CUDA GRAPHS AND DEVICE-INITIATED COMMUNICATION WITH NVSHMEM

JIRI KRAUS, PRINCIPAL DEVTECH COMPUTE

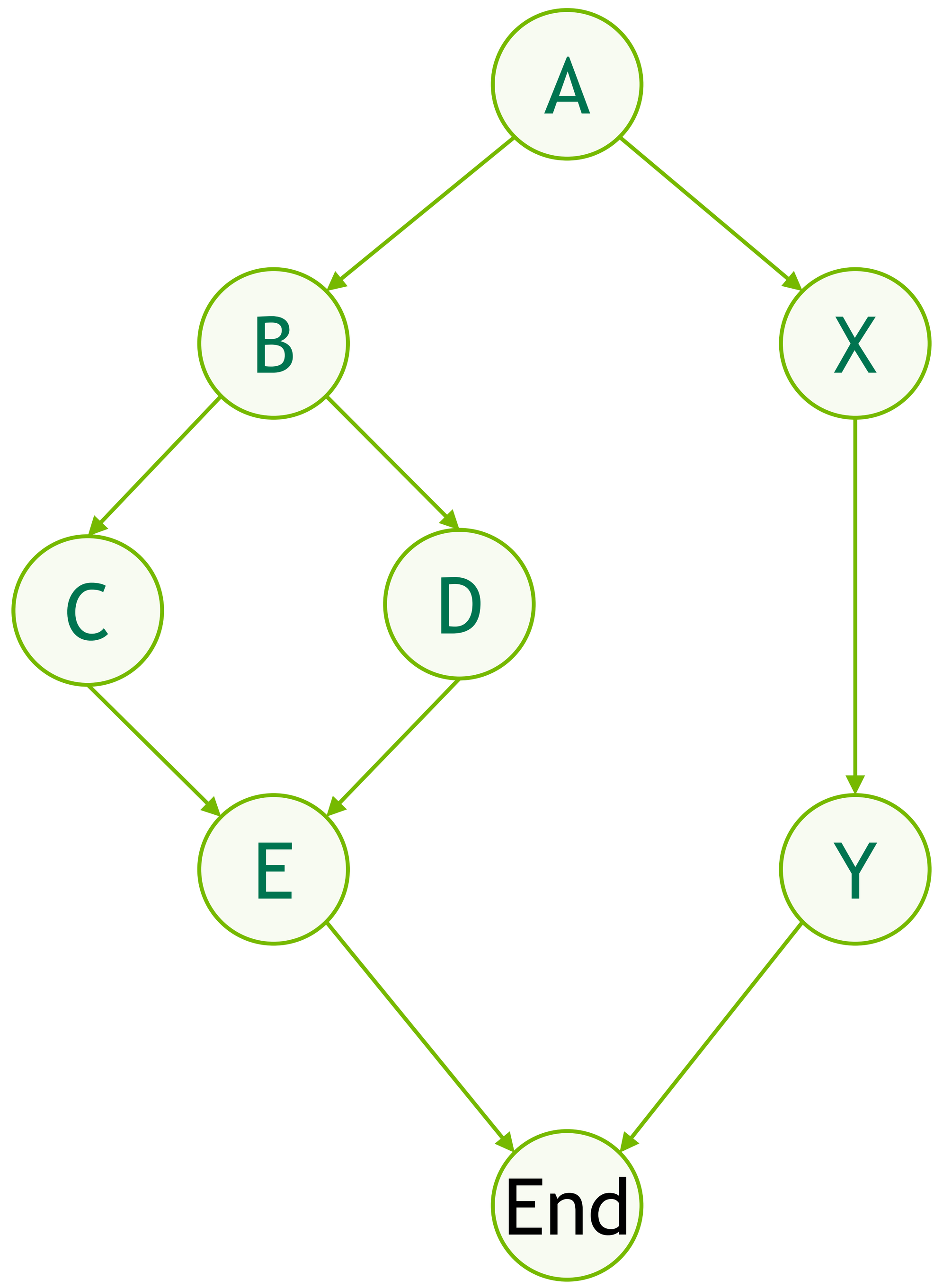
MULTI GPU JACOBI NSIGHT SYSTEMS TIMELINE

NCCL 8 NVIDIA A100 40GB on JUWELS Booster

ASYNCHRONOUS TASK GRAPH

A Graph Node Is A CUDA Operation

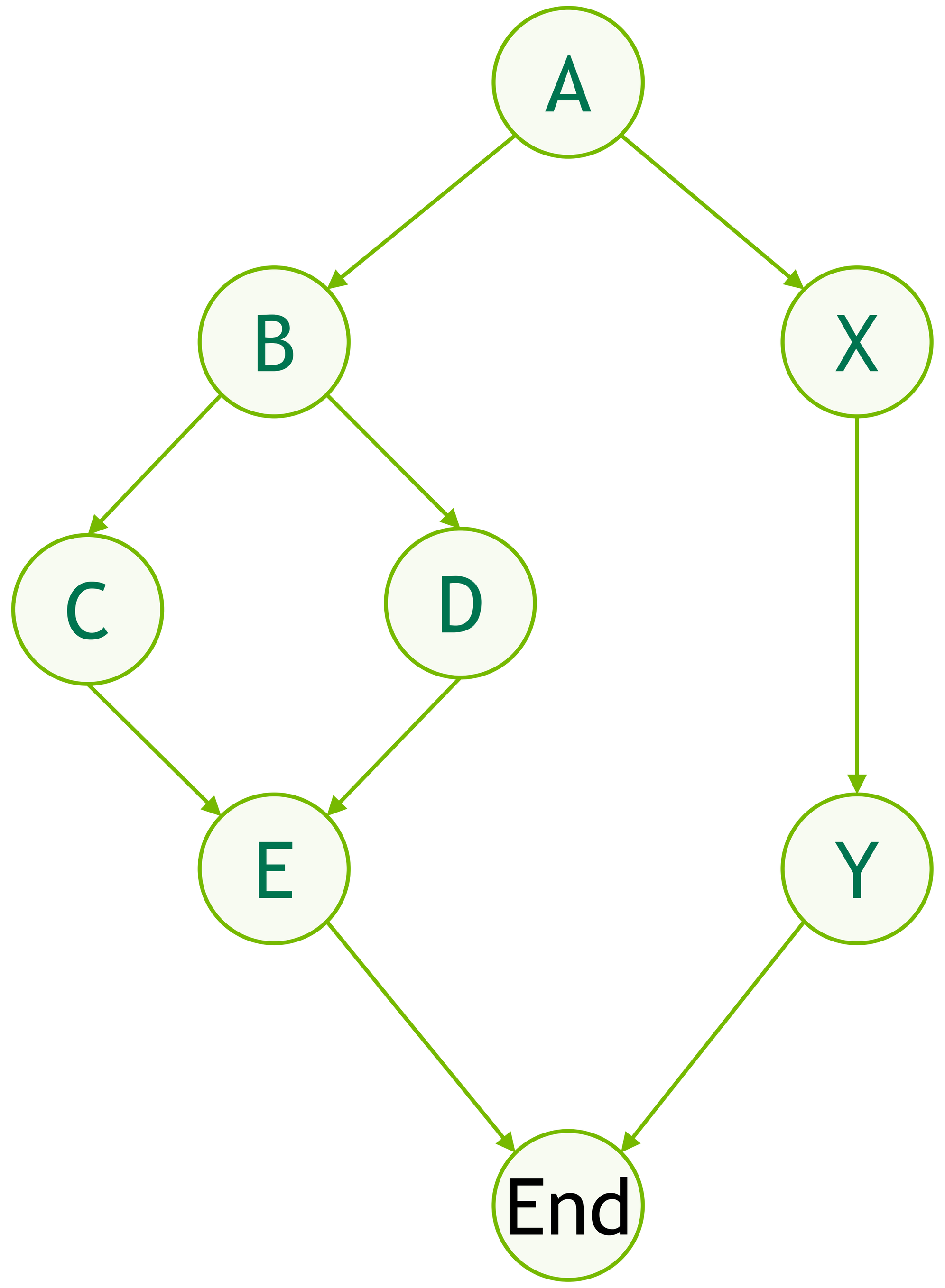
- Sequence of operations (nodes), connected by dependencies
- Operations are one of:
    - Kernel Launch — CUDA kernel running on GPU
    - CPU Function Call — Callback function on CPU
    - Memcopy/Memset — GPU data management
    - Mem Alloc/Free — Memory management
    - External Dependency — External semaphores/events
    - Sub-Graph — Graphs are hierarchical
- Nodes within a graph can also span multiple devices



THREE-STAGE EXECUTION MODEL

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

Define

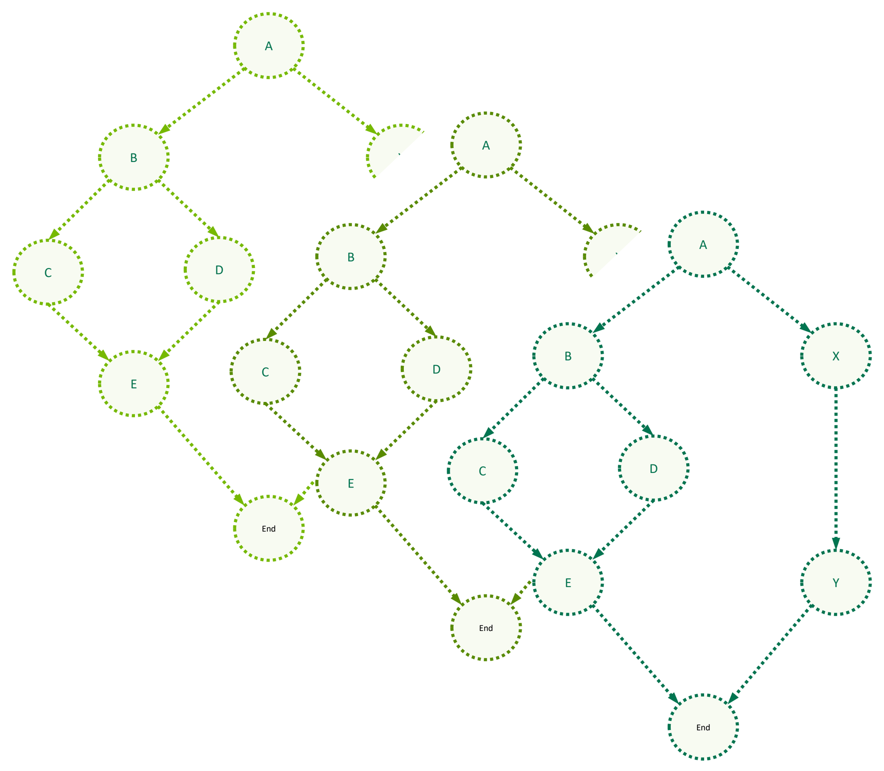


Single Graph “Template”

