NVSHMEM: A PARTITIONED GLOBAL ADDRESS SPACE LIBRARY FOR NVIDIA GPU CLUSTERS

Sreeram Potluri, Anshuman Goswami - NVIDIA 3/28/2018

NVIDIA

AGENDA

GPU Programming Models

Overview of NVSHMEM

Porting to NVSHMEM

Performance Evaluation

Conclusion and Future Work

GPU CLUSTER PROGRAMMING

Offload model
- Compute on GPU
- Communication from CPU

Synchronization at boundaries

Performance overheads
- Offload latencies
- Synchronization overheads

Limits scaling

Increases code complexity

More CPU means more power

```
void 2dstencil (u, v, …)
{
  for (timestep = 0; …) {
    interior_compute_kernel <<<…>>> (…)
    pack_kernel <<<…>>> (…)
    cudaStreamSynchronize(…)
    MPI_Irecv(…)
    MPI_Isend(…)
    MPI_Waitall(…)
    unpack_kernel <<<…>>> (…)
    boundary_compute_kernel <<<…>>> (…)
    …
  }
}
```

GPU CLUSTER PROGRAMMING

Offload model
- Compute on GPU
- Communication from CPU

Synchronization at boundaries

Performance overheads
- Offload latencies
- Synchronization overheads

Limits scaling

Increases code complexity

More CPU means more power

```
void 2dstencil (u, v, …)
{
  for (timestep = 0; …) {
    interior_compute_kernel <<<…>>> (…)
    pack_kernel <<<…>>> (…)
    MPI_Irecv_on_stream(…,stream)
    MPI_Isend_on_stream(…,stream)
    MPI_Wait_on_stream(…,stream)
    unpack_kernel <<<…>>> (…)
    boundary_compute_kernel <<<…>>> (…)
    …
  }
}
```

MPI-async and NCCL help improve this, but!

GPU-INITIATED COMMUNICATION

Removing reliance on CPU for communication avoids overheads

Parallelism for implicit compute – communication overlap

Continuous fine-grained accesses smooths traffic over the network

Direct accesses to remote memory simplifies programming

Improving performance while making it easier to program

COMMUNICATION FROM CUDA KERNELS

Long running CUDA kernels

Communication within parallel compute

```
void 2dstencil (u, v, …)
{
    stencil_kernel <<<…>>> (…)
}
```

```
__global__ void 2dstencil (u, v, sync, …)
{
   for(timestep = 0; …) {

      u[i] = (u[i] + (v[i+1] + v[i-1] . . .

      //data exchange
      if (i+1 > nx) {
          shmem_float_p (v[1], v[i+1], rightpe);
      }
      if (i-1 < 1) {
          shmem_float_p (v[nx], v[i-1], leftpe);
      }

      //synchronization
      if (i < 2) {
          shmem_fence();
          shmem_int_p (sync + i, 1, peers[i]);
          shmem_wait_until (sync + i, EQ, 1);
      }
      //intra-kernel sync
      …
   }
}
```

AGENDA

GPU Programming Models

Overview of NVSHMEM

Porting to NVSHMEM

Performance Evaluation

Conclusion and Future Work

WHAT IS OPENSHMEM ?

