DiOMP-Offloading: Portable OpenMP Offloading For Distributed Heterogeneous Systems

Baodi Shan
baodi.shan@stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Mauricio Araya-Polo
TotalEnergies EP Research & Technology US, LLC
Houston, Texas, USA

Barbara Chapman
barbara.chapman@stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Abstract

Heterogeneous supercomputers challenge traditional MPI+X with redundant GPU memory management and limited portability. We present **DiOMP-Offloading**, a framework unifying OpenMP target offloading with a PGAS global memory space and OMPCCL device-side collectives. Built on LLVM OpenMP and GASNet-EX, it eliminates staging, simplifies programmability, and improves scalability. Benchmarks on large GPU systems demonstrate reduced latency, higher throughput, and performance gains over MPI+OpenMP.

CCS Concepts

• **Computer systems organization → Parallel architectures**.

Keywords

OpenMP, PGAS, Distributed Computing, GPGPU

ACM Reference Format:
Baodi Shan, Mauricio Araya-Polo, and Barbara Chapman. 2025. DiOMP-Offloading: Portable OpenMP Offloading For Distributed Heterogeneous Systems. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern HPC systems combine manycore CPUs and multiple GPUs, demanding programming models that support massive concurrency, heterogeneous data movement, and portability. While MPI+OpenMP dominates, developers still face redundant memory management, host–device staging, and reliance on vendor-specific collectives. We present **DiOMP-Offloading**, which unifies OpenMP target offloading with a PGAS global memory space, enabling direct GPU remote access, efficient collectives via **OMPCCL**, and scalable multi-GPU execution without vendor lock-in.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Design of DiOMP-Offloading

DiOMP-Offloading is a runtime system for scalable OpenMP execution on distributed CPU–GPU clusters. Built on LLVM OpenMP and GASNet-EX, it extends the original DiOMP framework with GPU support and tightly integrates PGAS-style global memory and device collectives.

2.1 Workflow and Memory Abstraction

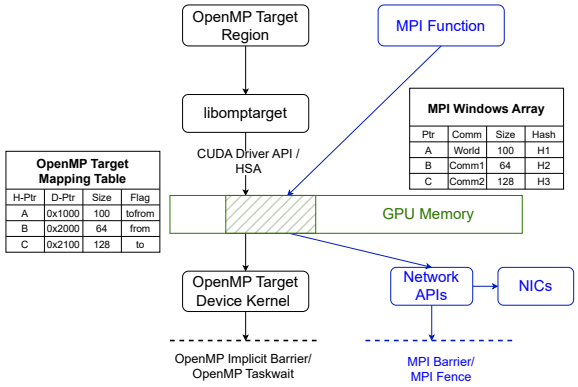
DiOMP intercepts device allocations from `libomptarget` and redirects them to a global segment managed by GASNet-EX, enabling remote `put`/`get` on GPU buffers without host staging. This shared memory space spans OpenMP offloading, GASNet P2P, and OMPCCL collectives, avoiding redundant registration and ensuring consistent lifecycles. As shown in Figure 1, this unified model couples computation, communication, and memory scheduling.

Global segments. DiOMP provides PGAS-style memory for CPU and GPU. Symmetric allocations use offset-based addressing, while asymmetric ones employ a pointer translation layer to preserve accessibility.

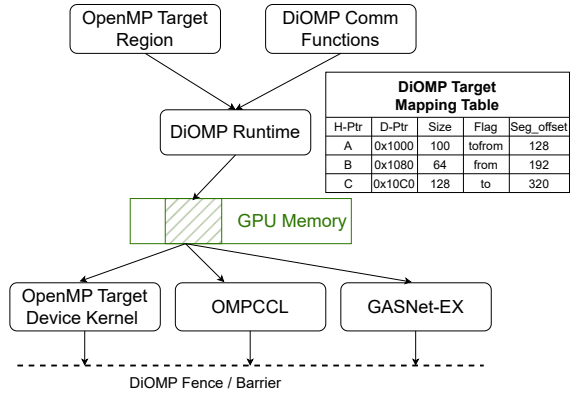
Topology-aware transfers. DiOMP chooses optimal paths: GASNet-EX for inter-node, IPC for intra-node, and GPUDirect P2P for GPU–GPU communication. A hybrid polling loop with bounded stream pools sustains throughput while limiting overhead.

