

Exploring OpenSHMEM Model to Program GPU-based Extreme-Scale Systems

Sreeram Potluri¹, Davide Rossetti¹, Donald Becker¹, Duncan Poole¹,
Manjunath Gorentla Venkata²(✉), Oscar Hernandez², Pavel Shamis²,
M. Graham Lopez², Mathew Baker², and Wendy Poole³

¹ NVIDIA Corporation, Santa Clara, USA

² Extreme Scale Systems Center (ESSC),
Oak Ridge National Laboratory (ORNL), Oak Ridge, USA
manjugv@ornl.gov

³ Open Source Software Solutions, Knoxville, USA

Abstract. Extreme-scale systems with compute accelerators such as Graphical Processing Unit (GPUs) have become popular for executing scientific applications. These systems are typically programmed using MPI and CUDA (for NVIDIA based GPUs). However, there are many drawbacks to the MPI+CUDA approach. The orchestration required between the compute and communication phases of the application execution, and the constraint that communication can only be initiated from serial portions on the *Central Processing Unit* (CPU) lead to scaling bottlenecks. To address these drawbacks, we explore the viability of using *OpenSHMEM* for programming these systems. In this paper, first, we make a case for supporting GPU-initiated communication, and suitability of the *OpenSHMEM* programming model. Second, we present *NVSHMEM*, a prototype implementation of the proposed programming approach, port Stencil and Transpose benchmarks which are representative of many scientific applications from MPI+CUDA model to *OpenSHMEM*, and evaluate the design and implementation of *NVSHMEM*. Finally, we provide a discussion on the opportunities and challenges of *OpenSHMEM* to program these systems, and propose extensions to *OpenSHMEM* to achieve the full potential of this programming approach.

1 Introduction

GPUs have become ubiquitous in High Performance Computing (HPC) clusters. Owing to the superior performance per watt that the GPU architecture delivers,

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan(<http://energy.gov/downloads/doe-public-access-plan>).

they will be a key component in the road to exascale computing. A wide range of scientific applications, many of which use the Message Passing Interface (MPI), have been ported to take advantage of GPUs. They typically use the MPI + *Compute Unified Device Architecture* (CUDA) model. CUDA is used to offload compute portions of the application onto the GPU. MPI [1] two-sided communication API is used from the CPU to manage data movement. This leads to distinct phases in the application: for computation, that run on GPU, and for communication, that run on the CPU. The transition between these phases is managed by the CPU and involves waiting for MPI communication to complete before launching CUDA kernels or waiting for CUDA kernels to complete before initiating communication.

The CPU-controlled execution model on GPU clusters incurs overheads that limit the strong scalability of applications. Strong scaling where the problem size is fixed while the execution resources are increased is a critical metric of efficiency of applications and the execution environment. There is an overhead associated with launching compute kernels on the GPU. Alternating compute and communication phases result in recurring launch kernels and lead to recurring overheads. The synchronizing activity on the GPU before MPI communication leads to under-utilization of GPU resources during communication and synchronization phases. Similarly, there is under-utilization of the network when computation is in progress. Some of these issues are handled by restructuring the application code to overlap independent compute and communication phases, using CUDA streams. These optimizations make the application code complex and their benefits usually diminishes as the problem size per GPU becomes smaller [2].

One of the approaches to address these limitations is by adding the capability to initiate communication on the GPUs. Self reliance of the GPU to initiate and complete communication without synchronizing with the CPU can result in long running kernels, providing better utilization of the GPU while avoiding launch and synchronization overheads. Communication from within CUDA kernels results in a highly concurrent, finer-grained communication model in applications. Using the MPI two-sided model for GPU-initiated communication requires GPUs to perform message matching, and deal with unexpected messages. To take advantage of GPU-initiated communication while overcoming the disadvantages of two-sided communication, we explore the viability of using one-sided communication model such as *OpenSHMEM* for GPU-initiated communication.

OpenSHMEM [3] specification defines a standard *Application Programming Interface* (API) that provides portability to applications that have traditionally used various vendor-specific SHMEM libraries. While the current standard provides an API for library initialization, data object management, communication and synchronization, it is still evolving to provide API and semantics for use of *OpenSHMEM* in multi-threaded environments. The effective use of *OpenSHMEM* on the GPU will require further extensions. For example, implementing the current *OpenSHMEM* semantics requires a Total Store Ordering(TSO) memory model on the target platform. This has to be relaxed to enable its use on GPUs which has a highly relaxed memory model. The current ordering and

completion API in OpenSHMEM apply to all communication initiated by the calling PE. In a highly-threaded environment, finer grained control of communication is required for efficiency and overlap.

In this paper, we first explore the advantages of using the *OpenSHMEM* model for GPUs. Then, we provide a prototype implementation, and evaluate the opportunities and challenges of this model. At the end, we provide a discussion to address the drawbacks and propose extensions to the *OpenSHMEM* programming model that could address the drawbacks.

