



# NVSHMEM

DG-08910\_001\_v0.3 | September 2019

## Developer Guide



# TABLE OF CONTENTS

<b>Chapter 1. Introduction.....</b>	<b>1</b>
1.1. Features.....	1
1.2. Strong Scaling.....	2
1.3. Partitioned Global Address Space.....	2
<b>Chapter 2. Programming, Communication, and Memory Models.....</b>	<b>4</b>
2.1. Programming Model.....	4
2.2. Communication Model.....	6
2.3. Memory Model.....	8
2.3.1. Modifications Of The OpenSHMEM Memory Model.....	9
2.3.2. Quiet Semantics.....	11
<b>Chapter 3. Programming Interface.....</b>	<b>12</b>
3.1. NVSHMEM Extensions To OpenSHMEM 1.4.....	12
3.2. NVSHMEM Extension APIs Invoked By GPU Thread Only.....	14
3.3. OpenSHMEM 1.4 APIs Not Supported In NVSHMEM.....	15
3.4. OpenSHMEM 1.4 APIs Not Supported Over InfiniBand In NVSHMEM.....	15
<b>Chapter 4. Initialization API.....</b>	<b>16</b>
4.1. shmemx_init_attr.....	16
<b>Chapter 5. CUDA Kernel Launch API.....</b>	<b>18</b>
5.1. shmemx_collective_launch.....	18
<b>Chapter 6. Communications API.....</b>	<b>20</b>
<b>Chapter 7. Useful Environment Variables.....</b>	<b>21</b>
<b>Chapter 8. Examples.....</b>	<b>23</b>
8.1. Attribute-Based Initialization Example.....	23
8.2. Collective Launch Example.....	24
8.3. On-Stream Example.....	25
8.4. Threadgroup Example.....	27
8.5. put_block Example.....	28
<b>Chapter 9. FAQs.....</b>	<b>30</b>

# Chapter 1.

## INTRODUCTION

One of the key goals of OpenSHMEM has been to provide an interface that can be implemented on the underlying hardware, with minimal software overheads. This is one of the main reasons NVSHMEM is based on the OpenSHMEM standard. The OpenSHMEM standard is also being actively improved upon for emerging node and cluster architectures.

The current version of NVSHMEM extends OpenSHMEM version 1.4 to provide an easy-to-use CPU-side interface to allocate pinned memory that is symmetrically distributed across a cluster of NVIDIA GPUs interconnected with NVLink or PCIe. It also provides a CUDA kernel-side interface that allows CUDA threads to access any location in symmetrically-distributed memory through OpenSHMEM data movement API calls or direct load and store (LD/ST) where the GPUs are P2P-accessible. NVSHMEM is also a low-level interface over which higher level and easier-to-use communication libraries and application frameworks can be built.

For more information on OpenSHMEM see the following web site:

<http://www.openshmem.org>

## 1.1. Features

NVSHMEM extends OpenSHMEM in the following ways:

- ▶ Symmetric heap allocation on GPU memory and support for GPU-initiated communication
- ▶ A new API call to collectively launch CUDA kernels across a set of GPUs
- ▶ Stream-based APIs that allow data movement operations initiated from the CPU to be offloaded onto the GPU with ordering with regard to a CUDA stream
- ▶ Threadgroup communication where threads from whole warps or whole thread blocks in a CUDA kernel can collectively participate in a single communication operation
- ▶ Differentiates synchronizing and non-synchronizing operations to benefit from strengths (weak or strong) of operations in GPU memory model

## 1.2. Strong Scaling

One of NVSHMEM's primary benefits is its utility in supporting strong scaling. Strong scaling is how the solution time of a fixed problem varies as the number of processors increases. This is a critical metric for scientific applications as they run on increasingly large clusters with many powerful GPUs. Current state-of-the-art scientific applications running on GPU clusters typically offload computation phases onto the GPU while relying on the CPU to manage cluster communication, using Message Passing Interface (MPI) and OpenSHMEM. Dependency on the CPU for communication limits strong scalability, owing to the overhead of repeated kernel launches, CPU-GPU synchronization, underutilization of the GPU during communication and synchronization, and underutilization of the network during compute phases. Some of these issues are handled by restructuring application code to overlap independent compute and communication phases using CUDA streams. These optimizations make the application code complex and their benefits usually diminish as the problem size per GPU becomes smaller.

Addressing the apparent [Amdahl's fraction](#) of synchronizing with the CPU for communication is critical for strong scaling of applications on GPU clusters. GPUs are designed for throughput and have enough state and parallelism to hide long latencies to global memory. Following the [CUDA programming model and best practices](#) enables developers to take advantage of these latency hiding capabilities. It is natural to also take advantage of these capabilities of the GPU and the CUDA programming model when tackling communications between GPUs.

## 1.3. Partitioned Global Address Space

NVSHMEM provides a Partitioned Global Address Space (PGAS) that spans memory across GPUs and provides an API for fine-grained GPU-GPU data movement and synchronization from within a CUDA kernel. Using NVSHMEM, developers can write long running kernels that include both communication and computation, reducing synchronization with the CPU. These composite kernels allow for overlap between computation and communication owing to warp scheduling on the GPU. This model avoids the overhead of additional kernel launches and calls to CUDA API and CPU-GPU synchronization for communication, allowing for better strong scaling of applications. When necessary, NVSHMEM also provides CPU-side calls for inter-GPU communication outside of CUDA kernels.

NVSHMEM facilitates inter-GPU communication inside CUDA kernels, characterized by highly concurrent fine-grained messaging. The MPI send-receive interface that is primarily used in applications today for GPU-GPU data movement does not serve well in such a communication regime. The main reasons for this are: locking (or atomics) for shared data structures around network end points, serialization inherent to message matching and queuing and handling of unexpected messages. While there have been efforts to reduce the overhead of critical sections using finer-grained locking and multiple network end-points per process, the problem with message matching is inherent to the send-receive model of communication in MPI. One-sided communication

primitives avoid this overhead by letting the initiating process/thread specify all the information required to complete a data transfer. They can be directly translated to RDMA primitives exposed by the network hardware or to LD/ST on a fabric like NVLink. One-sided primitives also make it programmatically easier to interleave computation and communication, thereby having the potential for better overlap.

# Chapter 2.

## PROGRAMMING, COMMUNICATION, AND MEMORY MODELS

This section describes the programming, communication, and memory models governing NVSHMEM usage.

### 2.1. Programming Model

NVSHMEM implements and extends the OpenSHMEM programming model. NVSHMEM interfaces are intended for use in SPMD (single program multiple data)-style programs. Each instance of the program is referred to as a PE (Processing Element) in NVSHMEM and can be executed by an OS process or a thread. Only PE-as-a-process is supported in version 0.3. Several executing instances of a program can declare their participation in an NVSHMEM job by collectively calling an initialization routine.

An NVSHMEM job is created by an initialization API call and ended when a finalization function is called. During initialization, each instance of the program is assigned a unique ID, called the PE ID. PEs within a job can share data through globally accessible memory that is allocated using the NVSHMEM allocation API. Memory allocated using any other method is considered private to the allocating PE and is not accessible by other PEs.

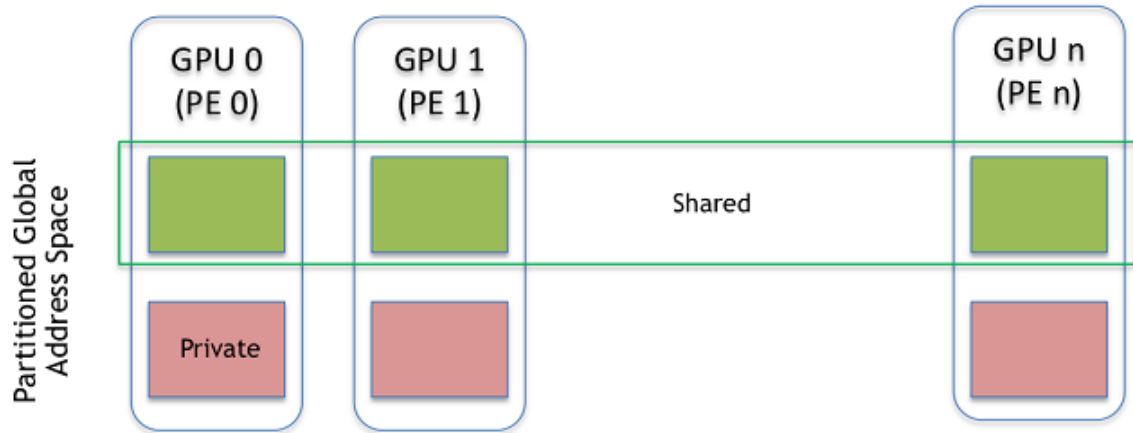


Figure 1 PE Shared and Private Memory

Shared memory object creation in NVSHMEM is collective and symmetric across all the PEs in a job. This means that each PE should participate in the allocation by making a call to the allocation API and every PE should pass the same value in the size argument for a given allocation. Each shared memory object creation involves allocation of a memory region of a given size in GPU device memory at each of the PEs. The memory region allocated at each PE is accessible to all the other PEs in the job (from inside CUDA kernels as well as through the CPU-side NVSHMEM API) through the communication API or direct load/stores (when supported by hardware).

