A Case Study of Porting HPGMG from CUDA to OpenMP Target Offload

Christopher Daley(✉), Hadia Ahmed, Samuel Williams, and Nicholas Wright

Lawrence Berkeley National Laboratory (LBNL),
1 Cyclotron Road, Berkeley, CA 94720, USA
{csdaley,hahmed,SWWilliams,NJWright}@lbl.gov

Abstract. The HPGMG benchmark is a non-trivial Multigrid benchmark used to evaluate system performance. We ported this benchmark from CUDA to OpenMP target offload and added the capability to use explicit data management rather than managed memory. Our optimized OpenMP target offload implementation obtains a performance of 0.73x and 2.04x versus the baseline CUDA version on two different node architectures with NVIDIA Volta GPUs. We explain how we successfully used OpenMP target offload, including the code refactoring required, and how we improved upon our initial performance with LLVM/Clang by 97x.

Keywords: HPGMG · Managed memory · CUDA · OpenMP target offload · NVIDIA · Volta · V100 · GPU

1 Introduction

The systems deployed at supercomputing centers increasingly consist of heterogeneous node architectures with both CPUs and GPUs. At the present time this includes the Summit supercomputer at ORNL and the Sierra supercomputer at LLNL. However, there are also many planned deployments which will use GPU accelerators from NVIDIA, AMD or Intel. It is important that user applications can run efficiently on a variety of accelerators. Non-portable programming approaches are not practical for a large number of application code teams because of lack of resources, no detailed knowledge of specific accelerators, or code maintainability concerns. OpenMP target offload is one approach to enable users to portably offload computation to accelerators using directives [29].

There are case studies of user experiences of OpenMP target offload and even an entire benchmark suite to evaluate OpenMP target offload performance (SPEC ACCEL [16]). However, the case studies often consider relatively simple micro-benchmarks or mini-apps. There is generally a gap between OpenMP target offload case studies and the complexity of full applications run at supercomputing centers. It is thus important to assess the ease and success of using OpenMP target offload in non-trivial applications. This can find gaps in the OpenMP specification, assist with developing best practices for other users to follow, and identify bugs and performance issues in OpenMP compilers.

In this work, we ported a non-trivial application named HPGMG [1,32] from CUDA to OpenMP target offload and extended the code to use explicit data management rather than managed memory. Managed memory is a capability enabling the CPU and GPU to transparently access the same data. It is used in many non-trivial applications [3], however it is not portable to all systems with GPUs and has potential performance issues [31]. We explain the code modifications required to use explicit data management as well as situations where a detailed understanding of the OpenMP specification is needed to correctly and efficiently manage data. We show performance of both code versions with multiple OpenMP compilers against the baseline CUDA performance. Our contributions include:

– We created an optimized OpenMP target offload implementation of HPGMG which achieves a performance of 0.73x and 2.04x versus the baseline CUDA version on two different node architectures with NVIDIA Volta GPUs.
– We describe how we successfully ported the managed memory CUDA version of HPGMG to OpenMP target offload and how we added explicit data management. This includes details about the refactoring required and the issues we encountered when mapping complicated data structures to the device.
– We compare the performance of 3 OpenMP offload compilers and detail a major bottleneck in the open-source LLVM/Clang compiler related to the size of the OpenMP present table. We describe our code changes to workaround LLVM/Clang compiler limitations and how we improved upon the initial LLVM/Clang performance by 97x.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the HPGMG application and discusses how we ported it to OpenMP target offload and added explicit data management. Section 4 introduces the systems and compilers used as well as the benchmarked HPGMG configuration. Section 5 shows the performance of the managed memory and explicit data management versions of HPGMG, and explains our progressive code optimizations for more efficient execution with the LLVM/Clang compiler. Section 6 discusses lessons learned. Section 7 concludes the paper.

2 Related Work

There are many examples of using OpenMP target offload to execute applications on platforms with GPUs, e.g. Nekbone [14], Lulesh [6,17], miniMD [30], Neutral [22] and other UK mini-apps [21,23]. The performance analysis in these papers mostly focuses on how well a compiler optimizes compute kernels for GPUs. User guidelines exist for achieving high performance with OpenMP target offload on CPU and GPU targets, e.g. using combined `teams distribute parallel for` OpenMP constructs and avoiding the use of explicit OpenMP schedules [15,24]. Similar guidelines were followed to port the entire SPEC ACCEL benchmark suite from OpenACC to OpenMP target offload [16]. Compiler optimization research exists that explores how to accelerate a broader range of OpenMP target regions on GPUs, specifically when user code appears between `target` and `parallel` constructs [5,11,15,34]. There has been some initial work done on identifying sources of overheads in OpenMP runtimes [26]. In this publication, we evaluate GPU compute performance with 3 OpenMP compilers and identify a significant LLVM/Clang OpenMP runtime overhead not previously reported.

