Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading

Alok Mishra
alok.mishra@stonybrook.edu
Stony Brook University

Lingda Li
lli@bnl.gov
Brookhaven National Laboratory

Martin Kong
mkong@bnl.gov
Brookhaven National Laboratory

Hal Finkel
hfinkel@anl.gov
Argonne National Laboratory

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

ABSTRACT

The latest OpenMP standard offers automatic device offloading capabilities which facilitate GPU programming. Despite this, there remain many challenges. One of these is the unified memory feature introduced in recent GPUs. GPUs in current and future HPC systems have enhanced support for unified memory space. In such systems, CPU and GPU can access each other's memory transparently, that is, the data movement is managed automatically by the underlying system software and hardware. Memory over subscription is also possible in these systems. However, there is a significant lack of knowledge about how this mechanism will perform, and how programmers should use it. We have modified several benchmarks codes, in the Rodinia benchmark suite, to study the behavior of OpenMP accelerator extensions and have used them to explore the impact of unified memory in an OpenMP context. We moreover modified the open source LLVM compiler to allow OpenMP programs to exploit unified memory. The results of our evaluation reveal that, while the performance of unified memory is comparable with thatof normal GPU offloading for benchmarks with little data reuse, it suffers from significant overhead when GPU memory is over subcribed for benchmarks with large amount of data reuse. Based on these results, we provide several guidelines for programmers to achieve better performance with unified memory.

KEYWORDS

GPU, unified memory, OpenMP offloading, benchmarking, performance evaluation

ACM Reference Format:
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: LLVM-HPC'17: Fourth Workshop on the LLVM Compiler Infrastructure in HPC, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3148173.3148184

1 INTRODUCTION

As the de facto directive based parallel programming model, OpenMP has been adopted extensively in the supercomputing world and gains more and more attention in general purpose computing [8]. Using OpenMP directives and its APIs, several parallel patterns such as loop level and task based parallelism can be expressed. Thus, OpenMP facilitates CPU parallel programming in shared memory systems significantly.

Instead of CPUs, today's systems rely more on hardware accelerators like GPUs to achieve performance goals. As the current most popular accelerator, the massive multi-threading ability of GPUs can especially benefit applications with large amount of parallelism, such as scientific computations and machine learning. Therefore, GPUs will remain a crucial component in supercomputing systems in the foreseeable future. For instance, the next generation supercomputer in ORNL, Summit, will be equipped with 6 NVIDIA Volta GPUs on each node [1].

In order to leverage accelerators like GPUs, OpenMP 4.0 introduced the ability to offload computations to accelerators [3]. It is called device offloading. The offloading devices can include GPUs, FPGAs, DSPs, etc. GPU offloading is arguably the most important one among them currently. Compared to native GPU programming models such as CUDA [19] and OpenCL [22], 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 [2], OpenMP has a broader user community and is more widely available. Therefore, we expect the number of OpenMP+GPU users to continue to grow.

Despite this, many challenges remain to be addressed to make OpenMP GPU offloading performance competent. One of the most important ones is the lack of support for unified virtual memory (UVM)[1]. Normally, CPU and GPU have separate memory. Before the introduction of unified memory, host and devices would use separate memory space. As a result, CPU and GPU could not access each other's memory,

[1]We use terms unified virtual memory, UVM, and unified memory interchangeably in this paper.

and communication between CPU and its devices had to be managed by programmers explicitly.

Now, with unified memory, a single memory space shared by both CPU and GPU is used to unify the previously separated memory spaces. This permits the host to refer to memory locations on the attached devices, and the devices to access addresses on their host. In addition, the data movement is managed by the underlying system software automatically. Unified memory on pre-Pascal GPUs behaves identical to cases when not using it: all data needs to be transferred to GPU before execution. The only difference is that the programmer is not required to handle it explicitly, rather this is managed automatically by the software. On the other hand, Pascal and later GPUs support on-demand page migration, which is a big step forward, and brings two major advantages.

First, the copying process of complex data structures is greatly simplified; there is no need for address translation, and CPU and GPU can use the same pointer. Second, leveraging the unified memory feature makes it feasible to run kernels with memory footprints larger than the GPU memory capacity. Without on demand page migration, GPU offloading is possible only when dataset fits into GPU memory. While with it, part of data can reside in CPU memory, and the GPU will fetch them to its own memory when they are actually required in the computation. These advantages, promote the use of unified memory by using OpenMP as a vehicle to ease the process of GPU programming.

However, there is a significant lack of knowledge regarding how unified memory performs and how programming models should use it, especially with the on-demand page migration support in Pascal and later GPUs[2]. To the best of our knowledge, there is no comprehensive performance study for it yet. Programmers tend to think unified memory can deliver similar performance compared with cases when it is not used, and no extra work needs to be done. However, as we will show in Section 3, straightforward unified memory usage can result in significant performance degradation without careful coding and tuning. Therefore, we believe a comprehensive study of unified memory is in great need and critical.