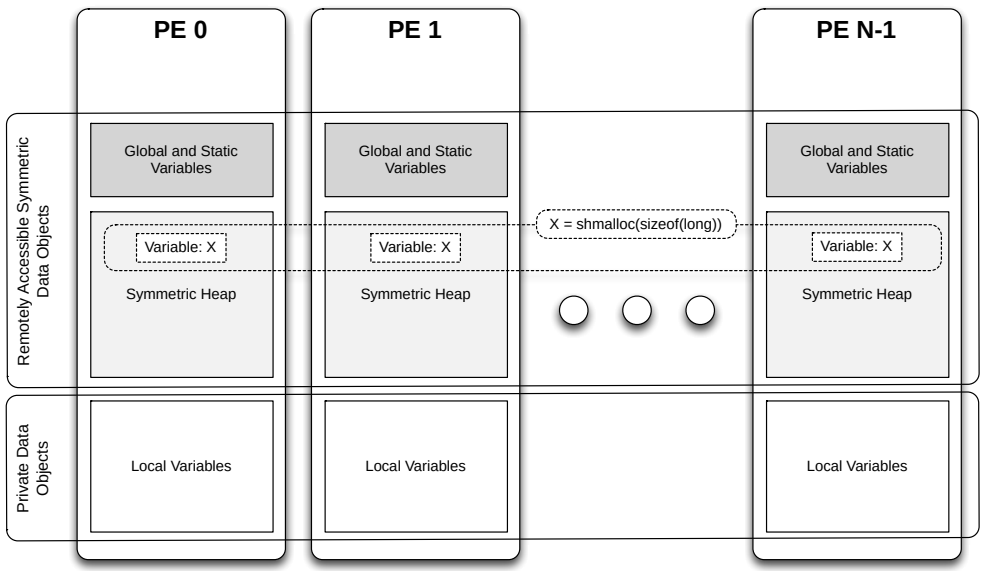
OpenSHMEM is a PGAS library interface specification

Distributed shared memory - defined locality of segments to application instances

OpenSHMEM constructs:

Programming Elements (PEs) – Execution Context

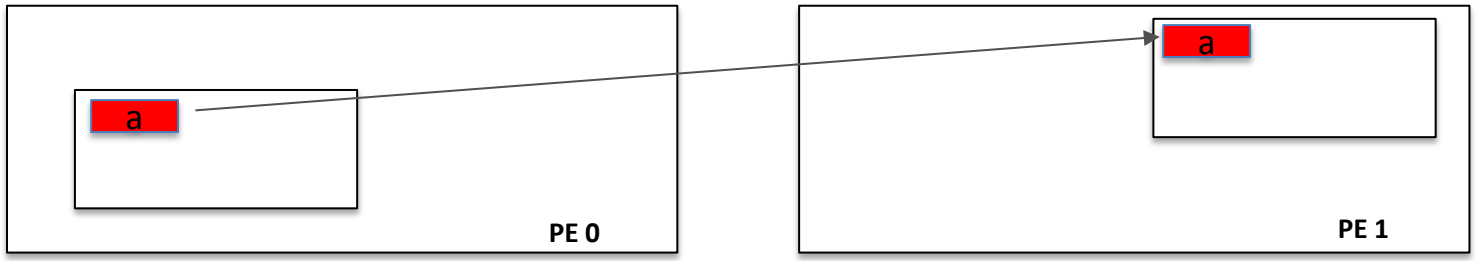
Symmetric objects – Global memory constructs which have same address offsets across all PEs



Symmetry allows

- Ease of use
- Fast address translation

QUICK EXAMPLE



Virtual Address Space

```
int *a, *a_remote;
Int value = 1;

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

if (shmem_my_pe() == 0) {
     //accessing remote memory using PutAPI
     shmem_int_p (a/*remote addr*/, value, 1/*remote PE*/);

    //can do the same using a ST
    a_remote = shmem_ptr(a, 1);
    *a_remote = value;
}
```

OPENSHMEM FEATURES

Point-to-point and group data movement operations

- Remote Memory Put and Get
- Collective (broadcast, reductions, etc)

Remote Memory Atomic operations

Synchronization operations (barrier, sync)

Ordering operations (fence, quiet)

NVSHMEM

Experimental implementation of OpenSHMEM for NVIDIA GPUs
Symmetric heap on GPU memory
Adds CUDA-specific extensions for performance

HOST ONLY	HOST/GPU	GPU
Library setup, exit and query Memory management Collective CUDA kernel launch CUDA stream ordered operations	Data movement operations Atomic memory operations Synchronization operations Memory ordering	CTA-wide operations

COLLECTIVE CUDA KERNEL LAUNCH

CUDA threads across GPUs can use NVSHMEM to synchronize or collectively move data

These kernels should be concurrently launched and be resident across all GPUs

OpenSHMEM extension built on top of CUDA cooperative launch

shmemx_collective_launch (…) //takes same arguments as a CUDA kernel launch

Can use regular CUDA launch if not using any synchronization or collective APIs

CTA-WIDE OPERATIONS

Parallelism on the GPU can be used to optimize OpenSHMEM operations

Extensions allow threads within a CTA to participate in a single OpenSHMEM call

Collective operations
- translate to a multiple point-to-point interactions between PEs
- threads can be used to parallelize this
- Eg: shmemx_barrier_all_cta(…), shmemx_broadcast_cta(…),
- semantic is still as if a single collective operation is executed