There are fewer publications detailing data management challenges. The challenge of mapping structs containing pointers to the device is described in [22]. Here, the authors suggest a user code transformation of adding a new pointer variable that points to a struct pointer, and then mapping and operating on the new pointer variable. Cleaner methods to map structs containing pointers are explained in [12]. In a follow on paper, the authors show how managed memory allocations via `cudaMallocManaged` simplifies the use of C++ objects and enables `std::vector` to be used in OpenMP target offload applications [13]. There has been a successful study modifying OpenMP target offload applications and the LLVM/Clang compiler to use managed memory allocations [25]. However, managed memory is not available on all CPU/GPU systems and there have been several studies reporting higher than expected overheads when using managed memory [25,31]. One method to successfully use explicit data management is to create data management abstractions using OpenMP runtime API calls only, e.g. RAJA OpenMP target offload backend [4], and GenAsis [7]. The use of data management directives can be problematic because of lack of compiler support [35]. In this publication, we explain how we successfully used OpenMP data management directives to map a complicated nested data structure to the GPU.

3 The HPGMG Mini Application

HPGMG is a finite-volume geometric Multigrid HPC benchmark [1]. It is written in C99 and has been parallelized using MPI and optionally OpenMP. The benchmark creates a Multigrid grid hierarchy once and repeats the same Multigrid solve operation a user-specified number of times. The benchmark performance metric is a throughput metric of Degrees of Freedom per second (DOF/s). The metric does not include the time to build the Multigrid grid hierarchy.

A Multigrid solver is an iterative linear solver which achieves fast convergence by solving an $Ax = B$ equation at different resolutions. Multigrid solvers often use a V-cycle computational pattern. The fine-to-coarse part of the V-cycle consists of a smoothing operation on the finest structured grid, the calculation of a residual, and restriction of this data to the next coarsest grid. A direct solver is used on the coarsest level. The coarse-to-fine part of the V-cycle consists of interpolation of data to finer grids followed by a smoothing operation. An alternative to a V-cycle computational pattern is an F-cycle computational pattern which consists of multiple V-cycles using progressively more levels.

The coarsest level in the Multigrid hierarchy consists of a level with 2^3 grid points. Each successive finer level of the hierarchy has 4^3, 8^3, 16^3, ... grid points. The level data is divided into blocks of variable size up to a user-specified maximum size, typically 32^3 or 64^3 grid points. These blocks are distributed between

MPI ranks to balance computational load and memory footprint. HPGMG uses a nested data structure named `level_type` to hold all block data, communication buffers, and block neighbor metadata for a single level.

3.1 HPGMG-CUDA

HPGMG-CUDA is a CUDA port of HPGMG [32]. It depends on CUDA managed memory allocations using `cudaMallocManaged` to enable the same data to be accessed by CPU and GPU. A single execution of HPGMG-CUDA uses 14 different CUDA kernels. HPGMG-CUDA includes an optimization where operations for coarse levels are run on the CPU and operations for fine levels are run on the GPU.

3.2 Porting HPGMG-CUDA to OpenMP Target Offload

The approach we took to port HPGMG-CUDA to OpenMP target offload involved mixing the original CUDA memory allocation API calls with newly-created OpenMP target offload regions. In HPGMG-CUDA, the CUDA kernels access block data through a `level` structure variable of type `level_type` which is passed by value as part of the CUDA kernel launch. This data structure contains many scalar and pointer variables, where pointer variables accessed by CPU and GPU point to memory allocated using `cudaMallocManaged`. This makes OpenMP data management as simple as adding `map(to:level)` to the OpenMP target region because the targets of the pointer variables can be accessed by the CPU and the GPU.

Our OpenMP target offload code regions look nearly identical to the original CUDA kernels. The only difference is that the CUDA launch configuration is replaced with loops inside the OpenMP target region. There is a repeating pattern in our OpenMP target regions of a coarse-grained loop over blocks, followed by extraction of block data, followed by a fine-grained loop over grid points in a block. We parallelized and work-shared these loops using the `teams distribute` combined construct on the outer loop and the `parallel for` combined construct on the inner loop.