In this paper, we develop a set of benchmarks and use them to study the performance of unified memory. Our benchmarks have been ported from the Rodinia benchmark suite [6], which is widely used in GPU performance evaluation. Since there are very few public available benchmarks for OpenMP GPU offloading, we first develop a set of OpenMP GPU offloading benchmarks. We also plan to make these benchmarks publicly available later. Upon them, we further add unified memory support by modifying the LLVM compiler [14] and using OpenMP directives/clauses in various ways. Finally, we conduct a performance study on these benchmarks. The contributions of this paper are the following:

- We develop a set of benchmarks for OpenMP GPU offloading both using and not using unified memory. Using these benchmarks, we analyze and identify the inefficiency existing in the default unified memory management policy. These benchmarks also demonstrate that using unified memory can simplify OpenMP GPU programming significantly, especially when transferring complex data structures is needed.
- To the best of our knowledge, we conduct the first comprehensive performance study for unified memory. We show cases when unified memory performs well and when it does not. We believe the findings in this paper can help programmers make better use of unified memory capabilities.
- We introduce a light-weight method to use unified memory within the current OpenMP framework, with negligible modifications to the OpenMP runtime.
- Based on the analysis results, we provide several programming guidelines for unified memory. Users can improve the performance of their applications following these guidelines when using unified memory.

The rest of this paper is organized as follows: Section 2 presents the developed benchmarks for OpenMP GPU offloading and unified memory. Section 3 evaluates the performance using the ported benchmarks. In Section 4, we describe how to improve the performance of unified memory. Section 5 discusses some related work. Finally, Section 6 consists of the conclusion of this paper.

2 BENCHMARK DEVELOPMENT

Although there are plenty of OpenMP benchmark suites publicly available, little effort has been put into OpenMP GPU offloading benchmarking since it is relatively new in OpenMP. To the best of our knowledge, there are very few public available benchmarks for OpenMP GPU offloading. (Benchmarks like SPEC for OpenMP 4.5 are still under construction.)

As the first step, we develop a set of benchmarks that use OpenMP GPU offloading. We choose the OpenMP benchmark subset of the Rodinia benchmark suite [6] as our baseline, and extend it to make use of GPU offloading. Once we complete the porting the complete suite we will release our code and try to make it an official part of Rodinia.

Rodinia was originally developed by Che *et al.* at University of Virginia to benchmark GPUs. Since its first release, many people have made contributions to Rodinia, and it is still evolving. Rodinia is one of the most widely used benchmark suites for GPU benchmarking nowadays.

Compared to other GPU benchmark suites such as Parboil [23], SHOC [9], and PolyBench/ACC [10], Rodinia has several advantages. First, benchmarks exhibit more irregular behavior, and thus are more challenging to implement and optimize for GPU execution. Next, Rodinia consists of a wider class of benchmarks compared to others. It covers the spectrum of scientific simulation, machine learning, graph processing, data processing, linear algebra, etc. Moreover, Rodinia benchmarks are written in multiple programming models, including CUDA, OpenCL, and OpenMP, which makes it relatively easy to extend. Note that the current

[2]In the rest of this paper, unified memory means unified memory with on demand page migration support unless otherwise stated.

Application	Domain	Description
Back Propagation (Backprop)	Pattern Recognition	Back Propagation is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network.
Breadth First Search (BFS)	Graph Algorithms	Graph algorithms are fundamental and widely used in many disciplines and application areas. Large graphs involving millions of vertices are common in scientific and engineering applications. Breadth First Search traverses all the connected components in a graph.
CFD Solver (CFD)	Fluid Dynamics	The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow.
K-means	Data Mining	K-means is a clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity. Many data-mining algorithms show a high degree of data parallelism.
k-Nearest Neighbors (NN)	Data Mining	Nearest Neighbor finds the k-nearest neighbors from an unstructured data set. The sequential NN algorithm reads in one record at a time, calculates the Euclidean distance from the target latitude and longitude, and evaluates the k nearest neighbors. The parallel versions read in many records at a time, execute the distance calculation on multiple threads, and the master thread updates the list of nearest neighbors.
Speckle Reducing Anisotropic Diffusion (SRAD)	Image Processing	SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features. SRAD consists of several pieces of work: image extraction, continuous iterations over the image (preparation, reduction, statistics, computation 1 and computation 2) and image compression. The sequential dependency between all of these stages requires synchronization after each stage (because each stage operates on the entire image).

Table 1: Benchmark details.

OpenMP version in Rodinia is for CPU and does not include GPU offloading.

By default, the current OpenMP implementation does not allocate data in unified memory. Users need to explicitly specify how to map data to GPU, and OpenMP runtime will use the provided information to manage data transfers between host and device. Therefore extra effort needs to be made to investigate the unified memory performance for OpenMP GPU offloading. In Section 2.2, we discuss the modifications done to the OpenMP implementation in the LLVM framework in order to provide mechanisms that allow using unified memory allocation. We also exploit existing OpenMP offloading features to prevent the OpenMP runtime from moving data explicitly. Table 1 summarizes the subset of benchmarks used in the paper.

