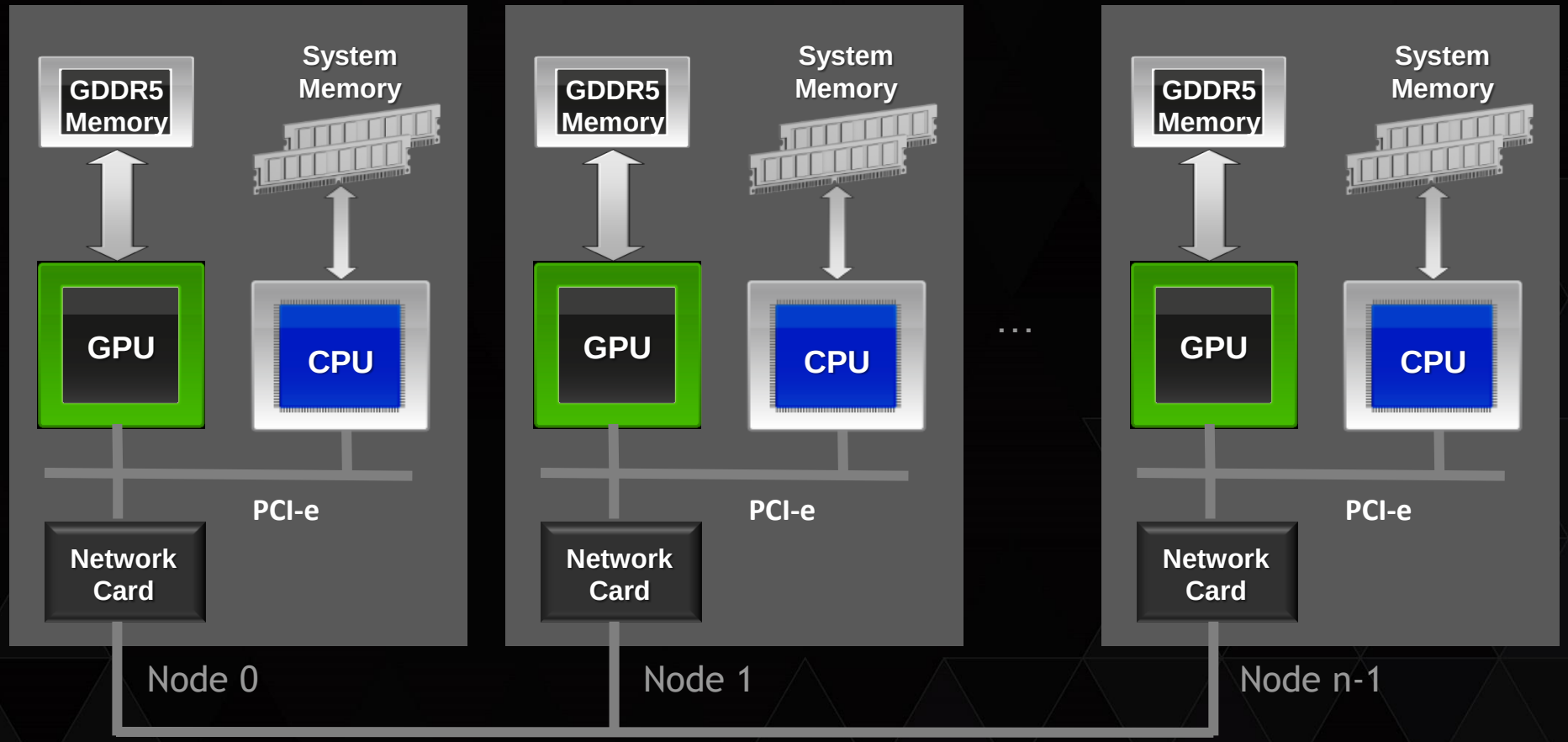