We implemented an incremental porting approach by creating a wrapper layer that dispatched to the original CUDA kernel or our newly-created function containing an OpenMP target region. This allowed us to test the correctness of one OpenMP function at a time. If the numerical results are not identical then we know we made a mistake in the OpenMP function or there is a compiler bug. This methodology requires compiling all code without fused-multiply-adds and fast math in order to expect a numerically identical solution.

3.3 Adding Explicit Data Management to HPGMG

Efficient explicit data management requires minimizing the number of data transfers between host and device. Our approach involved creating the `level` structure variable once on the device at program initialization. Most fields in the device version of `level` never need to be accessed by the host. The exception is the raw block data which must be transferred to the host every solution step because some HPGMG functions do not have GPU implementations. In order to use this approach, we had to refactor the code so that our modified OpenMP target regions access level data through a pointer to the device version of `level`. The code transformation is shown in Fig. 1.

Example initial code using managed memory

```
void smooth(level_type level, ...) {
// Map "level" to the device. All pointer variables in "level" point to
// data allocated with cudaMallocManaged. These addresses are thus valid
// on host and device
#pragma omp target teams distribute map(to:level)
  for (int blk=0; blk < level.num_my_blocks; blk++) {
```

Example refactored code using explicit data management

```
void smooth(level_type *level, ...) {
// Map zero-length array section of "level". This attaches the "level"
// pointer in the target region to the device copy of "level" which
// is already present on the device
#pragma omp target teams distribute map(to:level[:0])
  for (int blk=0; blk < level->num_my_blocks; blk++) {
```

Fig. 1. The code transformation used to efficiently implement explicit data movement in HPGMG.

The `level` structure variable contains a small number of data buffers for the entire level. This enables the data buffers to be copied efficiently between CPU and GPU in bulk data transfers. However, it has a software consequence that the blocks for each level must contain multiple pointers to different offsets within the larger data buffers. This necessitates additional OpenMP data management to ensure that the pointers point to the appropriate device data buffer and not the original host data buffer. The way we attached the appropriate device address to the device block pointers used the `[:0]` syntax and is shown in Fig. 2. This syntax has an additional effect of creating an association between the host and device address in the OpenMP runtime.

We implemented two techniques to ensure correctness of the explicit data management version of HPGMG. The first technique involved adding a print statement in the wrapper layer to enable tracing of the executed GPU functions in both managed memory and explicit data management versions. This allowed us to find a case where a missing `target update` construct caused the BiCGStab iterative solver to terminate early. Our second technique involved creating functions that calculated mean and L1 norm summary statistics of the level data in the location that owns the level, i.e. fine levels are owned by the GPU and coarse levels are owned by the CPU. We called these functions after each function and compared results against the managed memory version.

Attaching a device address to a device pointer

```
for (shape=0; shape<STENCIL_MAX_SHAPES; shape++) {
  for (block=0; block<3; ++block) {
    for (b=0; b<level->exchange_ghosts[shape].num_blocks[block]; ++b) {

#pragma omp target enter data \
  map(alloc:level->exchange_ghosts[shape].blocks[block][b].read.ptr[:0])

    }
  }
}
```

Fig. 2. OpenMP target offload `[:0]` syntax to make device pointers point to device addresses and not host addresses.

4 Experimental Methodology

4.1 Hardware and Software Environment

We used the Summit supercomputer at OLCF [18] and the Cori-GPU testbed at NERSC [27]. The characteristics of the two systems which are most relevant for this study are shown in Table 1.

Table 1. Overview of the Cori-GPU and Summit systems.

	Cori-GPU	Summit
Node architecture	Cray CS-Storm 500NX	IBM AC922
Node CPUs	2 × Intel Skylake	2 × IBM Power 9
Available cores per CPU	20 @ 2.40 GHz	21 @ 3.07 GHz
Node GPUs	8 x 16 GB NVIDIA V100	6 x 16 GB NVIDIA V100
CPU-GPU interconnect	PCIe 3.0 switch	NVLink 2.0

We evaluated multiple compilers on both systems to assess the OpenMP offload performance of HPGMG. We compared performance against the original CUDA version and also an OpenACC version which was ported to the GPU in an identical way as the OpenMP offload version. The OpenACC version was included to provide additional performance results for directive-based GPU offload. The benefit of including OpenACC in our study is that the PGI compiler provides mature OpenACC support and is available on both systems. The full list of compilers is shown in Table 2.