2 Background

2.1 Current Programming Model Approach for GPU-based Systems

Most HPC applications ported to clusters accelerated with NVIDIA GPUs currently use an MPI+CUDA hybrid programming model. The application is split into phases of communication and computation with CPU orchestrating their execution. Computation is typically offloaded onto the GPUs while MPI communication is managed from the CPU. To demonstrate the use of this programming approach, consider the stencil example listed in Fig. 1. The data grid is split into interior and boundary portions to achieve overlap between computation and communication. The execution in each iteration goes through the following steps:

Traditional	Envisioned
<pre> Loop { interior_compute <<<..., stream0>>>(...) pack_boundaries<<<..., stream1>>>(...) cudaStreamSynchronize (stream1) Exchange (MPI/OpenSHMEM) unpack_boundaries<<<..., stream1>>>(...) boundary_compute<<<..., stream1>>>(...) cudaDeviceSynchronize(); } </pre>	<pre> compute_exchange<<<...,stream0>>>(...) </pre>

Fig. 1. Execution models: traditional and envisioned

1. A CUDA kernel is launched via a CUDA stream to compute interior of the data grid.
2. While this is in progress, another CUDA kernel is launched to pack the boundary cells. This is launched on a different stream. Both these CUDA streams are asynchronous from the CPU's perspective. The pack kernel is launched on

the stream with higher priority, so that it does not block behind the CUDA kernel computing the interior. This enables better overlap between computation and data exchange.

3. `cudaStreamSynchronize`, a blocking synchronization call on CPU, checks the completion of the packing kernel.
4. After the completion of packing, MPI (or other communication API) is used to exchange data between processes. The data exchange pattern can be classified as near-neighbor exchange. The CPU blocks until the MPI communication is complete.
5. After the data is exchanged, two CUDA kernels are used for processing the data. The first kernel unpacks the exchanged data, and the second kernel computes the boundary portion.
6. On the last step, the call to `cudaDeviceSynchronize` ensures the completion of computation on the GPU. The `cudaDeviceSynchronize` is a blocking call on the CPU and marks the end of an iteration

As this example illustrates, the existing model requires frequent synchronization between GPU and CPU. The CPU has to be running at full-speed to ensure fast synchronization and hence will stay in a high power state even though it does little useful work. The synchronization phase at end of each iteration requires the GPU to be completely drained before starting the next iteration which reduces the utilization of the GPU and also kills any opportunity of data locality and reuse. Further, when the application is scaled strongly, the interior compute time and hence the opportunity for overlap decreases. The application runtime is limited by the latency for CPU-GPU synchronization which includes: launch and synchronization of CUDA kernels and MPI communication. The envisioned approach is explained in Sect. 3.

2.2 OpenSHMEM

OpenSHMEM is a PGAS library interface specification. It includes routines, environment variables, and constants to implement the PGAS programming model as a library. It also defines bindings in C and Fortran languages, enabling the applications to invoke the interfaces.

OpenSHMEM presents a PGAS view of execution contexts and memory model. The execution contexts in *OpenSHMEM* is an OS process identified by integer called Processing Element (PE). An *OpenSHMEM* program has private address space and shared address space. A PE allocates and stores its private data and control structures in the private address space. The shared address space in *OpenSHMEM* is presented as symmetric objects, which are accessible by all PEs in an *OpenSHMEM* program. The symmetric objects are allocated using a collective allocation operation in *OpenSHMEM*.

OpenSHMEM provides routines for communication and synchronization with other PEs. The communication in *OpenSHMEM* is primarily one-sided. It provides many variants of Put, Get and Atomic operations to access and modify symmetric data objects that are located on remote PEs. It provides Quiet and Fence which complete communication and orders communication, respectively.

It provides interfaces for collective communication and synchronization. The group of PEs involved in a collective communication is defined by three integers called Active Set. The collectives include barrier, reductions, and data gather operations.

3 Our Approach: GPU-Initiated Communication using *OpenSHMEM*

We propose a programming approach using GPU-initiated communication which is motivated by the throughput oriented architecture of the GPU. The GPUs are designed to support tens of thousands of threads to achieve maximum parallel throughput. These threads are extremely light weight and thousands of threads are queued up for work (in groups called warps). If one warp must wait on a memory access, another warp can start executing in its place. As a separate set of registers is allocated for all active threads, there is no need for swapping of registers or state. As a consequence, this execution model has inherent latency hiding capabilities with minimal scheduling overheads. With the increasing amount of parallelism, GPU architectures can have enough state to hide latencies not only to local GPU device memory but also to remote GPU memory over a network. GPU-initiated communication can be used to take advantage of this inherent capability of the GPU hardware while relying on the CUDA programming paradigm that has been used for scaling within a GPU. Further, this improves programmability as developers will not have to rely on a hybrid model to orchestrate and overlap between different phases of the application.

Figure 1 shows the contrast of the CPU code of stencil application using the hybrid model and the model with GPU-initiated communication. The communication initiated and synched from within CUDA kernels will not only reduce the reliance on the CPU, additionally it also avoids existing synchronization overheads that limit the strong scaling. Also, to enhance the efficiency within a warp and reduce the pressure on the memory sub-system, the loads and stores can be coalesced by the hardware when alignment and access pattern requirements are met.

Using PGAS programming model such as *OpenSHMEM* suits GPU-initiated communication better. The one-sided communication model of *OpenSHMEM* requires only the caller (origin) of the interface to be active, which matches the massive parallelism and dynamic scheduling model on the GPU. Further, a combination of global address space approach and semantics of *OpenSHMEM* communication interfaces such as Put and Get is a close match to load and stores on shared-memory space of GPU.