2.1 Benchmarking OpenMP GPU Offloading

OpenMP device offloading includes two essential parts: 1) mapping data between host (i.e., CPU) and device (GPU in this paper) since they cannot address each other's memory by default, and 2) offloading computations from host to device. In this subsection, we discuss how these two parts work.

Data Mapping. Fig. 1 shows how GPU offloading generally works in OpenMP. Suppose we have two 2D arrays A and B and perform matrix addition on them to create array C. Initially all the data is located in the CPU memory. Recall that the GPU cannot access these data without the support of unified memory. In order for the GPU to access host data, OpenMP needs some mechanisms to facilitate GPU data allocation and transfer between the CPU and the GPU. This is achieved by using the `target data` directives and `map` clauses which instruct the OpenMP runtime to allocate the corresponding data copy in the device, followed by a synchronization between the copies of the host and device. The C code for the example in Figure 1 is given in Listing 1.

Listing 1: OpenMP GPU offloading code example.

```
#pragma omp target data map(to: A, B) map(from: C)
{
#pragma omp target teams distribute
for (int i = 0; i < N; i++) {
    #pragma omp parallel for
    for (int j = 0; j < M; j++)
        C[i][j] = A[i][j] + B[i][j];
}
}
```

In this code segment, the `map` clauses instruct OpenMP to copy the values of array A and B from the CPU to GPU at

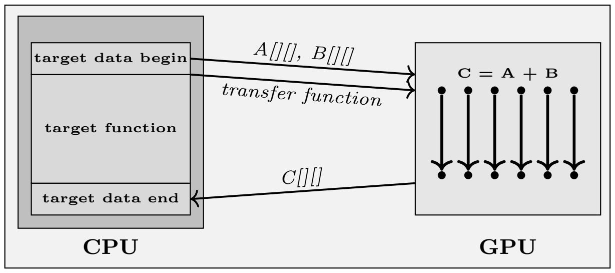


Figure 1: Traditional OpenMP GPU offloading without unified memory.

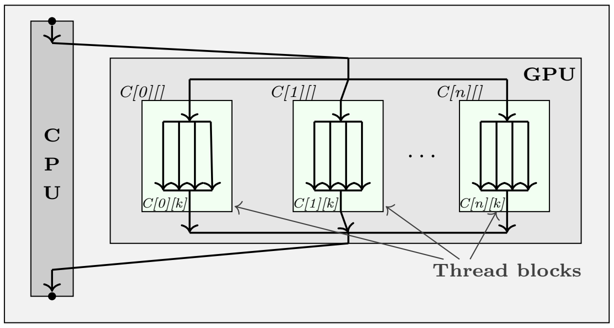


Figure 2: Two level of parallelism.

the start of the target data region with the keyword `to`. It also instructs to copy array C from GPU to CPU at the end of the target data region with the keyword `from`. Note that the keyword `tofrom` can be used to enforce data synchronizations both at the beginning and at the end of the data region (i.e., when copying-in and copying-out).

Computation Offloading. Once the data mapping has completed, control is transferred to the GPU. This is done by using the `target` directive as shown in Listing 1. Then, the computation is offloaded to the device when the data synchronization is finished. After this, the compiler packs the target region into a function, which is transferred to the target device, i.e., to the GPU. Upon termination of the offloaded kernel, the control returns to the CPU.

Unlike OpenMP's flat threading model on CPUs, GPUs have a hierarchical thread organization: a GPU kernel is composed by multiple thread blocks, and each thread block consists of multiple threads[3]. Different thread blocks can execute in parallel, as well as different threads within the same thread block. To fully exploit this two level of parallelism, OpenMP introduces a new concept called *team* besides traditional thread. A team corresponds to a thread block in the GPU context, and it can include multiple threads, which correspond to threads within a GPU thread block.

At the beginning of a target region, only one team and one member thread is active. To have multiple teams, we use the directive `teams distribute` at first, which distributes the entire loop iteration space among all teams. Further, if there are additional nested parallel loops, we use the directive `parallel for` upon them to distribute the nested loop iterations among threads within a team as shown in Listing 1. In scenarios when there is only one level of parallel loops, or when there is enough parallelism in the outer loop, we make use of the combined directive `teams distribute parallel for` to distribute the iteration space of one loop both among teams and threads within a team. Figure 2 illustrates a typical workload partition among teams and threads for the example of Listing 1.

Deep Copy Challenge. During the process of implementing GPU offloading on our benchmarks, we found that the most challenging and error-prone part is to map complex data structures between the host and the device, many of which can make use of pointers to access different memory regions. This constitutes a strong case for the need of *deep copy*. Listing 2 shows the main data structure `BPNN` in the Backprop benchmark. Besides scalar variables, it also includes many pointers that refer to the actual data storage of a neural network. It is not sufficient to just map an instance of `BPNN` to the GPU memory in this scenario, since the pointers stored in it are still pointing to the CPU memory, where the GPU cannot access.

