Extending OpenMP and OpenSHMEM for Efficient Heterogeneous Computing

Wenbin Lu, Shilei Tian, Tony Curtis, and Barbara Chapman
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, United States
{wenbin.lu, shilei.tian, anthony.curtis, barbara.chapman}@stonybrook.edu

***Abstract*—Heterogeneous supercomputing systems are becoming the mainstream thanks to their powerful accelerators. However, the accelerators' special memory model and APIs increase the development complexity, and calls for innovative programming model designs. To address this issue, OpenMP has added target offloading for portable accelerator programming, and MPI allows transparent send-receive of accelerator memory buffers. Meanwhile, Partitioned Global Address Space (PGAS) languages like OpenSHMEM are falling behind for heterogeneous computing because their special memory models pose additional challenges.**

We propose language and runtime interoperability extensions for both OpenMP and OpenSHMEM to enable portable remote access on GPU buffers, with minimal amount of code changes. Our modified runtime systems work in coordination to manage accelerator memory, eliminating the need for staging communication buffers. Comparing to the standard implementation, our extensions attain 6x point-to-point latency improvement, 1.3x better collective operation latency, 4.9x random access throughput, and up to 12.5% better performance in strong scaling configurations.

***Index Terms*—Heterogeneous Computing, LLVM, OpenMP, UCX, OpenSHMEM, Hybrid Programming**

I. Introduction

Accelerators like GPUs are extremely popular in modern supercomputers, due to their high computational throughput and excellent power efficiency. As of June 2022, nine out of the ten fastest supercomputers on the TOP500 list use accelerators as their main source of FLOPS [1]. For example, over 96% of the computing power of the Summit supercomputer at the Oak Ridge National Laboratories comes from the six NVIDIA V100 GPUs installed on each node, while the CPU's job is mainly to provide enough work to keep the GPUs occupied during application execution.

On the software side, porting old applications to, as well as writing new applications for the accelerators, proves to be challenging. Firstly, we already have accelerators from two different vendors (NVIDIA, AMD) running on production systems, with a third vendor (Intel) appearing on the horizon. Each of the three vendors uses their own programming model and compilation toolchain (CUDA, ROCm, oneAPI) for application development. These vendor-specific solutions have made writing performance portable HPC applications more challenging for the developers. Additionally, the accelerators' memory is physically disjoint from the main memory of the compute node. This requires not only host-accelerator data transfers, but also staging communication buffers in the main memory if the inter-process communication model lacks support for the accelerator and therefore cannot access its memory directly, as demonstrated in Figure 1. Although vendors provide memory coherence mechanisms like CUDA unified memory to simplify accelerator memory management, the performance loss is non-trivial [2].

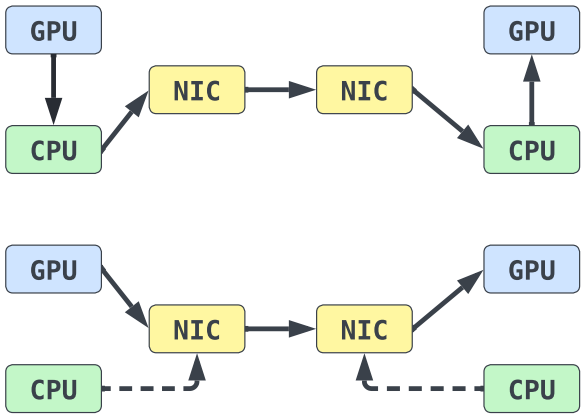

Fig. 1. Two different ways to perform inter-node GPU-to-GPU data transfers. NIC stands for network interface controller. The data flow diagram on top demonstrates the need for CPU staging communication buffer if the hardware and/or the software does not support remote direct GPU memory access. The bottom diagram shows that the extra memory copies can be eliminated with direct GPU-GPU transfers. The CPUs only need to submit read/write requests to the NICs in the second case (the dotted lines).

To address these issues, community-driven parallel programming models have been working with vendors to add support for accelerators. For portable on-node parallelism, the OpenMP [3] programming model is among one of the most popular choices. Continuing its compiler directives-based approach to parallelize loops on the CPU, OpenMP has added device offloading support for executing annotated loops on the accelerators. By abstracting away low-level vendor-

This research was 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.

specific details and providing a unified high-level interface, users can harness the computing power of HPC accelerators using the same code base, regardless of the hardware vendor. Similarly, as the most popular distributed HPC programming model, Message Passing Interface (MPI) [4] has also seen continuous accelerator-related improvements since the early days of heterogeneous supercomputing. From the application developer's point of view, instead of copying data back and forth between the accelerator and dedicated communication buffers in the main memory, passing pointers to accelerator buffers directly to the MPI communication routines significantly simplifies application development. Under the hood, MPI implementations can use optimizations such as pipelining to hide host-accelerator copy latency and improve communication performance.

However, the two-sided message passing communication paradigm is not the best fit for all classes of applications, especially the ones that are highly dynamic and/or irregular in nature. Partitioned Global Address Space (PGAS) programming models were developed to enable low-overhead one-sided Remote Memory Access (RMA) in HPC applications. In PGAS programming models, each processing element (PE) manages a portion of a globally shared memory address space. Data objects allocated inside the local PE's partition of globally shared memory are accessible by remote PEs without active participation of the local PE. Since these one-sided communication operations are expected to progress automatically (in contrast to calling `MPI_Test` to progress MPI messages), PGAS programming models are particularly suited to implement dynamic and/or irregular algorithms like graph analytics. OpenSHMEM [5] is one of the most successful PGAS programming models. It is a library-based programming model, providing point-to-point RMA, atomic operations (AMO) and collective operations on data objects allocated inside its symmetric heap. The simple and high-performance API of OpenSHMEM has made it an attractive alternative to MPI. In addition to dedicated implementations like OSSS-UCX, Sandia OpenSHMEM and Cray OpenSHMEMX, Open-MPI and MVAPICH2 also provide their own OpenSHMEM implementation.

However, OpenSHMEM is falling behind when it comes to HPC accelerator support. The latest OpenSHMEM specification [6], version 1.5, still lacks functionality for accessing accelerator memory buffers. The *Symmetric Partitions* [7] proposal is one of the earliest attempts for adding accelerator memory support to OpenSHMEM, but it has not gain much traction. Currently, the OpenSHMEM specification committee is working on the *Memory Spaces* proposal to provide teams-based creation/attachment of device memory buffers. It is not finalized yet, and the implementations are not ready. As a result, hybrid OpenSHMEM+X applications, where X is an accelerator programming model, have to use the CPU symmetric heap as a staging buffer, and perform manual host-accelerator data transfers before and after each RMA/AMO/collective operation.