4 *NVSHMEM*: A Prototype for GPU-Initiated Communication using *OpenSHMEM*

4.1 Overview of Implementation

NVSHMEM is a prototype implementation of *OpenSHMEM* with support for GPU-initiated communication. We do not depart significantly from the

OpenSHMEM execution model that is currently used on non-GPU clusters. Currently, each PE can be implemented using an OS process. All the resources and state associated with the use of a GPU by a host process or thread is encapsulated in a CUDA context. A context is associated with a single GPU device and only one context can be active in a OS process/thread at a given time. In our *OpenSHMEM* execution model, we extend the scope of a PE to include an OS process or thread and a CUDA context. This respects the execution boundary of the GPU and the state boundary (device address space, etc.) of the CUDA context. We consider a single symmetric heap per PE, that will be enabled by the unified memory capability of NVIDIA GPUs. We started with an intuitive separation of API that is supported on the host from that is supported on the GPU. In the current version of *NVSHMEM*, API for runtime initialization/termination, memory management and collective communication is supported from serial portions on the host. API for one-sided communication, one-sided ordering/completion, and point-to-point synchronization interfaces are supported from the inside parallel regions, on the GPU. *NVSHMEM* can be seen as a subset of the *OpenSHMEM* standard and follows the semantics defined in the standard except for a few exceptions. It assumes multi-threading support in *OpenSHMEM*, allowing all the *OpenSHMEM* calls supported on the GPU to be made concurrently by multiple threads. Current *OpenSHMEM* standard assumes a strongly consistency memory model on the architectures it is implemented on. This is relaxed in the *NVSHMEM* implementation to allow implementation on the GPU which has a highly relaxed memory model. This extension is discussed in more detail in Sect. 5.5.

NVSHMEM data movement interfaces are implemented by leveraging CUDA data movement mechanisms. CUDA has a protocol built on top of PCIe that enables direct data movement between GPUs that are connected to the same PCIe root complex. This data path completely bypasses CPU/system memory avoiding additional copies. This is referred to as the P2P protocol in CUDA and was originally limited to inter-GPU data movement within a single process. Since CUDA 4.2, it has been extended to support inter-process communication. The capability is exposed to the developer through CUDA IPC (Inter-Process Communication) API. We have used these existing technologies to build an emulation platform for experimenting with GPU initiated communication. With this environment, we can scale unto 8 GPUs connected to the same PCIe root complex. Technologies like NVLink [4] and PLX Express Fabric [5] are intended to allow larger number of GPUs to be deployed in this fashion.

Akin to shared memory on Linux clusters, CUDA IPC provides API for one process to map GPU device memory allocated by another process, into its own address space. Once the mapping has been established it can access the memory as it would access its own. The handle creation and mapping mechanism are exposed through the *cudaIpc** interface provided by the CUDA toolkit. In our *NVSHMEM* implementation, symmetric heap allocation and setup happens as part of the initialization function (*shmем.init*). Each process allocates its heap in device memory and shares the IPC handle to all other process. An all-to-all

mapping of the heaps of all processes is established. Memory allocation is managed by returning symmetric chunks of memory from the heap. *NVSHMEM* also supports `shmem_ptr` API, which exposes remote symmetric regions for direct loads and stores in the application code. This is implemented as a simple address translation using displacement of the pointer in the local heap and calculating the corresponding pointer in the remote heap that was mapped using CUDA IPC. *NVSHMEM* also provides collective communication operations which are implemented as CUDA kernels.

As GPU-initiated communication is usually fine-grained, we use direct loads/stores to implement Put/Get routines in *NVSHMEM* to avoid the overheads of making the CUDA API calls for small transfers. The `shmem_quiet` call is implemented as a `__threadfence_system()` call in CUDA. This waits until all prior memory requests have been performed with respect to all clients, including those over PCIe. Point-to-point synchronization calls such as `shmem_wait` and `shmem_wait_until` are implemented as active polling loops on the memory locations and have to include a call to `__threadfence_system()` in order to guarantee ordering of loads. Both `shmem_quiet` and `shmem_fence` have similar implementations on our multi-GPU platform.

4.2 Evaluation of the Prototype Implementation

To evaluate the design and implementation of *NVSHMEM*, we ported 2dstencil and Transpose kernels to use *OpenSHMEM* with GPU-initiated communication. These kernels are representative of commonly found computation and communication patterns in several scientific applications. Rest of the section details the implementation of kernels and its performance characteristics.

2DStencil: Fig. 2 shows the use of *NVSHMEM* in the 2DStencil code. The code snippet on the left shows the CPU code for setting up global memory and code snippet on the right shows the use of *NVSHMEM* inside the CUDA kernel. In the traditional version of the 2dstencil code, as shown in Fig. 1, computation of the interior and the boundary elements are separate to allow for overlap between a major chunk of computation and communication. The overlap is achieved using multiple streams in CUDA. The traditional version involves four kernel launches per step: for interior computation, for boundary pack-unpack and for boundary compute. There also has to be explicit CPU-based synchronization between the CUDA kernels and MPI communication. These add performance overheads and code complexity.

In the *NVSHMEM*-based implementation, the host code involves a single kernel launch that runs for the lifetime of the application. It does not require any further intervention from the CPU. The communication and synchronization are expressed as part of the kernel, using *OpenSHMEM* API. Boundary threads fetch data from the neighboring processes using `shmem_float_get` and kernels across neighboring processes synchronize at the end of each step using `shmem_float_wait`. In the current version of the code, a hierarchical synchronization is implemented with one thread per block calling `shmem_int_wait` while other

```

...
u = (void *) nvshmmalloc (size)
v = (void *) nvshmmalloc (size)
sync= (void *) nvshmmalloc (sizeof(int)*npeers)
...
stencil_kernel <<< >>> (u, v, sync ...)
...
cudaDeviceSynchronize();