The Cori-GPU MPI stack was always OpenMPI-4.0.3, except for the CCE compiler which was limited to using MPICH-3.3.2. The OpenMPI library was built with UCX support enabling CUDA-aware MPI communication, but not GPUDirect support which would have enabled direct peer-to-peer data transfers between GPUs. The Summit MPI stack was always IBM Spectrum MPI 10.3.1.2-20200121. Our Summit job launch scripts always specified `--smpiargs="-gpu"`

Table 2. Compilers and GPU offload methods evaluated on the Cori-GPU and Summit systems.

Compiler	GPU offload	Cori-GPU version	Summit version
GCC + NVCC	CUDA	7.3.0 + 10.1.243	7.4.0 + 10.1.243
NVIDIA/PGI	OpenACC	20.4	20.1
Cray CCE	OpenMP	9.1.0 (LLVM version)	–
IBM XL	OpenMP	–	16.1.1-5
LLVM/Clang	OpenMP	11.0.0-git (#17d8334)	11.0.0-git (#17d8334)

to enable the use of CUDA-aware MPI with GPUDirect. We used the NVIDIA nvprof profiler on both platforms to measure the time spent in GPU kernels and data movement operations between CPU and GPU. nvprof was active for all results shown in this study. We used the ECP-funded HPCToolkit profiler [2] on Cori-GPU to identify bottlenecks in the LLVM/Clang OpenMP runtime.

4.2 Application Configuration

We configured the HPGMG benchmark to use the out-of-place Gauss Seidel Red Black (GSRB) smoother, 4th order boundary conditions, and a Multigrid F-Cycle. A single run executes 3 different problem sizes with a grid spacing of h, 2 h, and 4 h. In this work we only consider the performance of the largest problem: that is the problem with a grid spacing of h. Our chosen problem has a grid spacing of $h = \frac{1}{512}$, a maximum box size of 32^3, and is executed between 3 and 100 steps depending on the throughput of the benchmark for each compiler. This problem has a memory footprint of approximately 38 GB and thus exceeds the memory capacity of the 16 GB GPUs in both our test platforms. In our tests we choose to use a single CPU socket and optionally 3 GPUs. The CPU-only configurations are executed with 1 MPI rank per core, and the CPU+GPU configurations are executed with 1 MPI rank per GPU.

5 Performance Evaluation

5.1 Performance When Using Managed Memory

Figure 3 shows the performance of the managed memory versions of HPGMG. The GPU versions of HPGMG generally performed better on Summit than Cori-GPU because of higher data transfer bandwidth between CPU and GPU (NVLink-2.0 versus PCIe 3.0), fewer GPU page faults, and less data movement between CPU and GPU. The system-level reasons for the differences are out of scope for this paper. Our performance evaluation will thus only compare compilers on the same system and not between systems. Figure 3a shows that the CCE and LLVM/Clang OpenMP compilers were not competitive with CUDA on Cori-GPU. CCE performed poorly because the code needed to be compiled at -O0 to workaround a compiler bug [10] (upstream issue at [19]). LLVM/Clang performed poorly because of significant time spent in `cuMemAlloc` and `cuMemFree` functions which are used to allocate and free device memory. These functions were called when mapping the `level` structure variable to the device. Figure 3b shows that the XL OpenMP compiler achieved 0.70x of CUDA performance on Summit. The PGI OpenACC compiler achieved 0.76x and 0.89x of CUDA performance on Cori-GPU and Summit, respectively, indicating that directive-based programming can deliver performance competitive with CUDA.

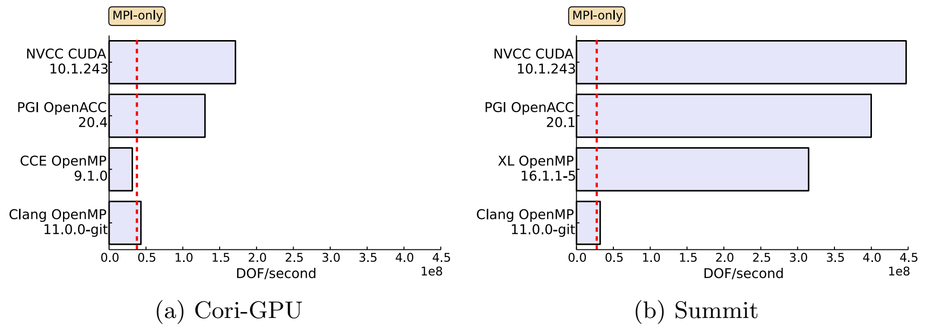

Fig. 3. HPGMG throughput for the managed memory version of HPGMG on Cori-GPU and Summit (higher is better). All configurations used 1 CPU socket and 3 GPUs. The dashed line shows the best MPI-only performance out of all available compilers when using 1 CPU socket.