To solve this problem, we need to explicitly map all the memory regions that the pointers within `BPNN` point to. We also need to create another instance of `BPNN` where all the pointers have been modified to point to the corresponding GPU memory regions before mapping it to the GPU. Potential pointer aliasing makes this process even more complicated because the programmer needs to guarantee that all aliasing pointers are translated correctly. It is a non-trivial job to map all data correctly. It requires deep knowledge about the ported application. This process took the most time in our benchmark development. As we will see in Section 2.2, unified memory helps alleviate this problem significantly.

Listing 2: The `BPNN` structure in Backprop.

```
typedef struct {
  int input_n;
  int hidden_n;
  int output_n;
  float *input_units;
  float *hidden_units;
  float *output_units;
  float *hidden_delta;
  float *output_delta;
  float *target;
  float **input_weights;
  float **hidden_weights;
  float **input_prev_weights;
  float **hidden_prev_weights;
} BPNN;
```

2.2 Benchmarking Unified Memory

As shown in Section 2.1, in current OpenMP GPU offloading, `map` clauses are used to perform the GPU data allocation as well as the data movement between CPU/GPU. In such scenarios, it is the responsibility of the OpenMP runtime to allocate data in the GPU memory and to generate explicit data transfers to make sure the data is in a coherent state. However, when using unified memory, we would like the underlying system to manage these things automatically. Figure 3 illustrates how the function and data transfers work in the presence of unified memory. Therefore, in order for OpenMP to use unified memory, first, we need its runtime to relinquish the data movement management.

To do this, `map` clauses should not be used in the presence of unified memory. To make things more complicated, the

[3]NVIDIA terminology is used in this paper. Thread block is called workgroup in AMD terminology.

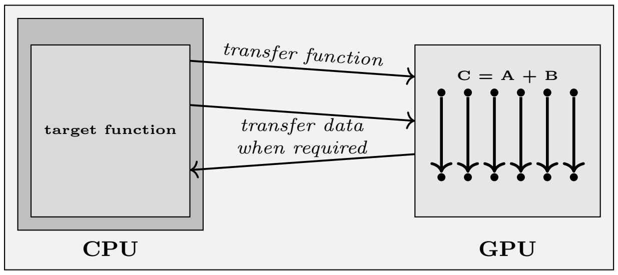


Figure 3: OpenMP GPU offloading with unified memory.

current implementation of the OpenMP runtime keeps records of all the GPU and CPU data and their relationship. If the `map` clause is not used for a pointer in the target region, the OpenMP runtime will assume that this pointer is the CPU copy and try to find the corresponding GPU copy in its records. This raises another problem since there is no such copy in the presence of unified memory, and CPU and GPU just refer to the same data. The approach we follow to address this issue, is to use `is_device_ptr` clauses to inform the OpenMP runtime that a particular pointer can in fact be used by the GPU.

The last problem encountered is that the current OpenMP implementation never allocates GPU data using unified memory allocation APIs. Therefore, we modify the OpenMP runtime in the LLVM framework to make it allocate data in unified memory. To be more specific, we modify the OpenMP runtime API `omp_target_alloc` so that it will use the unified memory allocator by default. We also try to pass the data allocated by the CUDA unified memory allocator `cudaMallocManaged` to the OpenMP target region directly. However, the latter method does not work because the data allocated by `cudaMallocManaged` is invisible to the OpenMP runtime.

Overall, to enable unified memory in OpenMP GPU offloading we use the modified `omp_target_alloc` for data allocation in unified memory and `is_device_ptr` clauses to pass them to target regions. The rest of the implementation remains unchanged, and works as the default GPU offloading introduced in Section 2.1. The C example code that shows how we use unified memory, where we assume A, B, and C are integer arrays is given in Listing 3.

Listing 3: Unified memory code example.

```
A = omp_target_alloc(N*M*sizeof(int));
B = omp_target_alloc(N*M*sizeof(int));
C = omp_target_alloc(N*M*sizeof(int));
#pragma omp target data is_device_ptr(A, B, C)
{
#pragma omp target teams distribute
for (int i = 0; i < N; i++) {
    #pragma omp parallel for
    for (int j = 0; j < M; j++)
        C[i][j] = A[i][j] + B[i][j];
}
}
```

Address Deep Copy with Unified Memory. As we discussed earlier, programming complex/deep data structure copy is time consuming and error-prone in cases where the data communication is managed explicitly by programmers. On the other hand, this job is much easier in the presence of unified memory. The reason behind this is that, with unified memory, both the GPU and the CPU can use the same address space to access memory regions, i.e., there is no need for pointer translation. For the Backprop example shown in Listing 2, With this new mechanism, we can conveniently pass the structure pointer of `BPNN` to the GPU without further concerning about the mapping of pointers inside it.