The symmetric management of the global address space allows a remote address to be simply referenced in terms of a tuple of `<local_address>`, `<destination_PE>`. This simplifies communication in applications. The NVSHMEM runtime can translate the local address to the actual remote address with little or no translation overhead.

NVSHMEM can be used in conjunction with OpenSHMEM or MPI, making it easier for existing OpenSHMEM and MPI applications to be incrementally ported to use NVSHMEM. The code snippet below shows a simple example of NVSHMEM usage within an MPI program, where there is a 1:1 correspondence between MPI rank and NVSHMEM PEs.

```
__global__ void simple_shift (int *destination) {
    int mype = shmem_my_pe() ;
    int n_pes = shmem_n_pes();
    int peer = (mype + 1)%n_pes;

    shmem_int_p(destination, mype, peer);
}

int main (int c, char *v[])
{
    int *source;
    int *destination;
    int rank;

    shmemx_init_attr_t attr;
    attr.mpi_comm = MPI_COMM_WORLD;

    MPI_Init(&c, &v);
    MPI_Comm_rank (&rank, MPI_COMM_WORLD));
```

```

cudaSetDevice(rank);
shmemx_init_attr (SHMEMX_INIT_WITH_MPI_COMM, &attr);

destination = (int *) shmem_malloc (sizeof(int));

simple_shift<<<1, 1>>> (destination);

shmem_barrier_all();

shmem_free(source);
shmem_free(destination);

shmem_finalize();
MPI_Finalize();
}

```

In the example, the CUDA kernel implements a circular shift of values in a global array distributed across the PEs. It creates two shared memory objects of **sizeof(int)** bytes using **shmem\_malloc** and assigns it to pointers **source** and **destination**. Each of these calls allocates a buffer of **sizeof(int)** bytes at each PE and the pointer returned at each PE points to the buffer that is local to itself.

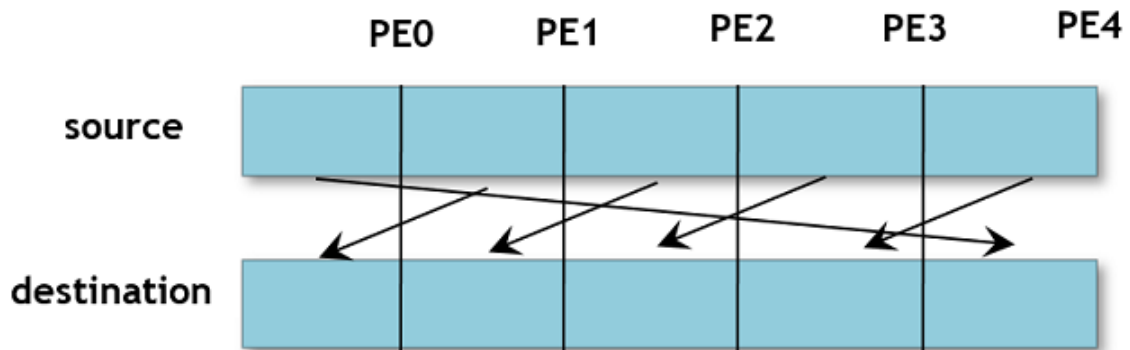


Figure 2 NVSHMEM in Conjunction with MPI

Data at the destination element at the neighboring/target PE is written using the **shmem\_int\_p** API, by passing the local destination pointer and the target PE ID. The runtime translates the local destination pointer to a pointer to destination at the target PE. The value passed to **shmem\_int\_p** is assigned to the local element of the destination array.

## 2.2. Communication Model

NVSHMEM provides fine-grained and low-overhead remote data access from inside CUDA kernels and enables applications to benefit from the intrinsic latency-hiding capabilities of the GPU warp scheduling hardware. In this spirit, NVSHMEM provides APIs to move data to or from global memory allocations. It also provides APIs to query virtual addresses that point to portions of a distributed global memory allocation. This allows applications to issue direct loads/stores to global memory. The same API or load/stores (when the hardware allows) can be used to access portion of the global memory that is local to the accessing PE or is physically located at a different PE.

NVSHMEM extends OpenSHMEM in the following ways:



- ▶ All symmetric memory that is allocated using the NVSHMEM allocation API is pinned GPU device memory.
- ▶ NVSHMEM supports both GPU- and CPU-side communication and synchronization APIs, provided that the memory involved is GPU device memory. In other OpenSHMEM implementations, these APIs can only be called from the CPU.

NVSHMEM is a stateful library. It detects which GPU a PE is using when the PE calls into an NVSHMEM initialization routine. This information is stored inside the NVSHMEM runtime. All symmetric allocation calls made by the PE return device memory of the selected GPU. All NVSHMEM calls made by the PE are assumed to be made with respect to the selected GPU or from inside kernels launched on this GPU. This requires certain restrictions on PE-GPU mappings in applications using NVSHMEM.

An NVSHMEM program should adhere to the following:

- ▶ The PE selects its GPU (with `cudaSetDevice`, for example), before the first allocation, synchronization, communication, or collective kernel API launch call.
- ▶ The PE uses one and only one GPU throughout the lifetime of an NVSHMEM job.

NVSHMEM relies on data coalescing features in GPU hardware to achieve efficiency over the network when the data access API is used. It is important that application developers follow CUDA programming best practices that promote data coalescing when using fine-grained communication APIs in NVSHMEM.

NVSHMEM also allows any two CUDA threads within a job to synchronize on locations in global memory using the OpenSHMEM point-to-point synchronization API `shmem_wait_until` or collective synchronization APIs like `shmem_barrier`.



CUDA kernels that use synchronization APIs must be launched using the collective launch API to guarantee deadlock-free progress and completion. For information on the collective launch API, see [CUDA Kernel Launch API](#).

CUDA kernels that do not use the NVSHMEM synchronization APIs but use other NVSHMEM communication APIs can be launched with either the normal CUDA launch interfaces or the collective launch API. These kernels can still use other NVSHMEM device side APIs such as the one-sided data movement API.

An NVSHMEM program that uses collective launch and CUDA kernel-side synchronization APIs should adhere to the following for correctness, and all NVSHMEM programs should adhere to the following for performance predictability:

- ▶ Multiple PEs should not share the same GPU.
- ▶ NVSHMEM PEs should have exclusive access to the GPU. The GPU cannot be used to drive a display or for another compute job.

## 2.3. Memory Model

The NVSHMEM memory model is defined in this section. It follows the semantics defined in the OpenSHMEM 1.4 specification except in the cases described in [Modifications Of The OpenSHMEM Memory Model](#) in this document.

Local or remote symmetric memory can be accessed in the following ways:

- ▶ Remote memory access (RMA: **PUT/GET**)
- ▶ Atomic memory operations (AMO)
- ▶ LD/ST (In the case of remote symmetric memory, using a pointer returned by **shmem\_ptr**)
- ▶ Collective functions (broadcast, reductions, and others)
- ▶ The wait function (local symmetric memory only)

Two memory accesses by a PE are guaranteed to be ordered only in the following cases:

- ▶ The accesses are the result of different collective function calls that happen in program order.
- ▶ The first access is a wait call, followed by a read operation, both of which target local symmetric memory.
- ▶ The accesses are the result of two different API calls or LD/ST operations and are separated by an appropriate ordering operation based on the following table:

Type of first access/target PE	Same-target PE	Different-target PE
Blocking	Fence/quiet/barrier	Quiet/barrier
Non-blocking	Quiet/barrier	Quiet/barrier



Two accesses from blocking operations may be ordered by the underlying system if there is a dependency relationship between them, such as data or address.