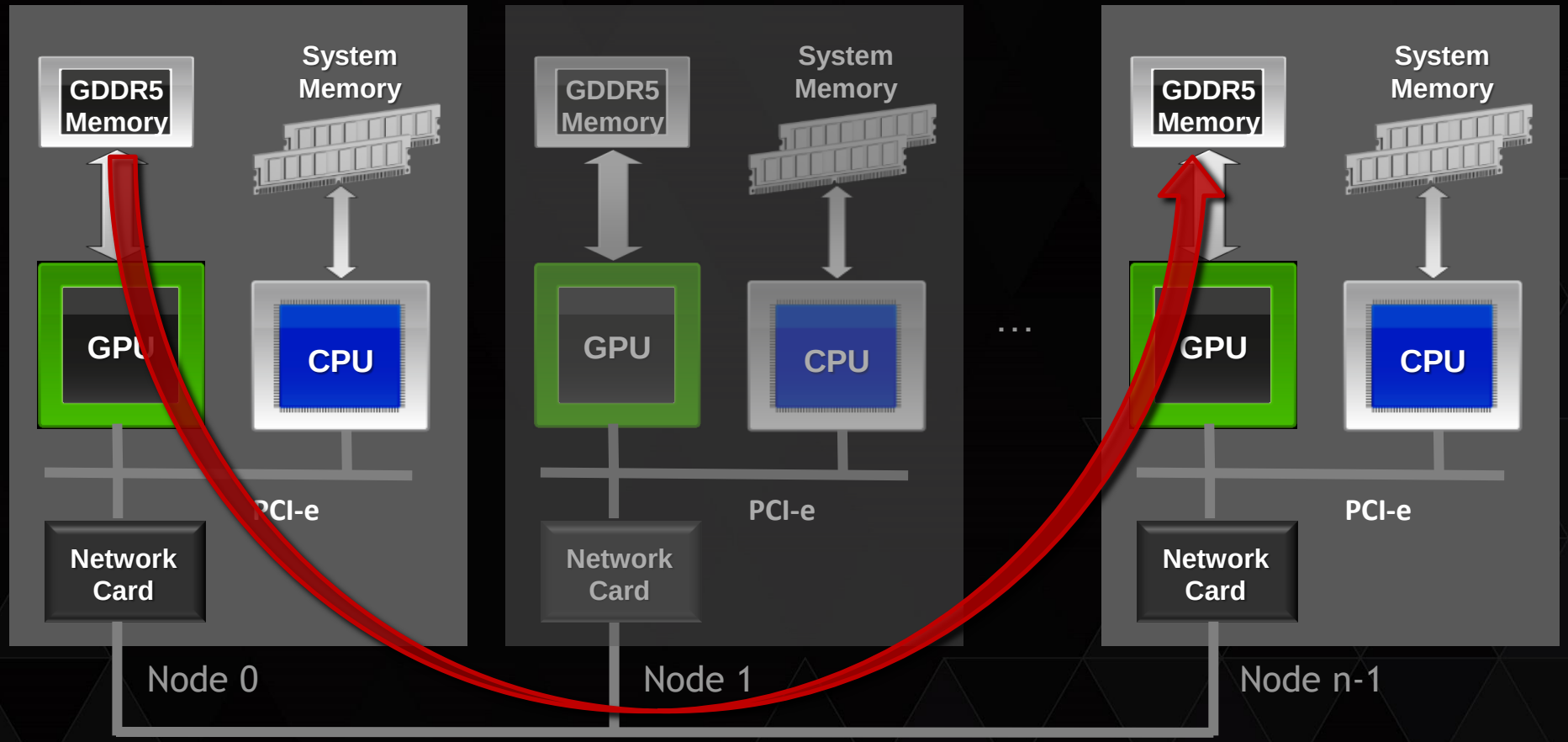
MULTI GPU PROGRAMMING WITH MPI