3 EVALUATION

3.1 Experimental Methodology

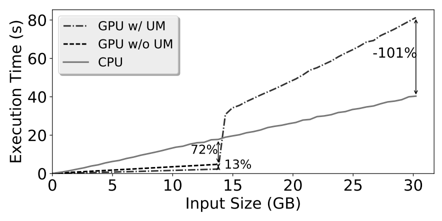
To evaluate the performance of our benchmarks, we use the next generation supercomputer prototype at ORNL, SummitDev. SummitDev has a total of 54 nodes. Each node is equipped with 2 POWER8 CPUs and 4 Tesla P100 GPUs, connected through NVLink 1.0. Table 2 provides the detailed configuration of CPUs and GPUs in the compute node of SummitDev.

Parameter	Value
CPU	2 POWER8
Cores / Socket	10
Threads / Core	8
CPU Clock	2.0 GHz
Main Memory	256GB DDR4
GPU	4 Tesla P100
SMs / GPU	56
SM Clock	1328 MHz
Register File Size / SM	256 KB
FP32 CUDA Cores / SM	64
FP64 CUDA Cores / SM	32
GPU L2 Cache Size	4096 KB
GPU Memory	16GB HBM2
CPU & GPU Interconnect	NVLink 1.0

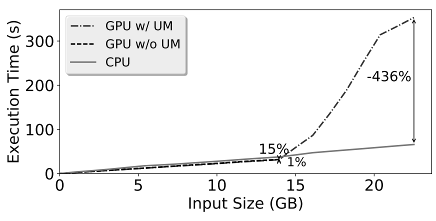
Table 2: Compute node configuration on SummitDev.

We use Clang 4.0 [4] to compile benchmarks for both CPU parallel computing and GPU offloading. To enable offloading for NVIDIA GPUs, along with `-fopenmp` we need to pass `-fopenmp-targets=nvptx64-nvidia-cuda` to Clang. All benchmarks are compiled under the O2 optimization level. Note that the Clang version used in our experiments includes enhancement to support unified memory. The Linux kernel version is 3.10.0 and the CUDA version is 8.0.61 on SummitDev.

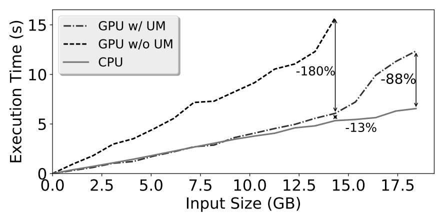
We evaluate the performance of GPU offloading without unified memory (GPU w/o UM), GPU offloading with unified memory (GPU with UM), and the CPU version where all computation is performed by the CPU. We were not able to



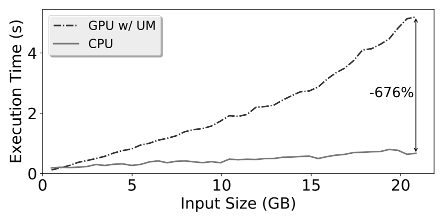
(a) BFS.



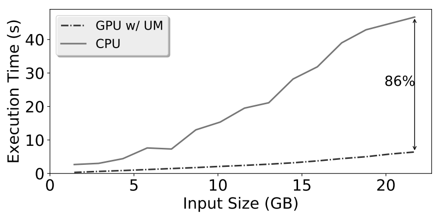
(b) CFD.



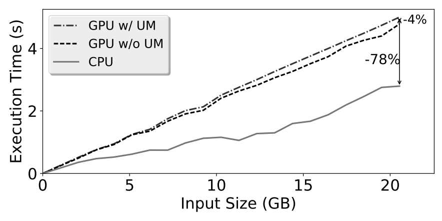
(c) SRAD.



(d) Backprop.



(e) K-means.



(f) NN.

Figure 4: Performance of CPU, GPU without unified memory, and GPU with unified memory.

port K-means and Backprop to GPU without unified memory due to the complexity of their data structures. These two benchmarks require deep copy, which significantly increases the difficulty of normal GPU offloading as described in Section 2.1. Thus, K-means and Backprop only have the CPU version and the GPU with unified memory version. Nevertheless, these benchmarks will also be included in the final public suite once their porting is completed.

For each benchmark, we generate inputs of various sizes to study their performance impact. For instance, the input sizes of BFS range from 0.2GB to 30GB, whereas for CFD it grows from 0.5GB to 22GB. We make sure the largest data size for each benchmark is always larger than the GPU memory capacity, so that we are able to study the performance impact of GPU memory over subscription. To evaluate the performance, we measure the execution time from the start till the end of the target data region. By doing this, the measured time captures both the GPU computation time and other overheads associated with OpenMP GPU offloading, such as data transfer time between CPU and GPU. We run each experiment 5 times and use the average execution time as the final result.

3.2 GPU Offloading without Unified Memory

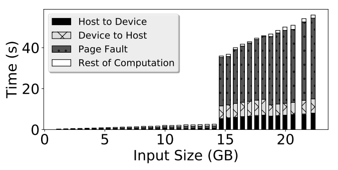
Figure 4 shows the results of our evaluation. In this subsection, we focus on comparing the execution time of benchmarks when running on CPU vs. GPU without unified memory. The results show that GPU offloading benefits 2 out of 4 benchmarks when the input size is smaller than that of the GPU memory. While BFS enjoys the most performance benefit from GPU offloading, which is on average 68.8% across different inputs, CFD shows a moderate gain from GPU offloading (21.7% improvement on average). The reason why BFS and CFD benefit from GPU offloading is that they are more computation bound and show a moderate amount of data reuse. Therefore, their performance is moderately improved by the GPU's massive threading ability.

