Towards Efficient Remote OpenMP Offloading

Wenbin Lu[1], Baodi Shan[1], Eric Raut[1], Jie Meng[2], Mauricio Araya-Polo[2], Johannes Doerfert[3], Abid M. Malik[4], and Barbara Chapman[1,4]

[1] Stony Brook University, Stony Brook NY 11794, USA
`{wenbin.lu,baodi.shan,eric.raut,barbara.chapman}@stonybrook.edu`
[2] TotalEnergies EP R&T, Houston TX 77002, USA
[3] Argonne National Laboratory, Lemont, IL 60439, USA
`jdoerfert@anl.gov`
[4] Brookhaven National Laboratory, Upton NY 11793, USA
`amalik@bnl.gov`

Abstract. On modern heterogeneous HPC systems, the most popular way to realize distributed computation is the hybrid programming model of MPI+X (X being OpenMP/CUDA/etc.), as it has been proven to perform well with various scientific applications. However, application developers prefer to use a single coherent programming model over a hybrid model, as maintainability and portability decrease per additional model. Recent work [14] has shown that the OpenMP device offloading model could be used to program distributed accelerator-based HPC systems with minimal changes to the application.
In this paper, we improve the performance of OpenMP remote offloading through various runtime optimizations, guided by a detailed overhead analysis. Evaluation of our work is conducted using an industrial-level seismic modeling code, *Minimod*, as well as two proxy-apps, *XSBench* and *RSBench*. Results show that, compared to the baseline version, our optimizations can reduce offloading latencies by up to 92%, and raise application parallel efficiency by at least 25.2% when running with 16 GPUs. We then point out why strong scaling is still difficult with OpenMP remote offloading, and propose further improvements to the runtime to increase scalability.

Keywords: OpenMP · GPGPU · distributed computing

1 Introduction

As we move towards extreme heterogeneity, it is increasingly important to utilize HPC accelerators like GPUs efficiently in the distributed setting. Also, the great variety in accelerator software/hardware makes the portability and maintainability of applications as important as reducing the time to solution.

As many of the distributed programming models were designed with only the CPUs as their primary processing elements, users often have to add additional layers of domain decomposition and use vendor-specific APIs to take advantage of the accelerators. For code initially written in intra-node programming models, they often require even more effort to port to a hybrid programming model that can run across multiple nodes. This increases the development burden and poses great portability challenges.

OpenMP [13] is the *de facto* HPC programming model for shared-memory parallelism. Version 4.0 of OpenMP introduced device offloading to execute code on accelerators, without the user having to write device kernels in vendor-specific APIs. Recent work by Patel and Doerfert [14] has shown that through extensions in the LLVM/OpenMP runtime system, specification-conforming OpenMP offloading applications can seamlessly utilize accelerators attached to remote compute nodes. This shows the potential for transforming OpenMP to an all-encompassing programming model for writing performance portable and maintainable scientific application in the era of heterogeneous supercomputing, as an alternative to hybrid programming models like MPI+X.

In this paper, we optimize the previous work and push OpenMP remote offloading performance further. We then point out limitations that are deeply rooted in the design of the LLVM/OpenMP runtime, in the hope to stir up discussions about remote offloading in the OpenMP community, and inspire future OpenMP runtime designs. Our main contributions are the following:

– A detailed overhead analysis of the OpenMP remote offloading plugin.
– A series of runtime optimizations that significantly reduce the overhead of the remote offloading process.
– An evaluation of the optimized plugin using proxy-apps of real-world HPC applications.
– A discussion of the limitations of the plugin, as well as proposals for further improvements.

The paper is organized as follows: section 2 describes LLVM/OpenMP remote offloading and the related work, section 3 identifies performance issues and present the corresponding optimizations, evaluation of our work is described in section 4, and finally section 5 concludes the work and talk about future directions.

2 Background

In this section, we describe the details of how OpenMP remote offloading works in LLVM, as well as related work about the technologies and cases used later in the experimental section.

2.1 OpenMP Offloading in LLVM/OpenMP

An OpenMP application's offloading directives are lowered to functions calls into the Host Runtime (`libomptarget.so`), which is the entry point of all offloading operations in the LLVM/OpenMP offloading workflow shown in Figure 1. The Host Runtime is target-agnostic, and it loads the Device Runtime plugin

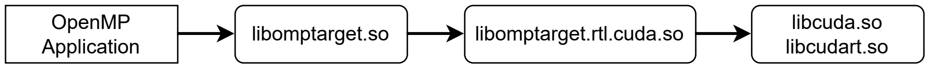

Fig. 1. LLVM/OpenMP device offloading workflow for CUDA devices

according to the type of device code embedded in the fat application binary. For example, it can load the CUDA plugin `libomptarget.rtl.cuda.so`, which makes CUDA calls to transfer data and launch kernels.