Created in host code

or built up from libraries

Instantiate

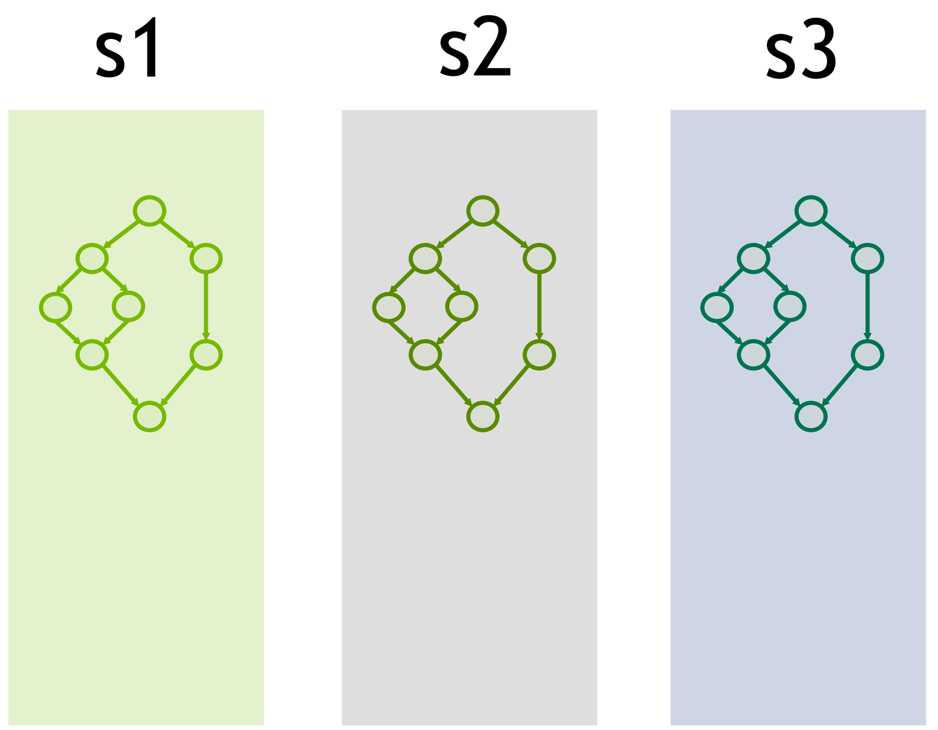


Multiple “Executable Graphs”

Snapshot of templates

Sets up & initializes GPU execution structures (create once, run many times)

Execute



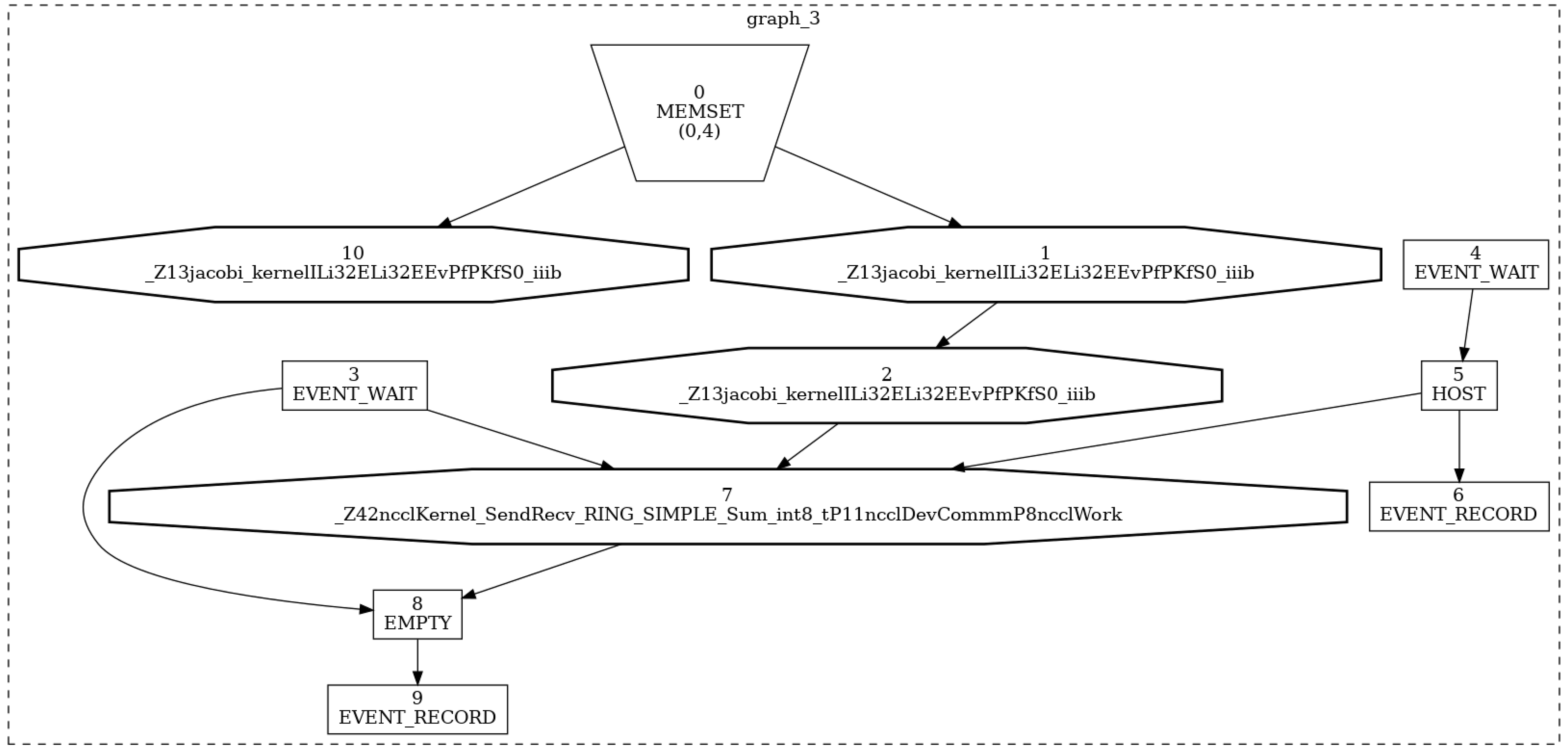
Executable Graphs Running in CUDA Streams

Concurrency in graph **is not** limited by stream

NEW EXECUTION MECHANISM

Graphs Can Be Generated Once Then Launched Repeatedly

```
while (l2_norm > tol && iter < iter_max) {

    cudaGraphLaunch(graph_calc_norm_exec[iter%2],
                    compute_stream);
    cudaStreamSynchronize(compute_stream);
    MPI_Allreduce(l2_norm_h, &l2_norm, 1,
                  MPI_REAL_TYPE, MPI_SUM,
                  MPI_COMM_WORLD);
    l2_norm = std::sqrt(l2_norm);

    if (!csv && 0 == rank && (iter % 100) == 0) {
        printf("%5d, %0.6f\n", iter, l2_norm);
    }
}
```



Generated with

```
cudaGraphDebugDotPrint(graphs[calculate_norm][0],
                       "jacobi_graph.dot",0)
```

and

```
dot -Tpng jacobi_graph.dot -o jacobi_grap.png
```

WHERE IS PERFORMANCE COMING FROM?

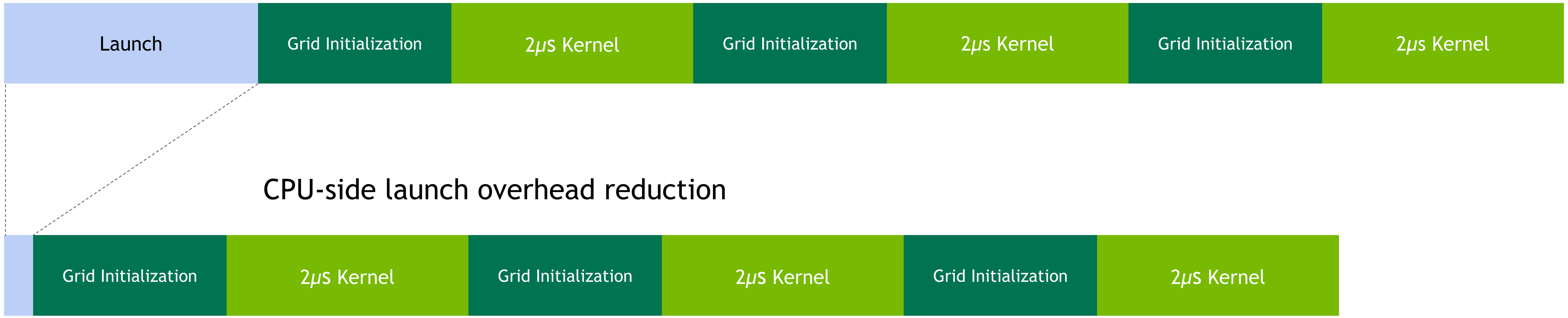
Reducing System Overheads Around Short-Running Kernels

Breakdown of time spent during execution

WHERE IS PERFORMANCE COMING FROM?