On the other hand, NN and SRAD do not benefit from OpenMP GPU offloading compared to their CPU counterparts. NN and SRAD suffer from an average performance degradation of 46.2% and 60.3% respectively. For these two benchmarks, the data transfer overhead outweighs the computing throughput benefit of GPU, because there is little data reuse and limited computation volume on each data item.

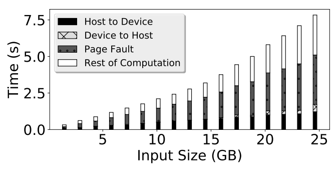
We also observe that the normal GPU offloading fails when the input size is comparable or larger than the GPU memory size. This is because in traditional GPU offloading, all the data must first be copied to the device before starting the kernel execution. When the required data exceeds the GPU's memory capacity the data allocation/copy will fail. We observe the failure threshold is roughly 14GB. It is not possible to utilize the full memory capacity 16GB, GPU memory, since part of the memory space is reserved for thread private data and system usage. NN does not fail because its GPU memory requirement is constant and does not change with inputs. On the other hand, the CPU version is able to deal with large inputs since 1) the CPU memory is much larger (256GB per node on SummitDev), and 2) virtual memory system could exchange data between memory and disk dynamically if necessary.

3.3 Unified Memory

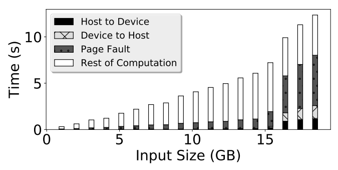
Figure 4 also shows the GPU offloading performance of different benchmarks with unified memory. These benchmark



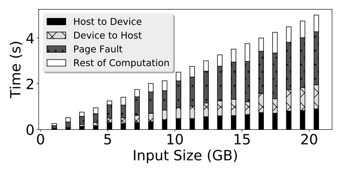
(a) BFS.



(b) K-means.

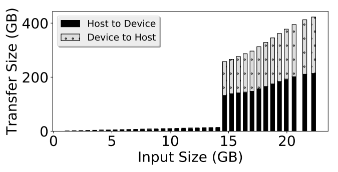


(c) SRAD.

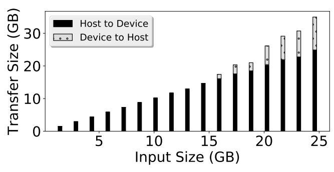


(d) NN.

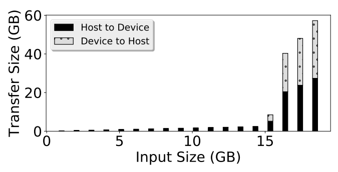
Figure 5: Execution time breakdown.



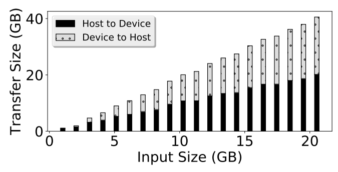
(a) BFS.



(b) K-means.



(c) SRAD.



(d) NN.

Figure 6: Data transfer breakdown.

variants show significantly different performance depending on the program behavior. We distinguish two categories of benchmarks here, those with large amounts of data reuse (BFS, CFD, K-means and SRAD), and those with little to none data reuse (Backprop and NN). We discuss their performance separately in this subsection.

Benchmarks with Significant Data Reuse. For the first category of benchmarks, we observe that unified memory brings two major advantages. First, it helps improve performance compared with GPU offloading without unified memory when the working set fits into the GPU memory. For instance, unified memory outperforms traditional offloading by an average of 18% for BFS when the dataset fits into GPU memory. The reason why BFS enjoys the most performance improvement is that, in each iteration, BFS accesses a fraction of nodes. Then, the next iteration will access the neighbors of current accessed nodes until all nodes have been traversed. In such a scenario, unified memory does not bring all the data at the beginning. Instead, data transfers happen on demand during the entire traversal process and thus the data movement overhead is amortized. Without unified memory, all the data would have to be moved to the GPU before any GPU computation can start. Therefore unified memory achieves better performance in such cases.

The second advantage of unified memory is that it helps solve the problem of GPU memory over subscription, although the performance may be poor. BFS, CFD, and SRAD suffer from significant performance degradation when the data size exceeds the GPU memory capacity. For instance, when using 14GB inputs (i.e., GPU memory is not yet over subscribed), the performance gain of GPU w/ UM is 86% for BFS compared with CPU, while this is 16% for CFD. Then, as the input size increases, the GPU offloading crashes without unified memory, but the benchmarks are able to run correctly using unified memory, though with significantly degraded performance. Depending on the benchmark, the performance degradation varies. CFD suffers from the most severe degradation, whose execution time is 4.4x of that of the CPU version at the 22GB input.