The Host Runtime talks to the Device Runtime through the Device Plugin Interface, which is a small set of target-agnostic C functions, to perform offloading operations (e.g. `__tgt_rtl_data_alloc()`). All Device Runtime plugins must implement this interface, hiding low-level details from the Host Runtime.

2.2 The Remote Offloading Plugin

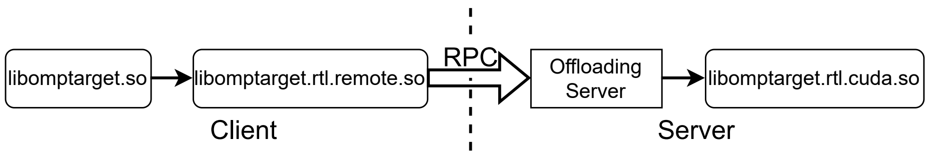

Fig. 2. LLVM/OpenMP remote offloading workflow for CUDA devices

The idea of the remote offloading plugin is to use remote procedure calls (RPC) to relay the plugin interface calls to remote processes, so the application can access devices on other machines transparently. This plugin implements the core set of the Device Plugin Interface to talk to the Host Runtime. OpenMP application are compiled as usual and no code modification is required except for doing multi-device offloading using the `device(n)` clause. The plugin is also compatible with OpenMP asynchronous offloading (e.g. the `nowait` clause), thanks to the orthogonal design of the concurrent offloading mechanism in LLVM/OpenMP [24].

In Figure 2, the Client stands for the OpenMP application, while the Server is a binary provided by the remote offloading plugin. The user runs one instance of the Client, and one instance of the Server per GPU node. Once connected, the Client can offload to all the GPUs managed by all the Servers. Both the Host Runtime and the Device Runtime plugin are working as they normally would, unaware of the Client-Server pair in the middle.

For each plugin interface call, the Client serializes the function arguments and sends one blocking RPC request to the Server. The server will deserialize incoming requests and execute the corresponding device operation, then reply to the Client, so it can proceed. For a `map(to/from)` operation, the buffer must also be serialized. The Server uses a task queue and a pool of threads to handle the RPC requests, so that offloading activities on one GPU does not block activities on other GPUs managed by the same Server.

Originally, the remote offloading plugin uses gRPC [5] as its RPC backend, since it is the natural choice. Then an UCX [22] backend was added since it can utilize high-performance interconnects like Infiniband, thus bringing the potential to improve the offloading application's performance to a level comparable to that of its MPI+X equivalent.

2.3 Related Work

OpenMP single-node multi-GPU offloading has been previously explored [7] [28], but standard OpenMP does not go beyond the node boundary and the most popular approach to program distributed-memory GPUs is still the hybrid model of MPI+X, X being a local GPU programming model such as CUDA or OpenMP. An alternative that has received some attention is to replace MPI with a task-based programming model that also interoperates with CUDA; among them are Charm++ [1], UPC++ [2], Legion [3], and Chapel [4]. NVSHMEM [6] enables remote communication directly from the CUDA kernels. Intel Cluster OpenMP uses a Distributed Shared Memory runtime system to run OpenMP CPU parallel regions across nodes [23]. Kokkos Remote Spaces [8] is an extension to the Kokkos programming model [27] to support distributed shared memory for programming GPUs and other devices. rCUDA [19] is a framework for remote GPU virtualization, in which a set of GPUs can be shared and remotely accessed by several clients simultaneously.

Minimod [10] is one of the applications used in this study. Implementations of Minimod have been evaluated with OpenMP tasks [18], and in distributed setups using the Legion programming model targeting CPUs [17] and GPUs [16,21]. The present paper evaluates a version of Minimod using OpenMP target regions wrapped in tasks to make use of multiple GPUs simultaneously.

3 Performance Analysis and Optimizations

This work focuses on the UCX backend of the remote offloading plugin for its performance potentials. Additionally, the devices are running the exact same kernel, so scalability issues mostly arise from runtime and communication overheads.