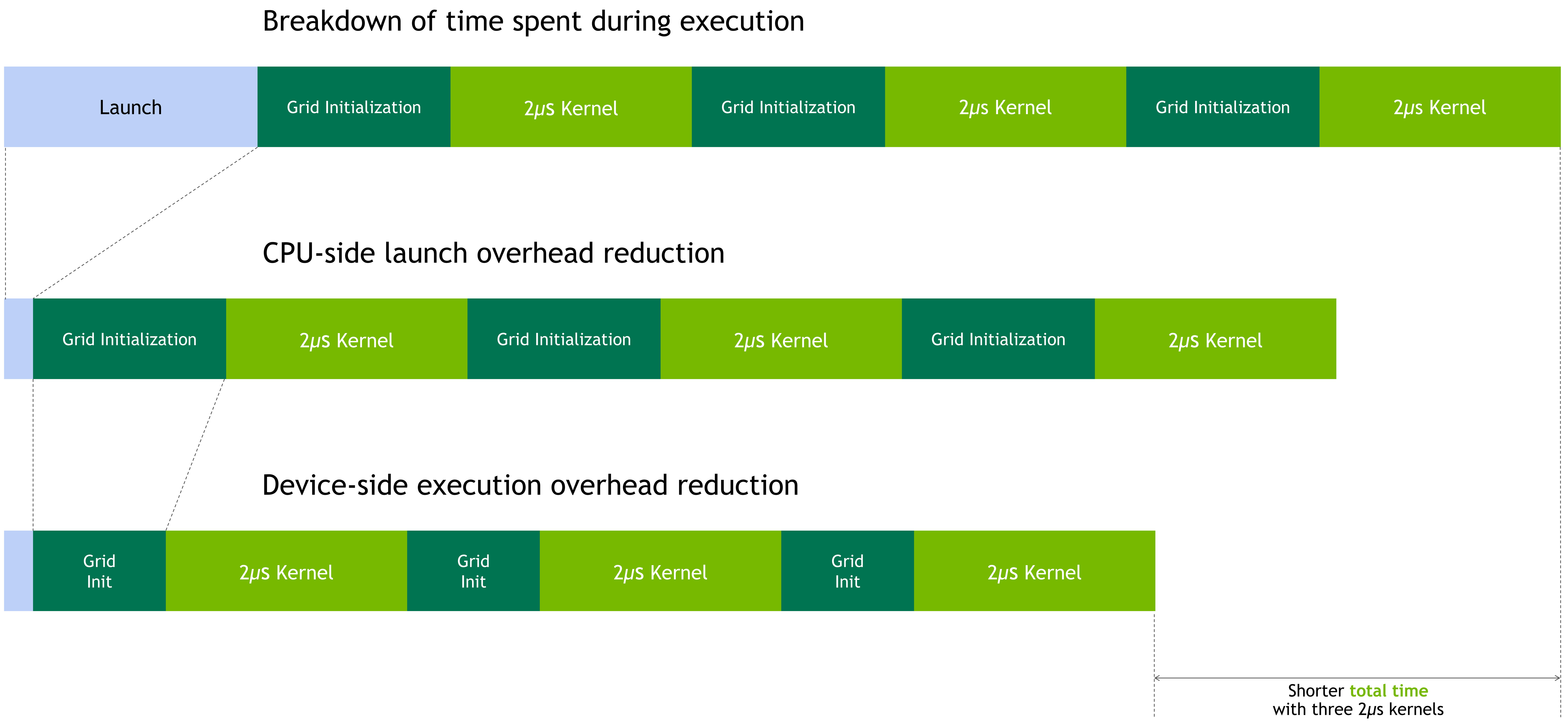
Reducing System Overheads Around Short-Running Kernels

Breakdown of time spent during execution



WHERE IS PERFORMANCE COMING FROM?

Reducing System Overheads Around Short-Running Kernels



MULTI GPU JACOBI NSIGHT SYSTEMS TIMELINE

NCCL with CUDA Graphs 8 NVIDIA A100 40GB on JUWELS Booster

#	Name	Start	Duration	TID
1	cudaGraphLaunch	5.73812s	29.860 µs	32375
2	cudaStreamSynchronize	5.73815s	366.443 µs	32375

CAPTURE STREAM WORK INTO A GRAPH

Create A Graph With Two Lines of Code

```
{

    cudaStreamBeginCapture(compute_stream, cudaStreamCaptureModeGlobal);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream));
    cudaEventRecord(reset_l2norm_done, compute_stream);

    ...

    cudaStreamWaitEvent(compute_stream, push_done, 0);

    cudaStreamEndCapture(compute_stream, graphs[calculate_norm]+is_even);

    std::swap(a_new, a);
    iter++;
}
```

CUDA GRAPH MANAGEMENT API

Instantiate CUDA Graphs

```
__host__cudaError_t cudaGraphInstantiateWithFlags ( cudaGraphExec_t* pGraphExec, cudaGraph_t graph,
                                                     unsigned long long flags )

__host__cudaError_t cudaGraphInstantiate ( cudaGraphExec_t* pGraphExec, cudaGraph_t graph,
          cudaGraphNode_t* pErrorNode, char* pLogBuffer, size_t bufferSize )
```

- `pGraphExec` [OUT]: Returns instantiated graph
- `graph` [IN]: Graph to instantiate
- `flags` [IN]: Flags to control instantiation (`cudaGraphInstantiateFlagAutoFreeOnLaunch |` **`cudaGraphInstantiateFlagUseNodePriority`**).
- `pErrorNode` [OUT]: In case of an instantiation error, this may be modified to indicate a node contributing to the error
- `pLogBuffer` [OUT]: A character buffer to store diagnostic messages
- `bufferSize` [IN]: Size of the log buffer in bytes

Returns: `cudaSuccess`, `cudaErrorInvalidValue`

CUDA GRAPH MANAGEMENT API

free resources

```
__host__cudaError_t cudaGraphDestroy ( cudaGraph_t graph )
```

- `graph` [IN]: Graph to destroy

Returns: `cudaSuccess`, `cudaErrorInvalidValue`

Destroys the `graph` specified by graph, as well as all of its nodes.

```
__host__cudaError_t cudaGraphExecDestroy ( cudaGraphExec_t graphExec )
```

- `graphExec` [IN]: Executable graph to destroy

Returns: `cudaSuccess`, `cudaErrorInvalidValue`

Destroys the executable graph specified by `graphExec`.

CPU-INITIATED COMMUNICATION

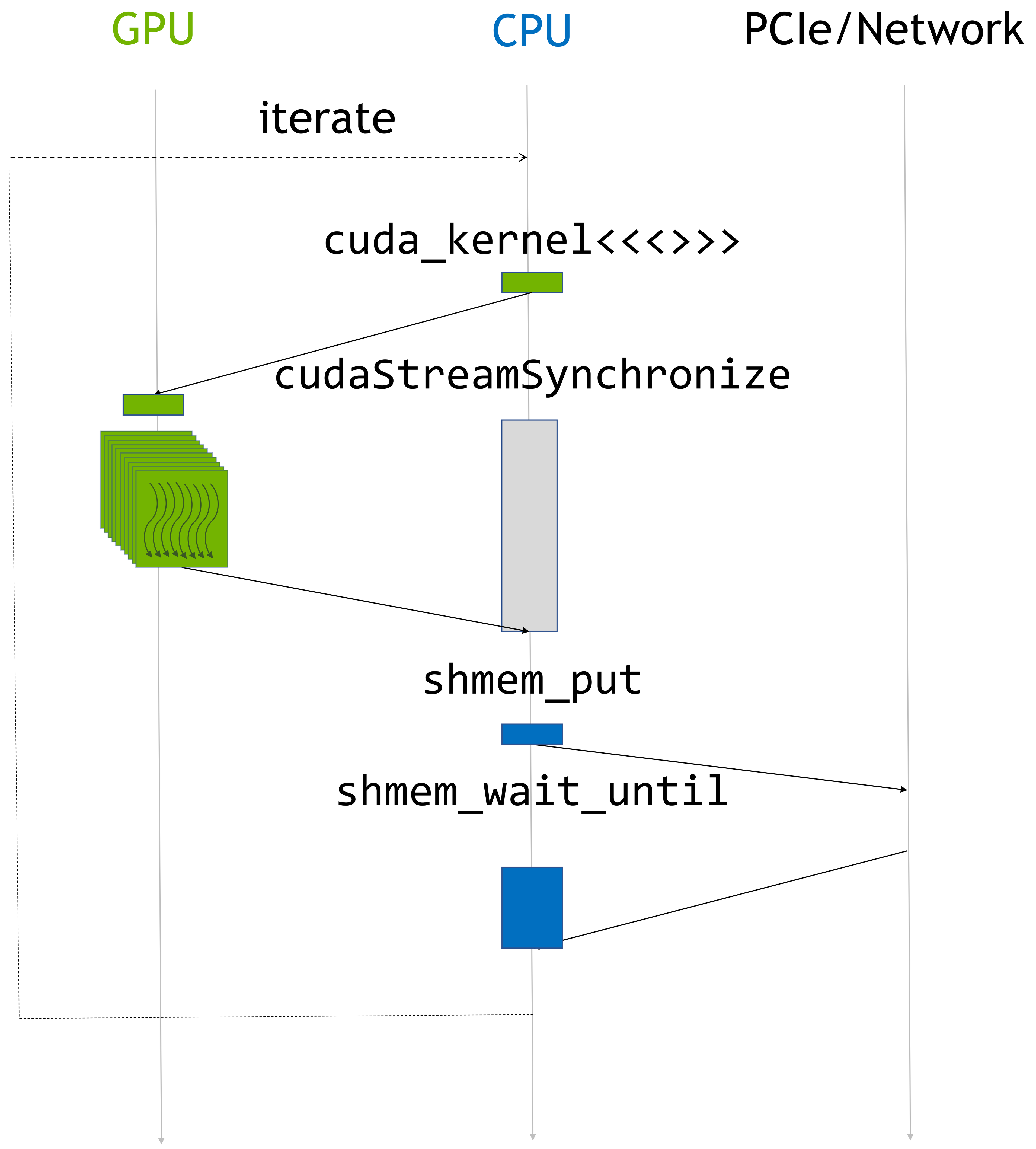
- Compute on GPU
- Communication from CPU