One API call has a reads-from relationship with another API call when the second call takes its value from the first call. For example, a **GET** has a reads-from relationship with a **PUT** if the **GET** reads the data that the **PUT** wrote. A collective operation involves an API call at each of the participating PEs and calls at PEs with target buffers have a read from relationship with the calls at PEs with source buffers.

An update made using an API call is guaranteed to become eventually visible to another API call. This prevents a runtime from buffering updates indefinitely. The update is stable in the sense once it is visible to another API call, the update remains until replaced by another update. This guarantees that synchronization as described above finishes in a finite amount of time.

An access **A** at PE **X** happens before (and is visible to) access **B** at PE **Y** if any of the following are true:

- ▶ **A** is an AMO/**shmem\_p** operation, **B** is a wait/AMO/**shmem\_g** operation and **B** has a reads-from relationship with **A**.
- ▶ **A** is ordered before an AMO/**shmem\_p** operation **C** at PE **x**, a wait/AMO/**shmem\_g** operation **D** at PE **y** has a reads-from relationship with **C**, and **D** is ordered before **B**.
- ▶ **A** is ordered before an access **C** at PE **x** and **C** happens before **B**.
- ▶ **A** happens before **C** at PE **z**, and **C** happens before **B**.



An implication of the above is when an API call **A** synchronizes with an API call **B**, and operations (LD/ST or API) that are ordered before **A** are visible to operations (LD/ST or API) that are ordered after **B**.

Two accesses are concurrent if any of the following are true:

- ▶ The accesses are performed by different threads within a PE and are unordered.
- ▶ The accesses are performed by the same thread and are unordered.
- ▶ The accesses are performed by two different PEs and one did not happen before the other.

Two accesses conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location.

Two concurrent conflicting accesses result in undefined behavior. AMOs, single-element RMA (**shmem\_p**, **shmem\_g**) operations and wait calls are an exception to this when the conflicting accesses are using the same datatype. In this case, it is as if the two operations have been executed in some order, meaning, they exhibit single-copy atomicity.

### 2.3.1. Modifications Of The OpenSHMEM Memory Model

NVSHMEM relaxes the memory ordering semantics as are defined in the OpenSHMEM specification to allow more efficient implementation on NVIDIA GPUs. The changes are described in detail in the following sections.

A blocking get operation (**shmem\_g**, **shmem\_get**, **shmem\_iget**), as defined in the OpenSHMEM specification, returns to the destination array at the local PE after the data has been delivered. The OpenSHMEM specification also implicitly guarantees that any two get operations are always executed in program order. This total ordering of get operations requires the implementation to include an appropriate memory barrier in each get operation, resulting in sub-optimal performance.

NVSHMEM relaxes the total ordering requirement of get operations and requires the programmer to use a fence where such ordering is required. The completion semantics of get remain unchanged from the specification: the result of the get is available for any dependent operation that appears after it, in program order.

The examples below show how the ordering of get operations in NVSHMEM differs from that defined in the OpenSHMEM specification. In the figure below, line c is data dependent on the result of the get operation in line b. Lines b and c are guaranteed to execute in program order. Hence, the output where j takes value 0 is prohibited.

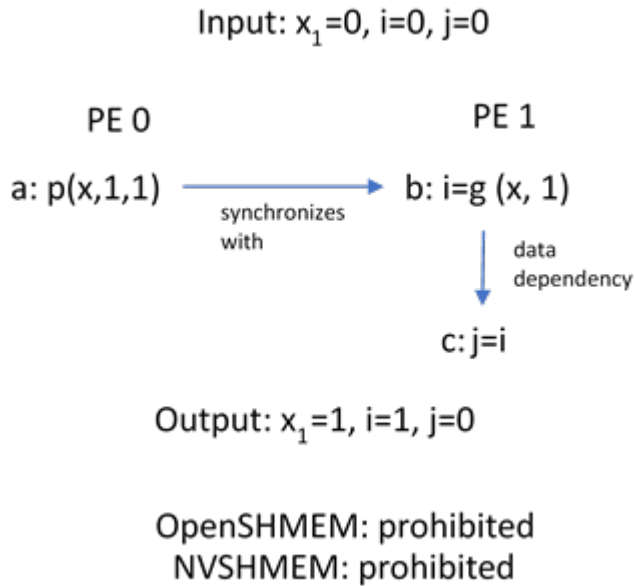


Figure 3 Completion semantics of **GET** operations. The Output in the figure is prohibited both in OpenSHMEM and NVSHMEM.

In the figure below, **GET** ( $g$ ) operations on lines c and d are unrelated and can be reordered in NVSHMEM. Hence the result where  $j$  takes value 0 is allowed.

*Notation:*  
 $p(x,1,1)$  – *shmem\_type\_p(variable, value, target PE)*  
 $g(y,1)$  – *shmem\_type\_g(variable, target PE) (returns the result)*  
 $x_n$  – *symmetric variable  $x$  at PE  $n$*   
 $x$  – *local variable*

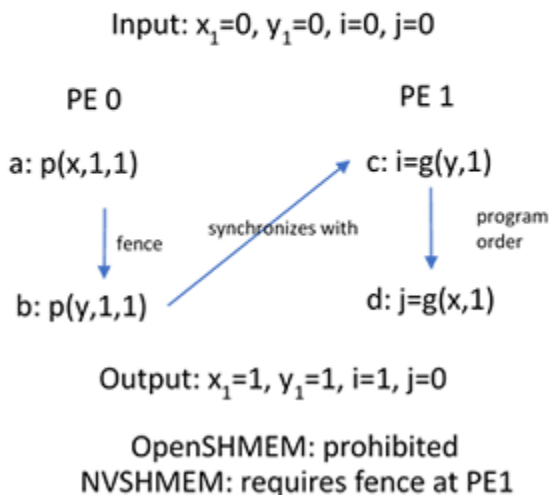


Figure 4 Another example of ordering of **GET** operations. The output above is prohibited in OpenSHMEM but not in NVSHMEM. In order for the above output to be prohibited in NVSHMEM, a *shmem\_fence()* operation is required between line c and line d at PE1.

NVSHMEM extends the semantics of **shmem\_fence** to order get operations. The **shmem\_fence** function does not order non-blocking get operations as specified in the OpenSHMEM specification.

### 2.3.2. Quiet Semantics

OpenSHMEM defines quiet semantics as follows: all put, AMO, memory store, and nonblocking put and get routines to symmetric data objects are guaranteed to be completed and visible to all PEs when **shmem\_quiet** returns. The guarantee that updates are visible when **shmem\_quiet** returns requires synchronizing with and having the necessary ordering APIs at the target PE, on all platforms.

NVSHMEM clarifies the semantics of quiet as follows: **shmem\_quiet** completes all non-blocking operations issued by the calling PE and ensures ordering between all accesses (to all PEs) that happen before it in program order and all the accesses (to all PEs) that happen after it in program order.

# Chapter 3.

## PROGRAMMING INTERFACE

The API in NVSHMEM is separated into three parts: invoked on the host, invoked on the GPU and invoked on the GPU or host. In the current version of NVSHMEM, the APIs for runtime initialization and termination, memory management, querying pointers, querying PE information, and collective kernel launch, are supported from the host.

One-sided remote memory access, one-sided remote atomic memory access, memory ordering, point-to-point synchronization and collectives are supported from both the host and the device. In addition, there are CUDA-specific extensions of remote memory access and collective APIs that are supported either only on the host or only on the device.

### 3.1. NVSHMEM Extensions To OpenSHMEM 1.4

The following tables summarize the NVSHMEM extension APIs and their functionality. The C11 form is used for all APIs.