JIRI KRAUS, NVIDIA

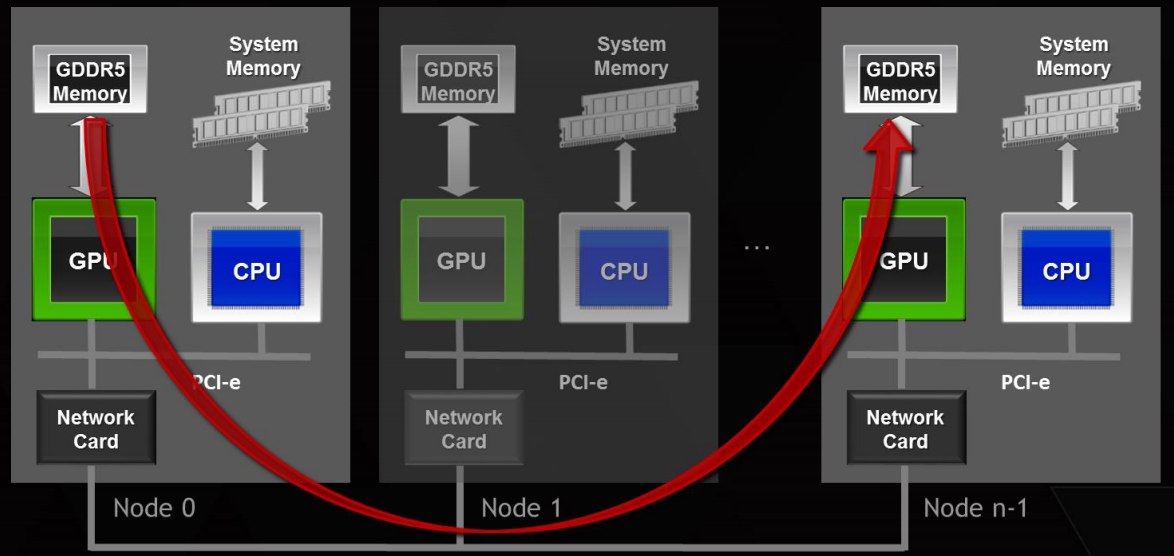
MPI+CUDA



MPI+CUDA



MPI+CUDA




```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,n-1,tag,MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

WHAT YOU WILL LEARN

- What MPI is
- How to use MPI for inter GPU communication with CUDA and OpenACC
- What CUDA-aware MPI is
- What Multi Process Service is and how to use it
- How to use NVIDIA Tools in an MPI environment
- How to hide MPI communication times

MESSAGE PASSING INTERFACE - MPI

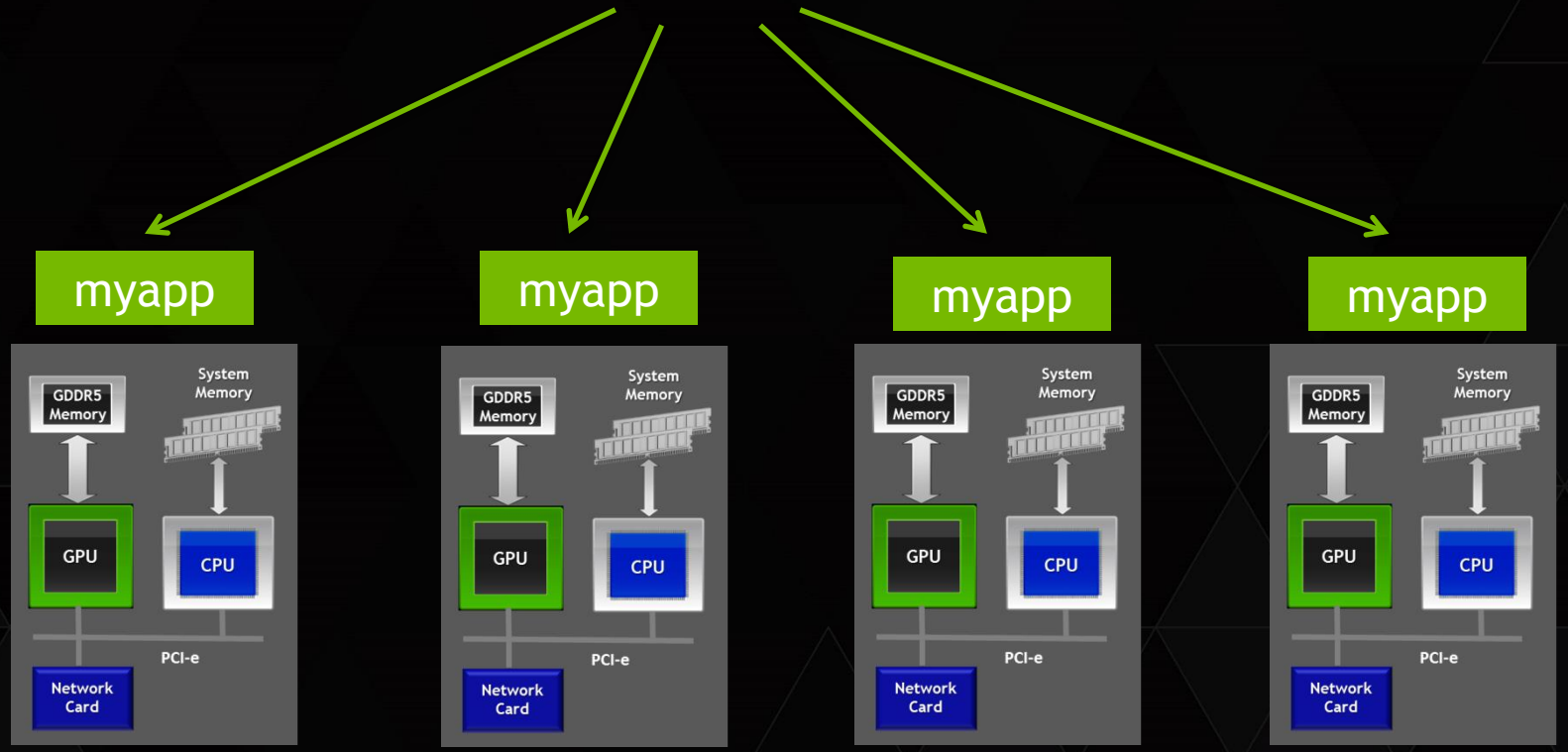
- Standard to exchange data between processes via messages
  - Defines API to exchanges messages
    - Pt. 2 Pt.: e.g. MPI_Send, MPI_Recv
    - Collectives, e.g. MPI_Reduce
- Multiple implementations (open source and commercial)
  - Binding for C/C++, Fortran, Python, …
  - E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, …

MPI - A MINIMAL PROGRAM

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank,size;
    /* Initialize the MPI library */
    MPI_Init(&argc,&argv);
    /* Determine the calling process rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
```