```

```

__global__ void stencil_kernel (u, v, sync, ...) {
    i = blockIdx.x*blockDim.x + threadIdx.x;
    for (...) {
        if (i+1 > nx) {
            v[i+1] = nvshmem_float_g (v[1], rightpe)
        }
        if (i-1 < 1) {
            v[i-1] = nvshmem_float_g (v[nx], leftpe)
        }
        u[i] = (v[i+1] + v[i-1]) . . . //compute
        /*peers array has left and right PE ids*/
        if (i < 1 || i > n-2) {
            nvshmem_int_p (sync[i], 1, peers[i]);
            nvshmem_quiet();
            nvshmem_wait_until (sync[i], EQ, 1);
        }
        //intra-process sync (code not shown)
        //compute v from u (code not shown)
    }
}

```

Fig. 2. Using GPU-initiated communication: example with *NVSHMEM*

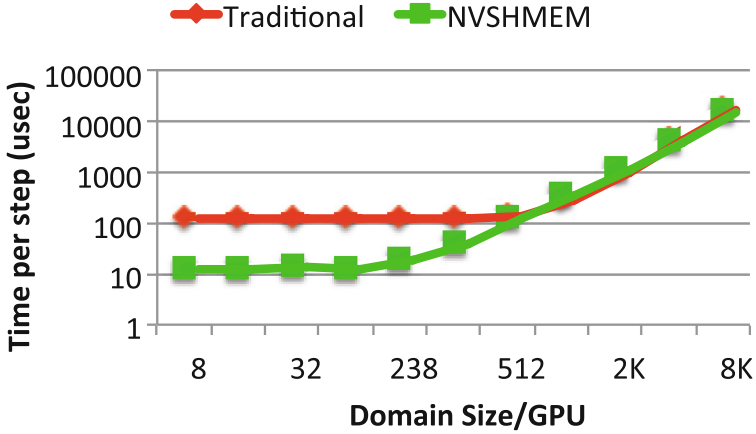


Fig. 3. Performance of 2D stencil code using traditional model and using *NVSHMEM*

threads block on an intra-block synchronization primitive. This is to reduce the number of global memory requests that are generated. The communication pattern between these two versions of the code changes from sparse coarse-grained to dense fine-grained. Further, the overlap between compute and communication changes from an explicit programmer managed design to an implicit GPU thread-scheduler managed one.

Figure 3 shows the preliminary results, comparing the performance of the traditional version of the 2D stencil code with the version using GPU-initiated

communication. Due to the non-preemptive execution of CUDA blocks, we have to be careful to avoid deadlocks when using inter-block or inter-kernel synchronization using point-to-point synchronization primitives like `shmem_int_wait`. We workaround this in our experiments by limiting the number of blocks equal to the number of Streaming Multiprocessors (SMs) available. This constraint of non-preemptive execution is a limitation only the current GPU architectures. For our experiments, we use a system with K40m GPUs, and we restrict the number of blocks to 15 for all our runs. From the performance results, we can observe that the version using GPU-initiated communication shows a significant improvement in per-step time, when the problem size per GPU is small (< 512). This improvement is a consequence of avoiding the overheads of multiple kernel launches and the use of CUDA API. Moreover, when strong scaling the applications on large GPU clusters, avoiding these overheads is critical for the application performance.

Transpose: Matrix transpose forms the basis of Fast Fourier transform (FFT) transformations and is therefore widely used in signal and image processing. We have ported a multi-GPU transpose code that traditionally uses MPI+CUDA to use *NVSHMEM*.

In the MPI+CUDA version, the algorithm involves pipelining three steps: local transpose using a CUDA kernel, all-to-all data transfers between processes using MPI, and copying data to the target location using `cudaMemcpy2D`. Note that it is important to perform the local transpose prior to the all-to-all. Doing the transpose upfront places memory to be sent to other GPUs in contiguous memory, thus ensuring an efficient transfer. This code is complex as it involves streams, non-blocking MPI communication, and asynchronous CUDA memory copies required to achieve its pipelining.

The *NVSHMEM* version uses fine-grained communication to implement the transpose. Remote data is fetched as required by using *OpenSHMEM* API from inside the CUDA kernel. To make the implementation simple, `shmem_double_get` calls are used to fetch the data, whether it is local or remote. As these calls translate into direct memory loads underneath, the overheads are negligible. The host code is trivial and involves a single kernel launch.

The performance of these two versions was compared on a node with three NVIDIA Tesla K40 GPUs connected to the same socket. Figure 4 shows the bi-directional PCIe bandwidth achieved as the size of matrix increases. We can see that the version using GPU-initiated fine grained communication is able to better saturate the PCIe bandwidth for smaller problem sizes than the MPI version. This is because of the overhead in CUDA and MPI API calls and overheads of DMA copies. Saturation of the PCIe channel shows that a network with higher bandwidth will benefit the performance of MGTranspose using *NVSHMEM*. The MPI version is not able to saturate the bandwidth even for larger matrix sizes because of a limitation in the current CUDA runtime. CUDA copies involving a buffer on a remote GPU (one not actively used by the calling process) are unable to use direct P2P transfer and are instead staged through system memory. This issue does not happen with *NVSHMEM* as the transfers are implemented using memory accesses rather than as CUDA memory copies.

Further, achieving a pipeline in the MPI version introduces significant code complexity. The number of lines of code for the transpose function is 280 in this version compared to 100 in the *NVSHMEM* version, and this actually understates the complexity of the MPI implementation compared with the *NVSHMEM* version.