Listing 1 shows two examples of only using specification-conforming OpenSHMEM and OpenMP to perform a `PUT` operation and a vector sum reduction. In the first example, the source PE must first copy the buffer from the accelerator to the CPU symmetric heap using the `omp target update from` directive. Then, the source PE writes the buffer to the symmetric heap of the remote PE, and sends a signal to notify it. The remote PE will wait on the signal until the buffer has been delivered, and then upload it to the accelerator using another OpenMP directive. Similarly, example 2 shows that we also have to perform manual copies before and after collective operations. These extra copies not only increase the development burden, but also impact application performance negatively. More importantly, requiring active participation from all PEs involved will break the one-sidedness of OpenSHMEM's communication model. For `PUT` operations, the target PE has to know in advance that there will be incoming `PUT` notifications from certain PEs, and will need to check these signal variables either periodically or will have to set up dedicated polling threads. It is even more complicated for the `GET` operation and atomic fetch/swap, for which we will show an example in Listing 5. Memory coherence mechanisms like CUDA unified memory could simplify the development process by eliminating manual memory copies, but the performance penalty of hidden host-accelerator data transfers remains to be a problem.

In this work, we aim to improve the interoperability between

```
float* X = shmem_malloc(M * sizeof(float));
float* U = shmem_malloc(N * sizeof(float));
float* V = shmem_malloc(N * sizeof(float));

#pragma omp target data map(tofrom:X[0:M]) \
                        map(from:U[0:N],V[0:N])
{
  // ==== Example 1: RMA PUT ====
  // ** Source PE
  // Copy X to main memory
  #pragma omp target update from(X[0:M])
  // PUT with notification
  shmem_float_put_signal(X, X, ...);

  // ** Target PE
  // Wait for the PUT signal
  shmem_signal_wait_until(...);
  // Copy X to device memory
  #pragma omp target update to(X[0:M])

  // ==== Example 2: Vector Reduction ====
  // Do device computation, store results in V
  #pragma omp target teams distribute parallel for
  for (...)
  // Copy V to main memory
  #pragma omp target update from(V[0:N])
  // Do reduction
  shmem_float_sum_to_all(U, V, ...);
  // Copy U to device memory
  #pragma omp target update to(U[0:N])
  // Do more device computation with U
  #pragma omp target teams distribute parallel for
  for (...)
}
```

Listing 1. Examples of hybrid OpenSHMEM and OpenMP target offloading without interoperability extensions.

OpenSHMEM and accelerator programming models, with the goal being to enable transparent remote access to accelerator memory, while maintaining the one-sided asynchronous communication semantics of OpenSHMEM. We choose to focus on supporting the OpenMP target offloading model, as it is the most portable solution across different accelerator vendors. To the best of our knowledge, this is the first attempt to extend OpenSHMEM and OpenMP target offloading for writing performance portable application running on heterogeneous supercomputers.

In our research prototype, we choose to implement the *Symmetric Partitions* proposal for its simplicity, but our interoperability workflow design is flexible enough to adapt to a different OpenSHMEM memory allocation interface. Once the OpenSHMEM specification committee finalizes the *Memory Spaces* proposal, our implementation can be ported to the new API without much effort.

This paper makes the following contributions:

- An optimized OpenSHMEM *Symmetric Partitions* implementation for managing OpenMP device buffers, without introducing vendor-specific code into the OpenSHMEM runtime or the application.
- An OpenMP memory allocator extension for mapping device objects that are interoperable with OpenSHMEM, built on top of the LLVM/OpenMP runtime system.
- A novel way to connect the runtime systems of the two programming models using our extensions, to enable transparent one-sided remote memory access on OpenMP mapped device objects.
- Evaluation of the proposed interoperability extensions using micro-benchmarks and mini-apps that cover the most common communication patterns in HPC applications.

Experiments have shown that our extensions can significantly improve the performance of OpenSHMEM+OpenMP hybrid applications, while requiring less code. In micro-benchmarks, our prototype implementation can attain 6x improvement of point-to-point latency and 1.3x improvement of collective operation latency, compared to a standard OpenSHMEM+OpenMP hybrid that uses CPU staging buffers. For mini-apps using communication patterns commonly seen in HPC applications, when compared to the standard implementation, we have obtained 4.9x higher random access throughput, 12.5% better strong scaling parallel efficiency in all-to-all exchange, and 8% better strong scaling parallel efficiency in nearest-neighbor exchange.

II. Background

A. Accelerator Programming with OpenMP

OpenMP is a directive-based on-node parallel programming model. In addition to traditional thread-based parallelism, its target offloading model provides a convenient and flexible mechanism to exploit the substantial computing power within the nodes of today's high-performance heterogeneous supercomputers [8]. It is one of the few programming models that will be supported on exascale systems from the United States Department of Energy, Exascale Computing Project (ECP). OpenMP introduced support for accelerators in OpenMP 4.0 via its `target` constructs, extended soon thereafter in OpenMP 4.5, e.g., to allow asynchronous execution of target regions [9]. Because of its performance portability and the relatively smooth transition from its (naturally grown) thread-parallel model into the target offloading realm [10], many ECP application proposals include OpenMP as part of their strategy for reaching exascale levels of performance.

```
// A is a regular buffer in the main memory
float* A = malloc(N * sizeof(float));
// Use OpenMP runtime API to allocate device buffer
float* B = omp_target_alloc(N * sizeof(float), 0);

// Device storage for A is automatically allocated
// by the map clause
// The mapping between A's host & device addresses
// is also created
#pragma omp target data map(tofrom:A[0:N])
{
  // Content of A automatically copied to the GPU

  // A can be used directly since OpenMP knows its
  // mapping
  // Use is_device_ptr to indicate that B is already
  // located on the device
  #pragma omp target teams distribute parallel for \
                    is_device_ptr(B)
  for (...) {
    A[i] += B[i];
  }

  // This prints the host address of A
  printf("%p\n", A);

  // This prints the device address of A
  #pragma omp target data use_device_ptr(A)
  printf("%p\n", A);

  // Content of A automatically copied to the host
}

// Release the buffers
omp_target_free(B, 0);
free(A);
```

Listing 2. OpenMP device buffer management examples. Buffer A is managed by the `map` clause, while buffer B is managed manually.

In the OpenMP device offloading model, when a `target` region is encountered, the host runtime automatically allocates device memory for the list of mapped variables, using vendor-specific API such as `cuMemAlloc` for NVIDIA GPUs. Initial values and computation results contained in the device buffers will also be copied automatically, if specified by the `map(to:...)` and `map(from:...)` clauses. Additionally, the user can also choose to manage device buffers manually using the OpenMP runtime APIs, an example is provided in Listing 2. The memory management abstractions provided by OpenMP hides low-level vendor-specific details from the application developer and improves portability.

In the many open-source and proprietary OpenMP compiler implementations, LLVM/OpenMP stands out in terms of both quality and completeness. Offloading backends for NVIDIA

and AMD GPUs are ready for use, while support for Intel GPUs is being worked on. Therefore, LLVM/OpenMP has seen great adoption in both production runs and research activities.