MPI - COMPILING AND LAUNCHING

```
$ mpicc -o myapp myapp.c
$ mpirun -np 4 ./myapp <args>
```



A SIMPLE EXAMPLE

EXAMPLE: JACOBI SOLVER

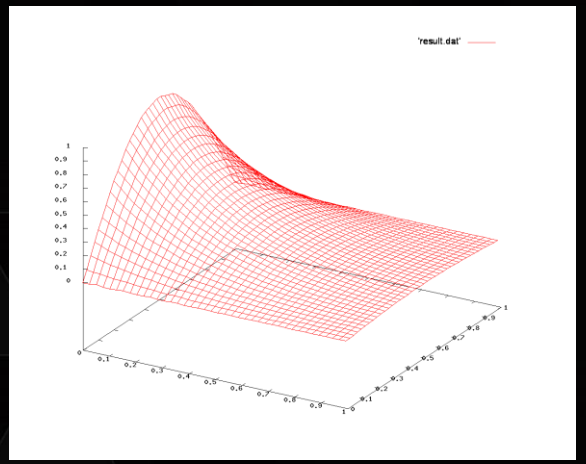
- Solves the 2D-Laplace equation on a rectangle

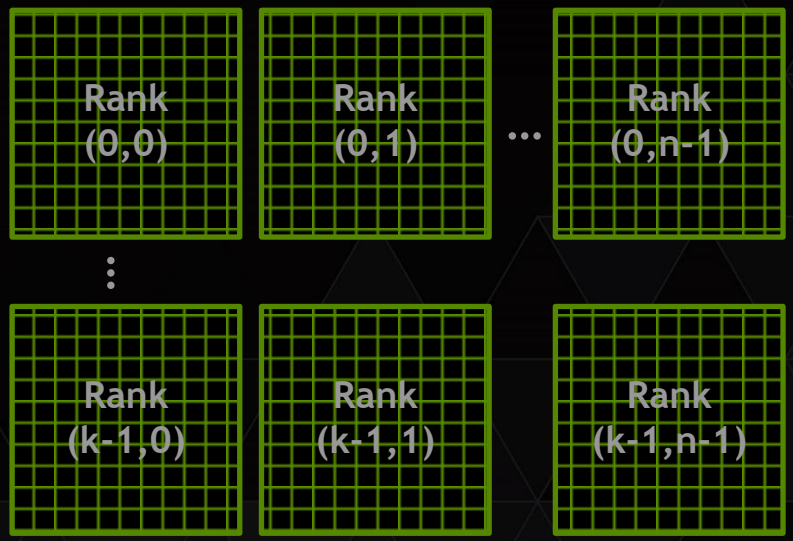
$$\Delta u(x,y) = 0 \; \forall \; (x,y) \in \Omega \backslash \delta\Omega$$

  - Dirichlet boundary conditions (constant values on boundaries)

$$u(x,y) = f(x,y) \in \delta\Omega$$

- 2D domain decomposition with n x k domains





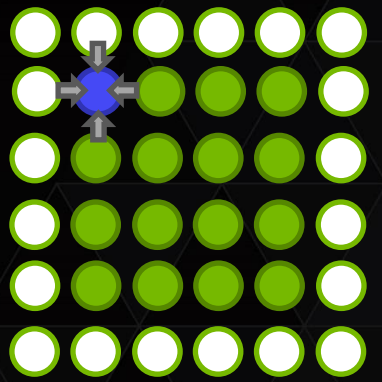
EXAMPLE: JACOBI SOLVER - SINGLE GPU

While not converged

- Do Jacobi step:

```
for (int i=1; i < n-1; i++)
  for (int j=1; j < m-1; j++)
    u_new[i][j] = 0.0f - 0.25f*(u[i-1][j] + u[i+1][j]
                               + u[i][j-1] + u[i][j+1])
```