3.1 Runtime Optimizations

To identify the performance issues of the remote offloading process, we profiled a microbenchmark that only does host-device transfers of fixed-size buffers, using the `map` clause. The selected results of running on a single remote GPU are shown in Figure 3. Clearly, the remote offloading plugin's internal overhead dominates the communication latency for all message sizes. This is a direct result of the RPC

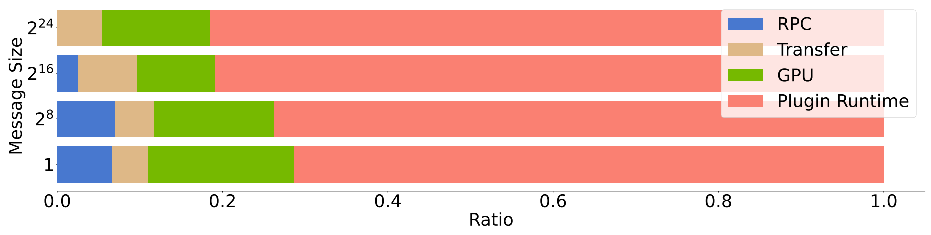

Fig. 3. Breakdown of remote `map(from)` latency

mechanism: the layers of abstractions to handle different offloading operations and communication backends, communication progression/completion tracking, data (de)serialization, and other bookkeeping operations. Since the plugin uses blocking RPCs, all these overheads are translated to `map` latencies, which lead to longer idle periods on the device.

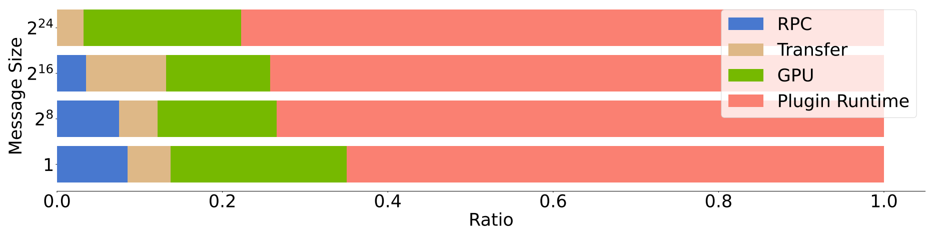

Fig. 4. Breakdown of remote `map(from)` latency, with runtime optimizations

UCX provides active messages for inter-node RPC, but its API is too primitive to relay the plugin interface functions. The original remote offloading plugin implemented its own RPC mechanism based on UCX's message passing API. We first improve the plugin from a software engineering point of view: use suitable C++ features to reduce the overhead of the serializer and other internal abstractions, speedup UCX communication progression to reduce send-receive latency, etc. The latency breakdown after applying our runtime optimizations is shown in Figure 4. Although we have achieved around 6% overhead reduction for small buffers, the optimizations' effectiveness drops for larger ones.

3.2 CUDA-Aware Communication

The plugin's RPC serializer is a major source of overhead. Similar to MPI, UCX is mostly designed to send contiguous chunks of data between buffers on different processes. To perform RPC, which is a high-level operation, the Client must serialize function arguments and all the buffers involved before passing them to UCX, and the Server must do the reverse. Additionally, for `map(to/from)` the

Server needs a staging buffer to store a temporary copy of the mapped buffer, to pass the mapped data between the serializer and the device API.

To map a host buffer to the device using the `__tgt_rtl_datasubmit` plugin interface, a regular device plugins does a device allocation, an `HtoD memcpy` and a device synchronization call. But for the remote offloading plugin, the RPC serializer does multiple extra memory allocations and `memcpy`'s. These overheads are repeated on both the Client and the Server and grows linearly with the size of the buffer, which is why the runtime overhead accounts for such a large percentage of the total latency.

To further reduce the overhead, we utilize the UCX's CUDA support to send and receive data directly from the GPU, eliminating the need for a staging buffer on the Server and other serializer overhead. When this mechanism is active, a `map(to/from)` operation will be broken into two steps: an RPC request that only sends the metadata (device buffer address, buffer size, etc.), and a second UCX send/receive request that transfers data directly between the Client's host memory and the Server's device memory. The two-step approach has its own associated overhead: if the mapped buffer is too small, then it may be faster to serialize everything and use a single RPC request. Therefore, we introduce an environment variable to specify the smallest buffer size to activate the CUDA-aware UCX mechanism (`SPLIT_THRESH`). This threshold is affected by many factors and should be experimentally determined for different software and hardware combinations.