2.2 OMPCCL and Collective Communication

DiOMP introduces a *Group* abstraction, like MPI communicators, to scope collectives and synchronization. OMPCCL builds on this to provide portable, device-side collectives across GPUs, abstracting NCCL/RCCL while ensuring consistency on both NVIDIA and AMD systems. New pragmas (e.g., `#pragma ompx target device_bcast`) and APIs (e.g., `ompx_bcast`) allow direct use in target regions. Decoupling groups from rank boundaries enables collectives across ar-



(a) OpenMP Target + MPI workflow



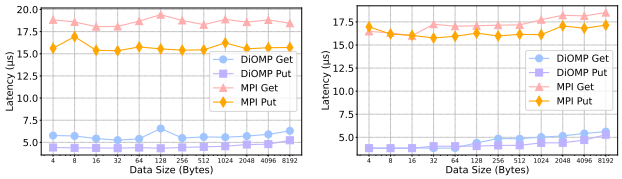
(b) DiOMP workflow

Figure 1: Comparison of workflows. (a) OpenMP+MPI separates memory registration and synchronization, creating redundant mappings and inconsistent barriers. (b) DiOMP unifies target offloading and communication, managing allocation and synchronization centrally.

bitrary device subsets, reducing metadata overhead and improving scalability.

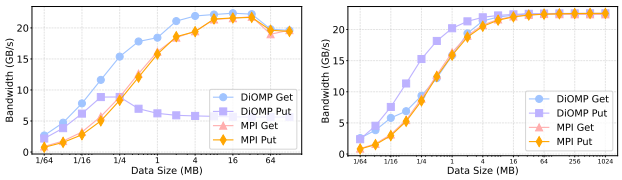
3 Evaluation

We evaluate DiOMP against MPI+OpenMP on two leadership GPU systems: (A) NERSC's Perlmutter with NVIDIA A100 GPUs and HPE Slingshot 11, and (B) LLNL's Tioga with AMD MI250X GPUs and Slingshot 11. DiOMP is built on GASNet-EX and a customized LLVM, with Cray MPICH as the MPI baseline. Figures 2 and 3[1] show that DiOMP lowers latency and improves bandwidth over MPI on these two systems.



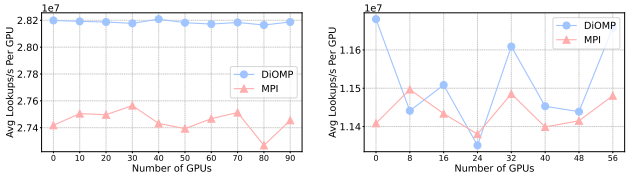
(a) Perlmutter (A100) (b) Tioga (MI250X)

Figure 2: Latency comparison of DiOMP and MPI from 4B–8KB. Lower is better.



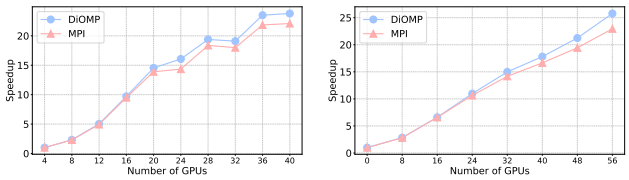
(a) Perlmutter (A100) (b) Tioga (MI250X)

Figure 3: Bandwidth comparison of DiOMP and MPI. Higher is better.



(a) Perlmutter (b) Tioga

Figure 4: XSBench comparison. Higher is better.

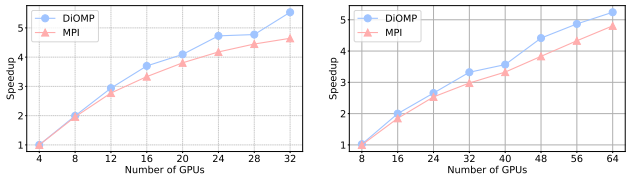


(a) Perlmutter (b) Tioga

Figure 5: Matrix multiplication speedup comparison. Higher is better.