5.2 Performance When Using Explicit Data Management

The performance results for the explicit data management version of HPGMG are limited because of various compiler issues: the XL compiler failed to correctly create the HPGMG device data structures [28] and the CCE 9.1.0 compiler does not support the OpenMP-5.0 pointer attachment rules required by HPGMG (the recently released CCE-10.0.0 compiler should provide this capability [9]). Henceforth, all performance results are obtained using the LLVM/Clang compiler. The initial results were disappointing compared to the OpenMP managed memory version: 12.0x slower on Cori-GPU and 5.7x slower on Summit.

The HPCToolkit profiler showed that most of the runtime was spent executing a `target update` construct used to copy data from GPU to CPU. The bottleneck was not data movement but instead time spent in a library function provided by `libstdc++` named `std::_Rb_tree_increment`. Our hypothesis is that this function is used by the LLVM/Clang OpenMP runtime to find out which host pointer corresponds to which device pointer before copying data between memories. OpenMP runtimes maintain an association between host and device pointers in a present table; it is expected to be efficient even when it contains many entries [8,14,36]. We describe our optimizations to workaround this LLVM/Clang bottleneck and other bottlenecks below. The impact of the individual optimizations are shown in Fig. 4 and explained below.

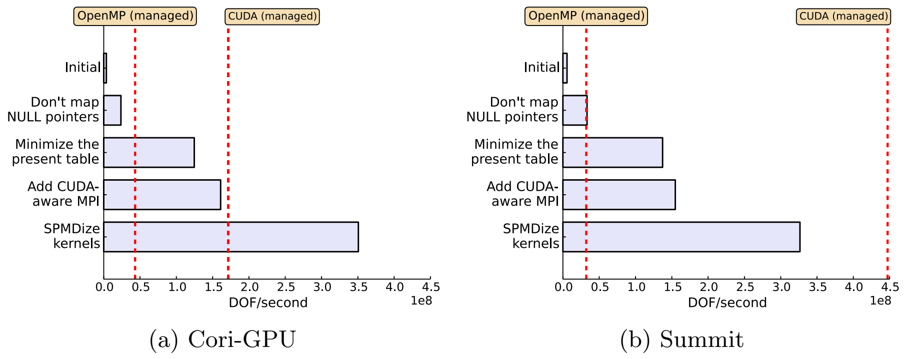

Fig. 4. HPGMG throughput for the explicit data management version of HPGMG on Cori-GPU and Summit using the LLVM/Clang compiler (higher is better). All configurations used 1 CPU socket and 3 GPUs. Optimizations apply additively, e.g. the code changes associated with the 4th bar down includes the code changes associated with the 2nd and 3rd bars. The dashed lines show the performance of the managed memory version of HPGMG when using OpenMP target offload and CUDA (values obtained from Fig. 3).

Don't Map NULL Pointers: Many HPGMG block pointers point to NULL for the duration of the application. These pointer addresses can safely be kept out of the OpenMP runtime present table. We added an `if` statement around the `target enter data` directive in Fig. 2 to only map data when it is not NULL. This improved the solve performance by 6.5x on Cori-GPU. It also reduced the initialization time from 392 s to 40 s on Cori-GPU. This is primarily because of less exclusive time in the OpenMP runtime but also because of significantly fewer CUDA HtoD memcpy transfers of 8 bytes. Here, an 8 byte transfer corresponds to setting a pointer in the device environment to a new device address.

Minimize the Present Table: The HPGMG block pointers are simply a convenience in the device kernels. Therefore, there is no need for the OpenMP runtime to maintain an association between block pointer host and device addresses. This is because all data transfers between CPU and GPU involve the larger `level` data buffers pointed to by the block pointers. We avoid creating an association by manually updating the device pointers in the device environment instead of using the `[:0]` syntax shown earlier (Fig. 2). Our function to do this is named `omp_attach` and is shown in Fig. 5. It performs the same task as the OpenACC runtime API function named `acc_attach`. This improved the performance by 5.3x on Cori-GPU and reduced initialization time to 9.3 s. The nvprof profiler shows that we called the OpenMP target region in `omp_attach` 103,644 times.