Synchronization at boundaries

Commonly used model, but –

- Offload latencies in critical path
- Communication is not overlapped

Hiding increased code complexity, not hiding limits strong scaling

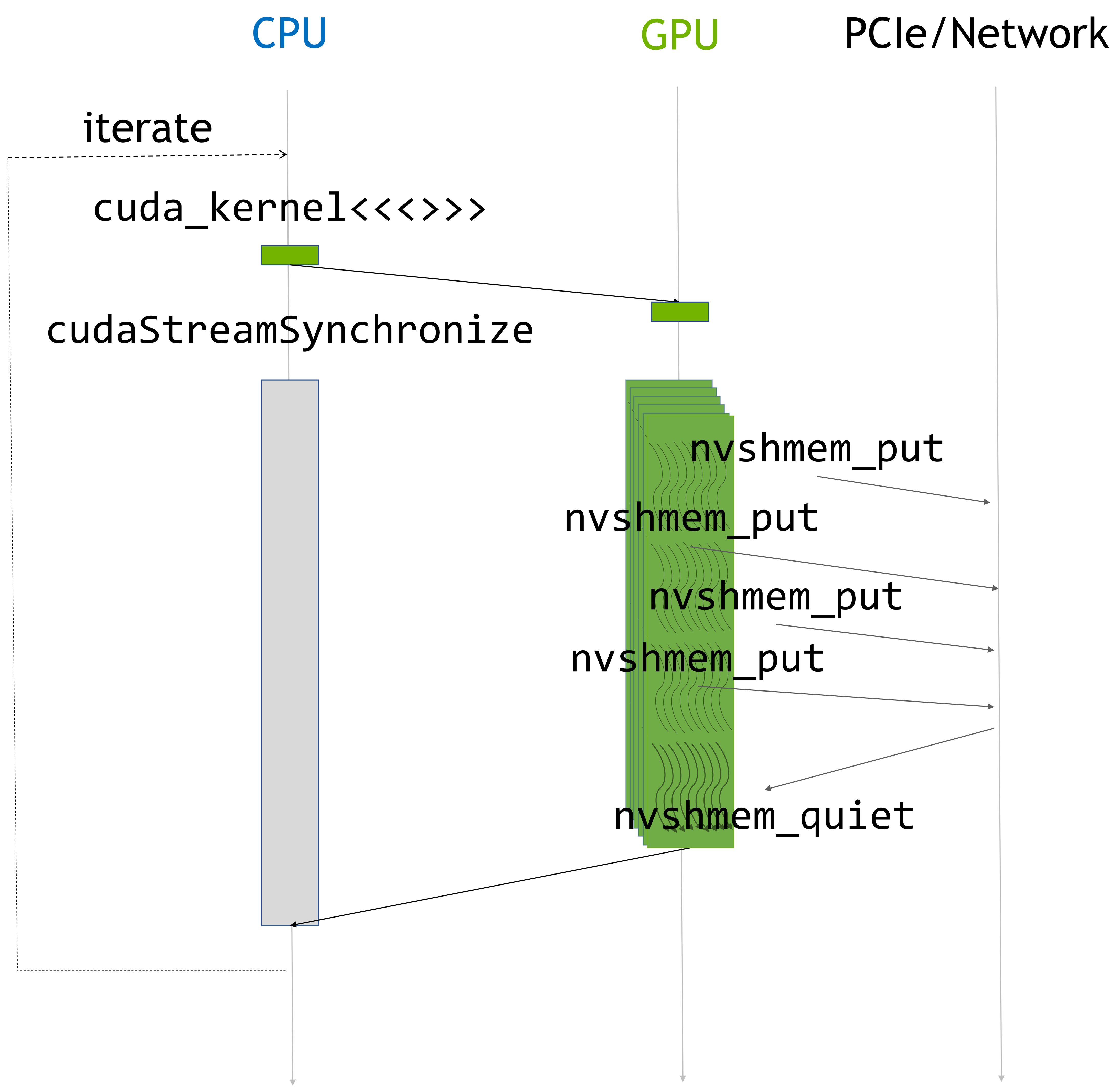


GPU-INITIATED COMMUNICATION

- Compute on GPU
- Communication from GPU

Benefits

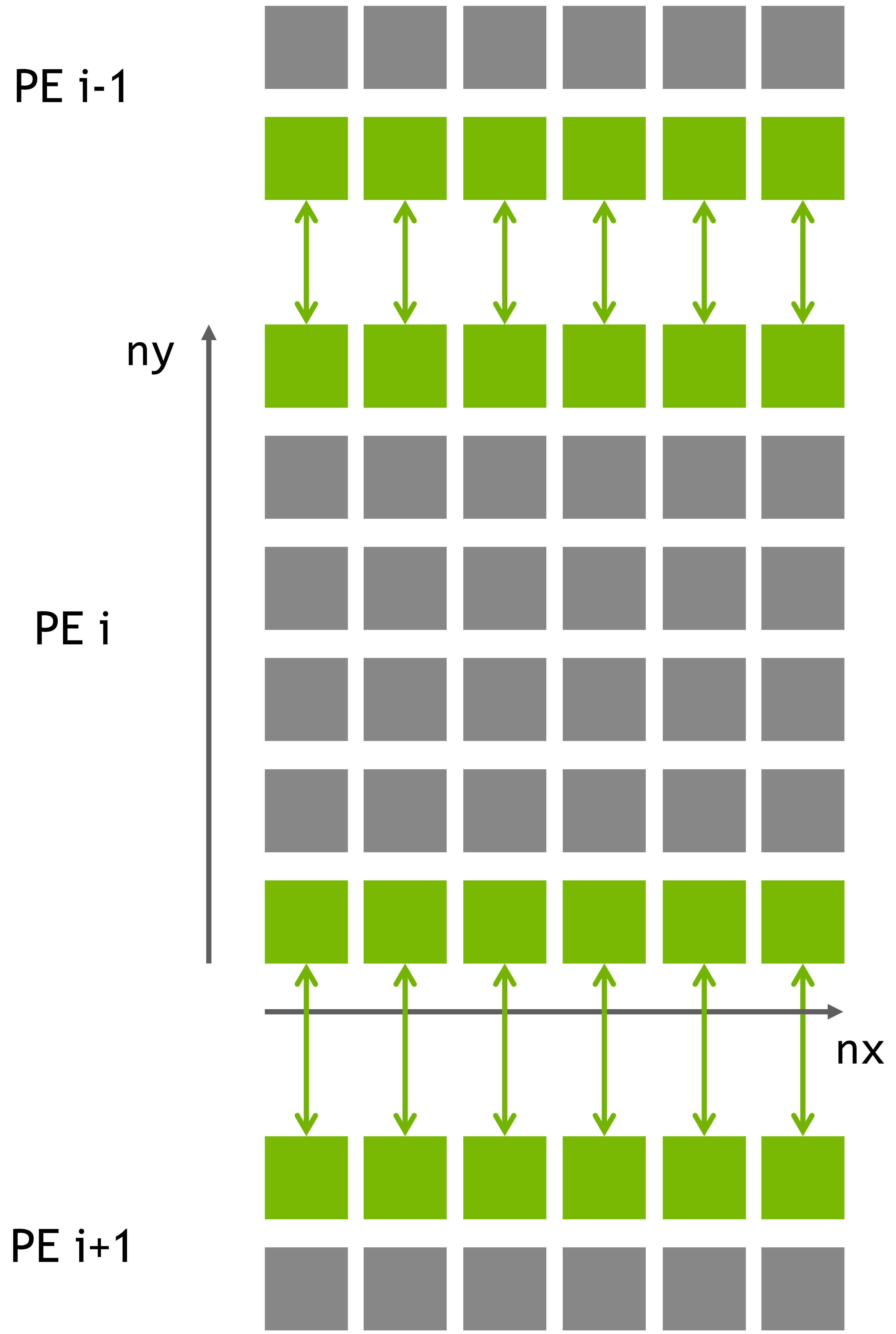
- Eliminates offloads latencies
- Compute and communication overlap by threading
- Easier to express algorithms with inline communication



THREAD-LEVEL COMMUNICATION

- Allows fine grained communication and computation overlap
- Efficient mapping to NVLink fabric on DGX systems

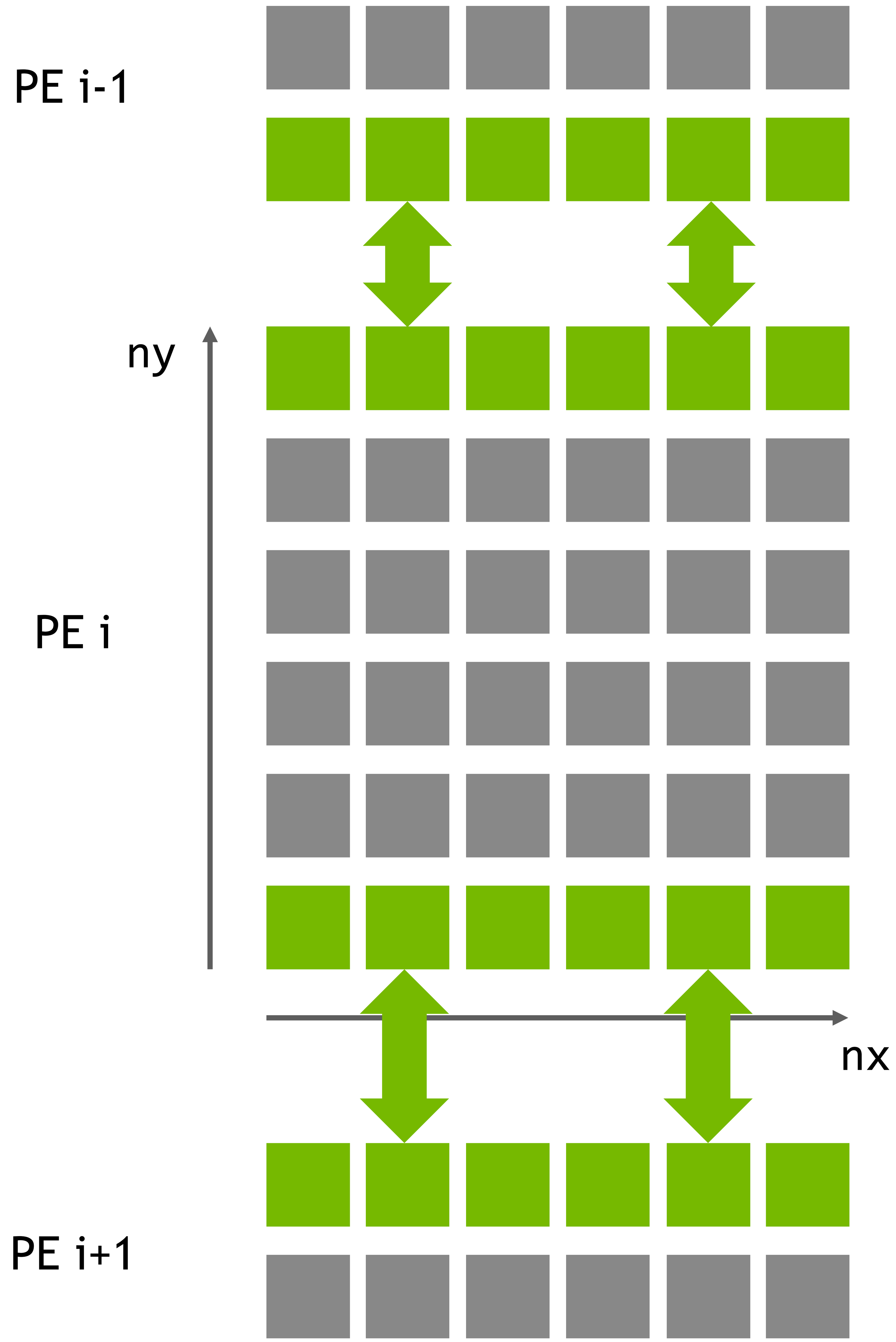
```
__global__ void stencil_single_step(float *u, float *v, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);
  compute(u, v, ix, iy);
  // Thread-level data communication API

  if (iy == 1)
    nvshmem_float_p(u+(ny+1)*nx+ix, u[nx+ix], top_pe);
  if (iy == ny)
    nvshmem_float_p(u+ix, u[ny*nx+ix], bottom_pe);
}
for (int iter = 0; iter < N; iter++) {
  swap(u, v);
  stencil_single_step<<<..., stream>>>(u, v, …);
  nvshmem_barrier_all_on_stream(stream);
}
```