Table 1 NVSHMEM Extension APIs Invoked by CPU Thread Only

Description	NVSHMEM (shmemx_*)	Notes
Initialization	<code>*_init_attr</code>	
Query	<code>*_my_pe (SHMEMX_TEAM_NODE or SHMEMX_TEAM_WORLD)</code> <code>*_n_pes (SHMEMX_TEAM_NODE or SHMEMX_TEAM_WORLD)</code>	Convenience API for querying index suitable for <code>cudaSetDevice</code> .
CUDA kernel launch	<code>*_collective_launch</code>	CUDA kernels that invoke NVSHMEM synchronization APIs such as <code>shmem_barrier</code> , <code>shmem_wait</code> , and others, must be launched using this API; otherwise behavior is undefined.
Remote memory access	<code>*_put_&lt;all_variants&gt;_on _stream,</code>	Asynchronous with respect to the calling CPU thread; takes a

Description	NVSHMEM (shmemx_*)	Notes
	*_get_<all_variants>_on_stream	cudaStream_t as argument and is ordered on that CUDA stream.
Memory ordering	*_quiet_on_stream	
Collective communication	*_broadcast_<all_variants>_on_stream, *_collect_<all_variants>_on_stream, *_alltoall_<all_variants>_on_stream, *_to_all_<all_variants>_on_stream (reductions)	
Collective synchronization	*_barrier_all_on_stream, *_barrier_on_stream, *_sync_all_on_stream, *_sync_on_stream	

Table 2 NVSHMEM Extension APIs Invoked by GPU Thread Only

Description	NVSHMEM (nvshmem_*)	Notes
Write buffer	*_put_block, *_put_warp	New APIs for GPU-side invocation are provided that can be called collectively by a threadblock or a warp.
Read buffer	*_get_block, *_get_warp	
Asynchronous write buffer	*_put_nbi_block, *_put_nbi_warp	
Asynchronous read buffer	*_get_nbi_block, *_get_nbi_warp	
Collective communication	*_broadcast_<all_variants>_block, *_broadcast_<all_variants>_warp, *_collect_<all_variants>_block, *_collect_<all_variants>_warp, *_alltoall_<all_variants>_block, *_alltoall_<all_variants>_warp, *_to_all_<all_variants>_block, *_to_all_<all_variants>_warp (reductions)	
Collective synchronization	*_barrier_all_block, *_barrier_all_warp, *_barrier_block, *_barrier_warp, *_sync_all_block, *_sync_all_warp, *_sync_block, *_sync_warp	

## 3.2. NVSHMEM Extension APIs Invoked By GPU Thread Only

Table 2 lists the NVSHMEM extension APIs that can be invoked by GPU threads. Each of the APIs corresponding to the entries in the description column have two variants each – one with the suffix **\_block** and the other with the suffix **\_warp**. For example, the OpenSHMEM API **shmem\_float\_put** has two extension APIs in NVSHMEM: **shmemx\_float\_put\_block** and **shmemx\_float\_put\_warp**.

These extension APIs are collective calls that must be called by *every* thread in the scope of the API and with exactly the same arguments. The scope of the **\*\_block** extension APIs is the block in which the thread resides. Similarly, the scope of the **\*\_warp** extension API is the warp in which the thread resides. For example, if thread 0 calls **shmem\_float\_put\_block**, then every other thread that is in the same block as thread 0 must also call **shmem\_float\_put\_block** with exactly the same arguments. Otherwise, the call will result in erroneous behavior or deadlock in the program. The NVSHMEM runtime may or may not leverage the multiple threads in the scope of the API to execute the API call.

The extension APIs are useful in the following situations:

- ▶ Converting **shmem\_float\_put** to **shmemx\_float\_put\_block** enables the NVSHMEM runtime to leverage all the threads in the block to concurrently copy the data to the destination PE if the destination GPU of the put call is p2p connected. If the destination GPU is connected via InfiniBand, then a single thread in the block can issue an RMA write operation to the destination GPU.
- ▶ **\*\_block** and **\*\_warp** extensions of the collective APIs can make use of multiple threads to perform collective operations, such as parallel reduction operations in case of a collective reduction operation or sending data in parallel.



- ▶ CUDA supports only a subset of the atomic operations included in the OpenSHMEM specification. The long type is currently unsupported for **add**, **fadd**, **inc**, and **finc** atomic operations in NVSHMEM.
- ▶ Bitwise atomic support for **and**, **fand**, **or**, **for**, **xor**, and **fxor** is part of OpenSHMEM 1.4 with names starting with the prefix **shmem\_atomic\_** followed by the name of the operation (**cswap**, **add** etc). These APIs exist in NVSHMEM but with the **shmemx\_** prefix followed by the name of the operation. The other atomic operations are named in the same way in OpenSHMEM 1.4. These APIs exist in NVSHMEM but with the **shmem\_** prefix followed by the name of the operation.
- ▶ The **shmem\_init\_thread** and **shmem\_query\_thread** APIs are in OpenSHMEM 1.4. These APIs exist with the **shmemx\_** prefix in NVSHMEM.



### 3.3. OpenSHMEM 1.4 APIs Not Supported In NVSHMEM

The following OpenSHMEM 1.4 APIs are not supported in NVSHMEM:

- ▶ `shmem_global_exit`
- ▶ `shmem_pe_accessible`
- ▶ `shmem_addr_accessible`
- ▶ `shmem_info_get_version`
- ▶ `shmem_info_get_name`
- ▶ `shpalloc`
- ▶ `shpclmove`
- ▶ `shpdealloc`
- ▶ `shmem_realloc`
- ▶ `shmem_ctx_create`
- ▶ `shmem_ctx_destroy`
- ▶ `shmem_fcollect`
- ▶ `shmem_alltoalls`
- ▶ `shmem_lock`
- ▶ `shmem_cache`

### 3.4. OpenSHMEM 1.4 APIs Not Supported Over InfiniBand In NVSHMEM

The following OpenSHMEM 1.4 APIs are not supported over InfiniBand in NVSHMEM:

- ▶ `shmem_iput`
- ▶ `shmem_iget`
- ▶ `shmem_g`
- ▶ `shmem_atomic_<all operations>`

# Chapter 4.

## INITIALIZATION API

NVSHMEM provides two forms of initialization API. The first one is **shmem\_init** as is defined in the OpenSHMEM 1.4 specification. The second variation is provided to enable easy porting of MPI and OpenSHMEM programs to NVSHMEM. It allows initialization of NVSHMEM based on an MPI communicator or inside an OpenSHMEM job. This is useful when an application is written to use NVSHMEM within a node and MPI across nodes. Or, when an application uses another OpenSHMEM implementation to manage communication across symmetric heap on the system memory.

### 4.1. shmemx\_init\_attr

The **shmemx\_init\_attr** function initializes the NVSHMEM library by allocating the resources used by the library and assigning a unique identifier to each PE. This collective operation should be called by all PEs before any other NVSHMEM routine.

With the **SHMEMX\_INIT\_WITH\_MPI\_COMM** option, the NVSHMEM library is initialized based on the MPI communicator that is provided with each rank in the MPI communicator participating as an NVSHMEM PE. A call to **shmem\_finalize** is required before the MPI communicator is destroyed. Do not make any calls to NVSHMEM routines after the MPI communicator has been destroyed.

With the **SHMEMX\_INIT\_WITH\_SHMEM** option, the NVSHMEM library is initialized based on the OpenSHMEM PE underlying each NVSHMEM PE. In this case, you must use the NVSHMEM APIs with **nv\_ prefixes**, so **nvshmemx\_init\_attr** should be called. A call to **nvshmemx\_finalize** is required before **shmem\_finalize** is called. Do not make any calls to NVSHMEM routines after the OpenSHMEM **shmem\_finalize** is called.

#### Parameters

##### **SHMEMX\_INIT\_WITH\_MPI\_COMM**

This flag is used to specify that an MPI communicator is provided by the user. A value of 0 indicates initialization similar to when **shmem\_init** is used.

##### **SHMEMX\_INIT\_WITH\_SHMEM**

This flag is used to specify that NVSHMEM is used inside an OpenSHMEM job. A value of 0 indicates initialization similar to when **shmem\_init** is used.

**mpi\_comm**

This attribute is a pointer to an MPI communicator.

**shmemx\_init\_attr\_t**

This attribute is a pointer to a structure containing input attributes, for example:

```
struct shmemx_init_attr_t {  
    void *mpi_comm;  
}
```

## Returns

Returns 0 on success and an error code on failure.

# Chapter 5.

## CUDA KERNEL LAUNCH API

NVSHMEM provides an interface that must be used to launch CUDA kernels on the GPU when the CUDA kernels use NVSHMEM synchronization APIs.