This implies that LLVM/Clang present table lookup time slows down significantly when the present table has O(100K) entries.

Attaching a device address to a device pointer - alternate method

```
void omp_attach(void **ptr)
{
  void *dptr = *ptr;
  if (dptr) {
#pragma omp target data use_device_ptr(dptr)
    {
#pragma omp target is_device_ptr(dptr)
      {
        *ptr = dptr;
      }
    }
  }
}
omp_attach((void**)&level->exchange_ghosts[shape].
                              blocks[block][b].read.ptr);
```

Fig. 5. The function `omp_attach()` attaches a device address to a device pointer without creating an entry in the present table of the OpenMP runtime. The function assumes it is passed the address of a pointer variable which is pointing to the host address of a mapped variable. We use the `use_device_ptr` clause to obtain the device address of the mapped variable. We then use an OpenMP target region to set the device pointer to the device address of the mapped variable.

Add CUDA-Aware MPI: The expensive `target update` code path can be avoided by exchanging GPU data between processes using CUDA-aware MPI communication. CUDA-aware MPI simplifies the OpenMP source code because it only involves adding a `target data` region with a `use_device_ptr` clause to pass the device address of a data buffer to a MPI communication call. The use of CUDA-aware MPI improved performance by 1.3x on Cori-GPU. However, CUDA-aware MPI is a capability that is not available in all MPI libraries.

SPMDize Kernels: The LLVM/Clang OpenMP compiler is known to perform poorly when there is user code in between `target` and `parallel` OpenMP directives [33]. As mentioned in Sect. 3.2, this code pattern happens frequently in HPGMG. It is possible to use the faster LLVM/Clang "SPMD" code generation scheme by creating all parallelism upfront to ensure that all threads execute the same code. It is impractical for us to use a combined `teams distribute parallel for` construct in the HPGMG OpenMP target regions because this would omit worksharing of the fine-grained loop over threads. Therefore we used strictly nested `teams` and `parallel` constructs with a manual distribution of the coarse-grained loop over teams based on the team ID. This was done because the OpenMP specification does not provide a combined `teams distribute parallel` construct and specifies that the `distribute` construct must be strictly nested inside a teams region. The SPMD code transformation improved LLVM/Clang performance by 2.2x on Cori-GPU. There was no benefit to the XL compiler because this compiler already implements interprocedural static compiler analysis to determine when all threads execute the same code [34].

6 Discussion

In this section we discuss whether the abstractions provided by the OpenMP specification were sufficient for our coding exercise as well as OpenMP compiler maturity and performance.

6.1 Assessment of OpenMP Abstractions

The directives and runtime API functions provided by the OpenMP specification enabled us to translate CUDA kernels into OpenMP target offload regions. They also enabled us to successfully implement explicit data management in a code that uses nested data structures with many pointer fields. We are concerned that only a small minority of users will be able to explicitly manage data movement in CPU/GPU systems in similarly complicated codes, however, this is no fault of the OpenMP specification. The barrier to entry is significantly lowered by relying on managed memory. We demonstrated that today's compilers correctly interoperate with CUDA managed memory and we are looking forward to compilers eventually supporting `requires unified_shared_memory` OpenMP directive to eliminate the need to mix OpenMP with lower-level non-portable APIs.

The only abstractions that could benefit similar coding efforts are related to performance. We found that a manual implementation of `acc_attach` in OpenMP enabled us to create a complicated data structure on the device in less time and assisted the LLVM/Clang OpenMP runtime to more quickly find the association between host and device addresses. However, this API function was only necessary because of a significant bottleneck in the LLVM/Clang present table implementation. We found that a manual implementation of a `teams distribute parallel` combined construct enabled us to use a faster LLVM/Clang code generation scheme for most of the HPGMG functions containing OpenMP target offload, however, it was detrimental to the performance of XL generated compute kernels.

6.2 Assessment of Compiler Maturity and Performance

We encountered issues with XL and CCE compilers which limited the OpenMP target offload results in Sect. 5. The only compiler which successfully compiled and executed the explicit data management version of HPGMG was LLVM/Clang (versions prior to LLVM/Clang-11.0.0 also had issues [19]). We found that the XL-compiled managed memory version of HPGMG achieved 0.70x of HPGMG-CUDA performance. This is encouraging because we made no specific optimizations to achieve high performance with the managed memory version of the code. The LLVM/Clang compiler performed poorly with the managed memory code version and abysmally with the initial explicit data management code version on both computing platforms.