Bulk point-to-point transfers benefit from concurrency and with coalescing
- Eg: shmemx_putmem_cta(…)

CUDA STREAM ORDERED EXTENSIONS

Not all communication can be moved into a CUDA kernels

Not all compute can be fused in to a single kernel

Synchronization or communication at kernel boundary is still required

Extension to offload CPU-initiated SHMEM operations onto a CUDA stream

Eg:

```
kernel1<<<…,stream>>>(…)
shmemx_barrier_all_on_stream(stream) //can be a collective or p2p operation
kernel2<<<…,stream>>>(…)
```

NVSHMEM STATUS

Working towards an early-access for external customers

Initial version will have support for P2P-connected GPUs (single-node)

- atomics not supported over PCIe
- full feature set on Pascal or newer GPUs

Non-P2P and Multi-node support in future

AGENDA

GPU Programming Models

Overview of NVSHMEM

Porting to NVSHMEM

Performance Evaluation

Conclusion and Future Work

PORTING TO USE NVSHMEM FROM GPU

Step I : Only communication from inside the kernel (on Kepler or newer GPUs)

Step II : Both communication and synchronization APIs from inside the kernel (on Pascal or newer Tesla GPUs)

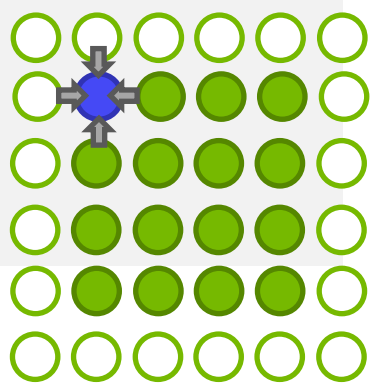
Using Jacobi Solver, we will walk through I and II and compare with MPI version

EXAMPLE: JACOBI SOLVER

While not converged

Do Jacobi step:

```
for( int iy = 1; iy < ny-1; iy++ )
for( int ix = 1; ix < ny-1; ix++ )
    a_new[iy*nx+ix] = -0.25 *
                      -( a[ iy   *nx+(ix+1)] + a[ iy   *nx+ix-1]
                       + a[(iy-1)*nx+ ix   ] + a[(iy+1)*nx+ix  ] );
```