B. Distributed Programming with OpenSHMEM

OpenSHMEM is a library-based partitioned global address space (PGAS) programming model that allows inter-process data exchange in shared and distributed memory machines. OpenSHMEM provides blocking and non-blocking point-to-point one-sided communication like PUT, GET and atomic operations, as well as collective operations like reductions and all-to-all exchange. Since communication and synchronization are decoupled, the asynchronous one-sided communication model of OpenSHMEM has made it a better fit for implementing dynamic and/or irregular algorithms, as it requires no active participation from the target process of a communication operation. However, regular objects that are allocated on the stack or heap are not accessible by remote processes. Users of OpenSHMEM should use its memory management routine to allocate buffers (in a collective fashion) on its symmetric heap to obtain symmetric data objects, and only these objects are remotely accessible.

OpenSHMEM has attracted users from both academia and industry, and continues to add new features. But as of OpenSHMEM specification version 1.5, it is still focused on CPU programming, without standardized support for heterogeneous accelerators.

C. Interoperability Issues of Hybrid OpenMP and OpenSHMEM

OpenMP and OpenSHMEM were not designed with each other in mind, and this has created issues for users that want to write hybrid applications. Although GPU vendors have developed drivers that allow the network card to access GPU memory directly (e.g. GPUDirect RDMA [11]), we cannot take advantage of them in hybrid OpenSHMEM+OpenMP applications, as OpenSHMEM controls the network stack but does not yet provide the necessary abstractions for GPU communications. In the current state, the users have to set up staging OpenSHMEM communication buffers in the main memory, and perform expensive manual data copies before and after the communication operations (example in Listing 1). Also, while OpenMP has greatly simplified accelerator programming by hiding low-level details (device contexts, execution streams, event notifications, etc.) from the users, it assumes full control of device memory management. When mixing OpenMP with OpenSHMEM, which will also act as a memory allocator once GPU support is added, it is unclear who should yield the control. This lack of interoperability considerations leads to increased development cost, and decreased application performance.

III. Design and Implementation

In this section, we will talk about the design and implementation of our interoperability extensions to both OpenSHMEM and OpenMP, as well as how they interact with each other. The term "accelerator" and "device" are used interchangeably in this paper.

```
// Environment variable for defining symmetric
// partitions
SHMEM_SYMMETRIC_PARTITION<ID>=SIZE=<size>
                              [:PGSIZE]=<pgsize>]
                              [:KIND=<kind>]
                              [:policy=<policy>]

// Symmetric partition memory allocation routine
void* shmem_partition_malloc(size_t size, int id)
```

Listing 3. The OpenSHMEM symmetric memory partitions extension.

A. OpenMP-Aware OpenSHMEM Extensions

To support heterogeneous computing, we would like to extend OpenSHMEM's symmetric heaps for accelerator memory. Previously, an attachment-based solution was proposed to add GPU symmetric heaps to OpenSHMEM [12]. The idea is as follows: the user allocates device buffers with vendor API like `cudaMalloc()`, the returned address ranges are then registered with the OpenSHMEM runtime using the special `shmemx_attach()` API extension, and host-device data transfers should continue to use the `cudaMemcpy()` routine. While this is sufficient for hybrid OpenSHMEM+CUDA applications due to how low-level CUDA is, it does not work well with OpenMP device offloading.

To reduce the effort for porting CPU-only applications to run on GPUs, OpenMP encourages the use of the `map` clause, which specifies how an original memory buffer is mapped from the current data environment to a corresponding buffer in the device data environment. The OpenMP runtime library will allocate/free/copy device memory buffers based on the map type without the extra effort from application developers. By delegating device memory management to OpenMP, the code base will be simplified, and the application can enjoy compiler-assisted data movement optimizations.

On the other hand, the per-buffer attachment approach forces the users to go back to manual device memory management and data transfers, thus negating one of the most important benefits of using OpenMP for device computation. Additionally, since these buffers are not allocated using the `map` clause, OpenMP does not know about them and all OpenMP target regions that access these buffers must be marked with an additional `is_device_ptr` clause[1]. As the result, the attachment-based solution leads to unnecessarily complicated code and prevents OpenMP-specific compiler optimizations.

A better approach would be to allow the OpenMP runtime to use OpenSHMEM as its internal device memory pool, so that the application can continue to use the `map` clause and transparently enable RMA for mapped device buffers. This approach requires minimal changes to both programming models because OpenSHMEM's global address space

[1]It indicates that those buffer pointers are device pointers.

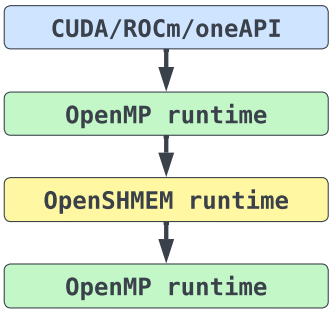

Fig. 2. Flow of memory address information in the proposed interoperability extensions. First, the OpenMP runtime allocates a large chunk of device memory using vendor API. Then, the address space is passed to OpenSHMEM to create a device memory pool and enable remote memory access. Finally, the OpenMP runtime uses OpenSHMEM to manage device computation buffers.

is already presented as a memory pool, it only needs a fixed memory address range `[start, end]` to perform allocations. Therefore, we can reserve a large chunk of device memory using OpenMP, pass its address range to OpenSHMEM during initialization, and let OpenMP allocate device buffers from this reserved space using OpenSHMEM memory management routines. A simplified diagram of this process is provided in Figure 2.

Therefore, we choose to implement a simplified version of the OpenSHMEM *Symmetric Partitions* extension proposed by Cray [7]. Instead of a single CPU symmetric heap controlled by the `SHMEM_SYMMETRIC_SIZE` environment variable, this extension allows OpenSHMEM to split the symmetric heap into multiple partitions, with their locations and sizes controlled by a series of environment variables shown in Listing 3. Multiple symmetric partitions can be created during `shmem_init` by defining environment variables with different `ID` and `SIZE`. Additionally, the optional `KIND` trait specifies the kind of memory used by the partition. Dependent on what is supported by the implementation, this can be set to non-volatile memory, high-bandwidth memory, and GPU memory of a specific vendor.

We have implemented the symmetric partitions extension on top of the reference OpenSHMEM implementation, OSSS-UCX [13], with support for memory kind `LIBOMP` to interoperate with the LLVM/OpenMP device offloading runtime. For example, if the user defines environment variables `SHMEM_SYMMETRIC_PARTITION0=8G` and `SHMEM_SYMMETRIC_PARTITION1=2G:KIND=LIBOMP`, OSSS-UCX will create an 8 GiB symmetric partition in main memory, and a 2 GiB symmetric partition in device memory.

During initialization, the OpenSHMEM runtime parses its environment variables, and allocates a device buffer using the `omp_target_alloc` routine[2]. The runtime then registers the device buffer with the UCX communication framework [14], so that they are considered "pinned" by the network card driver and allow Remote Direct Memory Access (RDMA). Since UCX has done the heavy lifting of identifying the vendor of the accelerator and setting it up for RDMA, we do not need to introduce vendor-specific code into the OpenSHMEM runtime. For the actual communication, we can continue to use standard OpenSHMEM communication routines, as Unified Virtual Addressing (UVA) guarantees that the address spaces of the main memory and the accelerator do not overlap. Therefore, when a communication routine is invoked with a pointer pointing into a device symmetric partition, we will calculate which symmetric partition this pointer points to, and select the correct UCX remote key (rkey) to perform the communication operation.