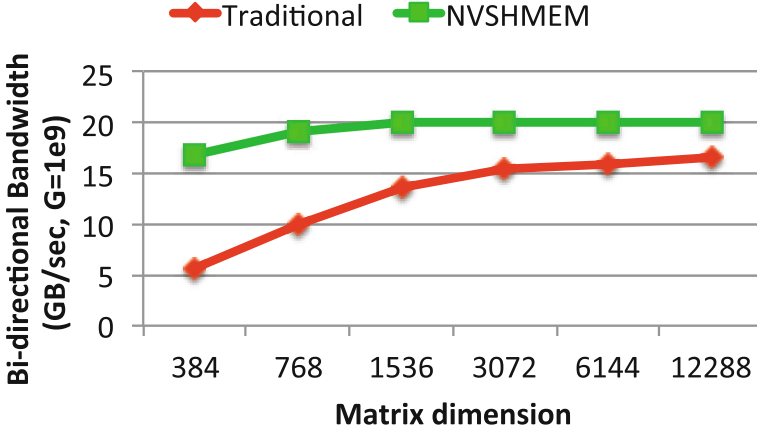


Fig. 4. Performance of Transpose code using traditional model and using *NVSHMEM*

5 Extending *OpenSHMEM* to Better Support GPU-Initiated Communication

This section highlights the extensions required to efficiently implement the *OpenSHMEM* model for GPU-based extreme scale systems with GPU-initiated communication. Firstly, the *OpenSHMEM* specification provides ordering guarantees that can only be implemented on systems with a TSO or stronger memory model. GPU on the other hand provides a highly relaxed memory model. This prevents *OpenSHMEM* to be implemented on the GPU in its current form. Secondly, the ordering and completion API apply to all the operations issued by the PE. This can be restrictive and expensive in a highly threaded environment. We see the need for finer grained isolation and control of operations within a PE. How collective operations behave within a threaded region is also not defined in the *OpenSHMEM* standard. The rest of the section explains the issues listed here in more detail and proposes extensions that can address them.

5.1 Memory Consistency Model in *OpenSHMEM*

In *OpenSHMEM*, the order in which updates become visible at the target PE (`shmem.put`) is controlled by the source PE, with ordering (`shmem.fence`) and completion (`shmem.quiet`) interfaces. No explicit *OpenSHMEM* operations are expected by the user to guarantee the correct visibility ordering on the target PE. Figure 5 shows the consistency guarantees *OpenSHMEM* provides when implementing a message passing idiom.

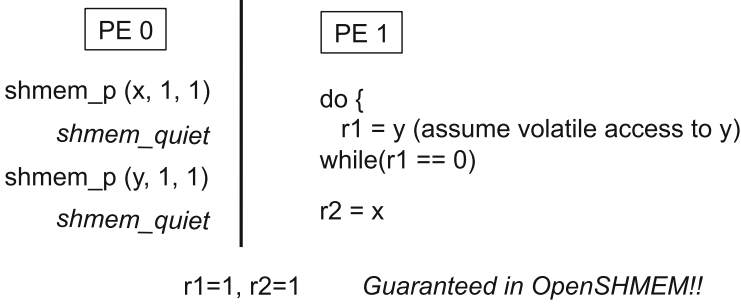


Fig. 5. Message passing idiom implemented using *NVSHMEM*

These semantics cannot be guaranteed on systems like NVIDIA GPUs, IBM Power and ARM64, which have relaxed memory models. Additional operations are required at the target PE to guarantee visibility ordering. Implementation of the same message passing idiom using CUDA to run on a NVIDIA GPU is shown in Fig. 6. Explicit memory barrier instruction is required at the target to prevent reordering of loads that can result the update to flag being observed before the update to data. Similar instructions (lwsync or isync) are required on IBM Power architecture. *OpenSHMEM* does not provide a way to express this using portable API. This forces applications to use platform specific ordering instructions in conjunction with *OpenSHMEM* leading to non-portable code.

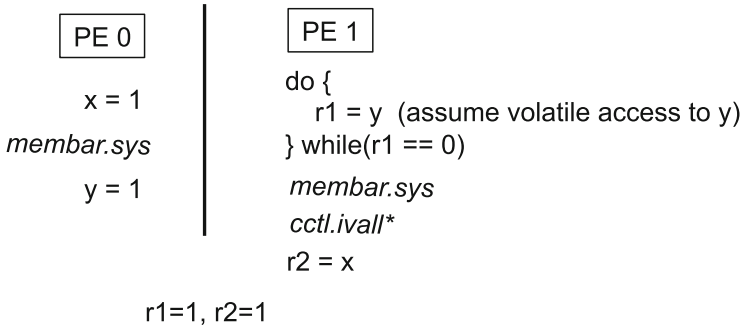


Fig. 6. Message passing idiom Implemented in CUDA

One way to address this is to require the target PE to use `shmem_wait` (`_until`) to poll on a flag in order to guarantee order it sees between any prior updates and update on the flag. Figure 7 shows the message passing idiom using `shmem_wait_until`. This call translates to the platform specific ordering instructions on the GPU.

While this extension to semantics solves the problem in the message passing pattern, we believe there is need for a holistic solution to express ordering