Apply periodic boundary conditions

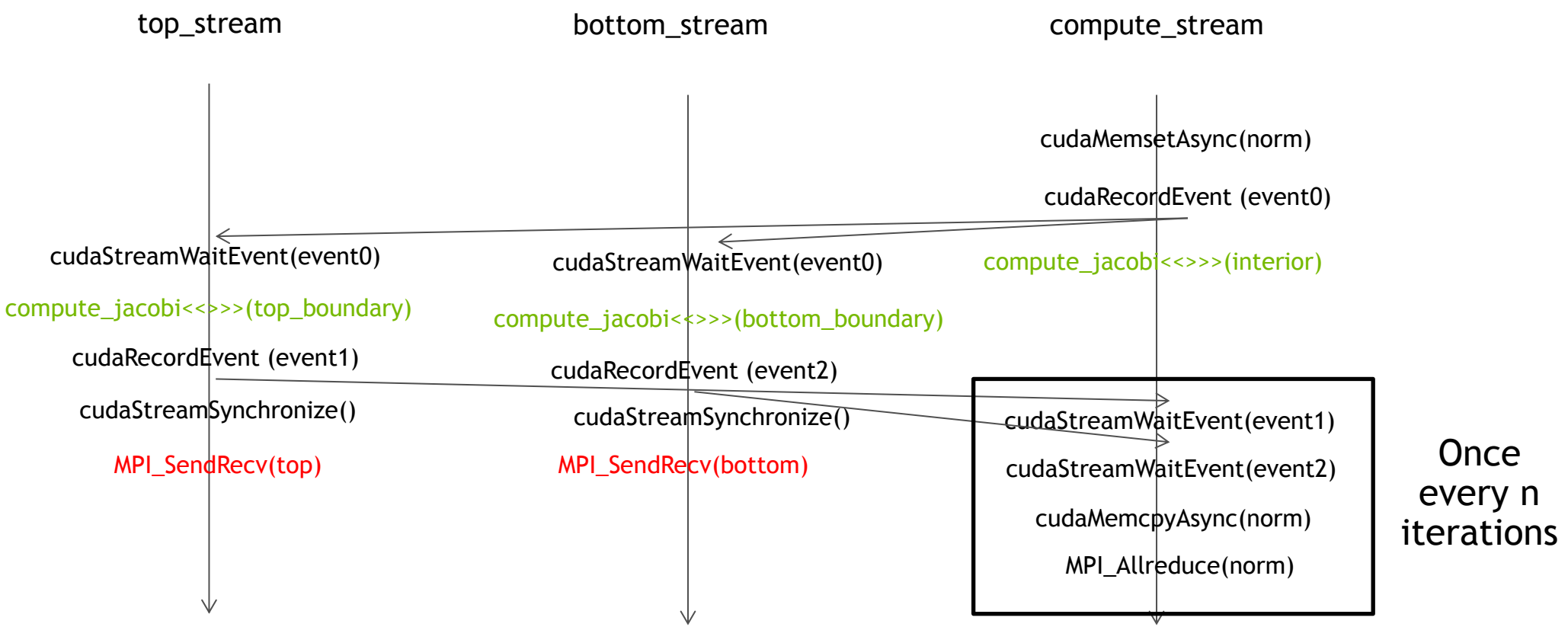
Swap `a_new` and `a`

Next iteration

COMPUTE KERNEL

```
__global__ void jacobi_kernel( ... ) {
    const int ix = bIdx.x*bDim.x+tIdx.x;
    const int iy = bIdx.y*bDim.y+tIdx.y + iy_start;
    real local_l2_norm = 0.0;
    for (int iy = bIdx.y*bDim.y+tIdx.y+iy_start; iy <= iy_end; iy += bDim.y * gDim.y) {
    for (int ix = bIdx.x*bDim.x+tIdx.x+1; ix < (nx-1); ix += bDim.x * gDim.x) {
        const real new_val = 0.25 * ( a[ iy * nx + ix + 1 ] + a[ iy * nx + ix - 1 ]
                                    + a[ (iy+1) * nx + ix ] + a[ (iy-1) * nx + ix ] );
        a_new[ iy * nx + ix ] = new_val;
        real residue = new_val - a[ iy * nx + ix ];
        atomicAdd( l2_norm, local_l2_norm ); }}
}
```

HOST CODE - MPI



HOST CODE - MPI

```
while (iter < iter_max ) {while ( l2_norm > tol && iter < iter_max )
{
        //reset norm
        CUDA_RT_CALL( cudaMemsetAsync(l2_norm_d, 0 , sizeof(real), compute_stream ) );
        CUDA_RT_CALL( cudaEventRecord( reset_l2norm_done, compute_stream ) );

        //compute boundary
        CUDA_RT_CALL( cudaStreamWaitEvent( push_top_stream, reset_l2norm_done, 0 ) );
        launch_jacobi_kernel( a_new, a, l2_norm_d, iy_start, (iy_start+1), nx, push_top_stream );
        CUDA_RT_CALL( cudaEventRecord( push_top_done, push_top_stream ) )
        CUDA_RT_CALL( cudaStreamWaitEvent( push_bottom_stream, reset_l2norm_done, 0 ) );
        launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_end-1), iy_end, nx, push_bottom_stream );
        CUDA_RT_CALL( cudaEventRecord( push_bottom_done, push_bottom_stream ) );

        //compute interior
        launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_start+1), (iy_end-1), nx, compute_stream );

        //Apply periodic boundary conditions
        CUDA_RT_CALL( cudaStreamSynchronize( push_top_stream ) );
                    MPI_CALL( MPI_Sendrecv( a_new+iy_start*nx,   nx, MPI_REAL_TYPE, top   , 0, a_new+(iy_end*nx), nx, MPI_REAL_TYPE, bottom, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE ));
        CUDA_RT_CALL( cudaStreamSynchronize( push_bottom_stream ) );
        MPI_CALL( MPI_Sendrecv( a_new+(iy_end-1)*nx, nx, MPI_REAL_TYPE, bottom, 0, a_new,             nx, MPI_REAL_TYPE, top,    0, MPI_COMM_WORLD, MPI_STATUS_IGNORE ));

        //Periodic convergence check
        if ( (iter % nccheck) == 0 || (!csv && (iter % 100) == 0) ) {
            CUDA_RT_CALL( cudaStreamWaitEvent( compute_stream, push_top_done, 0 ) );
            CUDA_RT_CALL( cudaStreamWaitEvent( compute_stream, push_bottom_done, 0 ) );
            CUDA_RT_CALL( cudaMemcpyAsync( l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream ) );
            CUDA_RT_CALL( cudaStreamSynchronize( compute_stream ) );
            MPI_CALL( MPI_Allreduce( l2_norm_h, &l2_norm, 1, MPI_REAL_TYPE, MPI_SUM, MPI_COMM_WORLD ) );
            l2_norm = std::sqrt( l2_norm );
}
```