THREAD-GPOUP COMMUNICATION

- NVSHMEM operations can be issued by all threads in a block/warp
- More efficient data transfers over networks like IB
- Still allows inter-warp/inter-block overlap

```
__global__ void stencil_single_step(float *u, float *v, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);
  compute(u, v, ix, iy);
  // Thread block-level communication API
  int boffset = get_block_offet(blockIdx,blockDim);
  if (blockIdx.y == 0)
    nvshmemx_float_put_nbi_block(u+(ny+1)*nx+boffset, u+nx+boffset, blockDim.x, top_pe);
  if (blockIdx.y == (blockDim.y-1))
    nvshmemx_float_put_nbi_block(u+boffset, u+ny*nx+boffset, blockDim.x, bottom_pe);
}
for (int iter = 0; iter < N; iter++) {
  swap(u, v);
  stencil_single_step<<<..., stream>>>(u, v, …);
  nvshmem_barrier_all_on_stream(stream);
}
```

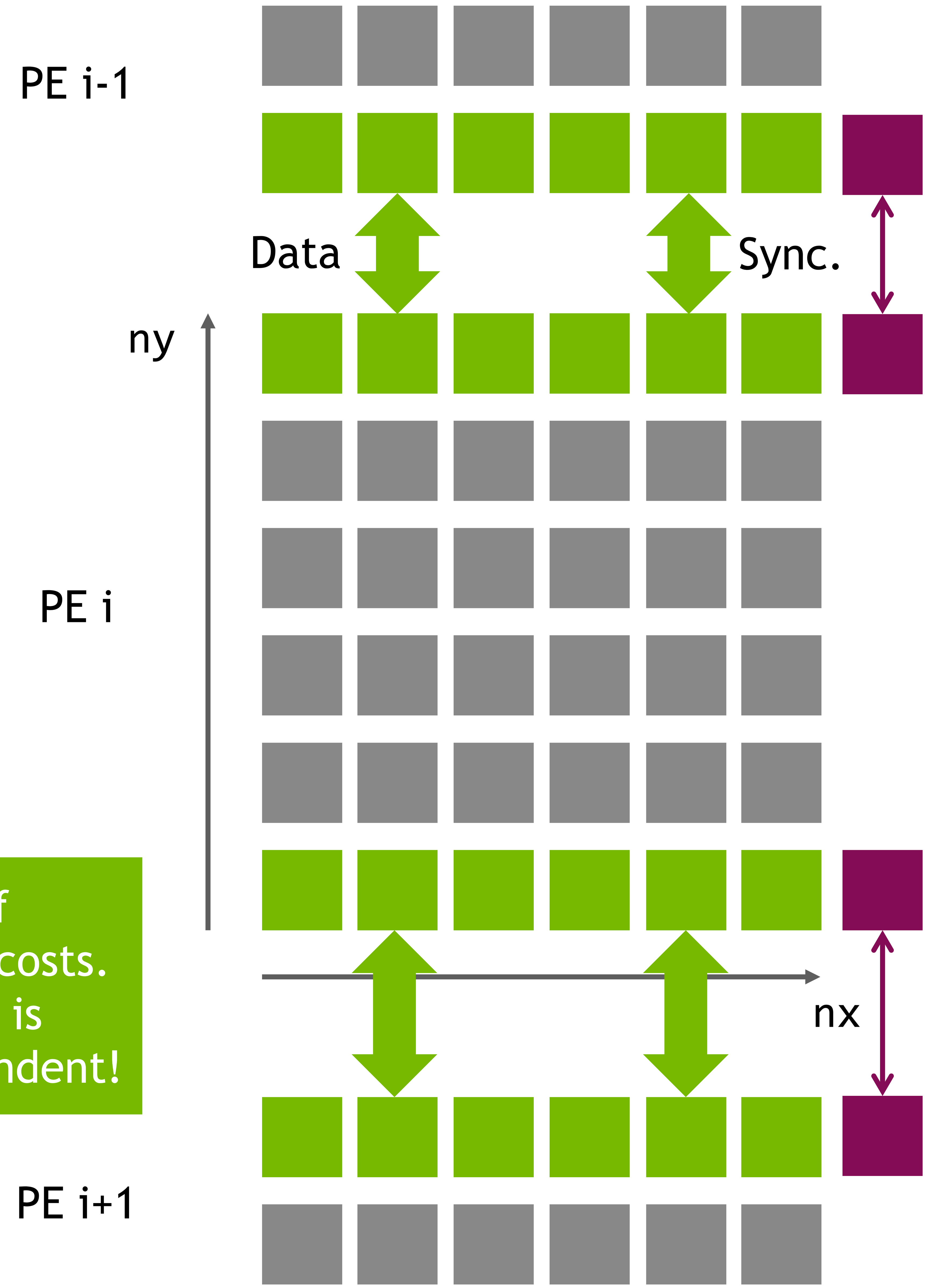


IN-KERNEL SYNCHRONIZATION

- Point-to-point synchronization across PEs within a kernel
- Enables kernel fusion

```
__global__ void stencil_multi_step(float *u, float *v, int N, int *sync, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);
  for (int iter = 0; iter < N; iter++) {
    swap(u, v); compute(u, v, ix, iy);
    // Thread block-level data exchange (assume even/odd iter buffering)
    int boffset = get_block_offet(blockIdx,blockDim);
    if (blockIdx.y == 0)
      nvshmemx_float_put_nbi_block(u+(ny+1)*nx+boffset, u+nx+boffset, blockDim.x, top_pe);
    if (blockIdx.y == (blockDim.y-1))
      nvshmemx_float_put_nbi_block(u + boffset, u+ny*nx+boffset,  blockDim.x, bottom_pe);
    if (blockIdx.y == 0 || blockIdx.y == (blockDim.y-1)) {
      __syncthreads();
      nvshmem_quiet();
      if (threadIdx.x == 0 && threadIdx.y == 0) {
        nvshmem_atomic_inc(sync, top_pe);
        nvshmem_atomic_inc(sync, bottom_pe);
    }}
    nvshmem_wait_until(sync, NVSHMEM_CMP_GT, 2*iter*gridDim.x);
}}
```

Be aware of synchronization costs. Best strategy is application dependent!



COLLECTIVE KERNEL LAUNCH

Ensures progress when using device-side inter-kernel synchronization

NVSHMEM Usage	CUDA Kernel launch
Device-Initiated Communication	Execution config syntax `<<<...>>>` or launch APIs
Device-Initiated Synchronization	`nvshmemx_collective_launch`

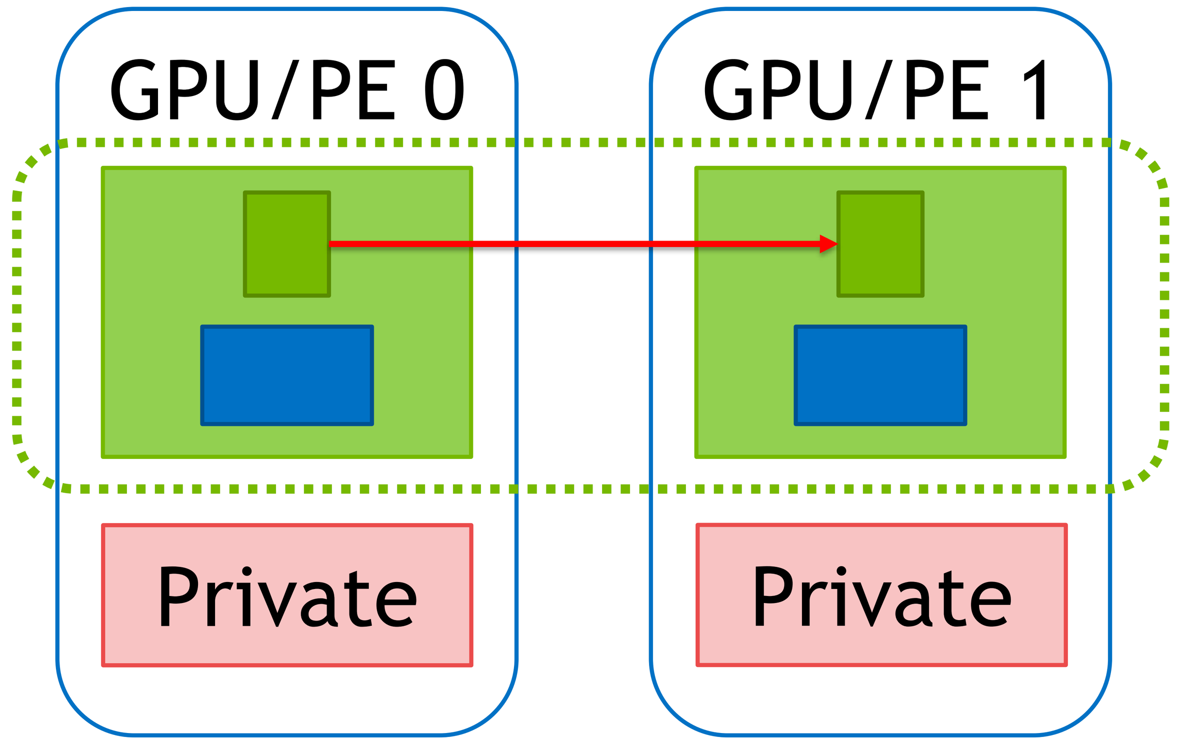
- CUDA's throughput computing model allows (encourages) grids much larger than a GPU can fit
- Inter-kernel synchronization requires producer and consumer threads to execute concurrently
- Collective launch guarantees co-residency using CUDA cooperative launch and requirement of 1PE/GPU