- Swap `u_new` and `u`
- Next iteration



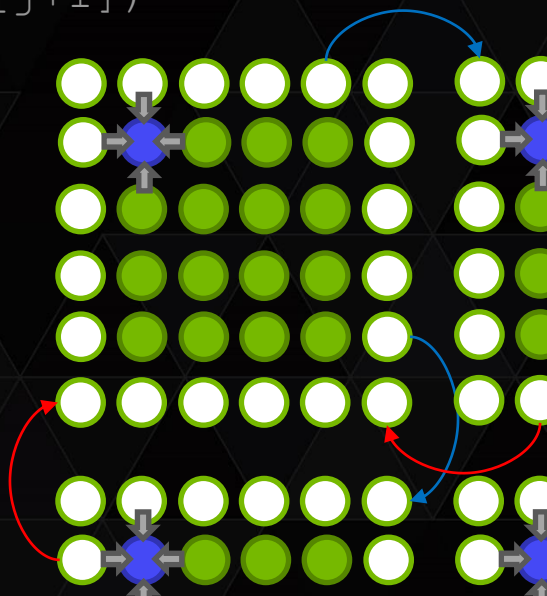
EXAMPLE: JACOBI SOLVER - MULTI GPU

While not converged

- Do Jacobi step:

```
for (int i=1; i < n-1; i++)
    for (int j=1; j < m-1; j++)
      u_new[i][j] = 0.0f - 0.25f*(u[i-1][j] + u[i+1][j]
                                  + u[i][j-1] + u[i][j+1])
```

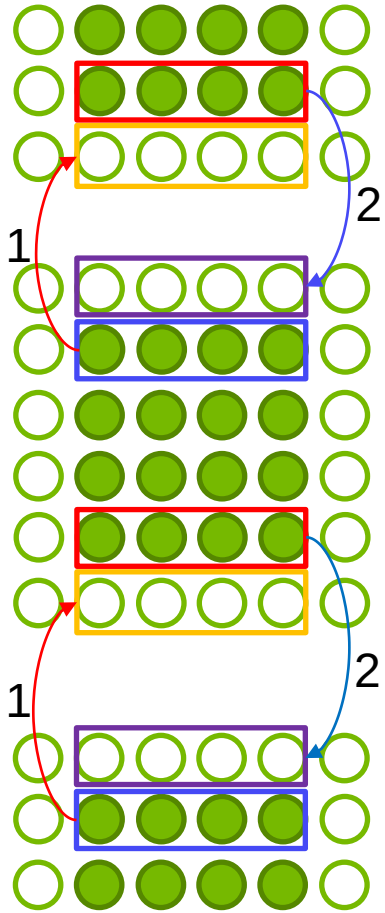
- Exchange halo with 2 4 neighbor
- Swap `u_new` and `u`
- Next iteration



EXAMPLE: JACOBI - TOP/BOTTOM HALO

```
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



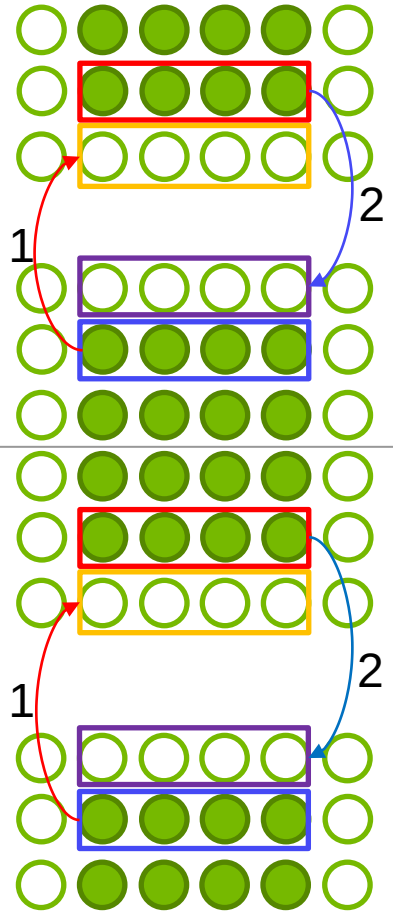
EXAMPLE: JACOBI - TOP/BOTTOM HALO

OpenACC

```
#pragma acc host_data use_device ( u_new ) {
  MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
               u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
               u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

