pages. https://doi.org/10.1145/3079079.3079092

[47] Jishen Zhao and Yuan Xie. 2012. Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration. In *ICCAD '12*. ACM, New York, NY, USA, 81–87. https://doi.org/10.1145/2429384.2429400

[48] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 345–357. https://doi.org/10.1109/HPCA.2016.7446077