NVSHMEM API

single element put

```
__device__ void nvshmem_TYPENAME_p(TYPE *dest, TYPE value, int pe)
```

- `dest` [OUT]: Symmetric address of the destination data object.
- `value` [IN]: The value to be transferred to `dest`.
- `pe` [IN]: The number of the remote PE.



See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#nvshmem-p

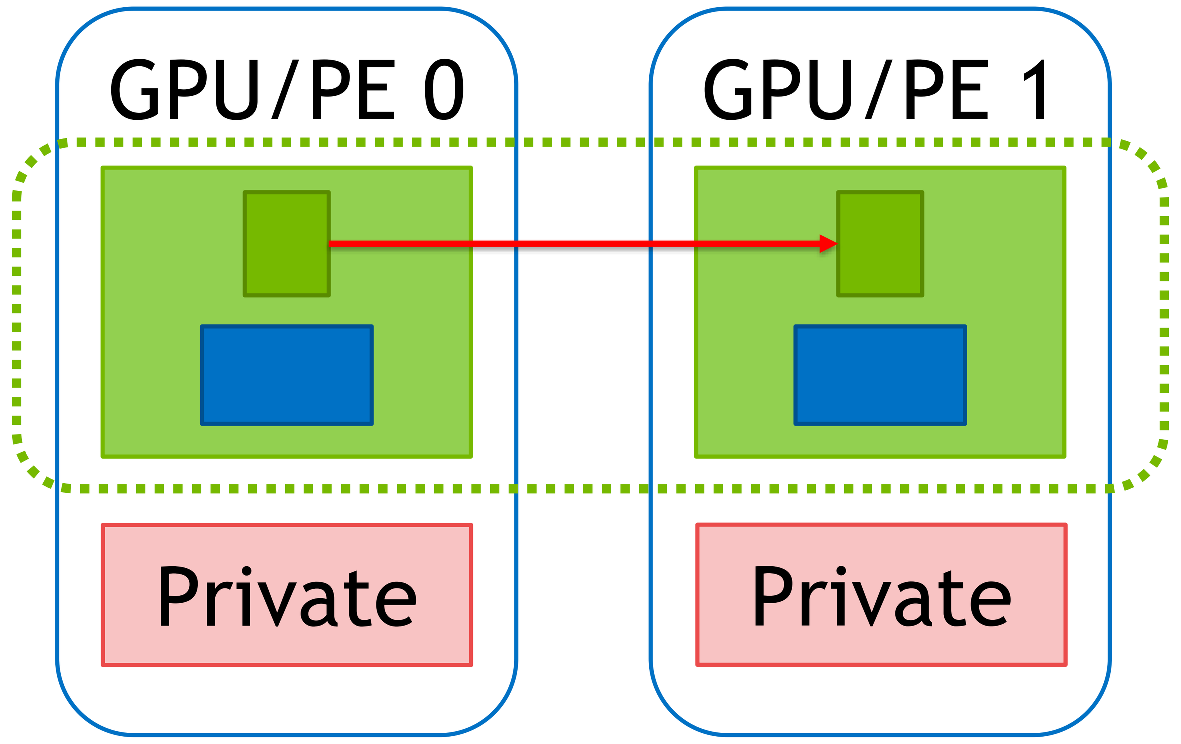
`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff`
(see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)

NVSHMEM API

nonblocking block cooperative put

```
__device__ void nvshmemx_TYPENAME_put_nbi_block(TYPE *dest, const TYPE *source, size_t nelems, int pe)
```

- `dest` [OUT]: Symmetric address of the destination data object.
- `source` [IN]: Symmetric address of the object containing the data to be copied.
- `nelems` [IN]: Number of elements in the `dest` and `source` arrays.
- `pe` [IN]: The number of the remote PE.



Cooperative call: Needs to be called by all threads in a block. thread and warp are also available.

x in `nvshmem`**`x`** marks API as extension of the OpenSHMEM APIs.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html?highlight=nvshmemx_typename_put_nbi_block#nvshmem-put-nbi

`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff`
(see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)

NVSHMEM API

ordering and completion

```
__device__ void nvshmem_quiet(void)
```

Ensures completion of all operations on symmetric data objects issued by the calling PE.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/ordering.html#nvshmem-quiet

NVSHMEM API

signal operation

```
__device__ inline void nvshmemx_signal_op(uint64_t *sig_addr, uint64_t signal, int sig_op, int pe)
```

- `sig_addr` [OUT]: Symmetric address of the signal word to be updated.
- `signal` [IN]: The value used to update `sig_addr`.
- `sig_op` [IN]: Operation used to update `sig_addr` with `signal`. (`NVSHMEM_SIGNAL_SET` or `NVSHMEM_SIGNAL_ADD`)
- `pe` [IN]: The number of the remote PE.

x in `nvshmem`**`x`** marks API as extension of the OpenSHMEM APIs.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/signal.html#nvshmemx-signal-op

NVSHMEM API

atomic operation

```
__device__ void nvshmem_TYPENAME_atomic_inc(TYPE *dest, int pe)
```

- `dest` [OUT]: Symmetric address of the signal word to be updated.
- `pe` [IN]: The number of the remote PE.

These routines perform an atomic increment operation on the `dest` data object on PE.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/amo.html#nvshmem-atomic-inc

`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff`
(see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)

NVSHMEM API

wait operations

```
__device__ void nvshmem_TYPENAME_wait_until_all(TYPE *ivars, size_t nelems, const int *status,
                                                                int cmp, TYPE cmp_value)

__device__ void nvshmem_TYPENAME_wait_until(TYPE *ivar, int cmp, TYPE cmp_value)
```

- `ivars`|`ivar` [IN]: Symmetric address of an array of remotely accessible data objects.|Symmetric address of a remotely accessible data object.
- `nelems` [IN]: The number of elements in the `ivars` array.
- `status` [IN]: Local address of an optional mask array of length `nelems` that indicates which elements in `ivars` are excluded from the wait set. Set to `NULL` when not used.
- `cmp` [IN]: A comparison operator (`NVSHMEM_CMP_EQ`, `NVSHMEM_CMP_NE`, `NVSHMEM_CMP_GT`, `NVSHMEM_CMP_GE`, `NVSHMEM_CMP_LT`, `NVSHMEM_CMP_LE`) that compares elements of `ivars`|`ivar` with `cmp_value`.
- `cmp_value` [IN]: The value to be compared with the objects pointed to by `ivars`.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/sync.html#nvshmem-wait-until-all and https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/sync.html#nvshmem-wait-until

`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff` (see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)

NVSHMEM PERFTESTS

shmem_p_bw and shmem_put_bw on JUWELS Booster – NVIDIA A100 40 GB

shmem_p_bw|shmem_put_bw from NVSHMEM perftest
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