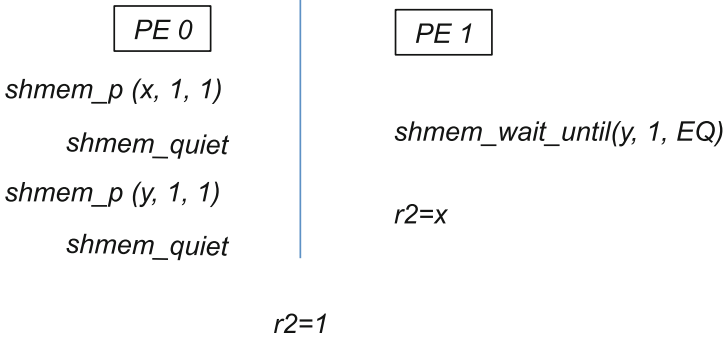


Fig. 7. Message passing idiom implemented using `shmем_wait_until`

in *OpenSHMEM*. For example, the ordering operation in the current standard, `shmем_fence`, does not identify the symmetric variable that is used for synchronization, for example `y` in Fig. 7. This information can allow the runtime to enforce ordering in a more efficient ways, e.g. setting `IBV_SEND_FENCE` flag on RDMA write to `y` in InfiniBand, use *STRL* to perform store release on `y` in ARMv8. Further, `shmем_fence` only guarantees ordering between updates to the same PE. The only way to guarantee consistency between updates to multiple PEs is by using `shmем_quiet` to complete all operations in-flight. Programming languages (C11, C++11 and Java5+) have converged on SC-DRF (Sequential Consistency - Data Race Free) as the default memory model [6]. This guarantees sequentially consistent behavior for data-race-free programs. *OpenSHMEM* standard can learn from these efforts and provide interfaces like the following which allows the flexibility to provide different ordering requirements around synchronization operations and variables.

```
void shmем_int_p_fence (int* x, int value, int pe, int ordering)
int shmем_int_g_fence (int* x, int pe, int ordering)
void shmем_int_wait_until_fence (int* x, int cond, int value, int ordering)
```

Possible values for ordering can be relaxed, release and acquire. Relaxed mode does not guarantee any ordering and the interfaces can be used only for synchronization. Release guarantees that all prior Put, AMOs, and memory store operations to symmetric data objects are delivered before the Put in `shmем_int_p_fence`. This can be used at the sender PE when updating `y` in the message passing example above. Acquire prevents reordering of later Get and memory load operations with the memory read(s) in `shmем_int_g_fence/shmем_int_wait_until_fence` operation. This can be used at the receiver PE when reading `y` in the message passing example. Such an API associates the synchronization variable with the synchronization operation allowing for optimizations as mentioned above. It also makes the API extensible to provide different ordering guarantees.

5.2 Support for Memory Locality

The future extreme scale systems are expected to have heterogeneous memories with hierarchies, including DRAM, HBM (High Bandwidth Memory), and NVRAM (Non-volatile RAM). The performance characteristics of the memories such as the access latency, bandwidth, and capacities varies. Further, for an extreme scale system with GPUs, the internal memory is hierarchical with global memory that is shared across SMs and shared memory which is local to a specific block(CTA) of threads. The latencies to these memories varies significantly.

The current *OpenSHMEM* model assumes that all memory is homogeneous and is flat with no hierarchies. Following this model, for GPU based systems, we have a single symmetric heap per PE, enabled by the unified memory capability on NVIDIA GPUs. While a single heap spanning the CPU and GPU provides simplifies programming model, the locality of data in the heap impacts the performance as we deal with multiple physical memories and migration between them. Thus, implementing *OpenSHMEM* memory model for these systems can result in a negative performance impact.

To achieve performance efficiency on emerging hardware architectures, the memory allocation and access interfaces of *OpenSHMEM* should provide a way to express explicit control over the locality of the symmetric variables. The user can provide hints to the implementation about the placement of symmetric variables, and as such set expectations of it's performance. For GPU-enabled systems, this would enable the user to place the memory primarily accessed by code on GPU on device memory, and memory primarily accessed by code on CPU on DDRAM.

5.3 Support for Fine-Grained Synchronization

In the *OpenSHMEM* model, all operations are completed when a `shmem_quiet` is invoked. Using this interface and its semantics for GPU translates to completing all `shmem_put` operations issued by all threads associated with the PE. Many applications involving stencil operations, near-neighbor communication, sparse communications require more fine-grained synchronization primitives. For example, in a stencil application, computation on a boundary is split among multiple groups of threads (blocks or warps). The communication and computation in one group of threads can progress independently from that handled by another group of threads. Such patterns can benefit from finer-grained isolation, ordering and completion of communication than a PE. The completion of communication issued from the threads of CTA/warp can allow for better overlap between different CTAs.

Independent Communication Streams with Contexts: Context is a programming construct, which allows independent streams of one-sided communications to be grouped together. This concept was introduced by Dinan et al. as an approach to achieve communication-computation overlap in single-threaded *OpenSHMEM* programs, and to avoid thread-interference and resource isolation

in multi-threaded *OpenSHMEM* programs [7]. We propose to use a similar construct to achieve fine-grained synchronization for GPU threads. However, we propose to add the flexibility to define a context based on a memory region. In distributed shared memory environments, we believe it can be natural for the user to isolate traffic based on the memory region being operated on rather than explicitly identifying all the operations accessing it. This can group the threads collaborating on a data region and complete their communication stream independent of other threads.

5.4 Multi-threaded *OpenSHMEM*

The current *OpenSHMEM* specification does not specify the interaction of *OpenSHMEM* and user-threads. In this absence, the thread-safe invocation of *OpenSHMEM* interfaces in the multi-threaded *OpenSHMEM* program is undefined i.e., implementing PE as a thread or invoking *OpenSHMEM* interfaces from multiple threads concurrently does not guarantee correct *OpenSHMEM* semantics. This is a significant bottleneck for using *OpenSHMEM* for many-thread systems such as GPUs, Xeon Phi, and CPUs with hundreds of hardware threads. The performance advantages of multi-thread support in the other programming model is well documented [8].