3.3 Thread Contention and NUMA-GPU Affinity

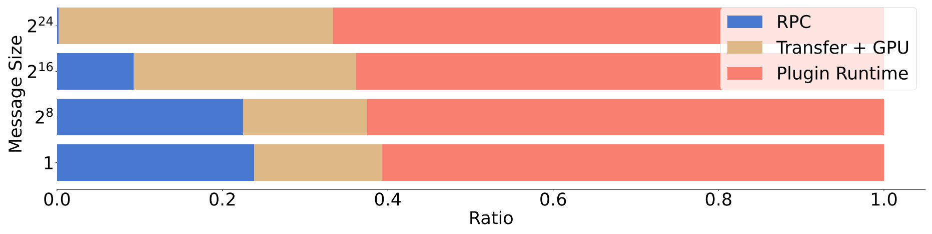

Fig. 5. Breakdown of remote `map(from)` latency, with all optimizations applied

There are several other factors that affect `map` latency. In UCX, the Worker object provides independent progression and completion of communication operations. This means that even if all traffic goes through the same network card, parallel injection from multiple Workers can still improve message throughput through overlapping send-receive operations in the higher levels of the runtime system, especially when the combined message flow is not large enough to saturate the hardware bandwidth. Originally, the Server uses a single Worker to serve all the GPUs it has access to. This means the Worker must use locks to prevent data races caused by concurrent access from different threads, essentially serializing many overlappable operations and reducing the injection rate [9]. In this work, we use one Server(Worker) per GPU, to reduce thread contention.

Also, it is important to make sure that the Server is running in the NUMA node that is the closest to the GPU it is offloading to. Modern heterogeneous systems tend to have more than one GPU per compute node, and different GPUs are local to different CPU sockets, if there is more than one socket. Additionally, HPC systems like ORNL Summit and LLNL Sierra split the PCIe lanes of the Infiniband network card evenly between the two sockets [29]. This means the application must drive communication from both NUMA nodes in a balanced fashion, to maximize communication performance. In our experiments, host-device transfers that cross the NUMA boundary can have up to 23% higher latencies than that of the transfers within the same NUMA node. In all our experiments, we pin the Server process to the NUMA node that the GPU is connected to, and UCX will pick up the closest network port.

Figure 5 shows the latency breakdown after applying the all optimizations mentioned in this section, with `SPLIT_THRESH` set to 2^{22} bytes. The results show that we have achieved at least 11% reduction in overheads when compared to the results of the original plugin in Figure 3, for all buffer sizes. Now, the plugin internal overhead is always below 70% of the remote data mapping latency. Note that the RPC overhead is a constant($5.328\mu s$), but its percentage in the total latency becomes over 20% for the smaller buffers, as a result of significantly reduced absolute latency.

4 Evaluation

We evaluate our optimizations using microbenchmarks and proxy-apps on the Cypress computing system at TotalEnergies R&T in Houston. Each Cypress node contains one AMD EPYC 7F52 16-core CPU, one Mellanox ConnectX-6 200 Gb/s Infiniband network card, and four NVIDIA A100 GPUs. The system runs CentOS 8 with Linux kernel 4.18.0, CUDA 11.5.119 and MOFED-5.1-2.5.8.0. We use UCX commit `5879c44` of the `v1.13x` branch, with the GPUDirect RDMA [12] and GDRCopy [11] transports enabled.

As the baseline, we use commit `6120be4` of the original remote offloading plugin, which is based on commit `67ab4c0` of the LLVM trunk. The remote offloading plugin with only runtime optimizations is referred to as Opt1; while plugin version Opt2 has all optimizations applied. Since we can eliminate unnecessary overhead by allowing the Client to offload to its local GPUs directly (instead of go through the remote plugin), we have included results that enabled Client-side offloading, which we refer to as Opt2L. All three proxy-apps are tested in small and large problem sizes.

4.1 Microbenchmarks

Figure 6 shows the remote GPU to/from mapping latencies of different buffer sizes. Compared to the baseline, the Opt2 version of the plugin reduces the

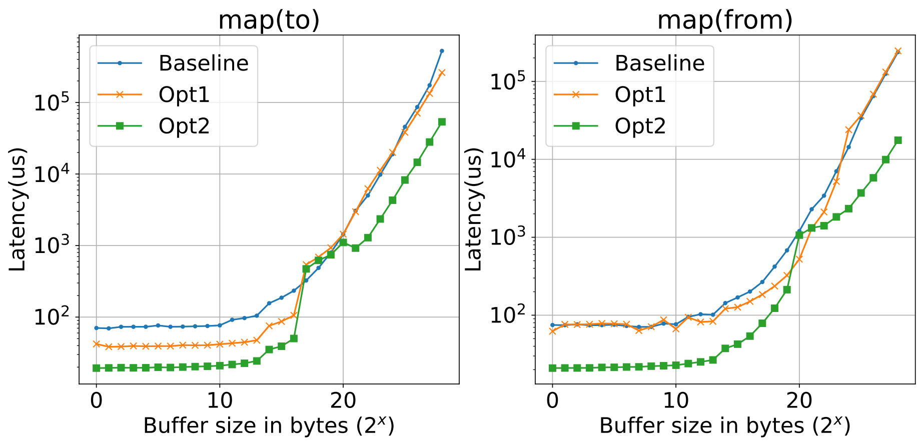

Fig. 6. Remote GPU `map(to/from)` latency

message latency by $\sim 72\%$ for small buffers, and $\sim 90\%$ for large ones. With Opt2's lower device buffer mapping latencies, the OpenMP application can launch `target` regions faster, thus obtaining better scalability.