To further analyze the performance results, especially the reasons behind the significant performance degradation when GPU memory is over subscribed, we breakdown the execution time and analyze its different components. The potential overhead of unified memory includes data transfer from host to device, that from device to host, and page fault processing. The NVIDIA GPU profiler nvprof is used to collect this information.

Figure 5 shows the execution time breakdown of 4 benchmarks. The "Rest of Computation" is computed using (total_execution_time − data_transfer_time − page_fault_processing_time). Therefore it does not necessarily reflect the actual GPU computing time since some computation overlaps with data transfer and page fault processing. For instance, we observe that the "Rest of Computation" time decreases when input size exceeds that of the GPU memory for SRAD because of the overlapping. We were not able to collect these data for CFD because of the failure of nvprof. For benchmarks with significant data reuse, including BFS, K-means and SRAD, the results show that the data transfer time and page fault processing increases significantly when the GPU memory is oversubscribed. This is caused by the inefficiency of the default unified memory management policy, which is presumably an Least Recently Used (LRU) like policy.

To show more details, Figure 6 illustrates the data transfer volume with unified memory for different benchmarks. Ideally, data transfers under unified memory should be near optimal because it only happens on demand which results in little redundant data movement. Unfortunately this is not true when the working set size exceeds that of the GPU memory. Because of data thrashing, both BFS and SRAD show a significant increment of data traffic from host to device and from device to host.

We use BFS as an example to explain the reasons of data thrashing for benchmarks with large amount of data reuse. There are two types of data reuse in BFS: 1) a node may be traversed multiple times based on the input graph structure, and 2) there is a lot spatial locality since neighbor nodes may get accessed in different iterations, resulting in page level reuse. These reuse happens in a relative long distance, and when the total data size is larger than that of GPU memory, it is likely that a data item will be evicted by the default LRU-like policy before it gets reused. This results in data thrashing: data are evicted before getting reused, and placing current accessed data in the GPU memory will further evict future reused data. As a result, little data locality is captured by the GPU memory under the LRU like management policy and the performance thus suffers [12, 15, 21].

Therefore, we conclude that reducing/eliminating data thrashing is critical to achieving better performance on benchmarks with significant data reuse. Improving unified memory management is an important means to achieve this goal.

Benchmarks with Little Data Reuse. For benchmarks such as Backprop and NN, the performance of unified memory is roughly proportional to the input size. It is also slightly outperformed by GPU offloading without unified memory. The reason behind this is that the data is effectively streamed (accessed without reuse). In such scenarios, the overhead associated with on demand data migration in unified memory becomes more dominant. As we observe in Figure 5 and 6, unified memory suffers from significant page fault processing overhead and on demand page movement overhead. The results for Backprop are similar. For these benchmarks, there is not enough data reuse to amortize the overhead compared to benchmarks with data reuse.

We also do not observe a large performance gap between inputs that fit in GPU memory and those that do not fit like those in BFS and CFD, since no data thrashing incurs without data reuse. Hence, we conclude that it is critical to reduce the data movement and page fault overhead to achieve better unified memory performance for these benchmarks.

4 IMPROVING UNIFIED MEMORY PERFORMANCE

As shown in Section 3.3, unified memory does not always perform well. Users need to put in extra efforts in order to achieve optimized performance. In this Section, we will introduce several programming guidelines we get from the performance results.

Applications with Significant Data Reuse. For such cases, we observe that unified memory is preferable when dataset fits into GPU memory, but it suffers from significant data thrashing when dataset does not fit.

In such scenarios, we suggest programmers to pin data with good locality into GPU memory and allow unified memory runtime to manage the data with low locality. We leverage the existing OpenMP APIs to achieve data pinning. For instance, in order for an array A to be pinned, its memory must be allocated in the regular fashion (without unified memory). Then, in an OpenMP target region, `map` clauses are required to allow the OpenMP runtime to explicitly move pinned data between CPU and GPU.

In doing this, the pinned data cannot be replaced by the unified memory management policy and thus will get consistent reuse. Data thrashing will be reduced since it will only happen to those unpinned data with poor locality. Note that it is not possible to pin data whose size is larger than the GPU memory size.

Applications with Little Data Reuse. In these cases, our evaluation shows that the achieved performance is slightly better without unified memory since all the data can be preloaded into the GPU memory, thereby avoiding the page fault overhead. This does not mean we should give up unified memory in these applications since the ease of programming with unified memory is still desirable. Instead, we provide suggestions to help unified memory achieve similar performance without significant modifications.

To reduce overhead associated with on-demand data migration, we suggest programmers to insert prefetching instructions before the data is accessed. By doing this, the data will be ready in GPU memory when needed and both data migration and page fault overhead will be minimal. OpenMP programs can use the CUDA runtime API `cudaMemPrefetchAsync` on the desired data to achieve this goal, thanks to the interoperability of OpenMP and CUDA.