While UCX handles PUTs, GETs and atomic operations required by the OpenSHMEM specification, we still need to make sure that collective operations work on device symmetric partitions. For operations that do not modify the values of the operands (broadcast, collect, all-to-all), implementations for CPU buffers will continue to work, as long as all local and remote access to the accelerator buffers are done through OpenSHMEM RMA communication routines. For reductions, we use the Unified Communication Collectives Library (UCC) [15], which automatically launch kernels to handle the computational parts of reduction operation.

One additional challenge is that the OpenSHMEM runtime needs to have an accelerator memory pool that does not access the managed buffer, so that we can avoid introducing vendor-specific code into the runtime. Also, since the OpenMP application could require multiple small allocations for each target region execution (e.g. for variables that store reduction results), it is crucial to have a high-performance memory manager. In our work, we have implemented a memory pool using the Buddy memory allocation algorithm [16], which stores allocation records outside the pool and is extremely efficient (around 40 nanoseconds per allocation).

With the symmetric memory partitions extension, OpenSHMEM can create accelerator buffers that are accessible by regular communication routines, and is ready to be used as the internal memory allocator of the LLVM/OpenMP device offloading runtime. Our design is completely vendor-agnostic and only requires setting a few environment variables.

B. OpenSHMEM-Aware OpenMP Extensions

By default, storage for variables listed in OpenMP's `map` clause are allocated by the LLVM/OpenMP host runtime, which wraps the memory management routines provided by the accelerator's vendor. Starting from version 5.0, OpenMP added support for specifying the type of memory allocator for the `target` construct using the `allocator` and `uses_allocators` clauses. The OpenMP specification has provided a list of predefined memory spaces (constant, high-bandwidth, low-latency, etc.), and a corresponding list of memory allocators.

To support using OpenSHMEM as an OpenMP device memory allocator, we extend the `allocator` related clauses to be used on `target data` regions. We also introduce an `omp_shmem_mem_space` memory space extension, as well

[2]It allocates memory in a device data environment and returns a device pointer to that memory.

```
// Register OpenSHMEM API, create the allocator
__tgt_set_shmem_allocator(shmem_partition_malloc,
                          shmem_partition_free,
                          0);
omp_memspace_handle_t shmems = omp_shmem_mem_space;
omp_allocator_handle_t shmema = ...;

auto X = new float[M]();
auto U = new float[N]();
auto V = new float[N]();

#pragma omp target data map(tofrom:X[0:M])      \
                        map(from:U[0:N],V[0:N]) \
                        uses_allocators(shmema) \
                        allocator(shmema:X,U,V)
{
  // ==== Example 1: RMA PUT ====
  // ** Source PE only
  // Pass device pointer of X to OpenSHMEM
  #pragma omp target data use_device_ptr(X)
  shmem_float_put(X, X, ...);

  // ==== Example 2: Vector Reduction ====
  // Do device computation, store results in V
  #pragma omp target teams distribute parallel for
  for (...)
  // Use device addresses of U & V for reduction
  #pragma omp target data use_device_ptr(U,V)
  shmem_float_sum_to_all(U, V, ...);
  // Do more device computation with U
  #pragma omp target teams distribute parallel for
  for (...)
}
```

Listing 4. Hybrid OpenSHMEM and OpenMP device offloading example, with interoperability extensions.

as its corresponding allocator `omp_shmem_mem_alloc`. This allows all `target data` regions annotated by the OpenSHMEM allocator to use OpenSHMEM to allocate storage for its mapped variables, thus enabling remote access for device objects. Hybrid programming examples shown in Listing 1 is modified to use our interoperability extensions in Listing 4. The use of staging communication buffers has been removed since now OpenSHMEM is capable of handling device buffers. Note that we cannot pass the variables `X`, `U` and `V` directly to OpenSHMEM, as their values still point to locations in the main memory. Instead, the `use_device_ptr` clause[3] is used to instruct the compiler to pass the corresponding device addresses to OpenSHMEM. This is a compile-time decision and therefore do not incur any run-time overhead.

Since OpenMP target allocators are still in the early stage of adoption, the compiler infrastructure for implementing our extensions is incomplete. Therefore, in our LLVM/OpenMP based prototype, all OpenMP mapped variables are allocated using our extended OpenSHMEM allocator.

C. Hybrid OpenSHMEM and OpenMP Workflow

The run-time interactions between the hybrid application, the LLVM/OpenMP runtime and the OSSS-UCX OpenSHMEM runtime are depicted in Figure 3. The detailed description of the four steps are as follows:

[3]It indicates that each list item is a pointer to an object that has corresponding storage on the device or is accessible on the device.

1) When the application starts its execution, the offloading-related code inside `__libc_start_main` will initialize the device runtime for all the accelerators (creating device contexts, streams, etc.).
2) When `shmem_init` is invoked, the OpenSHMEM runtime will parse the environment variables that define symmetric partitions, call `omp_target_alloc` to allocate symmetric partitions with the specified sizes on the accelerators, and register the base pointers (OSSS-UCX memory pool, UCX mapped memory), so that the device symmetric partitions are RDMA-ready.
3) Once we encounter an OpenMP target data mapping clause that requests OpenSHMEM symmetric partition memory, the LLVM/OpenMP runtime calls the registered `shmem_partition_malloc` routine, which returns a pointer to accelerator memory that can be accessed using the vendor `memcpy` routines and also from inside kernels.
4) When an OpenSHMEM communication routine like `shmem_putmem` is called with a pointer pointing into a device symmetric partition, OSSS-UCX will perform address range calculation, get the corresponding UCX remote key, and perform the communication operation.

Through our interoperability extensions to OpenSHMEM and OpenMP, runtime systems of the two programming models work seamlessly together to provide low-latency one-sided remote access to OpenMP mapped device buffers, with minimum efforts from the application developer.

IV. Evaluation

A. Experimental Setup

To evaluate the effectiveness of our interoperability extensions (referred to as "Extended OpenSHMEM"), we ran a series of benchmarks with different communication patterns on the Summit supercomputer at the Oak Ridge National Laboratories, and compare the results with unmodified OpenSHMEM+OpenMP (referred to as "Standard OpenSHMEM") and MPI+OpenMP running on the same setup. Each Summit node is equipped with a total of 42 IBM POWER 9 CPU cores in two sockets, 6 NVIDIA V100 GPUs connected to each other through NVLink, and a Mellanox ConnectX-5 EDR 100Gb/s network card connected to a fat-tree network.