CUDA KERNEL - NVSHMEM FOR COMMS

```
__global__ void jacobi_kernel( ... ) {
    const int ix = bIdx.x*bDim.x+tIdx.x;
    const int iy = bIdx.y*bDim.y+tIdx.y + iy_start;
    real local_l2_norm = 0.0;
    for (int iy = bIdx.y*bDim.y+tIdx.y+iy_start; iy <= iy_end; iy += bDim.y * gDim.y) {
    for (int ix = bIdx.x*bDim.x+tIdx.x+1; ix < (nx-1); ix += bDim.x * gDim.x) {
        const real new_val = 0.25 * ( a[ iy * nx + ix + 1 ] + a[ iy * nx + ix - 1 ]
                                    + a[ (iy+1) * nx + ix ] + a[ (iy-1) * nx + ix ] );
        a_new[ iy * nx + ix ] = new_val;
        if ( iy_start == iy )
             shmem_float_p(a_new + top_iy*nx + ix, new_val, top_pe);
        if ( iy_end == iy )
             shmem_float_p(a_new + bottom_iy*nx + ix, new_val, bottom_pe);
        real residue = new_val - a[ iy * nx + ix ];
        atomicAdd( l2_norm, local_l2_norm ); }}
}
```

HOST CODE - NVSHMEM FOR COMMS

```
a = (real *) shmem_malloc(nx*(chunk_size+2)*sizeof(real));
a_new = (real *) shmem_malloc(nx*(chunk_size+2)*sizeof(real));
…
while (iter < iter_max && l2_norm > tol ) {
    …
    jacobi_kernel<<<dim_grid,dim_block,0,compute_stream>>>(
        a_new, a, l2_norm_d, iy_start, iy_end, nx,
        top, iy_end_top, bottom, iy_start_bottom );
    shmemx_barrier_all_on_stream(compute_stream);

    //convergence check
    if ((iter % nccheck) == 0) {
       cudaMemcpyAsync( l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, 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 );
    }
}
…
```

CUDA KERNEL - NVSHMEM FOR COMMS + SYNC

```
__global__ void jacobi_uber_kernel( ... ) {
    grid_group g = this_grid();
    //comms only ...
     g.sync();
     if ( (iter % nccheck) == 0 ) {
       //reduction across shmem pes
       if (!tid_x && !tid_y) {
         shmem_barrier_all();
         shmem_float_sum_to_all (l2_norm + 1, l2_norm, 1, 0, 0, npes, NULL, NULL);
         l2_norm[1] = (float) __frsqrt_rn( (float)l2_norm[1] );
         l2_norm[0] = 0;}}
     g.sync();
}
```

HOST CODE - NVSHMEM FOR COMMS

```
a = (real *) shmem_malloc(nx*(chunk_size+2)*sizeof(real));
a_new = (real *) shmem_malloc(nx*(chunk_size+2)*sizeof(real));
…
void *args[] = {&a_new, &a, &l2_norm_d, &iy_start, &iy_end, &nx,
       &top, &iy_end_top, &bottom, &iy_start_bottom};
shmemx_collective_launch ( jacobi_kernel, dim_grid,
      dim_block,0,compute_stream);
…
```