Opt1 and Opt2 show little to no speedup for buffer size around 2^{19} bytes. This is caused by the compound effects of UCX switching its internal communication protocol and our `SPLIT_THRESH` setting for enabling CUDA-aware communication, since we used the same setting for all buffer sizes instead of the best settings for each size. For real applications, the user should adjust the `SPLIT_THRESH`, as well as UCX's protocol switching thresholds (zero-copy, rendezvous, etc.), so that the latencies of the most frequently mapped buffer sizes are optimal.

4.2 Weak Scaling - RSBench and XSBench

XSBench [26] and RSBench [25] are proxy-apps that capture the core computation of the Monte Carlo neutron transport code OpenMC [20], while XSBench is memory-bound and RSBench is compute-bound. We use the OpenMP offloading version of both proxy-apps, with the same modifications used in [14] to enable multi-device offloading. We run the proxy-apps on 4 to 16 GPUs, and normalized the run times with respect to the run time of using 4 local GPUs without the remote offloading plugin. We keep the amount of work and data transferred per GPU constant to evaluate the weak scalability of the plugin.

Kernels execution time, total host-device transfer size per kernel, and the total number of host-device transfers per kernel are listed in Table 1. The number of transfers per kernel launch is important since it is proportional to the number of RPC requests for device buffer allocation/free, and the actual data transfer. So XSBench has $19 \times 3 + 1 = 58$ RPC requests per kernel launch, while the number is $27 \times 3 + 1 = 82$ for RSBench.

Table 1. XSBench/RSBench kernel durations and per-kernel launch data transfers

	Kernel-Small	Kernel-Large	Transfer-Small	Transfer-Large	No. Transfers
XSBench	56.38 ms	271.6 ms	240.4 MB	5648 MB	19
RSBench	231.4 ms	1371 ms	5.325 MB	28.92 MB	27

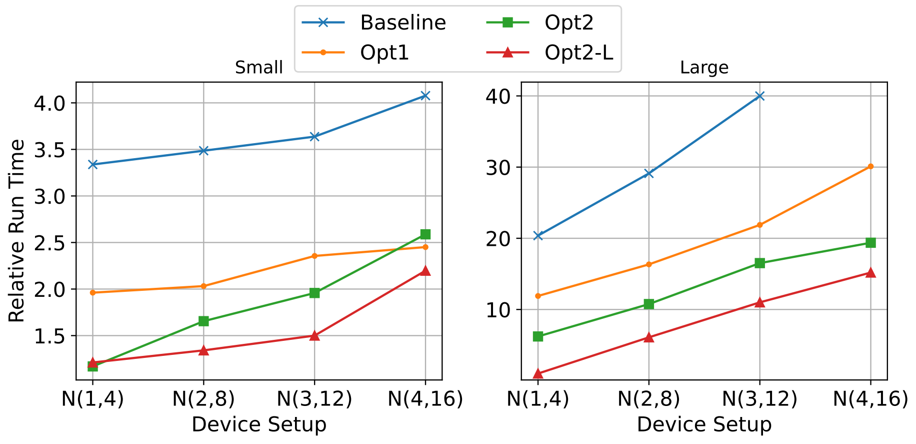

Fig. 7. Weak scaling results of XSBench

XSBench results are presented in Figure 7. For the horizontal axis, $N(n, 4n)$ stands for running the benchmark on n nodes and using all $4n$ GPUs. The baseline version running the large setup crashes since it exhausts all available host memory. Again, our optimized implementations are significantly faster than the baseline. For 16 GPUs, Opt2L increases the parallel efficiency by 25.2%. While we can achieve a 56% parallel efficiency on 16 GPUs for the small setup, only 6% was obtained on 16 GPUs for the large setup. This is because the XSBench-large transfers 5.5 GB of data per kernel launch, and all of them must go through the Client's network card without effective overlap. XSBench-large is therefore severely communication-bound and does not scale well.

RSBench results in Figure 8 shows the effectiveness of our optimizations for compute-bound applications with longer kernel execution times and smaller data transfers. On 16 GPUs, all three optimized versions obtained at least 66% parallel efficiency, while the baseline version goes as low as 34%.

4.3 Strong Scaling - Minimod

Minimod [10] is a proxy application that simulates the propagation of waves through subsurface models, by solving a finite difference discretized form of the wave equation. In this work, we use one of the kernels contained in Minimod: the acoustic isotropic propagator in a constant-density domain [15].