For software, we use RHEL 8.2, GCC 8.5.0, CUDA 11.0.3, MLNX_OFED 4.9, GDRCopy 2.0, UCX v1.12.x commit `dc92435`, UCC commit `cd393e4`. Our modified OSSS-UCX is based on upstream commit `3f565e0`, and our modified LLVM is based on upstream commit `d2792e7`. Additionally, we use IBM Spectrum MPI 10.4.0 to serve as a vendor-optimized MPI performance reference. Note that instead of UCX, Spectrum MPI uses the Parallel Active Messaging Interface (PAMI) as its communication backend, so the observed performance differences could also be affected by the differences in the communication middlewares. All benchmarks are compiled with our modified LLVM installation to keep GPU kernel performance the same.

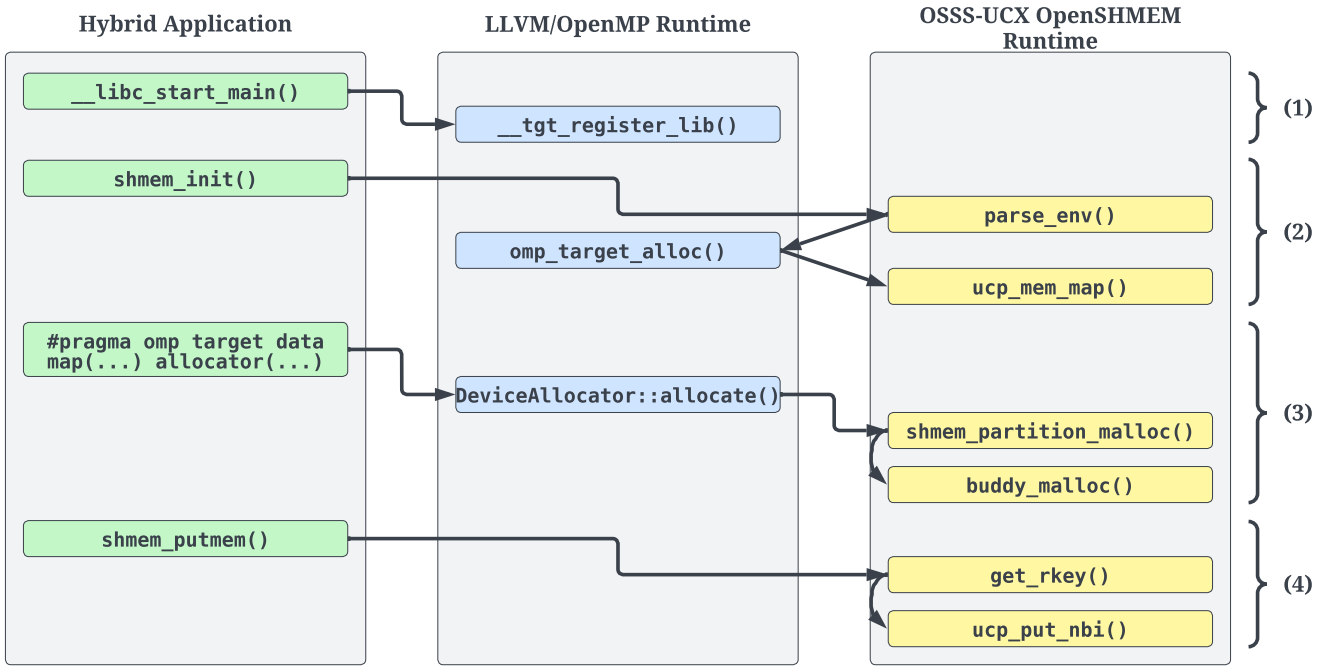

Fig. 3. Hybrid OpenSHMEM and OpenMP device offloading workflow, with interoperability extensions.

Point-to-point and collective micro-benchmarks are first used to demonstrate the basic overhead reduction capability of our extensions. Then, four mini-apps are used to evaluate the performance improvements of our extensions for applications that have different communication patterns. The four mini-apps and the corresponding communication patterns are: Random Access for random reads and writes, Fast Fourier Transform for all-to-all, matrix multiplication for ring exchange, and 3D heat equation for nearest-neighbor exchange. These communication patterns cover a wide variety of HPC applications: stencil PDE, N-body, particle methods, dense linear algebra, and graph analytics. We wrote all the mini-apps from scratch with reasonable amounts of optimizations (e.g. communication-computation overlap), while keeping them as close to each other as possible. In all our benchmarks, we use one GPU per PE/rank, which is the standard practice to avoid dealing with the complexity of handling multiple GPUs and/or NUMA nodes in a single process.

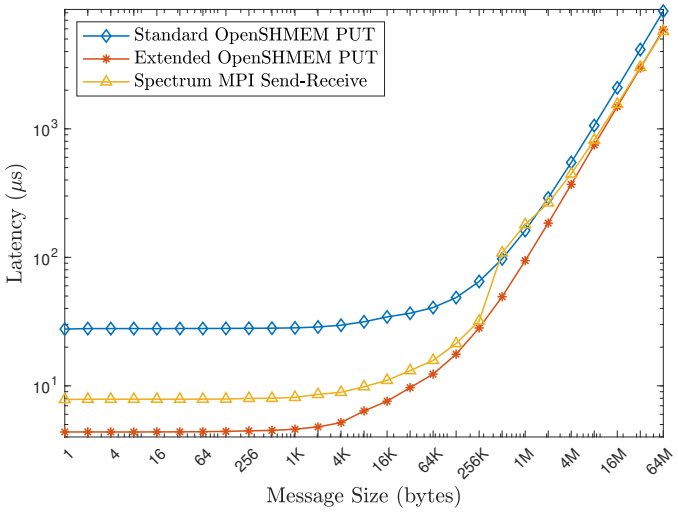

Fig. 4. Inter-node point-to-point communication latency between two GPUs. Lower is better.

B. Inter-Node Micro-benchmarks

We first present the latency numbers of communicating between a pair of GPUs located on two different nodes using `shmem_put`. Spectrum MPI CUDA send-receive latencies are also added as a reference. Results for inter-node CPU-GPU transfers are very similar and therefore omitted.

Results in Figure 4 show that for small messages, OpenSHMEM communication latency between two remote GPUs is 6 times faster with our interoperability extensions. The advantage decreases as message size increases, since the benchmark becomes more and more bandwidth-bound. But even for messages larger than 64 MiB, the Extended OpenSHMEM version is still at least 1.3 times faster than the Standard OpenSHMEM version. This shows that removing the need for manual CPU-GPU transfers through interoperability extensions yields significant performance benefits for hybrid OpenSHMEM + OpenMP offloading.