CUDA

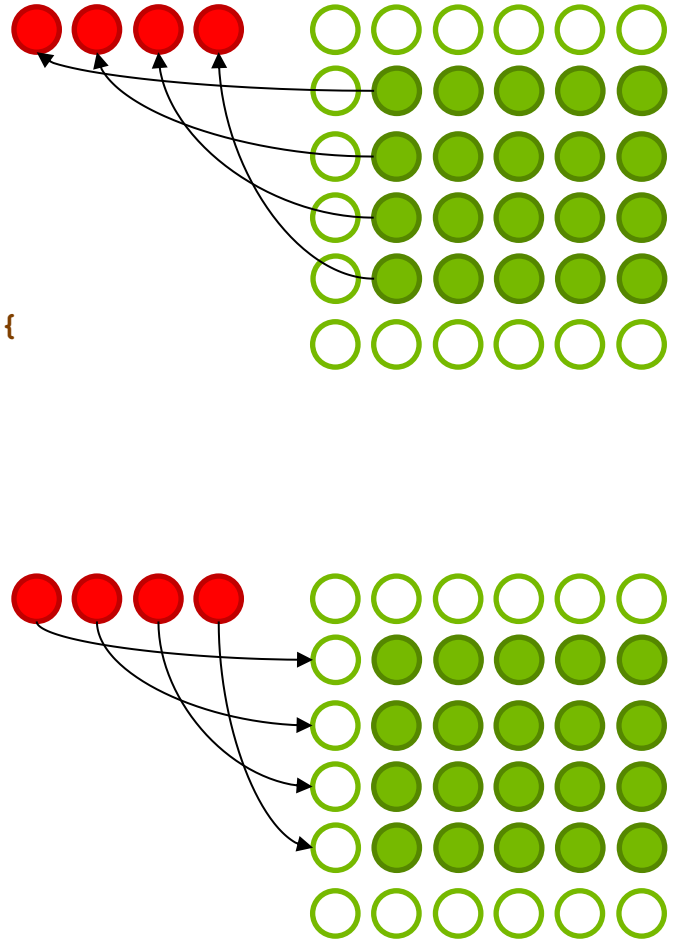
```
MPI_Sendrecv(u_new_d+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new_d+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new_d+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new_d+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



EXAMPLE: JACOBI - LEFT/RIGHT HALO

OpenACC

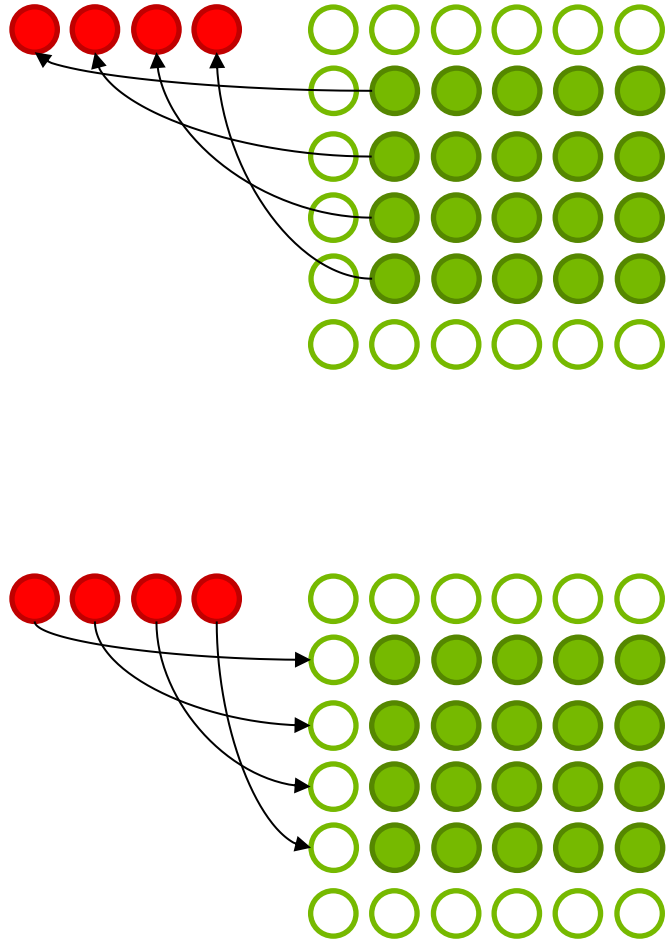
```
//right neighbor omitted
#pragma acc parallel loop present ( u_new, to_left )
for ( int i=0; i<n-2; ++i )
     to_left[i] = u_new[(i+1)*m+1];