We optimized the initial explicit data management code to achieve higher performance with the LLVM/Clang compiler: 2.04x of HPGMG-CUDA performance on Cori-GPU and 0.73x of HPGMG-CUDA performance on Summit. It should be mentioned that this is not the fairest of comparisons because HPGMG-CUDA does not include explicit data management to enable efficient bulk data transfers between CPU and GPU. One concern we have about our optimizations for LLVM/Clang are that they are unintuitive to the average OpenMP programmer and should instead be performed by a tuned OpenMP compiler and runtime. The runtime overheads in LLVM/Clang included excessive time spent in device memory management functions and a slow present table implementation (we have reported this issue [20]). Neither the CCE or XL compiler use device memory management functions as frequently as the LLVM/Clang compiler. We have not been able to test whether the same present table bottleneck exists in the CCE or XL compilers yet. The LLVM/Clang compiler also generated relatively slow device code without our manual SPMD code transformation. This hopefully will not be needed for much longer, since a prototype exists to use the faster code generation scheme in LLVM/Clang [11].

7 Conclusion

This paper describes how we ported HPGMG from CUDA to OpenMP target offload, added explicit data management, measured performance with multiple OpenMP compiler and runtimes on two different node architectures, and finally optimized HPGMG performance when using LLVM/Clang. Our work shows that OpenMP target offload compiler and runtimes still need to fix compiler bugs, implement more complete OpenMP 5.0 feature support, efficiently compile a broader range of application usage of OpenMP directives, and fix overheads in OpenMP runtimes. However, there were positive performance results compared to HPGMG-CUDA (managed memory CUDA implementation of HPGMG): the XL-compiled managed memory version of HPGMG achieved 0.70x of HPGMG-CUDA performance on Summit, and the LLVM/Clang-compiled explicit data management version of HPGMG achieved 2.04x of HPGMG-CUDA performance on Cori-GPU and 0.73x of HPGMG-CUDA performance on Summit.

Acknowledgments. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. The authors would like to thank Mat Colgrove of NVIDIA for the initial development of an explicit data management version of HPGMG using OpenACC. The authors would also like to thank Brian Friesen of LBNL for installing CUDA-aware versions of OpenMPI for several different compiler stacks.

References

1. Adams, M., Brown, J., Shalf, J., Van Straalen, B., Strohmaier, E., Williams, S.: HPGMG (2020). https://bitbucket.org/hpgmg/hpgmg
2. Adhianto, L., et al.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
3. Almgren, A.S., Bell, J.B., Lijewski, M.J., Lukić, Z., Van Andel, E.: Nyx: a massively parallel AMR code for computational cosmology. Astrophys. J. **765**, 39 (2013). https://doi.org/10.1088/0004-637X/765/1/39
4. Beckingsale, D.A., et al.: RAJA: portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81, November 2019. https://doi.org/10.1109/P3HPC49587.2019.00012
5. Bercea, G.T., Bataev, A., Eichenberger, A.E., Bertolli, C., O'Brien, J.K.: An open-source solution to performance portability for Summit and Sierra supercomputers. IBM J. Res. Dev. **64**(3/4), 12:1–12:23 (2020)
6. Bercea, G.T., et al.: 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 2015. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2832087.2832089
7. Budiardja, R.D., Cardall, C.Y.: Targeting GPUs with OpenMP directives on summit: a simple and effective Fortran experience. Parallel Comput. **88**, 102544 (2019)
8. Colgrove, M., Wolfe, M.: Personal Communication, May 2020
9. Crayport: Case 247291 - Cray CCE-9.0.0 has OpenMP offload bugs when mapping structs (2020). https://portal.cray.com
10. Crayport: Case 256571 - Test program must be compiled at -O0 when using CCE/9.1.0 (2020). https://portal.cray.com
11. Doerfert, J., Diaz, J.M.M., Finkel, H.: The TRegion interface and compiler optimizations for OpenMP target regions. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 153–167. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_11
12. Grinberg, L., Bertolli, C., Haque, R.: Hands on with OpenMP4.5 and unified memory: developing applications for IBM's Hybrid CPU + GPU systems (part I). In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 3–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_1
13. Grinberg, L., Bertolli, C., Haque, R.: Hands on with OpenMP4.5 and unified memory: developing applications for IBM's hybrid CPU + GPU systems (part II). In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 17–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_2
14. Hart, A.: First experiences porting a parallel application to a hybrid supercomputer with OpenMP4.0 device constructs. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 73–85. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_6
15. Hayashi, A., Shirako, J., Tiotto, E., Ho, R., Sarkar, V.: Performance evaluation of OpenMP's target construct on GPUS - exploring compiler optimisations. Int. J. High Perform. Comput. Network. **13**(1), 54–69 (2019). https://doi.org/10.1504/IJHPCN.2019.097051