Additionally, we benchmarked the all-to-all exchange operation on 8 nodes (48 GPUs) to demonstrate the benefits of

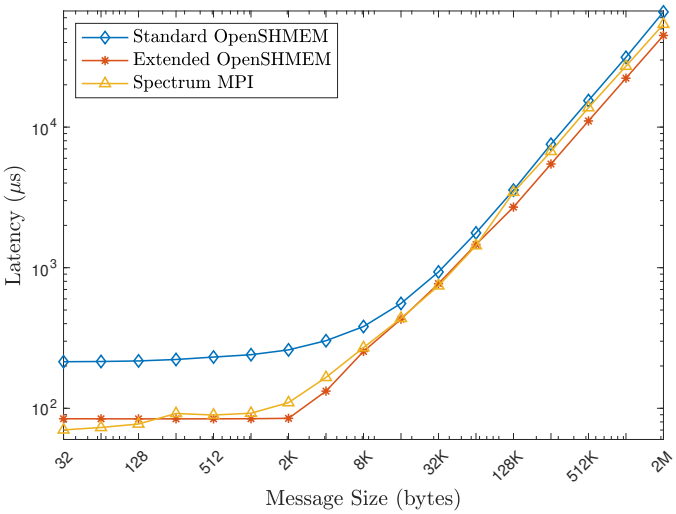

Fig. 5. All-to-all communication latency between 48 GPUs. Lower is better.

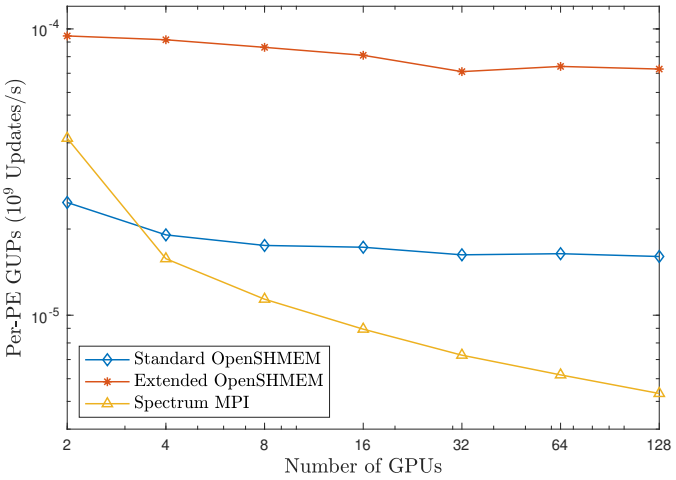

Fig. 6. HPC Challenge Random Access throughput in weak scaling configurations. Higher is better.

our extensions for collective operations. In all-to-all exchange, every process send a message of a given length to every other process. It is one of the most communication-intensive collective operations, and is commonly used to do Fast Fourier Transformation (FFT) or matrix transposition.

Similarly, all-to-all latency results in Figure 5 shows that the extended version is 2.6 times faster than the standard version for small messages, and is 1.4 times faster for large messages.

C. HPC Challenge Random Access (GUPs)

For mini-apps, we start with the HPC Challenge Random Access benchmark, which profiles the memory architecture of a system using random accesses. It measures the performance of random 64-bit read-modify-write with the unit of Giga updates per second (GUPs). For pure-CPU execution, this benchmark favors PGAS models since their simple one-sided GET-modify-PUT implementations adapts well with the randomness of the accesses, while MPI implementations require careful optimization and tuning for the send-receive mechanism to achieve good performance.

For GPU executions, the one-sided GETs initiated by the CPU cannot reach the GPU memory of the target PE, breaking the simple GET-modify-PUT implementation. For the Standard OpenSHMEM version of the mini-app, we have simulated GPU GETs using put-with-signal and an additional polling thread. The simplified code of the Standard OpenSHMEM version, as well as the Extended OpenSHMEM version, are shown in Listing 5. As we can see, the Standard OpenSHMEM version is quite complicated, and consumes an extra CPU core to handle incoming "GET" requests.

The weak scaling results of the GUPs benchmark are presented in Figure 6. In the evaluation, each PE/rank perform 2^{21} updates to remote GPUs, while we increase the number of GPUs from 2 to 128. Note that the data updates are performed on the CPU, so all remote data transfers are inter-node CPU-GPU. The results show that both OpenSHMEM implementations of the mini-app have good weak scalability, but our interoperability extensions improved the performance by up to 4.9 times. MPI's two-sided send-receive implementation does not scale well for random accesses.

D. Fast Fourier Transform

The next mini-app calculates the Fast Fourier Transform (FFT) of a 2D function stored in a square matrix. FFT is the core algorithm of many important HPC applications, such as molecular dynamics and spectral methods. In terms of communication pattern, FFT stresses the all-to-all capability of the system. Initially, the matrix is partitioned into block stripes of the same size and distributed to the participating GPUs. Then, all GPUs perform 1D FFT in one direction, transpose the matrix, and perform 1D FFT in that same direction. The matrix transposition is implemented in an all-to-all exchange and is not overlapped with the computation.

```
// Standard SHMEM read-modify-write initiator
shmem_atomic_set(...);  // Send "Ready to GET"
while (sig_s) {};       // Wait for "Data PUT"
foo(...);               // Modify
shmem_put_signal(...);  // Send "Data updated"

// Standard SHMEM read-modify-write polling thread
terminate(sig_t);       // Check for termination
check(sig_g, PE);       // Check for "Ready to GET"
#pragma omp target update from(...) // Read from GPU
shmem_put_signal(...); // Send "Data PUT"
check(sig_u, PE);       // Wait for "Data updated"
#pragma omp target update to(...)   // Write to GPU


// Extended SHMEM read-modify-write
#pragma omp target data use_device_ptr(table)
{
  auto v = shmem_uint64_g(...); // Read
  v = foo(v);                   // Modify
  shmem_uint64_p(...);          // Write
}
```

Listing 5. Simplified HPC Challenge Random Access implementations, Standard SHMEM v.s. Extended SHMEM.

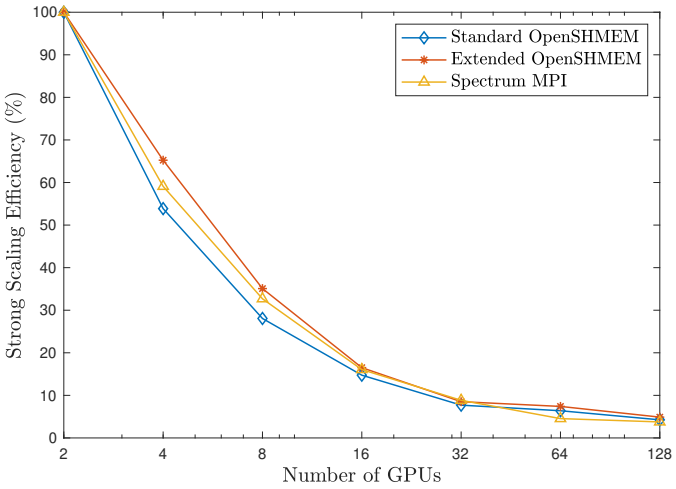

Fig. 7. Strong scaling parallel efficiency of Fast Fourier Transform. Higher is better.