### 5.1. shmemx\_collective\_launch

The **shmemx\_collective\_launch** call is collective across the PEs in the NVSHMEM job. It takes the same parameters as a CUDA kernel launch API. It uses a single device CUDA cooperative launch and hence provides all its guarantees. If a CUDA kernel in a PE calls NVSHMEM synchronization API (such as **shmem\_wait**, **barrier**, or **barrier\_all**), then it is required to be launched using this API. Any CUDA kernel not using NVSHMEM synchronization APIs (or not using NVSHMEM APIs at all), is not required to be launched by this API. Specify **gridDim** or set it to 0. When **gridDim** is set to 0, the NVSHMEM runtime picks the largest grid size that can be used for the given kernel with CUDA cooperative launch on the current GPU.

#### Parameters

**func**

A pointer to the function to launch on the device.

**gridDim**

The grid dimensions.

**blockDim**

The block dimensions.

**args**

Arguments to be passed to the device function.

**sharedMem**

The size of the shared memory.

**stream**

The stream on which the kernel should be launched.

**Returns**

Returns 0 on success and an error code on failure.

# Chapter 6.

## COMMUNICATIONS API

This section describes the following functions that can be called from a CPU or GPU thread in the context of NVSHMEM:

- ▶ Remote memory access (extended by NVSHMEM)
- ▶ Atomic
- ▶ Memory ordering (extended by NVSHMEM)
- ▶ Point-to-point synchronization
- ▶ Collectives (extended by NVSHMEM)

The Atomic and point-to-point synchronization functions in NVSHMEM are not extended and work exactly like the corresponding functions in OpenSHMEM. The extensions for remote memory access and collective functions have two types.

- ▶ CPU only: all CPU only extensions have the suffix `*_on_stream` and include the argument `cudaStream_t`, for specifying the CUDA stream (see [NVSHMEM Extensions To OpenSHMEM 1.4](#)).
  - ▶ These functions otherwise perform exactly as in OpenSHMEM. See the [OpenSHMEM Application Programming Interface, version 1.4](#), for documentation.
- ▶ GPU only: `*_<scope>`, where scope is `*_block` or `*_warp` (see [NVSHMEM Extensions To OpenSHMEM 1.4](#)),
  - ▶ These functions otherwise perform exactly as in OpenSHMEM. See the [OpenSHMEM Application Programming Interface, version 1.4](#), for documentation.



Ordering APIs (`fence`, `quiet`, and `barrier`) issued on the CPU and the GPU only order communication operations that were issued from the CPU and the GPU, respectively. To ensure completion of GPU-side operations from the CPU, the developer has to ensure completion of the CUDA kernel from which the GPU-side operations were issued, using operations like `cudaStreamSynchronize` or `cudaDeviceSynchronize`. Stream-based quiet or barrier operations have the effect of a barrier being executed on the GPU in stream order, ensuring completion and ordering of only GPU-side operations.

# Chapter 7.

## USEFUL ENVIRONMENT VARIABLES

The following environment variables are useful in certain circumstances.

Variable	Description
<code>NVSHMEM_ENABLE_NIC_PE_MAPPING</code>	When not set or set to 0, a PE is assigned the NIC on the node that is closest to it by distance. When set to 1, NVSHMEM either assigns NICs to PEs on a round-robin basis or uses <code>NVSHMEM_HCA_PE_MAPPING</code> or <code>NVSHMEM_HCA_LIST</code> when they are specified.
<code>NVSHMEM_HCA_PE_MAPPING</code>	Specifies an HCA per PE as a comma-separated list such that the GPU corresponding to the PE uses the given HCA for all transfers. Each entry in the comma separated list is of the form <code>hca_name:port:count</code> . For example, <code>m1x5_0:1:2,m1x5_0:2:2</code> means that PE0, PE1 can be mapped to port 1 of <code>m1x5_0</code> , and PE2, PE3 can be mapped to port 2 of <code>m1x5_0</code> .  <code>NVSHMEM_ENABLE_NIC_PE_MAPPING</code> must be set to 1 for this variable to be effective.
<code>NVSHMEM_HCA_LIST</code>	Specifies a comma-separated list of HCAs to use in the NVSHMEM application. This is useful to skip disabled HCAs or HCAs operating in a different mode than InfiniBand. Each entry in the comma separated list is of the form: <code>hca_name:port</code> . For example, <code>m1x5_1:1,m1x5_2:2</code> specifies use of port 1 of <code>m1x5_1</code> and port 2 of <code>m1x5_2</code> in the application. A ^ before the list indicates an exclusion list. For example, <code>^m1x5_1:1,m1x5_1:2</code> means not to use <code>m1x5_1</code> port 1 and <code>m1x5_1</code> port 2.  <code>NVSHMEM_ENABLE_NIC_PE_MAPPING</code> must be set to 1 for this variable to be effective.
<code>NVSHMEM_SYMMETRIC_SIZE</code>	Specifies the GPU memory per PE allocated for the symmetric heap. By default, NVSHMEM would allocate 1GB per PE.

Variable	Description
<code>NVSHMEM_MPI_LIB_NAME</code>	Name of the MPI library that is used for bootstrapping NVSHMEM. By default, NVSHMEM looks for library with name <code>libmpi.so</code> .
<code>NVSHMEM_SHMEM_LIB_NAME</code>	Name of the OpenSHMEM library that is used for bootstrapping NVSHMEM. By default, NVSHMEM looks for library with name <code>liboshmem.so</code> .
<code>NVSHMEM_BARRIER_DISSEM_KVAL</code>	Radix of the dissemination algorithm used for <code>barrier</code> and <code>barrier_all</code> collectives. By default, NVSHMEM uses radix 2.
<code>NVSHMEM_BARRIER_TG_DISSEM_KVAL</code>	Radix of the dissemination algorithm used for <code>barrier_warp</code> , <code>barrier_group</code> , <code>barrier_all_warp</code> , and <code>barrier_all_group</code> collectives. By default, NVSHMEM uses radix 2.
<code>NVSHMEM_DEBUG</code>	Controls the debug information that is displayed from NVSHMEM. Values accepted are <code>WARN</code> , <code>INFO</code> , and <code>TRACE</code> .
<code>NVSHMEM_DEBUG_FILE</code>	Set the filename where debug output is written.



# Chapter 8.

## EXAMPLES

Source code for the examples described in this section is available in the examples folder of the NVSHMEM package.

### 8.1. Attribute-Based Initialization Example

The following code shows an existing MPI version of the simple shift program that was explained in the Programming Model section of this document. It shows the use of the NVSHMEM attribute-based initialization API, where the MPI communicator can be used to setup NVSHMEM, where the MPI communicator can be used to set-up NVSHMEM.

```
#include <stdio.h>
#include "mpi.h"
#include "shmem.h"
#include "shmemx.h"

#define CUDA_CHECK(stmt) \
do { \
    cudaError_t result = (stmt); \
    if (cudaSuccess != result) { \
        fprintf(stderr, "[%s:%d] cuda failed with %s \n", \
            __FILE__, __LINE__, cudaGetErrorString(result)); \
        exit(-1); \
    } \
} while (0)

#define MPI_CHECK(stmt) \
do { \
    int result = (stmt); \
    if (MPI_SUCCESS != result) { \
        fprintf(stderr, "[%s:%d] MPI failed with error %d \n", \
            __FILE__, __LINE__, result); \
        exit(-1); \
    } \
} while (0)

__global__ void simple_shift (int *target, int mype, int npes) {
    int peer = (mype + 1)%npes;
    shmem_int_p(target, mype, peer);
}

int main (int c, char *v[])
```

```

{
    int *target;
    int rank, nranks;
    MPI_Comm mpi_comm;
    shmemx_init_attr_t attr;
    int mype, npes;

    MPI_CHECK(MPI_Init(&c, &v));
    MPI_CHECK(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
    MPI_CHECK(MPI_Comm_size(MPI_COMM_WORLD, &nranks));

    mpi_comm = MPI_COMM_WORLD;
    attr.mpi_comm = &mpi_comm;
    shmemx_init_attr (SHMEMX_INIT_WITH_MPI_COMM, &attr);
    mype = shmem_my_pe();
    npes = shmem_n_pes();

    //application picks the device each PE will use
    CUDA_CHECK(cudaSetDevice(mype));
    target = (int *) shmem_malloc (sizeof(int));

    simple_shift<<<1, 1>>> (target, mype, npes);
    CUDA_CHECK(cudaDeviceSynchronize());

    printf("[%d of %d] run complete \n", mype, npes);

    shmem_free(target);

    shmem_finalize();
    MPI_CHECK(MPI_Finalize());
    return 0;
}

```

## 8.2. Collective Launch Example

The following code shows an example implementation of a single ring-based reduction where multiple iterations of the code, including computation, communication and synchronization are expressed as a single kernel.