16. Juckeland, G., et al.: From describing to prescribing parallelism: translating the SPEC ACCEL OpenACC suite to OpenMP target directives. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 470–488. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_33
17. Karlin, I., et al.: Early experiences porting three applications to OpenMP 4.5. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 281–292. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_20
18. Vergara Larrea, V.G., et al.: Scaling the summit: deploying the world's fastest supercomputer. In: Weiland, M., Juckeland, G., Alam, S., Jagode, H. (eds.) ISC High Performance 2019. LNCS, vol. 11887, pp. 330–351. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34356-9_26
19. LLVM Bugzilla: Bug 44390 - Incorrect OpenMP target offload code at > -O0 optimization (2020). https://bugs.llvm.org
20. LLVM Bugzilla: Bug 46107 - Poor present table performance (2020). https://bugs.llvm.org
21. Martineau, M., McIntosh-Smith, S., Gaudin, W.: Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 338–347 (2016)
22. Martineau, M., McIntosh-Smith, S.: The productivity, portability and performance of OpenMP 4.5 for scientific applications targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 185–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_13
23. Martineau, M., et al.: Performance analysis and optimization of Clang's OpenMP 4.5 GPU support. In: Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, PMBS 2016, pp. 54–64. IEEE Press (2016)
24. Martineau, M., Price, J., McIntosh-Smith, S., Gaudin, W.: Pragmatic performance portability with OpenMP 4.x. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 253–267. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_18
25. Mishra, A., Li, L., Kong, M., Finkel, H., Chapman, B.: Benchmarking and evaluating unified memory for OpenMP GPU offloading. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC. LLVM-HPC 2017. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3148173.3148184
26. Monsalve Diaz, J.M., Friedline, K., Pophale, S., Hernandez, O., Bernholdt, D., Chandrasekaran, S.: Analysis of OpenMP 4.5 offloading in implementations: correctness and overhead. Parallel Comput. **89**, 102546 (2019). https://doi.org/10.1016/j.parco.2019.102546
27. NERSC: Cori GPU Nodes (2020). https://docs-dev.nersc.gov/cgpu/
28. OLCF Support: IBM ticket TS003552272 - IBM compiler OpenMP target offload data management bug (2020)
29. OpenMP Architecture Review Board: OpenMP application programming interface version 5.0, November 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

30. Pennycook, S.J., Sewall, J.D., Hammond, J.R.: Evaluating the impact of proposed OpenMP 5.0 features on performance, portability and productivity. In: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 37–46 (2018)
31. Rabbi, F., Daley, C.S., Aktulga, H.M., Wright, N.J.: Evaluation of directive-based GPU programming models on a block eigensolver with consideration of large sparse matrices. In: Wienke, S., Bhalachandra, S. (eds.) WACCPD 2019. LNCS, vol. 12017, pp. 66–88. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49943-3_4
32. Sakharnykh, N., Wang, P., Williams, S.: HPGMG-CUDA (2020). https://bitbucket.org/nsakharnykh/hpgmg-cuda
33. The Clang Team: Clang 11 Documentation, OpenMP Support (2020). https://clang.llvm.org/docs/OpenMPSupport.html
34. Tiotto, E., Mahjour, B., Tsang, W., Xue, X., Islam, T., Chen, W.: OpenMP 4.5 compiler optimization for GPU offloading. IBM J. Res. Dev. **64**(3/4), 14:1–14:11 (2020)
35. Vergara Larrea, V.G., Budiardja, R.D., Gayatri, R., Daley, C., Hernandez, O., Joubert, W.: Experiences in porting mini-applications to OpenACC and OpenMP on heterogeneous systems. Concurr. Comput.: Pract. Exp. e5780 (2020). https://doi.org/10.1002/cpe.5780. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5780. [Published online ahead of print (24 April 2020)]
36. Wolfe, M., Lee, S., Kim, J., Tian, X., Xu, R., Chandrasekaran, S., Chapman, B.: Implementing the OpenACC data model. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 662–672, May 2017. https://doi.org/10.1109/IPDPSW.2017.85