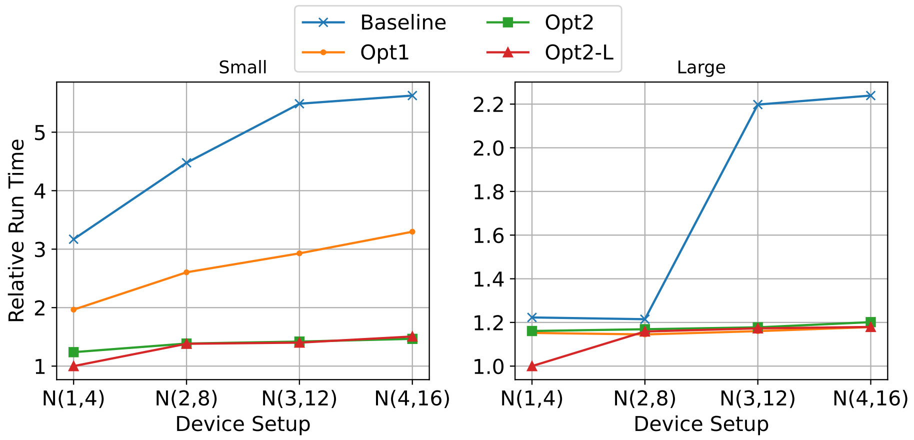

Fig. 8. Weak scaling results of RSBench

Table 2. Minimod total kernel durations and data transfers (per iteration)

Kernel-Small	Kernel-Large	Transfer-Small	Transfer-Large	No. Transfers
171.9 μs	6733 μs	182.2 KB	4032 KB	2

```
for (int g = 0; g < nGPUs; g++) {
  #pragma omp task depend(...)
  #pragma omp target teams distribute parallel for device(g)
  for (...)
    // Stencil computation
}
// Halo exchange
for (int g = 0; g < nGPUs; g++) {
    // Left halo region: DtoH
    #pragma omp task depend(...)
    #pragma omp target update from(...) device(g)
    // Left halo region: HtoD
    #pragma omp task depend(...)
    #pragma omp target update to(...) device(g-1)
    // Repeat for the right halo regions ...
}
```

Fig. 9. Simplified Minimod multi-GPU offloading and halo exchange workflow

Minimod natively supports multi-device OpenMP offloading using `target` regions wrapped in OpenMP tasks (see Figure 9), and is strong-scaling in nature [18]. The 3D grid used in Minimod is partitioned along the X-axis (i.e. sliced parallel to the YZ-plane), regardless of the number of devices it is running on. Therefore, the amount of halo data exchanged between the devices is only related to the size of the grid, since the area of the cross-section of the grid is always $dimY \times dimZ$. However, since the offloading Servers are only connected to the Client, not to each other, the halo data must all be relayed by the Client, creating a central communication bottleneck. Minimod's kernel durations and data transfer sizes for running on two devices are listed in Table 2.

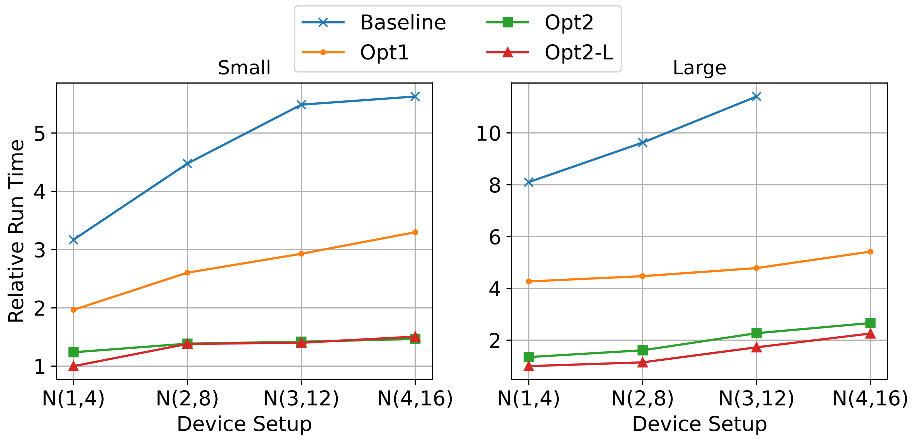

Fig. 10. Strong scaling results of Minimod

Figure 10 shows Minimod strong scaling results. Again, Opt2 and Opt2L outperforms other versions of the plugin significantly, showing the effectiveness of our optimizations. However, for all configurations, the run time increases as we use more GPUs. The reason is: as the number of device increases, the (already short) execution time of the kernel decreases linearly, but the halo exchange overhead grows linearly. Additionally, all halo exchange traffic must go through the Client, which leads to high communication contention. Since the communication overhead dominates the execution time, we see no performance improvement for all configurations.