This example also demonstrates the use of NVSHMEM collective launch, required when the NVSHMEM synchronization API is used from inside the CUDA kernel.

There is no MPI dependency for the example. NVSHMEM can be used to port existing MPI applications and develop new applications.

```

#include <stdio.h>
#include "shmem.h"
#include "shmemx.h"

#define CUDA_CHECK(stmt) \
do { \
    cudaError_t result = (stmt); \
    if (cudaSuccess != result) { \
        fprintf(stderr, "[%s:%d] cuda failed with %s \n", \
            __FILE__, __LINE__, cudaGetErrorString(result)); \
        exit(-1); \
    } \
} while (0)

#define SHMEM_CHECK(stmt) \
do { \
    int result = (stmt); \
    if (SHMEM_SUCCESS != result) { \

```

```

        fprintf(stderr, "[%s:%d] shmem failed with error %d \n", \
                __FILE__, __LINE__, result); \
        exit(-1); \
    } \
} while (0)

__global__ void reduce_ring (int *target, int mype, int npes) {
    int peer = (mype + 1)%npes;
    int lvalue = mype;

    for (int i=1; i<npes; i++) {
        shmem_int_p(target, lvalue, peer);
        shmem_barrier_all();
        lvalue = *target + mype;
    }
}

int main (int c, char *v[])
{
    int mype, npes;
    shmem_init();
    mype = shmem_my_pe();
    npes = shmem_n_pes();

    //application picks the device each PE will use
    CUDA_CHECK(cudaSetDevice(mype));
    double *u = (double *) shmem_malloc(sizeof(double));

    void *args[] = {&u, &mype, &npes};
    dim3 dimBlock(1);
    dim3 dimGrid(1);

    SHMEM_CHECK(shmemx_collective_launch ((const void *)reduce_ring, dimGrid,
    dimBlock, args, 0 , 0));
    CUDA_CHECK(cudaDeviceSynchronize());

    printf("[%d of %d] run complete \n", mype, npes);

    shmem_free(u);

    shmem_finalize();
    return 0;
}

```

## 8.3. On-Stream Example

The following example shows how **shmemx\_\*\_on\_stream** functions can be used to enqueue a SHMEM operation onto a CUDA stream for execution in stream order. Specifically, the example shows the following:

- ▶ How a collective SHMEM reduction operation can be made to wait on a preceding kernel in the stream.
- ▶ How a kernel can be made to wait for a communication result from a previous collective SHMEM reduction operation.

The example shows one use case for relieving CPU control over GPU compute and communication.

```

#include <stdio.h>
#include "shmem.h"
#include "shmemx.h"

```

```

#define THRESHOLD 42
#define CORRECTION 7

#define CUDA_CHECK(stmt) \
do { \
    cudaError_t result = (stmt); \
    if (cudaSuccess != result) { \
        fprintf(stderr, "[%s:%d] cuda failed with %s \n", \
            __FILE__, __LINE__, cudaGetErrorString(result)); \
        exit(-1); \
    } \
} while (0)

__global__ void accumulate(int *input, int *partial_sum)
{
    int index = threadIdx.x;
    if (0 == index) *partial_sum = 0;
    __syncthreads();
    atomicAdd(partial_sum, input[index]);
}

__global__ void correct_accumulate(int *input, int *partial_sum, int *full_sum)
{
    int index = threadIdx.x;
    if (*full_sum > THRESHOLD) {
        input[index] = input[index] - CORRECTION;
    }
    if (0 == index) *partial_sum = 0;
    __syncthreads();
    atomicAdd(partial_sum, input[index]);
}

int main (int c, char *v[])
{
    int mype, npes;
    int *input;
    int *partial_sum;
    int *full_sum;
    int input_nelems = 512;
    int to_all_nelems = 1;
    int PE_start = 0;
    int PE_size = 0;
    int logPE_stride = 0;
    long *pSync;
    int *pWrk;
    cudaStream_t stream;

    shmem_init ();
    PE_size = shmem_n_pes();
    mype = shmem_my_pe();
    npes = shmem_n_pes();

    CUDA_CHECK(cudaSetDevice(mype));
    CUDA_CHECK(cudaStreamCreate(&stream));

    input = (int *) shmem_malloc(sizeof(int) * input_nelems);
    partial_sum = (int *) shmem_malloc(sizeof(int));
    full_sum = (int *) shmem_malloc(sizeof(int));
    pWrk = (int *) shmem_malloc(sizeof(int) * SHMEM_REDUCE_MIN_WRKDATA_SIZE);
    pSync = (long *) shmem_malloc(sizeof(long) * SHMEM_REDUCE_SYNC_SIZE);

    accumulate <<<1, input_nelems, 0, stream>>> (input, partial_sum);
    shmemx_int_sum_to_all_on_stream(full_sum, partial_sum, to_all_nelems,
    PE_start, logPE_stride, PE_size, pWrk, pSync, stream);
    correct_accumulate <<<1, input_nelems, 0, stream>>> (input, partial_sum,
    full_sum);
}

```

```

    CUDA_CHECK(cudaStreamSynchronize(stream));

    printf("[%d of %d] run complete \n", mype, npes);

    CUDA_CHECK(cudaStreamDestroy(stream));

    shmem_free(input);
    shmem_free(partial_sum);
    shmem_free(full_sum);
    shmem_free(pWrk);
    shmem_free(pSync);

    shmem_finalize();
    return 0;
}

```

## 8.4. Threadgroup Example

The example in this section shows how **shmemx\_collect32\_block** can be used to leverage threads to accelerate a SHMEM collect operation when all threads in the block depends on the result of a preceding communication operation. For this instance, partial vector sums are computed across different PEs and have a SHMEM collect operation to obtain the complete sum across PEs.

```

#include <stdio.h>
#include "shmem.h"
#include "shmemx.h"

#define NTHREADS 512

#define CUDA_CHECK(stmt) \
do { \
    cudaError_t result = (stmt); \
    if (cudaSuccess != result) { \
        fprintf(stderr, "[%s:%d] cuda failed with %s \n", \
            __FILE__, __LINE__, cudaGetErrorString(result)); \
        exit(-1); \
    } \
} while (0)

__global__ void distributed_vector_sum(int *x, int *y, int *partial_sum, int
*sum, long *pSync, int use_threadgroup, int mype, int npes)
{
    int index = threadIdx.x;
    int nelems = blockDim.x;
    int PE_start = 0;
    int logPE_stride = 0;
    partial_sum[index] = x[index] + y[index];

    if (use_threadgroup) {
        /* all threads realize the entire collect operation */
        shmemx_collect32_block(sum, partial_sum, nelems, PE_start, logPE_stride,
npes, pSync);
    } else {
        /* thread 0 realizes the entire collect operation */
        if (0 == index) {
            shmem_collect32(sum, partial_sum, nelems, PE_start, logPE_stride, npes,
pSync);
        }
    }
}

```

```

int main (int c, char *v[])
{
    int mype, npes;
    int *x;
    int *y;
    int *partial_sum;
    int *sum;
    int use_threadgroup = 1;
    long *pSync;
    int nthreads = NTHREADS;

    shmem_init ();
    npes = shmem_n_pes();
    mype = shmem_my_pe();

    CUDA_CHECK(cudaSetDevice(mype));

    x = (int *) shmem_malloc(sizeof(int) * nthreads);
    y = (int *) shmem_malloc(sizeof(int) * nthreads);
    partial_sum = (int *) shmem_malloc(sizeof(int) * nthreads);
    sum = (int *) shmem_malloc(sizeof(int) * nthreads * npes);
    pSync = (long *) shmem_malloc(sizeof(long) * SHMEM_COLLECT_SYNC_SIZE);

    void *args[] = {&x, &y, &partial_sum, &sum, &pSync, &use_threadgroup, &mype,
    &npes};
    dim3 dimBlock(nthreads);
    dim3 dimGrid(1);
    shmemx_collective_launch ((const void *)distributed_vector_sum, dimGrid,
    dimBlock, args, 0, 0);
    CUDA_CHECK(cudaDeviceSynchronize());

    printf("[%d of %d] run complete \n", mype, npes);

    shmem_free(x);
    shmem_free(y);
    shmem_free(partial_sum);
    shmem_free(sum);
    shmem_free(pSync);

    shmem_finalize();

    return 0;
}

```

## 8.5. put\_block Example

In the example below, every thread in block 0 calls **shmemx\_float\_put\_block**. Alternatively, every thread can call **shmem\_float\_p**, but **shmem\_float\_p** has a disadvantage that when the destination GPU is connected via InfiniBand, there is one RMA message for every single element, which can be detrimental to performance.