5 RELATED WORK

A large amount of benchmarks have been developed to study GPU performance. Rodinia [6, 7] is one of the first suites developed for GPU benchmarking. Currently, it provides implementations for programming models such as CUDA, OpenCL and OpenMP. It is also arguably the most widely used one. Parboil benchmarks [23] were developed for throughput processors including GPUs, and they are written in CUDA, OpenCL, and OpenMP. SHOC [9] is a GPU benchmark suite designed for both OpenCL and CUDA with a focus on scalability. PolyBench/ACC [10] ports the PolyBench suite to run on GPUs. The NUPAR benchmark suite [24] supports the exploration of relatively new GPU features, including dynamic parallelism and concurrent kernel execution in CUDA and OpenCL. Valar [18] aims to study the interaction between CPU and GPU in heterogeneous systems. NAS parallel benchmarks [13] are widely used to benchmark supercomputers. Although OpenMP variants are provided, GPU support is not yet available. Furthermore, none of these benchmarks suites explore the unified memory features of recent GPUs, especially on the presence of on-demand page migration. Thus, we believe this work contributes to fill part of the knowledge gap.

Since the introduction of device offloading in OpenMP 4.0, several compilers have adopted this feature. For instance, [4] describes how to implement this extension in the LLVM compiler. Bercea *et al.* evaluated the performance of OpenMP 4.0 GPU offloading using a CORAL proxy application and found that its performance is comparable to that of CUDA after some tuning [5]. Martineau *et al.* evaluated the performance of LLVM OpenMP 4.5 GPU support using several simple kernels and mini-apps [16]. Based on the results, they propose to perform optimizations by combining directives, caching read-only data, etc. Cray's compiler also supports OpenMP GPU offloading currently. Martineau *et al.* studied the OpenMP GPU offloading performance of the Cray compiler on NVIDIA K20X and found that its performance is reasonably similar to that of CUDA [17]. None of these compilers have unified memory support currently, and thus existing studies do not pay attention to unified memory.

There are several proposals to simplify and optimize GPU memory management. CGCM [11] provides compiler and runtime support to automatize GPU memory management for CUDA programs. Pai *et al.* propose a software coherence mechanism to reduce redundant data transfer between CPU and GPU [20]. These works aim at traditional GPU programs where data movement is managed explicitly by the programmer, and does not consider unified memory.

6 CONCLUSION

In this paper, we describe how we develop benchmarks for OpenMP GPU offloading, with a focus on unified memory. Using these benchmarks, we evaluate the performance of unified memory and discover some inefficiencies existing in current unified memory system. We also provide several programming guidelines for users to achieve better performance with unified memory.

For the future work, we will continue to develop more benchmarks for unified memory study in the context of OpenMP GPU offloading. We will also design both programming and automatic strategies to optimize unified memory performance.

7 ACKNOWLEDGEMENT

We would like to thank our colleague Hashim Sharif for his feedback and helpful discussions. 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] Oak Ridge Leadership Computing Facility - Summit. https://www.olcf.ornl.gov/summit
[2] OpenACC. http://www.openacc.org
[3] 2013. OpenMP 4.0 Specifications. (2013). http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
[4] 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 *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC (LLVM-HPC '16)*. IEEE Press, Piscataway, NJ, USA, 1–11. https://doi.org/10.1109/LLVM-HPC.2016.6
[5] Gheorghe-Teodor Bercea, Carlo Bertolli, Samuel F. Antao, Arpith C. Jacob, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, and Kevin O'Brien. 2015. Performance Analysis of OpenMP on a GPU Using a CORAL Proxy Application. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems (PMBS '15)*. ACM, New York, NY, USA, Article 2, 11 pages. https://doi.org/10.1145/2832087.2832089
[6] 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.
[7] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on.* IEEE, 1–11.
[8] 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
[9] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702
[10] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012.* IEEE, 1–10.
[11] 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 *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/1993498.1993516
[12] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971
[13] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. (1999).
[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673
[15] Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu Cheng. 2012. Optimal Bypass Monitor for High Performance Last-level Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 315–324. https://doi.org/10.1145/2370816.2370862
[16] M. Martineau, S. McIntosh-Smith, C. Bertolli, A. C. Jacob, S. F. Antao, A. Eichenberger, G. T. Bercea, T. Chen, T. Jin, K. O'Brien, G. Rokos, H. Sung, and Z. Sura. 2016. Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 54–64. https://doi.org/10.1109/PMBS.2016.011
[17] M. Martineau, S. McIntosh-Smith, and W. Gaudin. 2016. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 338–347. https://doi.org/10.1109/IPDPSW.2016.70
[18] Perhaad Mistry, Yash Ukidave, Dana Schaa, and David Kaeli. 2013. Valar: A Benchmark Suite to Study the Dynamic Behavior of Heterogeneous Systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*

(GPGPU-6). ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/2458523.2458529

[19] NVIDIA. 2007. Compute unified device architecture programming guide. (2007).

[20] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2012. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/2370816.2370824

[21] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 381–391. https://doi.org/10.1145/1250662.1250709

[22] 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.

[23] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[24] Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David Kaeli. 2015. NUPAR: A Benchmark Suite for Modern GPU Architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/2668930.2688046