AGENDA

GPU Programming Models

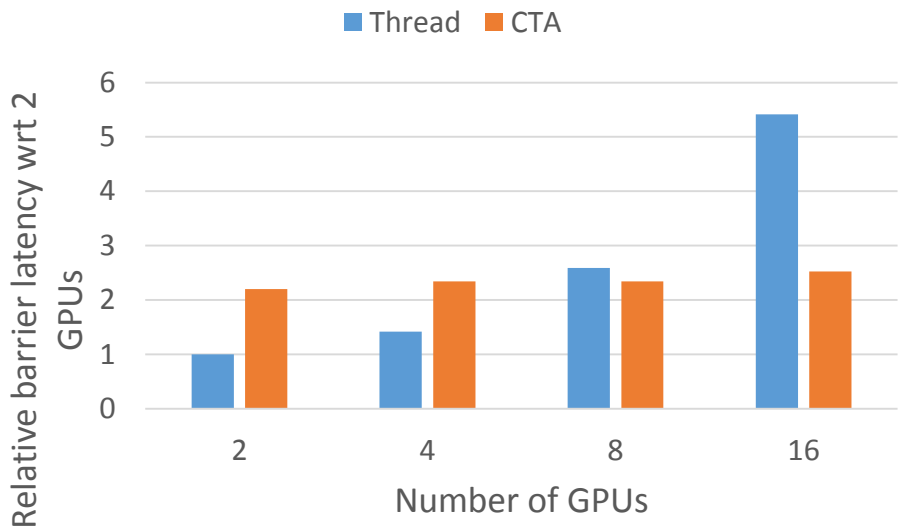
Overview of NVSHMEM

Porting to NVSHMEM

Performance Evaluation

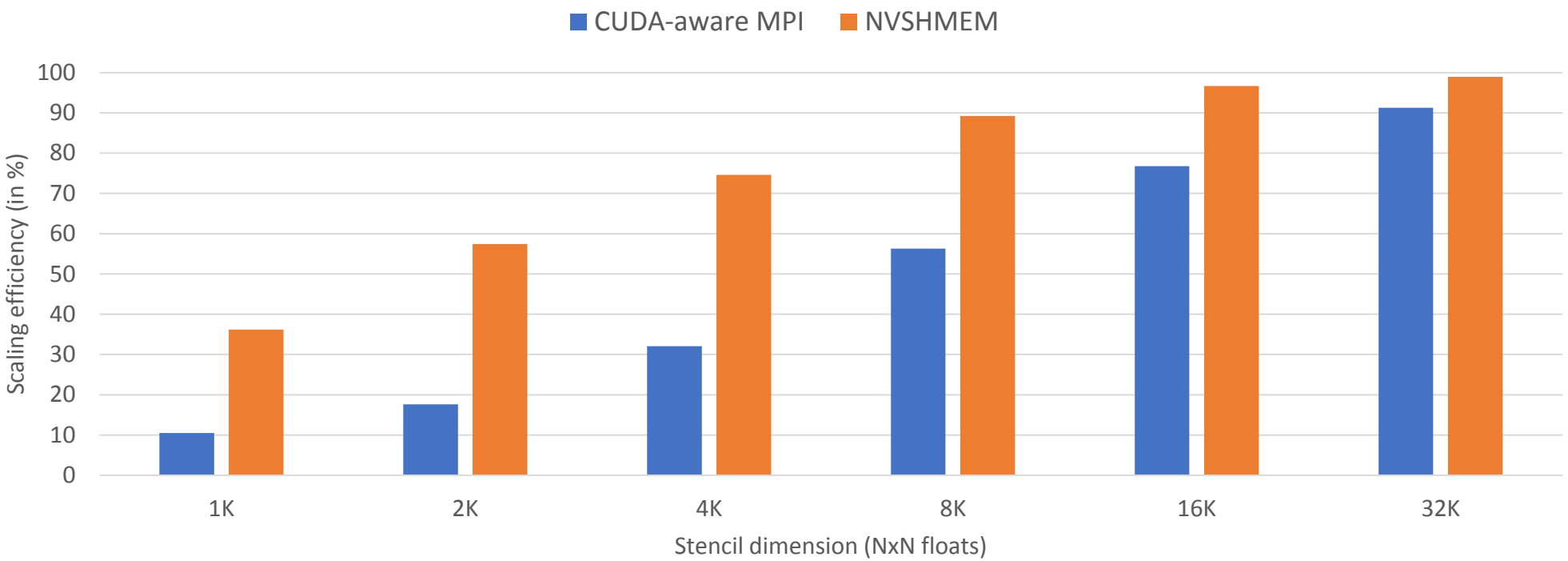
Conclusion and Future Work

CTA-WIDE BARRIER PERFORMANCE



Disclaimer: Results from a pre-production system
DGX-2:
- GPUs: 16 V100/32 GB
- Dual Socket Intel Xeon Platinum 8168 CPU 2.7 GHz, 24-cores

JACOBI SOLVER



8 V100 GPUs + NVLink (2x4)

MULTI-GPU TRANSPOSE

GPU 0

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

GPU 1

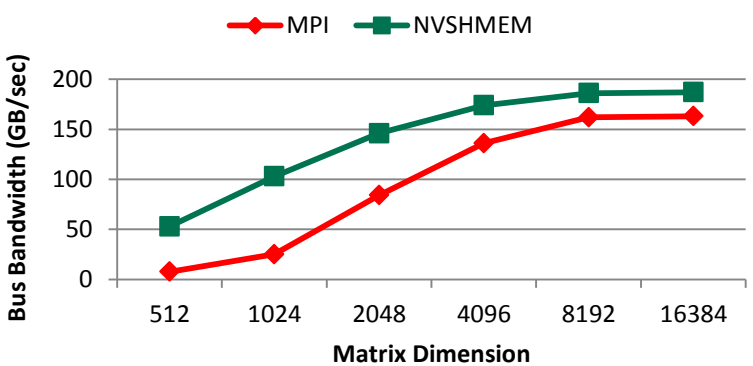