4.4 Discussions

Our work has reduced the overhead of the LLVM/OpenMP remote offloading plugin by a large margin, and is especially effective for weak scaling of compute-bound applications. But as shown in Figure 5, plugin overhead still accounts for more than 60% of the communication latency, which is a road blocker for higher strong scalability. One solution is to implement the asynchronous Device Plugin Interface functions in the remote plugin, using UCX asynchronous communication APIs. We will need to replicate CUDA Stream functionalities to keep track of asynchronous events and handle dependencies, but this will hide and/or reduce the aforementioned runtime overheads. We could also implement message aggregation to reduce the total number of RPC requests, as many transfers listed in Table 1 are only sending a single scalar.

OpenMP's flat device model can be extended to expose the device topology to the users, so they can do hierarchical computation decomposition. Similar to an MPI shared-memory communicator, an OpenMP `node` construct can enumerate the `legion` of devices attached to the same machine. We could also do implicit hierarchical offloading by presenting all GPUs attached to the same NUMA node as a single device, and `map` buffers to CUDA Managed Memory.

The current design of the LLVM/OpenMP runtime can also be extended to push the scalability further. Currently, the Device Plugins know very little about the big picture and rely on the Host Runtime's prescriptive offloading instructions, creating the central bottleneck. We could increase the autonomy of the Device Plugins and give descriptive orders whenever possible. A partitioned global address space model can also be introduced to support direct inter-node device-to-device transfers.

Lastly, the single-Client multi-Server architecture must be replaced with a more SPMD-like one, for the remote offloading plugin to work for a wider spectrum of applications. Then current centralized approach not only creates a communication bottleneck, but also limits the amount of host memory available to the application to be the amount of memory installed on the Client's node. One specification-breaking solution is to encourage the users to allocate all host buffers that will be interacting with the device i inside a `target data device(i)` construct. Then in run-time we "offload" the entire `target data` region to device i's node, so that the host code inside that region also runs on the remote node and can utilize its main memory. Alternatively, we could use a page migration-based mechanism to transparently extend the amount of host memory, similar to Intel Cluster OpenMP [23].

5 Conclusions and Future Work

Remote OpenMP device offloading is a promising alternative to MPI+X, as it improves the portability and maintainability of the application by covering both inter-node and intra-node computation in a single programming model. In this work, we analyzed the performance bottlenecks of the LLVM/OpenMP remote offloading plugin, and have identified the RPC serializer and the buffered communication mechanism as two major sources of overhead. We then applied optimizations that reduce the plugin's internal overhead, and enabled CUDA-aware UCX communications to accelerate the transportation of large buffers. Evaluation using microbenchmarks shows that our optimizations have reduced `map` latencies by up to 92%. With lower data transfer latencies, we have achieved a minimum of 25.2% increase of parallel efficiency in proxy-apps running on 16 GPUs. Our optimizations are especially effective for weak scaling proxy-apps that have relatively long-running kernels and small data transfers.

However, weak scaling with large data transfers and strong scaling still prove to be challenging despite our optimizations. We propose a few runtime modifications to further reduce the plugin's latency and improve its scalability, but ultimately we believe extending the OpenMP runtime design is required to reach performance comparable to MPI+X.

For future work, we can implement the asynchronous plugin interface to hide plugin internal overhead, add an MPI backend to further improve the plugin's portability, and extend the offloading Server to support direct Server-to-Server transfers.

Acknowledgements

We would like to thank TotalEnergies EP Research and Technologies for their support of this work. This research was supported in part 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, in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE).

This research was also funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory, operated by Triad National Security, LLC under Contract No. 89233218CNA000001.

References

1. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., Kale, L.: Parallel programming with migratable objects: Charm++ in practice. In: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 647–658 (2014). https://doi.org/10.1109/SC.2014.58
2. Bachan, J., Baden, S.B., Hofmeyr, S., Jacquelin, M., Kamil, A., Bonachea, D., Hargrove, P.H., Ahmed, H.: Upc++: A high-performance communication framework for asynchronous computation. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 963–973 (2019). https://doi.org/10.1109/IPDPS.2019.00104
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (Nov 2012). https://doi.org/10.1109/SC.2012.71
4. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications **21**(3), 291–312 (2007). https://doi.org/10.1177/1094342007078442
5. gRPC community: grpc. https://grpc.io/about/