The disadvantage with using **shmem\_float\_put** in this case is that when the destination GPU is P2P-connected, a single thread will copy the entire data to the destination GPU. While **shmemx\_float\_put\_block** can leverage all the threads in the block to copy the data in parallel to the destination GPU.

```

#include <stdio.h>
#include <assert.h>
#include "mpi.h"
#include "shmem.h"
#include "shmemx.h"

```

```

#define CUDA_CHECK(stmt) \
do { \
    cudaError_t result = (stmt); \
    if (cudaSuccess != result) { \
        fprintf(stderr, "[%s:%d] cuda failed with %s \n", \
            __FILE__, __LINE__, cudaGetErrorString(result)); \
        exit(-1); \
    } \
} while (0)

#define THREADS_PER_BLOCK 1024

__global__ void set_and_shift_kernel (float *send_data, float *recv_data, int
num_elems, int mype, int npes) {
    int thread_idx = blockIdx.x * blockDim.x + threadIdx.x;

    /* set the corresponding element of send_data */
    if (thread_idx < num_elems)
        send_data[thread_idx] = mype;

    int peer = (mype + 1) % npes;

    if (Int block_offset = blockIdx.x == 0)
        * blockDim.x;
    /* All threads in the block call API with same arguments */
    shmemx_float_put_block(recv_data + block_offset, send_data + block_offset,
min(blockDim.x, num_elems, - block_offset), peer);
}

int main (int c, char *v[])
{
    int mype, npes;
    float *send_data, *recv_data;
    int num_elems = 8192;
    int num_blocks;

    shmem_init ();
    mype = shmem_my_pe();
    npes = shmem_n_pes();

    //application picks the device each PE will use
    CUDA_CHECK(cudaSetDevice(mype));
    send_data = (float *) shmem_malloc(sizeof(float) * num_elems);
    recv_data = (float *) shmem_malloc(sizeof(float) * num_elems);

    assert(num_elems % THREADS_PER_BLOCK == 0); /* for simplicity */
    num_blocks = num_elems / THREADS_PER_BLOCK;

    set_and_shift_kernel<<<num_blocks, THREADS_PER_BLOCK>>>
        (send_data, recv_data, num_elems, mype, npes);
    CUDA_CHECK(cudaDeviceSynchronize());

    printf("[%d of %d] run complete \n", mype, npes);

    shmem_free(send_data);
    shmem_free(recv_data);

    shmem_finalize();
    return 0;
}

```

## Chapter 9.

### FAQS

#### **Q: Does NVSHMEM require CUDA?**

A: Yes. CUDA 9.0 or later must be installed to use NVSHMEM. NVSHMEM is a communication library intended to be used for efficient data movement and synchronization between two or more GPUs. It is currently not intended for data movement that does not involve GPUs.

#### **Q: Does NVSHMEM require MPI?**

A: No. NVSHMEM applications without MPI dependencies can use NVSHMEM and be launched with the Hydra launcher packaged with NVSHMEM. A stand-alone build of the Hydra launcher can also be used.

#### **Q: Can NVSHMEM be used in MPI applications?**

A: Yes. NVSHMEM provides an initialization API that takes an MPI communicator as an attribute. Each MPI rank in the communicator becomes an OpenSHMEM PE. Currently, NVSHMEM has been tested with OpenMPI 4.0.0. In principle, other OpenMPI derivatives such as SpectrumMPI (available on Summit and Sierra) are also expected to work.

#### **Q: Can NVSHMEM be used in OpenSHMEM applications?**

A: Yes. NVSHMEM provides an initialization API that supports running NVSHMEM on top of an OpenMPI/OSHMEM job. Each OSHMEM PE maps 1:1 to an NVSHMEM PE. NVSHMEM has been tested with OpenMPI 4.0.0/OSHMEM and OpenMPI3+/OSHMEM depends on UCX (NVSHMEM has been tested with UCX 1.4.0). The OpenMPI-4.0.0 installation must be configured with the `--with-ucx` flag to enable OpenSHMEM + NVSHMEM interoperability.



**Q: Can I use NVSHMEM to transfer data across GPUs on different sockets?**

A: Yes, if there is an InfiniBand NIC accessible to GPUs on both the sockets. Otherwise, NVSHMEM requires that all GPUs are P2P accessible.

**Q: Can I use NVSHMEM to transfer data between P2P-accessible GPUs that are connected by PCIE?**

A: Yes, NVSHMEM supports both PCIE and NVLink. Atomic memory operations are only supported between NVLink-connected GPUs.

**Q: Can I use NVSHMEM to transfer data between GPUs on different hosts connected by InfiniBand?**

A: Yes. NVSHMEM supports InfiniBand. Strided-RMA (`shmem_put/get`), single-element get (`shmem_g`), and atomic memory operations are not supported over InfiniBand.

**Q: Can two PEs share the same GPU with NVSHMEM?**

A: NVSHMEM assumes a 1:1 mapping of PEs to GPUs. NVSHMEM jobs launched with more PEs than available GPUs are not supported.

**Q: My NVSHMEM job runs on NVIDIA Volta GPUs but hangs on NVIDIA Kepler GPUs. What should I do?**

A: NVSHMEM Synchronizing APIs inside the CUDA kernel is only supported on NVIDIA Pascal and NVIDIA Volta GPUs.

**Q: Can I run NVSHMEM on a host without InfiniBand NICs?**

A: Yes. Support on P2P platforms remains unchanged.

**Q: Can I run NVSHMEM on a host with InfiniBand NICs where some NICs are disabled or configured in a non-InfiniBand mode?**

A: Yes. See the [Useful Environment Variables](#) section for how to explicitly specify NIC affinity to PEs.

**Q: When should I use the nv-prefix version of an API or APIs?**

A: If your application will use OpenMPI/OSHMEM, use the nv-prefix versions of the APIs. If not, either version can be used.

**Q: How should I allocate memory for NVSHMEM?**

A: NVSHMEM supports `shmem_malloc` and `shmem_align` memory allocation APIs. Per the OpenSHMEM specification v1.4, these APIs only require the remote pointer to be

from the symmetric heap (SHEAP). However, NVSHMEM also requires the local pointer to be from SHEAP for communication with a remote peer connected by InfiniBand. If the remote peer is P2P accessible (PCI-E or NVLink), the local pointer can be obtained using `cudaMalloc` and is not required to be from the SHEAP.

### Q: What does the following runtime warning imply?

```
WARN: IB HCA and GPU are not connected to a PCIe switch so InfiniBand
performance can be limited depending on the CPU generation
```

A: This warning is related to the HCA to GPU mapping of the platform. For more information, refer to the `SHMEM_HCA_PE_MAPPING` variable in the [Useful Environment Variables](#) section.

### Q: What does the following runtime error indicate?

```
NULL value could not find mpi library in environment
```

A: This occurs if `libmpi.so` is not present in the environment. For example, Spectrum MPI installs `libmpi_ibm.so`. For more information, refer to the `NVSHMEM_MPI_LIB_NAME` variable in the [Useful Environment Variables](#) section to specify the name of the MPI library installed.