In NVSHMEM, we experiment with using one PE per GPU and invoking *OpenSHMEM* interfaces from multiple CUDA threads within the PE. Thus, we present a use case for thread-safe invocation of *OpenSHMEM*. The interfaces and semantics proposed by Cray [9] accommodates most of the needs for implementing *NVSHMEM*. However, we believe the requirement that each thread has to call `shmem_thread_register` before making *OpenSHMEM* calls is too restrictive in a highly parallel and dynamic thread environment like the GPU. We think the ability of threads to initiate communication should be implicit once the thread safety level of the runtime has been specified during initialization. Any information that might help optimize the runtime (number of threads, for example) can be provided as hints during the initialization. Next subsection discusses the interfaces and semantics of collectives invoked from a multi-threaded *OpenSHMEM* program.

5.5 Collectives Extensions to Operate in Thread Environment

Assuming the thread model proposed by [9] is a part of the specification, we propose two extensions to the semantics of collectives to operate in the presence of the threads.

Collectives with Thread Collaboration: In current *OpenSHMEM* standard (and even after including thread models similar to MPI), a collective is initiated through a single call from a PE. In a highly-threaded environment, this typically means, synchronizing among all the threads in a PE before allowing one thread to call into the collective, as depicted in Fig. 8. Enforcing these synchronization points can be expensive and can bring down the overall utilization of highly

parallel environments such as GPU. Parallelism can be spawned transparently within the collective but this can lead to additional overheads, and deadlocks due to scheduling and interaction between runtime and the application.

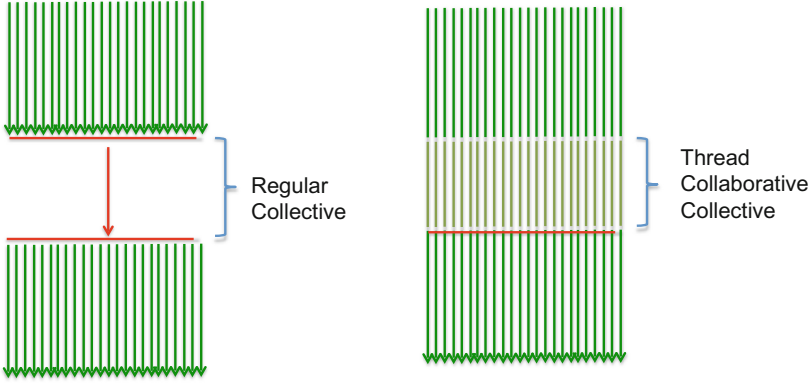


Fig. 8. Making collective a collaborative threaded operation

To address this, we propose that the *OpenSHMEM* allow multiple threads to call into a collective and thus explicitly declaring their participation. This allows the collective to be optimized using the parallelism while avoiding the synchronization point before the collective. Collectives in *OpenSHMEM* are one-sided which means the application guarantees that the target buffers on all the processes are ready to be used when the collective call is made on any process. Further in *OpenSHMEM* collectives, the transformation on each data element is independent of the transformation on other data elements at a PE. This allows parallelism to be naturally applied inside the collective without affecting the semantics or correctness. The collective implementation can suppress parallelism by partial synchronization after considering the hierarchy and topology in the hardware. On the GPU, scheduler can inherently overlap between the data movement and computation components of the collective happening at different groups of threads. This is not the same as each thread calling a collective independently as it would change the placement of data.

To demonstrate the usage, consider the example with *collect* interface. Invoking multiple independent collect operations gathers the data from all PEs for each collect operation into a contiguous location. This leaves data contributed by each PE into non-contiguous locations. This changes the nature of the collective and also prevents efficient coalescing due to non-contiguity of accesses. In a thread-collaborative collective, calls made by all the threads constitute one collective operation and result in a data arrangement where data contributed by each PE is contiguous, similar to that achieved when using a single collect operation.

Persistence in Collective Operations: To implement and optimize the collective, runtime needs information about the amount of parallelism used for a particular collective operation [10]. The synchronization of parallelism within a PE after the collective operation can be efficiently staggered and hidden if the collective is non-blocking and the application checks for its completion later in its run. These are possible with extensions in *OpenSHMEM* to define the collective ahead its use, to launch non-blocking collective and wait for its completion using request. The following API presents an example for collect operation. The parameter *ncalls* gives the number of threads/calls per PE that constitute the collective. The info argument can be used to pass information about the way collective is called among threads (for example, one call per CTA/warp).

```
void shmem_collect32_init (void *target, const void *source, size_t nelems,
int ncalls, int PE_start, int logPE_stride, int PE_size, long *pSync, shmem_info
info, shmem_request_t *request)
```

```
void shmem_start(request)
void shmem_wait(request)
```

6 Related Work

There have been a fair number of contributions from vendors and researchers that are geared towards enabling efficient communication from GPUs. NVIDIA's GPUDirect technologies [11,12] have helped reduce communication overheads by removing the need to copy data to CPU memory before it can be moved to other GPUs, within and across nodes. However, the CPU is still involved in initiating communication and synchronizing between computation and communication in the application. This demands a powerful CPU for best performance and incurs overheads that limit strong scaling. GPUDirect Async [13] addresses this by allowing GPU to trigger and synchronize network operations in order with compute kernels that are queued by the CPU. This will allow CPU to be put in low power states as computation and communication is progressed by the GPU. However, communication operations are scheduled only at kernel boundaries. Computation and communication phases have to be defined and queued from the CPU. This will result in multiple kernels and bulk synchronization phases whose overheads can dominate as applications are scaled strongly. In this paper, we consider communication from inside CUDA kernels as an alternative approach and show how it can enable better performance. It also improves programmability by being inline with the CUDA programming model and taking advantage of the native GPU execution model.