On XSBench (Figure 4), DiOMP delivers ~4% higher throughput on Perlmutter, while matching MPI on Tioga. For matrix multiplication (Figure 5), DiOMP sustains superlinear scaling. On Minimod [1] (Figure 6), DiOMP outperforms MPI across nodes, highlighting its ability to combine performance with programmability.

[1]An anomalous bandwidth behavior on Perlmutter has been observed and reported, but is unrelated to DiOMP (see GASNet-EX users group).



(a) Perlmutter (b) Tioga

Figure 6: Minimod speedup comparison. Higher is better.

4 Discussion and Conclusion

DiOMP-Offloading differs from MPI+X and OpenSHMEM by emphasizing portability and ease of use. MPI requires extra directives to expose device memory, while non-vendor OpenSHMEM stacks (e.g., Cray-OpenSHMEMX [2]) lack GPU-Direct support. In contrast, DiOMP integrates directly with `libomptarget`, avoiding host staging, lowering overhead, and unifying communication and offloading in an OpenMP-centric model. Vendor libraries such as NVSHMEM and rocSHMEM, although GPU-Direct capable, are tied to CUDA/HIP and incompatible with OpenMP target offloading, limiting their portability. DiOMP overcomes these issues with a centralized global memory layer, enabling true cross-platform interoperability. Figure 7 illustrates the implementation of DiOMP, MPI+OpenMP, and OpenSHMEM for a typical one-sided "get" operation. A central feature, OMPCCL, further enables portable collectives inside OpenMP target regions by reconciling allocation semantics and backend setup, ensuring consistent behavior on both NVIDIA and AMD GPUs.

Conclusion. DiOMP-Offloading provides high performance, portability, and simpler programming across heterogeneous clusters. Future directions include deeper task-parallel integration with OpenMP and tighter LLVM compiler support for automatic remote data optimizations.

References

[1] Jie Meng, Andreas Atle, Henri Calandra, and Mauricio Araya-Polo. 2020. Minimod: A Finite Difference solver for Seismic Modeling. *arXiv* (2020). arXiv:2007.06048 [cs.DC] https://arxiv.org/abs/2007.06048

[2] Naveen Namashivayam, Bob Cernohous, Dan Pou, and Mark Pagel. 2019. Introducing Cray OpenSHMEMX - A Modular Multi-communication Layer OpenSHMEM Implementation. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, Swaroop Pophale, Neena Imam, Ferrol Aderholdt, and Manjunath Gorentla Venkata (Eds.). Springer International Publishing, Cham, 41–55.

```
// 1. Directly issue a non-blocking get operation
ompx_get( ... );
// or memcpy
omp_target_memcpy_async( ... );

// 2. Synchronize to ensure the operation completes
ompx_fence();
```

(a) DiOMP Offloading

```
// 1. Setup Window: A window must be created.
MPI_Win_create( ... , &win);

// 2. Communication & Synchronization: Requires Fence/Lock.
MPI_Win_fence(0, win);

// 3. Ensure MPI can access the device pointer.
#pragma omp target use_device_ptr(dst_ptr)
MPI_Get( ... , win);

MPI_Win_fence(0, win); // End epoch

// 4. Cleanup
MPI_Win_free(&win);
```

(b) MPI One-Sided Communication

```
if (my_rank == sender_rank) {
    // Asynchronous send returns a request handle
    MPI_Request req;

    #pragma omp target use_device_ptr(dst_ptr)
    MPI_Isend( ... , &req);

    MPI_Wait(&req, MPI_STATUS_IGNORE);
} else if (my_rank == receiver_rank) {
    MPI_Request req;

    #pragma omp target use_device_ptr(dst_ptr)
    MPI_Irecv( ... , &req);

    MPI_Wait(&req, MPI_STATUS_IGNORE);
}
```

(c) MPI Two-Sided Communication

```
// 1. Allocate symmetric memory using a special function
src_ptr = shmem_malloc(size);

// 2. Copy data from device to host
#pragma omp target data update(from: ... )

// 3. Communication: Performed between host memories
shmem_get( ... );

// 4. Synchronize SHMEM operation
shmem_quiet();

// 5. Copy data from host back to device
#pragma omp target data update(to: ... )
```

(d) OpenSHMEM

Figure 7: A programmability comparison of a remote get operation across models.