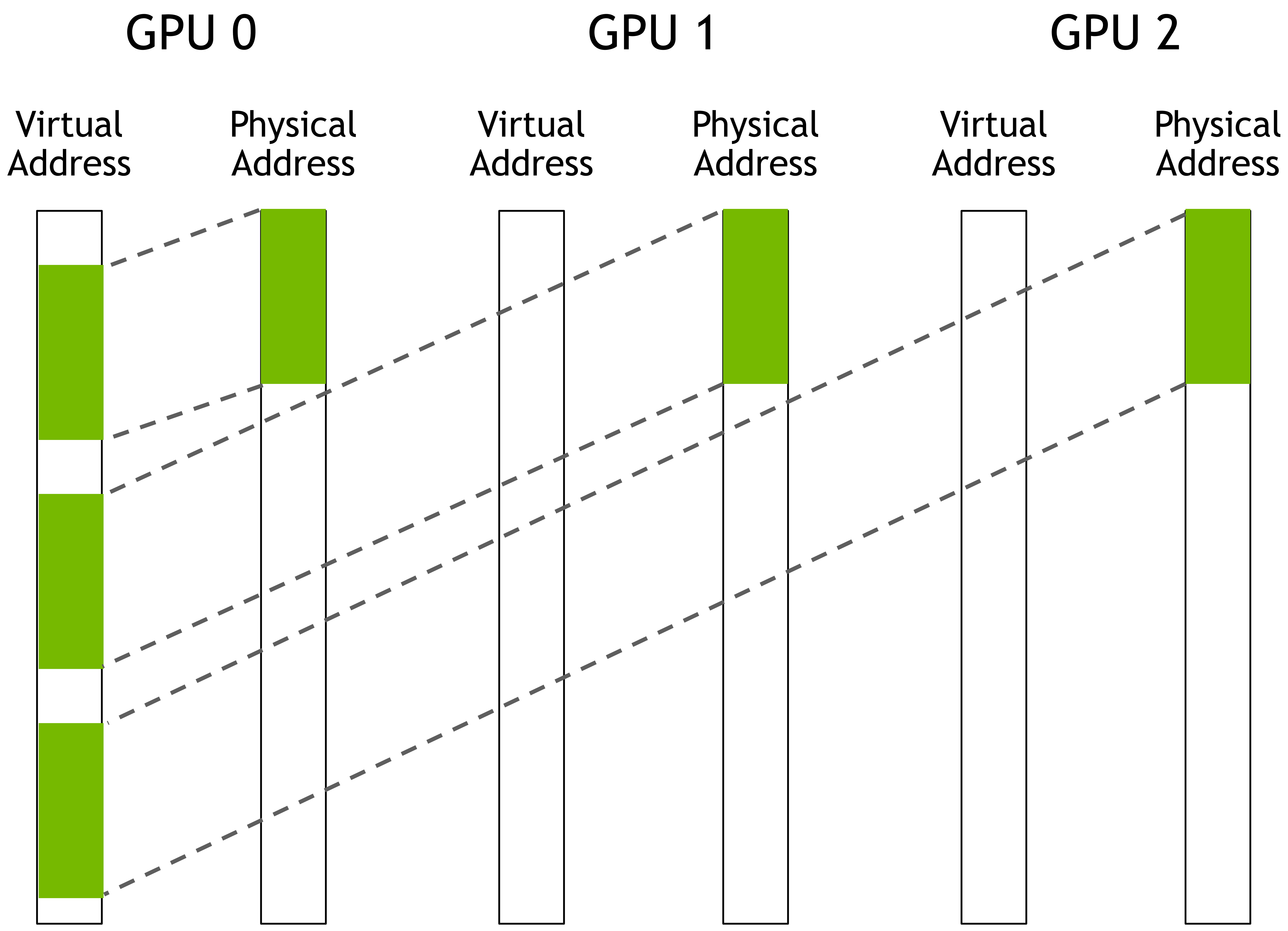
NVSHMEM PERFTESTS

shmem_p_bw and shmem_put_bw on JUWELS Booster – NVIDIA A100 40 GB

shmem_p_bw|shmem_put_bw from NVSHMEM perftest
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

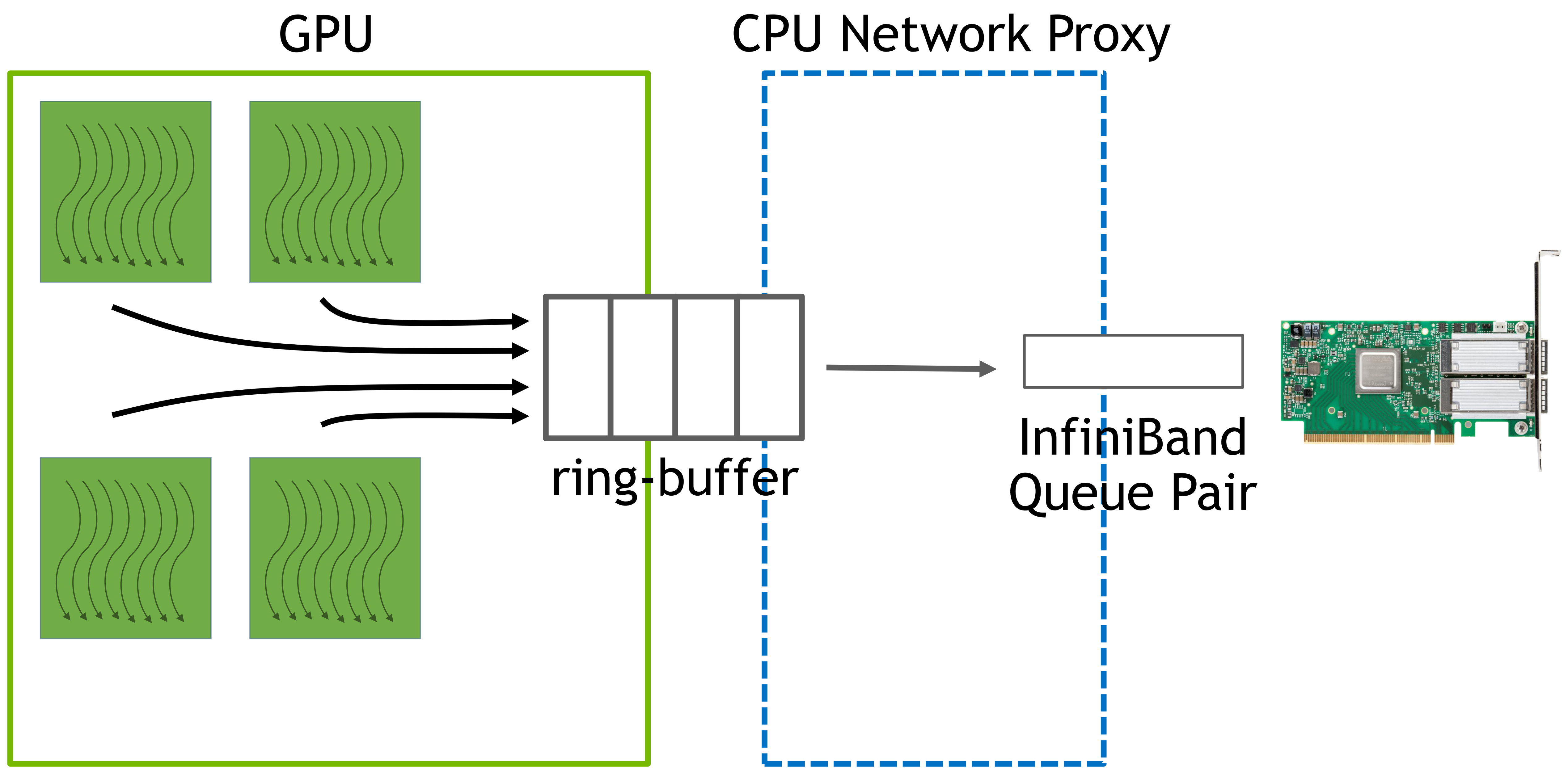
OPTIMIZED INTRA-NODE COMMUNICATION

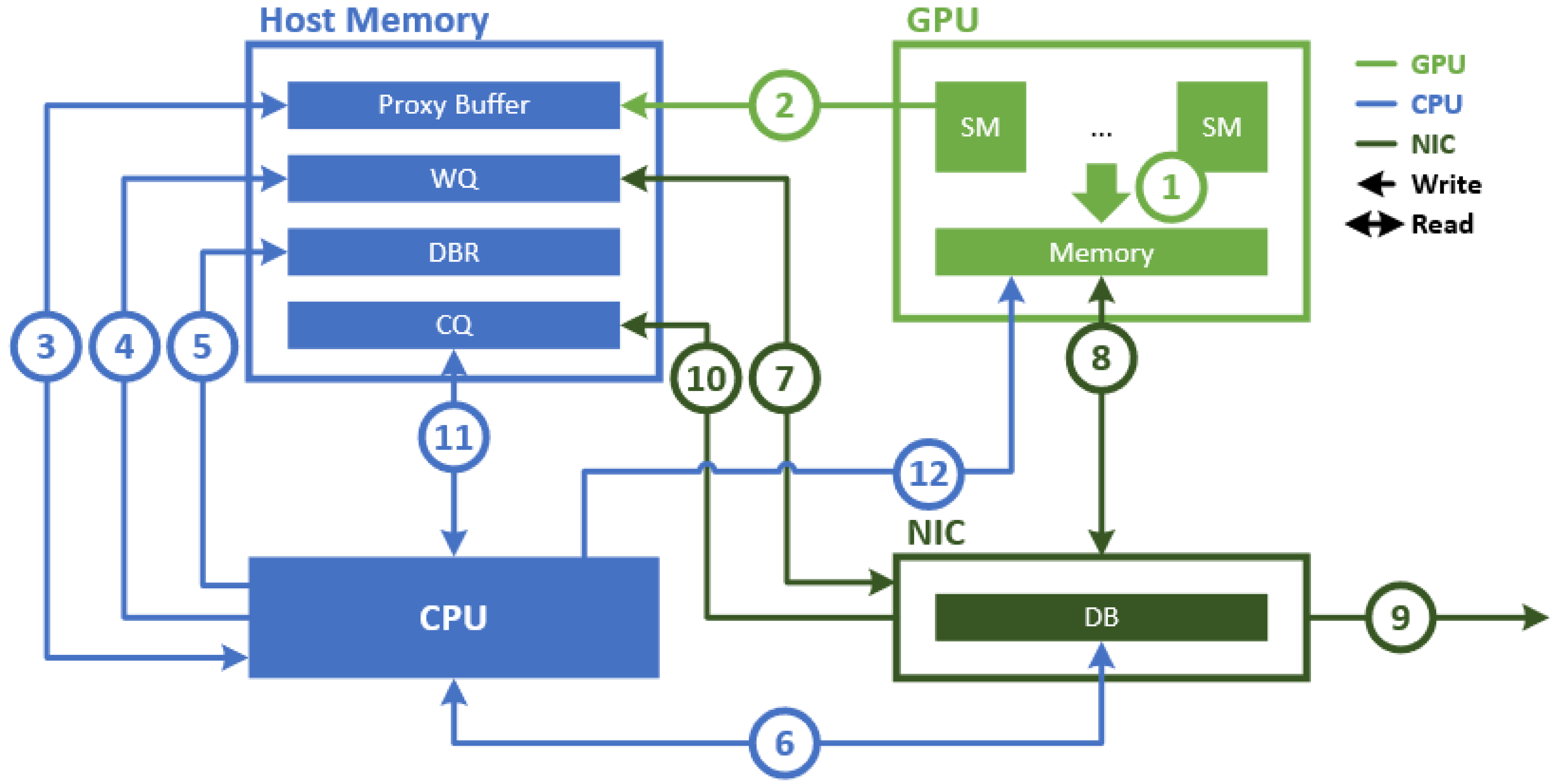
- Supported on NVLink and PCI-E
- Use CUDA IPC or `cuMem*` API to map symmetric memory of intra-node PEs into virtual address space
- `nvshmem_[put|get]` on device -> load/store
- `nvshmem_[put|get]_on_stream` -> `cudaMemcpyAsync`