Several solutions have been proposed to enable efficient use of GPUs with programming models such as MPI and PGAS. Wang et al. and Potluri et al. [14,15] has proposed CUDA-aware MPI with MVAPICH2 [16] which allows use of standard MPI interfaces for moving data from GPU or host memories. They take advantage of unified virtual addressing (UVA) feature from NVIDIA. Ashwin et al. proposed use of datatype attributes to identify GPU memory in communication using standard MPI interfaces [17]. Potluri et al. proposed CUDA-aware approach for

OpenSHMEM [18]. The aforementioned works address CPU-initiated communication involving GPU device memory. Cunningham et al. have proposed the use of X10 to program GPUs and CPUs across clusters as part of their APGAS programming model [19]. Miyoshi et al. have contributed work that allows embedding MPI calls within GPU kernels in their FLAT GPU framework [20]. In our approach, we allow the GPU user to rely on the familiar CUDA programming model while using OpenSHMEM to express inter-GPU data movement inline with the CUDA model.

7 Conclusion

This paper explores the viability of using OpenSHMEM with GPU-initiated communication to program GPU-based extreme scale systems. To evaluate this approach, we design and implement, NVSHMEM, and also port compute kernels, Transpose and Stencil, from MPI+CUDA model to OpenSHMEM. Our results show that in addition to performance advantages, this approach has productivity advantages as it does not require orchestrating compute and communication phases. Based on the experience with NVSHMEM, we identify potential extensions to OpenSHMEM that can improve performance and usability of this model to program GPU-based extreme scale systems.

Acknowledgments. The work at NVIDIA is funded by U.S. Department of Energy under subcontract 7078610 with Lawrence Berkeley National Laboratory. The work at Oak Ridge National Laboratory (ORNL) is supported by the United States Department of Defense and used the resources of the Extreme Scale Systems Center located at the ORNL. In addition the authors would like to thank Stephen Poole (DoD) for his review of this work and many technical discussions that help shape the ideas presented in the paper.

References

1. The MPI Forum: MPI: A Message Passing Interface. Technical report (Version 3.0, 2012)
2. Tariq, S.: Lessons learned in improving scaling of applications on large GPU clusters. In: HPC Advisory Council Stanford Conference (2003)
3. OpenSHMEM Org.: OpenSHMEM Specification (2015). <http://openshmem.org/>
4. NVIDIA: Nvlink high-speed interconnect (2015). <http://www.nvidia.com/object/nvlink.html>
5. AVAGO: Expressfabric technology (2015). <http://www.plxtech.com/applications/expressfabric>
6. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence, 1st edn. Morgan & Claypool Publishers, San Rafael (2011)
7. Dinan, J., Flajslik, M.: Contexts: a mechanism for high throughput communication in openshmem. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6–10, 2014, pp. 10:1–10:9 (2014)

8. Gropp, W.D., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringer, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)
9. ten Bruggencate, M., Roweth, D., Oyanagi, S.: Thread-safe SHMEM extensions. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 178–185. Springer, Heidelberg (2014)
10. Skjellum, A.: High performance MPI: extending the message passing interface for higher performance and higher predictability. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) (1998)
11. NVIDIA: Gpudirect (2015). <https://developer.nvidia.com/gpudirect>
12. NVIDIA: Gpudirect RDMA (2015). <http://docs.nvidia.com/cuda/gpudirect-rdma>
13. Rossetti, D.: Gpudirect: integrating the GPU with a network interface. In: GPU Technology Conference (2015)
14. Wang, H., Potluri, S., Luo, M., Singh, A.K., Sur, S., Panda, D.K.: Mvapi2-GPU: optimized GPU to GPU communication for infiniband clusters. *Comput. Sci.* **26**, 257–266 (2011)
15. Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., Panda, D.K.: Efficient inter-node MPI communication using gpudirect RDMA for infiniband clusters with nvidia gpus. In: Proceedings of the 2013 42Nd International Conference on Parallel Processing. ICPP 2013, pp. 80–89. IEEE Computer Society, Washington (2013)
16. MVAPICH: MPI over infiniband, 10gige/iwarp and roce (2015). <http://mvapich.cse.ohio-state.edu>
17. Aji, A.M., Dinan, J., Buntinas, D., Balaji, P., Feng, W.c., Bisset, K.R., Thakur, R.: MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems. In: 14th IEEE International Conference on High Performance Computing and Communications, Liverpool, UK (2012)
18. Potluri, S., Bureddy, D., Wang, H., Subramoni, H., Panda, D.K.: Extending open-shmem for GPU computing. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IPDPS 2013, pp. 1001–1012. IEEE Computer Society, Washington (2013)
19. Cunningham, D., Bordawekar, R., Saraswat, V.: GPU programming in a high level language: compiling x10 to cuda. In: Proceedings of the 2011 ACM SIGPLAN X10 Workshop. X10 2011, pp. 8:1–8:10. ACM, New York (2011)
20. Miyoshi, T., Irie, H., Shima, K., Honda, H., Kondo, M., Yoshinaga, T.: Flat: A GPU programming framework to provide embedded MOI. In: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units. GPGPU-5, pp. 20–29. ACM, New York (2012)

OpenSHMEM and Related Technologies. Experiences,
Implementations, and Technologies
Second Workshop, OpenSHMEM 2015, Annapolis, MD,
USA, August 4-6, 2015. Revised Selected Papers
Gorentla Venkata, M.; Shamis, P.; Imam, N.; Lopez, M.G.
(Eds.)
2015, X, 199 p. 84 illus. in color., Softcover
ISBN: 978-3-319-26427-1