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

Bandwidth limited

MPI version carefully pipelines local transposes (packing and unpacking) and inter-process data movement

NVSHMEM moves data in-place that significantly reduces code complexity



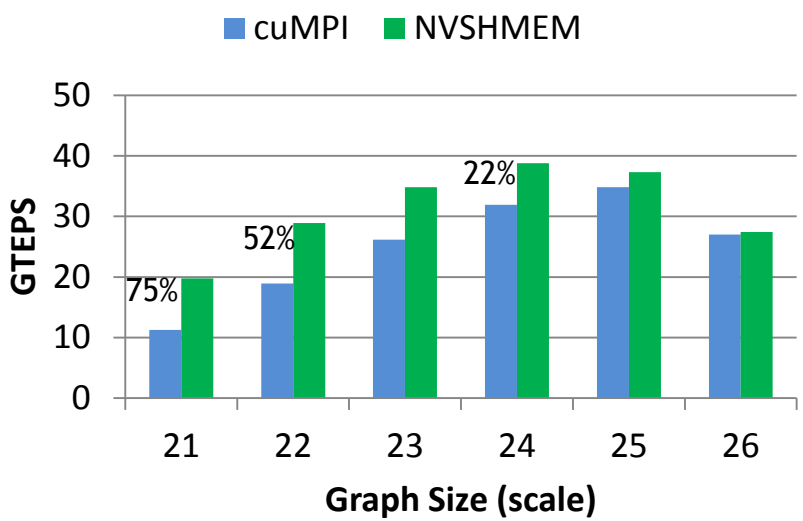
4 GP100s connected with NVLink

MULTI-GPU BREADTH FIRST SEARCH

Key subroutine in several graph algorithms, naturally leads to random access

MPI Version implementations: pack or use a bitmap to exchange frontier at end of each step