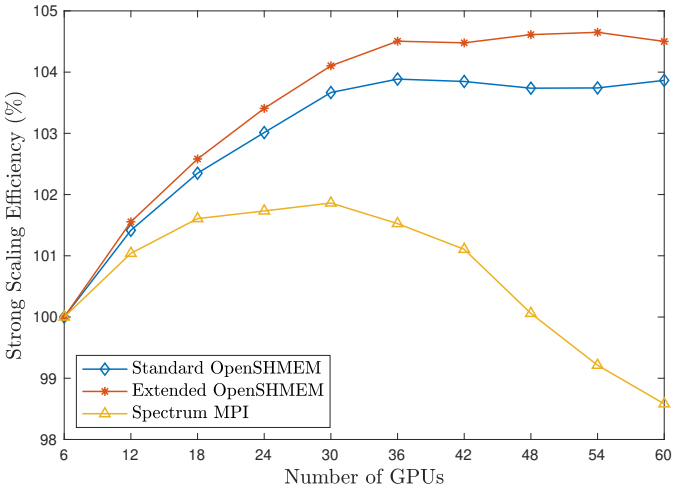

Fig. 8. Strong scaling parallel efficiency of matrix multiplication. Higher is better.

The strong scaling results of performing FFT on a 4096×4096 complex matrix on 2 to 128 GPUs are presented in Figure 7. All three versions of the mini-app does not scale well, as the communication-intensive all-to-all exchange does not scale well, even with IBM's optimized implementation in Spectrum MPI. However, the Extended OpenSHMEM version still consistently beats the other two versions, by up to 12.5%.

E. Matrix Multiplication

We then test the ring exchange communication pattern using a mini-app that implements the Cannon's algorithm to calculate square matrix multiplication product $C = A \times B$. All three versions of the mini-app use an extra block stripe for matrix B so that computation and communication can be overlapped. The Standard OpenSHMEM version must perform manual CPU-GPU transfers before and after the exchange of the block stripes. These transfers happen serially and are not overlapped with the computation.

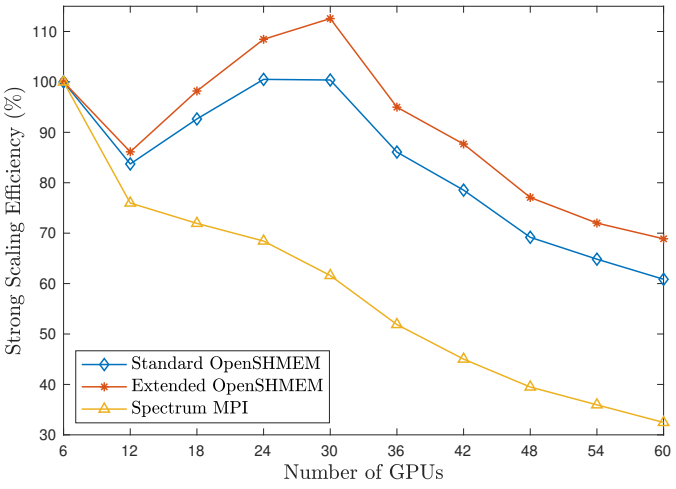

Fig. 9. Strong scaling parallel efficiency of 3D heat equation solving. Higher is better.

Figure 8 shows the strong scaling results of multiplying two 30240^2 matrices together using 6 to 60 GPUs. Ring exchange is the simplest communication pattern when compared to the other three patterns, as every PE/rank only communications with its left and right neighbor in each iteration. Therefore, the differences between the three versions are not significant, with the Extended OpenSHMEM version having around 1% higher scaling efficiency. The super-linear scaling of the mini-apps is caused by improved cache utilization when the matrices are partitioned amongst increasing numbers of GPUs.

F. 3D Heat Equation

Finally, we evaluate our interoperability extensions using a mini-app that solves the heat equation on a 3D domain to see how it performs under the nearest-neighbor communication pattern. The mini-app uses spatial discretization and a 7-point finite difference scheme to calculate the propagation of heat in a solid square block of material. The grid is partitioned evenly between all participating compute units, and the neighboring compute units will have grid points on the boundaries duplicated (halo regions), to correctly handle the data dependencies created by the 7-point stencil. Since we use the periodic boundary condition, each compute unit will have exactly six neighbors that it needs to perform halo exchange with.

In all three versions of the mini-app, we use non-blocking communication routines to exchange the halo regions, so that the communication can overlap with the computation of the inner grid points. Again, the Standard OpenSHMEM version requires extra steps to copy halo regions between the CPU and the GPU, which cannot be overlapped with computation and will result in negative performance impacts.

Figure 9 shows the strong scaling efficiency of all three versions of the mini-app, solving a 1260^3 simulation grid for 10000 time steps. Results are normalized to the execution time of the mini-app on 6 GPUs. As we can see, the version of the mini-app that uses our interoperability extensions con-

sistently beats the Standard OpenSHMEM version, attaining an 8% higher strong scalability on 60 GPUs. The super-linear scaling that appeared for 24 and 30 GPUs is caused by communication-optimal X-Y-Z partitioning and ranking of the simulation grids among the GPUs. The MPI version has worse scalability, possibly due to the lack of effective communication progression for CUDA buffers (no overlap).

V. Related Work

Researchers have been working on extending programming models for heterogeneous computing since the dawn of general-purpose computing on graphics processing (GPGPU). For MPI, MVAPICH-GDR [17], OpenMPI [18] and MPICH [19] are the most well-known open source implementations that have support for direct communication on GPU buffers. Vendor implementations like Cray MPICH and IBM Spectrum MPI support GPUs as well. Unfortunately, although MPI-3 has introduced one-sided communication similar to the PGAS languages, the majority of the implementations above do not support one-sided access to GPU buffers.

OpenMP is primarily designed to handle on-node parallelism, but efforts [20] were still made to extend its device offloading model to program remote GPUs. This approach has better programmability than hybrid solutions, but is facing scalability challenges stemming from the limitations of the OpenMP specification.

For the PGAS family of programming languages, OpenSHMEM has received the most attention for interoperability improvements. The MVAPICH team has worked on CUDA-aware OpenSHMEM and has obtained good results [12] [21] [22]. Their GPU attach-based OpenSHMEM extension requires the user to manually attach each GPU buffer that needs remote access, and is limited to CUDA devices. GPU vendors are also putting efforts to develop their OpenSHMEM-like programming models. Both NVSHMEM [23] [24] and ROCM_SHMEM [25] enable GPU-initiated communications that have very low overheads. However, their solutions are not portable to other vendors' accelerators. Other attempts [26] [27] [28] to improve OpenSHMEM+X, with X being CUDA/OpenMP/OpenACC, were either focusing on traditional thread-based parallelism, or did not enable RDMA for GPU and had to use staging buffers. Other PGAS languages that have support for heterogeneous computing include X10 [29] and UPC++ [30].