#pragma acc host_data use_device ( from_left, to_left ) {
  MPI_Sendrecv( to_left, n-2, MPI_DOUBLE, l_nb, 0,
                from_left, n-2, MPI_DOUBLE, l_nb, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}

#pragma acc parallel loop present ( u_new, from_left )
for ( int i=0; i<n-2; ++i )
     u_new[(i+1)*m] = from_left[i];
```



EXAMPLE: JACOBI - LEFT/RIGHT HALO

CUDA

```
//right neighbor omitted
pack<<<gs,bs,0,s>>>(to_left_d, u_new_d, n, m);
cudaStreamSynchronize(s);




MPI_Sendrecv( to_left_d, n-2, MPI_DOUBLE, l_nb, 0,
              from_left_d, n-2, MPI_DOUBLE, l_nb, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );




unpack<<<gs,bs,0,s>>>(u_new_d, from_left_d, n, m);
```



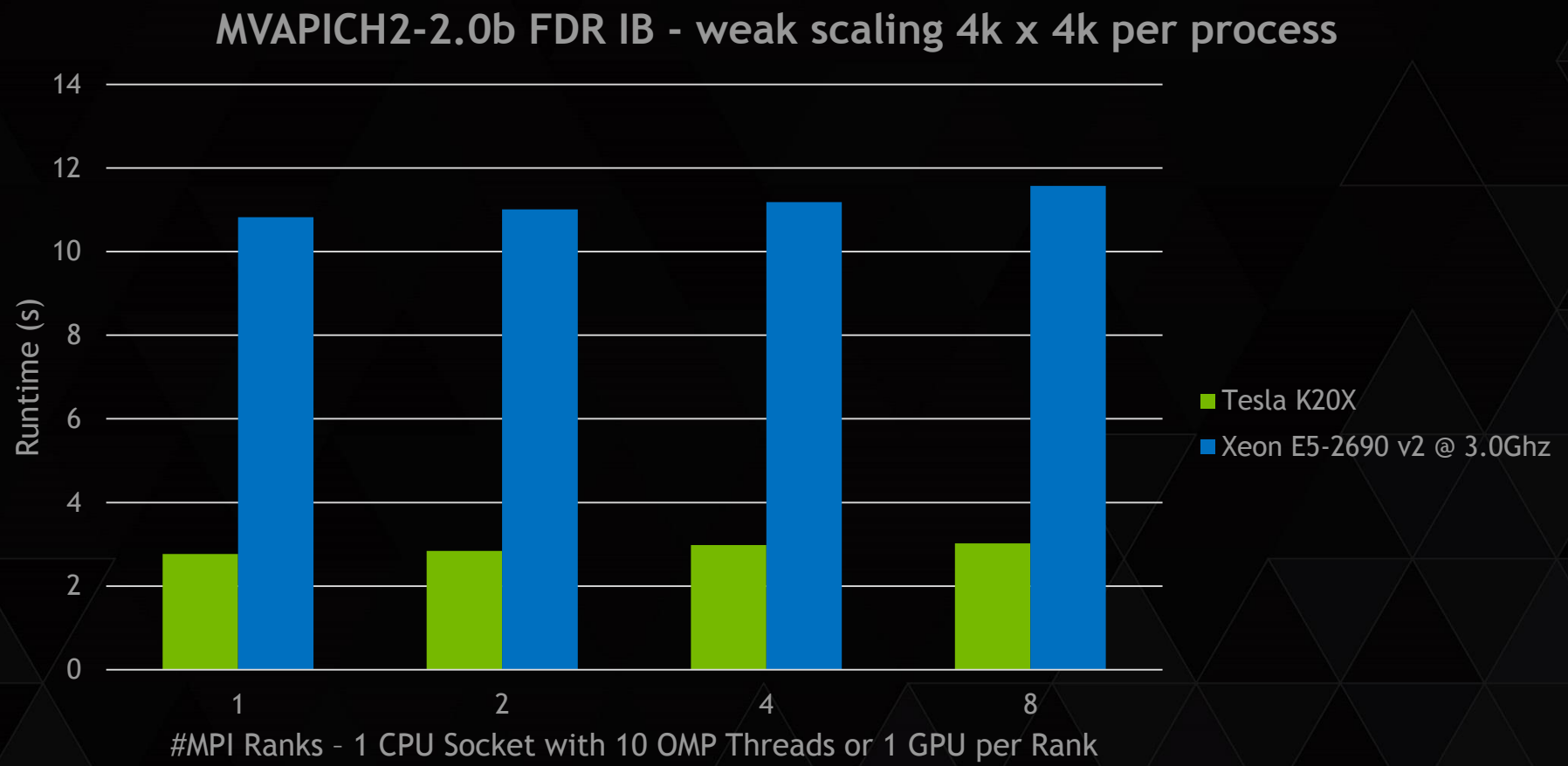
LAUNCH MPI+CUDA/OPENACC PROGRAMS

- Launch one process per GPU
  - MVAPICH: `MV2_USE_CUDA`