### Q: What does the following runtime error indicate?

```
src/comm/transport/ibrc/ibrc.cpp:867: NULL value mem registration failed
```

A: This occurs if `GPUDirectRDMA` is not enabled on the platform, thereby preventing registration of `cudaMalloc` memory with the InfiniBand driver. This usually indicates that the `nv_peer_mem` kernel module is absent. When `nv_peer_mem` is installed, output from `lsmod` is similar to the following:

```
~$ lsmod | grep nv_peer_mem
nv_peer_mem      20480 0
ib_core          241664 11
rdma_cm,ib_cm,iw_cm,nv_peer_mem,mlx4_ib,mlx5_ib,ib_ucm,ib_umad,ib_uverbs,rdma_ucm,ib_ipoib
nvidia           17596416 226
nv_peer_mem,gdrdrv,nvidia_modeset,nvidia_uvm
nv_peer_mem is available here
https://github.com/Mellanox/nv_peer_memory
```

### Q: Why does NVSHMEM package provide installation script for hydra process manager?

A: NVSHMEM packages the installation script for the Hydra Process Manager to enable standalone NVSHMEM application development. Specifically, you can write an NVSHMEM program and run a multi-process job using the Hydra Process Manager. This eliminates any dependency on installing MPI to use NVSHMEM. The Hydra launcher is called `mpiexec.hydra` and the default Hydra build system installs two symbolic links, `mpiexec` and `mpirun`. Run `mpiexec.hydra -h` for help information.

**Q: Are there environment variables, configuration files, or parameters that need to be set for `mpirun`?**

A: Run `mpiexec.hydra -h` for comprehensive information on these topics. The following is the minimum required command line for a multi-node run on 2 hosts with 2 GPUs on each host:

```
mpirun -n 4 -ppn 2 -hosts hostname1,hostname2 /path/to/nvshmem/app/binary
```

For a single node, run:

```
mpirun -n 2 /path/to/nvshmem/app/binary
```

**Q: Why does my NVSHMEM Hydra job become non-responsive on Summit?**

A: Summit requires the additional option `--launcher ssh` to be passed to `mpiexec.hydra` at the command line.

**Q: How do I dump debugging information?**

A: This is currently not available as a runtime switch in NVSHMEM. Report any errors to NVIDIA for assistance.

**Q: How is an MPI+NVSHMEM app launched on Summit? Should `jsrun`, SMPI's `mpirun`, `mpirun` from local install of OpenMPI or the `mpirun` shipped with NVSHMEM be used?**

A: We have successfully tested with `jsrun`, OpenMPI's `mpirun` and Hydra `mpirun` shipped with NVSHMEM. Only `jsrun` is officially supported on Summit. The following is a sample command line to use 2 nodes, 4 GPUs/node with `jsrun`:

```
jsrun -n 2 -r 1 -a 4 -c 40 -g 4 -b packed:10 -d packed /path/to/nvshmem/app/binary
```

One possible way to query hostnames for using OpenMPI and Hydra's `mpirun` is the following:

```
jsrun -n 2 -r 1 hostname
```

**Q: My application uses the CMake build system. Adding NVSHMEM to the build system breaks for a CMake version below 3.11. Why?**

A: Device linking support was added in version 3.11 which NVSHMEM requires.

**Q: What's the difference between, say, `shmemx_putmem_on_stream` and `shmemx_putmem_nbi_on_stream`? It seems both are asynchronous to the host thread and ordered with respect to a given stream.**

A: The function `putmem_nbi_on_stream` is implemented in a more deferred way by not issuing the transfer immediately but making it wait on an event at the end of the

stream. If there is another transfer in process at the same time (on another stream), bandwidth could be shared. If the application can avoid this, `nbi_on_stream` gives the flexibility to express this intent to NVSHMEM. But NVSHMEM currently does not track activity on all CUDA streams. The current implementation records an event on the user provided stream, makes an NVSHMEM internal stream wait on the event, and then issues a put on the internal stream. If all nbi puts land on the same internal stream, they are serialized so that the bandwidth is used exclusively.

**Q: Can I issue multiple `barrier_all_on_stream` on multiple streams concurrently and then `cudaStreamSynchronize` on each stream?**

A: Multiple concurrent `barrier_all_on_stream` calls are not valid. Only one barrier (or any other collective) among the same set of PEs can be in-flight at any given time. To use concurrent barriers among partially overlapping active sets, `syncneighborhood_kernel` can be used as a template to implement a custom barrier. See the following for an example of a custom barrier: [multi-gpu-programming-models](#).

**Q: Suppose there are in-flight `putmem_on_stream`. Does `shmem_barrier_all()` ensure completion of the pending shmem operations on streams?**

A: The `StreamSynchronize` function needs to be called before calling `shmem_barrier_all`. `barrier_all_on_stream` appears to hang non-deterministically.

**Q: Is there any hint to diagnose the hang?**

A: Check if there are stream 0 blocking CUDA calls from the application, like `cudaDeviceSynchronize` or `cudaMemcpy`, especially in the iterative phase of the application. Stream 0 blocking calls in the initialization and finalization phases are usually safe. Check if the priority of the user stream used for `NVSHMEM_on_stream` calls is explicitly set with `cudaStreamCreateWithPriority`. Check that the determinism of the hang changes with single-node (all pairs of GPUs connected by NVLink or PCI-E only) compared to single-node (GPUs on different sockets connected by Infiniband loopback) or multi-node (GPUs connected by InfiniBand).

**Q: Why does a CMake build of my NVSHMEM application fail with version 3.12 but does not with an earlier version?**

A: A new CMake policy adds `-pthread` to the `nvcc` device linking causing the linking failure. Before 3.12, the default policy did not add `-pthread`. For 3.12 and newer, add the following:

```
cmake_policy(SET CMP0074 OLD) " to CMakeLists.txt
```

**Q: Why is `shmem_quiet` necessary in the `syncneighborhood_kernel`?**

A: It is required by `shmem_barrier` semantics. As stated in [multi-gpu-programming-models](#), "...`shmem_barrier` ensures that all previously issued stores and remote memory updates, including AMO and RMA operations, done by any of the PEs in the active set on the default context are complete before returning."

**Q: If a kernel uses `shmem_put_block` instead of `shmem_p`, is the `shmem_quiet` still required?**

A: It is required per OpenSHMEM's requirement of put semantics which do not guarantee delivery of data to the destination array on the remote PE. For more information, see [multi-gpu-programming-models](#).

**Q: I use the host-side blocking API, `shmem_putmem_on_stream`, on the same CUDA stream that I want to be delivered at the target in order. Is `shmem_quiet` required even though there is no non-blocking call and they are issued in separate kernels?**

A: In the current implementation, `shmem_putmem_on_stream` includes quiet. However, it is only required to release the local buffer and not necessarily deliver at the target by the OpenSHMEM spec.

**Q: Is it sufficient to use a `shmem_fence` (instead of a `shmem_quiet`) in the above case if the target is the same PE?**

A: In the current implementation, all messages to the same PE are delivered in the order they are received by the HCA, which follows the stream order. So, even `shmem_fence` is not required. These are not the semantics provided by the OpenSHMEM specification, however. The `putmem_on_stream` function on the same CUDA stream only ensures that the local buffers for the transfers will be released in the same order.

**Q: When `shmem_quiet` is used inside a device kernel, is the quiet operation scoped within the stream the kernel is running on? In other words, does it ensure completion of all operations or only those issued to the same stream?**

A: It ensures all operations that are GPU-initiated. A `shmem_quiet()` call on the device does not quiet in-flight operations from the host.

**Q: Does pointer arithmetic work with `shmem` pointers? For example, `int* outmsg = (int *) shmem_malloc(2* sizeof(int)); shmem_int_p(target + 1, mype, peer)?`**

A: Yes.

**Q: Can I avoid `cudaDeviceSynchronize` + `MPI_Barrier` to synchronize across multiple GPUs?**

A: Yes, `shmem_barrier_all_on_stream` with `cudaStreamSynchronize` can be called from the host thread. If multiple barrier synchronization events can happen before synchronizing with the host thread, this gives better performance. Calling `shmem_barrier_all` from inside the CUDA kernel can be used for collective synchronization if there are other things that can be done by the same CUDA kernel after a barrier synchronization event. For synchronizing some pairs of PEs and not all, pair-wise `shmem_atomic_set` calls by the initiator and `shmem_wait_until` or `shmem_test` calls by the target can be used.

## Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

## Trademarks

NVIDIA, the NVIDIA logo, DGX, DGX-1, DGX-2, and DGX Station are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

## Copyright

© 2019 NVIDIA Corporation. All rights reserved.