VI. Conclusions and Future Work

In this paper, we have presented the design and implementation of a set of interoperability extensions for OpenSHMEM and OpenMP device offloading to support transparent remote access of GPU device memory. Using our extended runtime systems, developers of hybrid OpenSHMEM+OpenMP applications only need a minimum amount of vendor-agnostic code to enable point-to-point and collective operations between GPUs, eliminating the need for staging communication buffers. Additionally, our design reuses existing work done by LLVM and UCX for interacting with vendor-specific APIs, this simplifies the OpenSHMEM runtime design and increases portability. Through these interoperability improvements, our extensions make OpenSHMEM better for programming heterogeneous supercomputers, which have become mainstream in the exascale era. In experiments, we observe significant speedups over standard OpenSHMEM+OpenMP implementations: 6 times better point-to-point latency, 1.3 times faster collective operations, 4.9 times higher random access throughput, 12.5% better strong scalability in FFT, and finally 8% better strong scalability in solving 3D heat equations.

For future work, we can improve our OpenMP device offloading extension, so that the users can selectively enable RDMA for accelerator communication buffers, and let OpenMP use vendor memory management API for other mapped buffers. Also, we can evaluate our work on AMD Instinct GPUs and Intel Xe GPUs with real-world HPC applications to provide a stronger argument for its performance portability across different vendors, and fix bugs and issues we encounter in the porting process. Finally, we could push for tighter integration between OpenSHMEM and OpenMP, by letting LLVM/Clang recognize OpenSHMEM communication routines, and replace remote device-to-device transfers with NVSHMEM or ROC_SHMEM calls. By switching from CPU-initiated communications to GPU-initiated communications, we could get even lower latencies, which should lead to even better strong scalability in applications.

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] "Top 500," https://www.top500.org/.

[2] L. Li and B. Chapman, "Compiler assisted hybrid implicit and explicit gpu memory management under unified address space," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356141

[3] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, Nov 2021, version 5.2. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[5] B. M. Chapman, T. Curtis, S. Pophale, S. W. Poole, J. A. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *PGAS*, 2010.

[6] "OpenSHMEM Application Programming Interface Version 1.5," http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf.

[7] N. Namashivayam, B. Cernohous, K. Kandalla, D. Pou, J. Robichaux, J. Dinan, and M. Pagel, "Symmetric memory partitions in openshmem: A case study with intel knl," in *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, M. Gorentla Venkata, N. Imam, and S. Pophale, Eds. Cham: Springer International Publishing, 2018, pp. 3–18.

[8] S. Bak, C. Bertoni, S. Boehm, R. D. Budiardja, B. M. Chapman, J. Doerfert, M. Eisenbach, H. Finkel, O. R. Hernandez, J. Huber, S. Iwasaki, V. Kale, P. R. C. Kent, J. Kwack, M. Lin, P. Luszczek, Y. Luo, B. Pham, S. Pophale, K. Ravikumar, V. Sarkar, T. Scogland, S. Tian, and P. K. Yeung, "OpenMP application experiences: Porting to accelerated nodes," *Parallel Computing*, vol. 109, p. 102856, 2022.

[9] S. Tian, J. Doerfert, and B. M. Chapman, "Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads," in *Languages and Compilers for Parallel Computing Workshop (LCPC)*, 2020, pp. 41–56.

[10] J. Doerfert, A. Patel, J. Huber, S. Tian, J. M. M. Diaz, B. M. Chapman, and G. Georgakoudis, "Co-Designing an OpenMP GPU Runtime and Optimizations for Near-Zero Overhead Execution," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 504–514.

[11] "GPUDirect RDMA - Direct Communication between NVIDIA GPUs." https://developer.nvidia.com/gpudirect.

[12] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D. Panda, "Extending openshmem for gpu computing," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1001–1012.

[13] "Open Source Software Solutions (OSSS) OpenSHMEM Implementation on top of OpenUCX (UCX) and PMIx." https://github.com/openshmem-org/osss-ucx.

[14] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "UCX: An Open Source Framework for HPC Network APIs and Beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.

[15] "Unified Communication Collectives Library." https://github.com/openucx/ucc.

[16] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, no. 10, p. 623–624, oct 1965. [Online]. Available: https://doi.org/10.1145/365628.365655

[17] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 80–89.

[18] "Open MPI: Open Source High Performance Computing." https://www.open-mpi.org/.

[19] "MPICH: a high performance and widely portable implementation of the Message Passing Interface (MPI) standard." https://www.mpich.org/.

[20] A. Patel and J. Doerfert, "Remote openmp offloading," *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022.

[21] K. Hamidouche, A. Venkatesh, A. A. Awan, H. Subramoni, C.-H. Chu, and D. K. Panda, "Exploiting gpudirect rdma in designing high performance openshmem for nvidia gpu clusters," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 78–87.

[22] K. Hamidouche, A. Venkatesh, A. Ahmad Awan, H. Subramoni, C.-H. Chu, and D. K. Panda, "Cuda-aware openshmem: Extensions and designs for high performance openshmem on gpu clusters," *Parallel Computing*, vol. 58, pp. 27–36, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819116300345

[23] S. Potluri, D. Rossetti, D. Becker, D. Poole, M. Gorentla Venkata, O. Hernandez, P. Shamis, M. G. Lopez, M. Baker, and W. Poole, "Exploring openshmem model to program gpu-based extreme-scale systems," in *Revised Selected Papers of the Second Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies - Volume 9397*, ser. OpenSHMEM 2015. Berlin, Heidelberg: Springer-Verlag, 2015, p. 18–35. [Online]. Available: https://doi.org/10.1007/978-3-319-26428-8_2

[24] S. Potluri, A. Goswami, D. Rossetti, C. Newburn, M. G. Venkata, and N. Imam, "Gpu-centric communication on nvidia gpu clusters with infiniband: A case study with openshmem," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 253–262.

[25] K. Hamidouche and M. LeBeane, "Gpu initiated openshmem: Correct and efficient intra-kernel networking for dgpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 336–347. [Online]. Available: https://doi.org/10.1145/3332466.3374544

[26] M. Grodowitz, E. F. D'Azevedo, S. Powers, and N. Imam, "Using hybrid model openshmem + cuda to implement the shoc benchmark suite," in *OpenSHMEM*, 2016.

[27] M. B. Baker, S. S. Pophale, J.-C. Vasnier, H. Jin, and O. R. Hernandez, "Hybrid programming using openshmem and openacc," in *OpenSHMEM*, 2014.

[28] W. Lu, T. Curtis, and B. Chapman, "Enabling low-overhead communication in multi-threaded openshmem applications using contexts," in *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, 2019, pp. 47–57.

[29] D. Cunningham, R. R. Bordawekar, and V. A. Saraswat, "Gpu programming in a high level language: compiling x10 to cuda," in *X10 '11*, 2011.

[30] D. Waters, C. A. MacLean, D. Bonachea, and P. Hargrove, "Demonstrating upc++/kokkos interoperability in a heat conduction simulation (extended abstract)," *PAW-ATM2021: Parallel Applications Workshop, Alternatives to MPI+X, 19 Nov 2021*, 11 2021. [Online]. Available: https://www.osti.gov/biblio/1820128