OPTIMIZED INTER-NODE COMMUNICATION

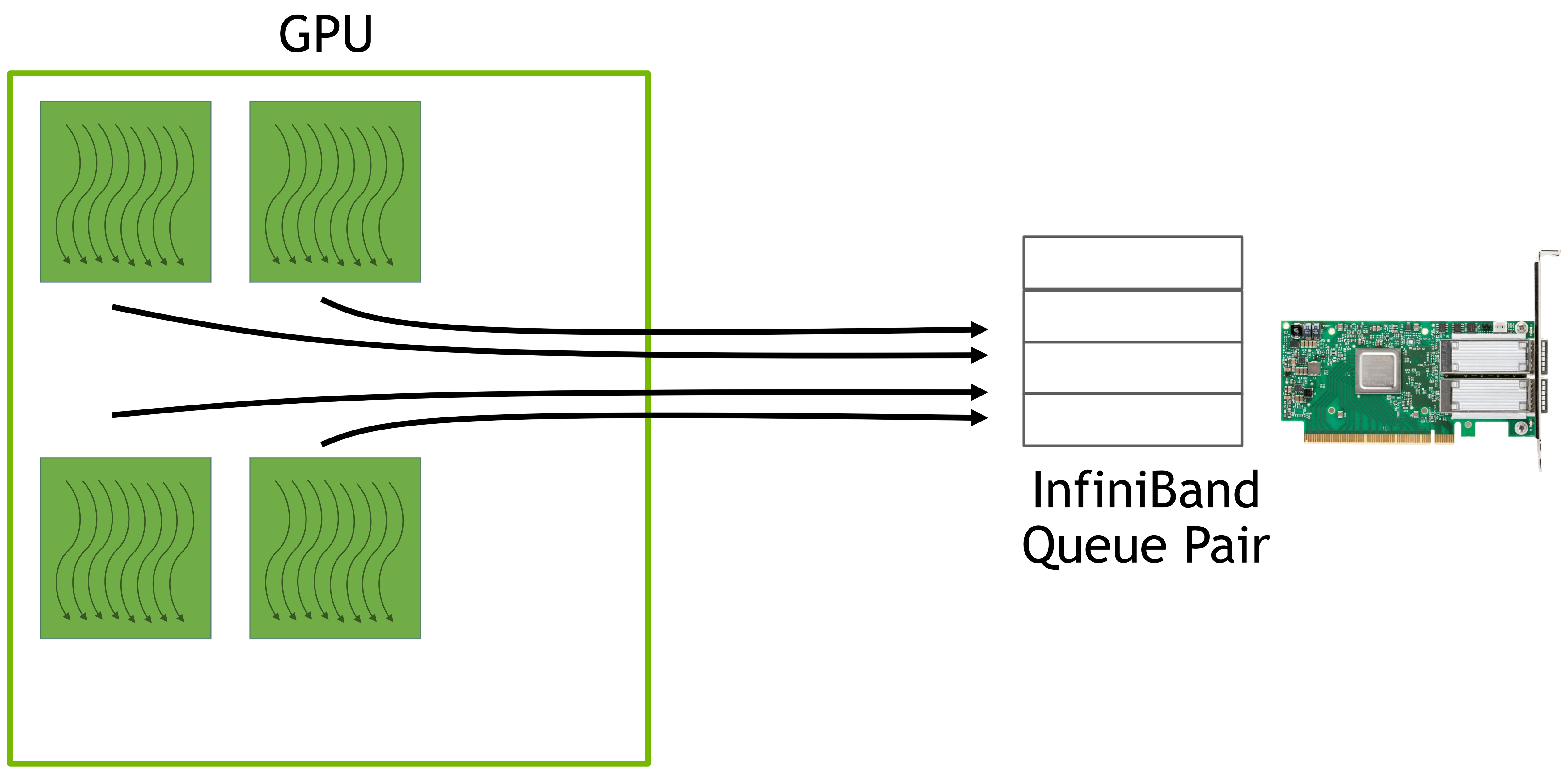
- NVSHMEM supports inter-node communication over InfiniBand, RoCE, and UCX (experimental)
- Using GPUDirect RDMA (data plane)
- Reverse offloads network transfers from GPU to the CPU (control plane)
- Ring buffer implementation avoids memory fences when interacting with CPU network proxy

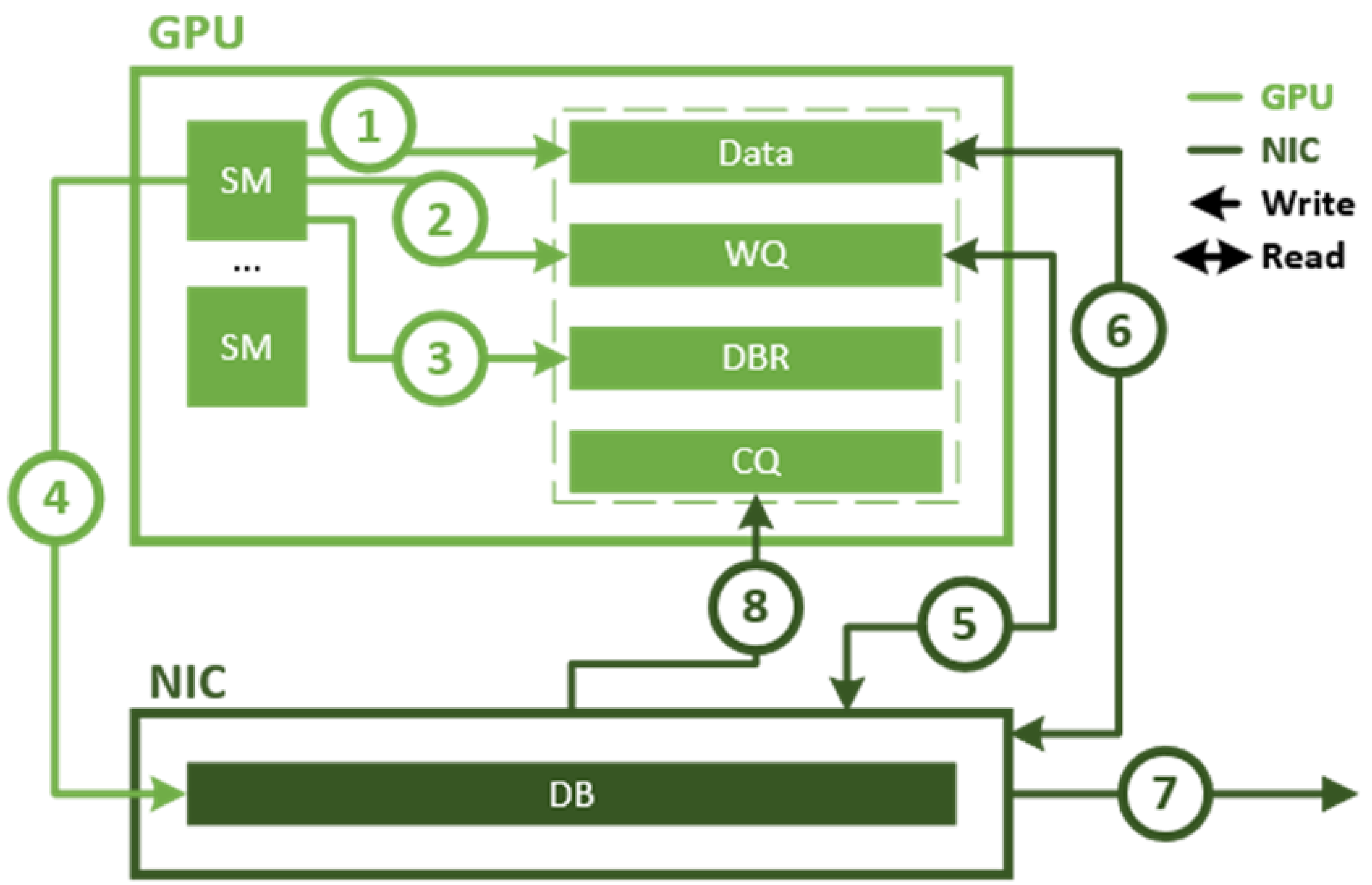




OPTIMIZED INTER-NODE COMMUNICATION IMPROVED

- **IB GPUDirect Async** (IBGDA) over InfiniBand
- Using GPUDirect RDMA (data plane)
- GPU directly initiates network transfers involving the CPU only for the setup of control data structures





NVSHMEM PERFTESTS WITH IBGDA

shmem_p_bw and shmem_put_bw on JUWELS Booster – NVIDIA A100 40 GB

shmem_p_bw|shmem_put_bw from NVSHMEM perftest
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

NVSHMEM VERSION WITH NVL, CPU PROXY AND IBGDA

NVSHMEM 2.8.0 prerelease – JUWELS Booster – NVIDIA A100 40 GB – Jacobi on 17408x17408

Source: https://github.com/NVIDIA/multi-gpu-programming-models
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

QUDA STRONG SCALING ON SELENE

Lattice Quantum ChromoDynamics

local lattice: 16^4

up to 1.6x Speedup (512 GPUs half precision)

https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s21673/

NVIDIA

SUMMARY AND MORE INFORMATION

- CUDA Graphs help minimize CPU-side launch overhead and Device-side execution overhead
- Device-initiate communication enables:
  - fine grained communication and computation overlap with sometimes less coding effort
  - kernel fusion not possible with host initiate communication models like MPI and NCCL
- With **IB GPUD**irect **A**sync (IBGDA) NVSHMEM can achieve peak Network message rates
- Without IBGDA for good intranode device-initiated communication performance it is necessary to aggregate larger messages (`nvshmemx_TYPENAME_put_nbi_block`)

- CUDA Graphs documentation: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs
- NVSHMEM: CUDA-Integrated Communication for NVIDIA GPUs (a Magnum IO session): https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41044/
- Overcoming Latency Barriers: Strong Scaling HPC Applications with NVSHMEM: https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s21673/
- https://developer.nvidia.com/blog/scaling-scientific-computing-with-nvshmem/
- https://developer.nvidia.com/blog/accelerating-nvshmem-2-0-team-based-collectives-using-nccl/