6. Hsu, C.H., Imam, N., Langer, A., Potluri, S., Newburn, C.J.: An initial assessment of nvshmem for high performance computing. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 1–10 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00104
7. Kale, V., Lu, W., Curtis, A., Malik, A.M., Chapman, B., Hernandez, O.: Toward supporting multi-gpu targets via taskloop and user-defined schedules. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) OpenMP: Portable Multi-Level Parallelism on Modern Systems. pp. 295–309. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_19
8. Kokkos: Kokkos remote spaces, https://github.com/kokkos/kokkos-remote-spaces
9. Lu, W., Curtis, T., Chapman, B.: Enabling low-overhead communication in multi-threaded openshmem applications using contexts. In: 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM). pp. 47–57 (2019). https://doi.org/10.1109/PAW-ATM49560.2019.00010
10. Meng, J., Atle, A., Calandra, H., Araya-Polo, M.: Minimod: A finite difference solver for seismic modeling. arXiv (2020), https://arxiv.org/abs/2007.06048
11. NVIDIA: Gdrcopy. https://github.com/NVIDIA/gdrcopy
12. NVIDIA: Nvidia cuda gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html
13. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Nov 2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, version 5.0
14. Patel, A., Doerfert, J.: Remote openmp offloading. In: Varbanescu, A.L., Bhatele, A., Luszczek, P., Marc, B. (eds.) High Performance Computing. pp. 315–333. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-07312-0_16
15. Qawasmeh, A., Hugues, M.R., Calandra, H., Chapman, B.M.: Performance portability in reverse time migration and seismic modelling via openacc. The International Journal of High Performance Computing Applications **31**(5), 422–440 (2017). https://doi.org/10.1177/1094342016675678
16. Raut, E., Anderson, J., Araya-Polo, M., Meng, J.: Evaluation of distributed tasks in stencil-based application on gpus. In: 2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2). pp. 45–52 (2021). https://doi.org/10.1109/ESPM254806.2021.00011
17. Raut, E., Anderson, J., Araya-Polo, M., Meng, J.: Porting and evaluation of a distributed task-driven stencil-based application. In: Proceedings of the 12th International Workshop on Programming Models and Applications for Multicores and Manycores. p. 21–30. PMAM'21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3448290.3448559
18. Raut, E., Meng, J., Araya-Polo, M., Chapman, B.: Evaluating performance of openmp tasks in a seismic stencil application. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) OpenMP: Portable Multi-Level Parallelism on Modern Systems. pp. 67–81. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_5
19. Reaño, C., Silla, F., Shainer, G., Schultz, S.: Local and remote gpus perform similar with edr 100g infiniband. In: Proceedings of the Industrial Track of the 16th International Middleware Conference. Middleware Industry '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2830013.2830015

20. Romano, P.K., Forget, B.: The openmc monte carlo particle transport code. Annals of Nuclear Energy **51**, 274–281 (2013). https://doi.org/https://doi.org/10.1016/j.anucene.2012.06.040
21. Sai, R., Mellor-Crummey, J., Meng, X., Araya-Polo, M., Meng, J.: Accelerating high-order stencils on gpus. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 86–108 (2020). https://doi.org/10.1109/PMBS51919.2020.00014
22. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G., Bouteiller, A.: Ucx: An open source framework for hpc network apis and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. pp. 40–43 (2015). https://doi.org/10.1109/HOTI.2015.13
23. Terboven, C., An Mey, D., Schmidl, D., Wagner, M.: First experiences with intel cluster openmp. In: Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism. p. 48–59. IWOMP'08, Springer-Verlag, Berlin, Heidelberg (2008)
24. Tian, S., Doerfert, J., Chapman, B.: Concurrent execution of deferred openmp target tasks with hidden helper threads. In: Chapman, B., Moreira, J. (eds.) Languages and Compilers for Parallel Computing. pp. 41–56. Springer International Publishing, Cham (2022)
25. Tramm, J.R., Siegel, A.R., Forget, B., Josey, C.: Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations. In: EASC 2014 - Solving Software Challenges for Exascale. Stockholm (2014). https://doi.org/10.1007/978-3-319-15976-8_3
26. Tramm, J.R., Siegel, A.R., Islam, T., Schulz, M.: XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In: PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future. Kyoto (2014), https://www.mcs.anl.gov/papers/P5064-0114.pdf
27. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., Wilke, J.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
28. Yan, Y., Lin, P.H., Liao, C., de Supinski, B.R., Quinlan, D.J.: Supporting multiple accelerators in high-level programming models. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores. p. 170–180. PMAM '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2712386.2712405, https://doi.org/10.1145/2712386.2712405
29. Zimmer, C., Atchley, S., Pankajakshan, R., Smith, B.E., Karlin, I., Leininger, M.L., Bertsch, A., Ryujin, B.S., Burmark, J., Walker-Loud, A., Clark, M.A., Pearce, O.: An evaluation of the coral interconnects. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3295500.3356166