NVSHMEM version: directly updates the frontier map at target using atomics



8 P100 GPUs + NVLink (2x4)

SUMMARY

NVSHMEM aims to enable communication from inside CUDA kernels

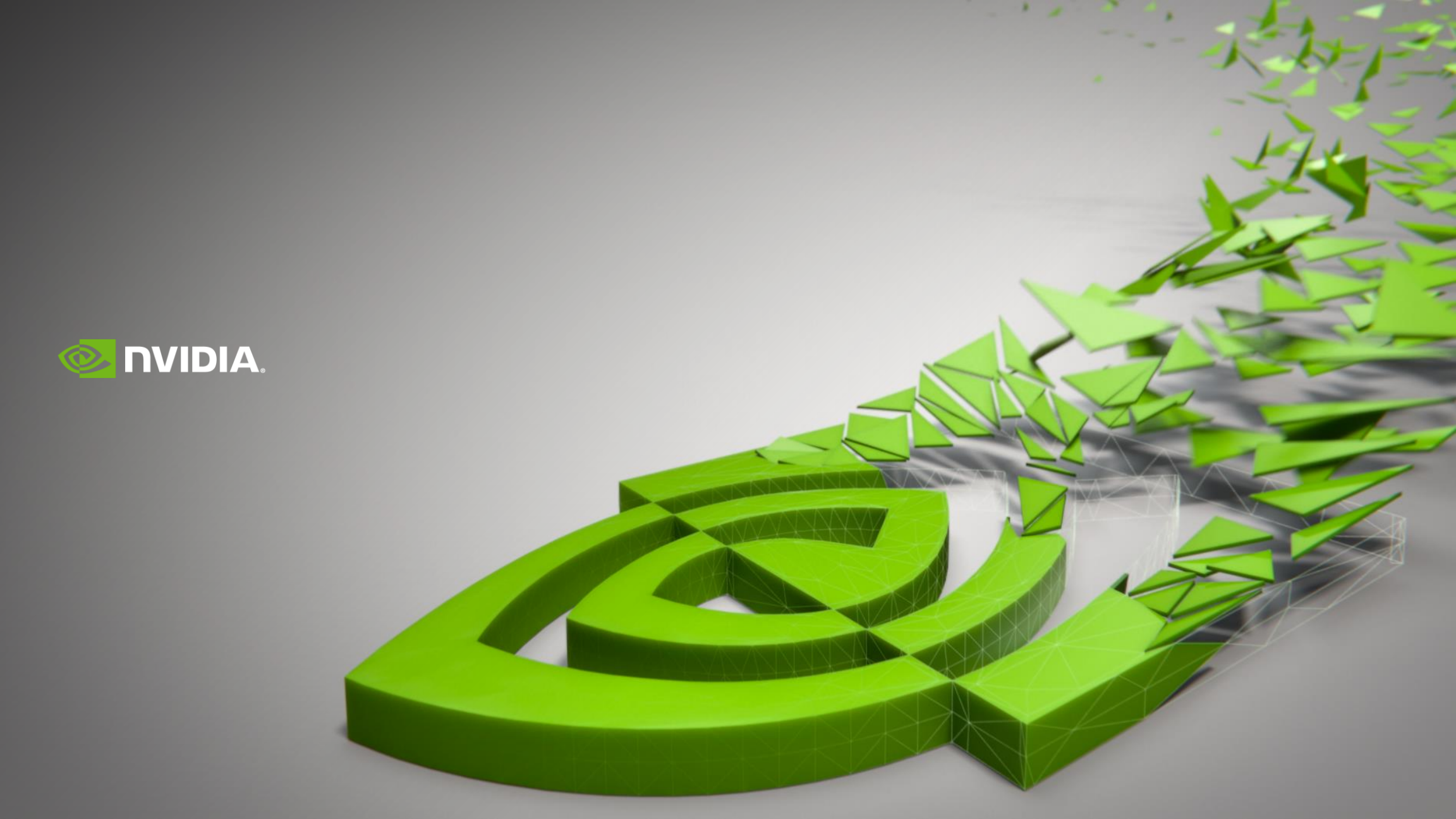
Evaluation with mini-apps shows promise

Still experimental but working towards an EA release

Initial version will have support for P2P-connected GPUs

Future work

- to support multi-node communication over IB
- to support single-node communication between GPUs not accessible by P2P


NVIDIA