```
$ MV2_USE_CUDA=1 mpirun -np ${np} ./myapp <args>
```

  - Open MPI: CUDA-aware features are enabled per default
  - Cray: `MPICH_RDMA_ENABLED_CUDA`
  - IBM Platform MPI: `PMPI_GPU_AWARE`

JACOBI RESULTS (1000 STEPS)



EXAMPLE: JACOBI - TOP/BOTTOM HALO

OpenACC

```
#pragma acc update host( u_new[1:m-2], u_new[(n-2)*m+1:m-2] )
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
#pragma acc update device( u_new[0:m-2], u_new[(n-2)*m:m-2] )
```

CUDA

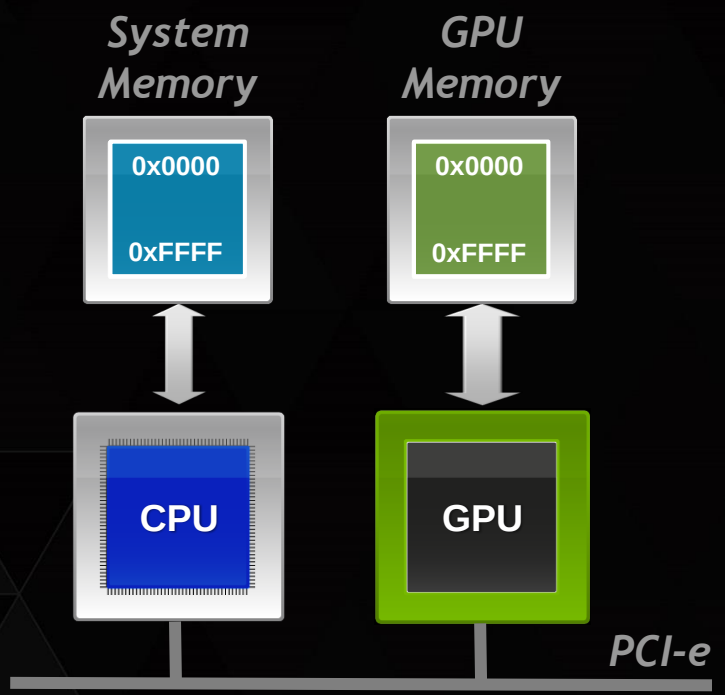
```
//send to bottom and receive from top - top bottom omitted

cudaMemcpy(u_new+1, u_new_d+1, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
cudaMemcpy(u_new_d, u_new, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
```

without CUDA-aware MPI

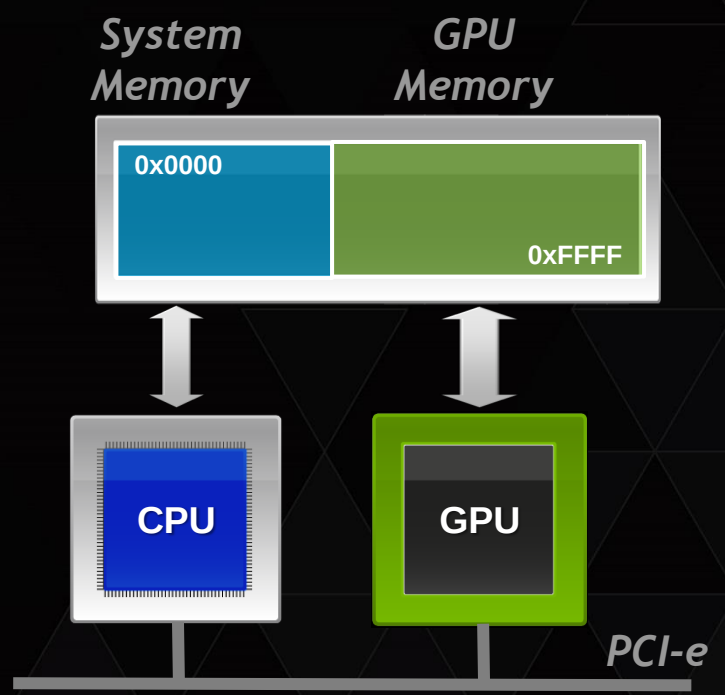
THE DETAILS

UNIFIED VIRTUAL ADDRESSING

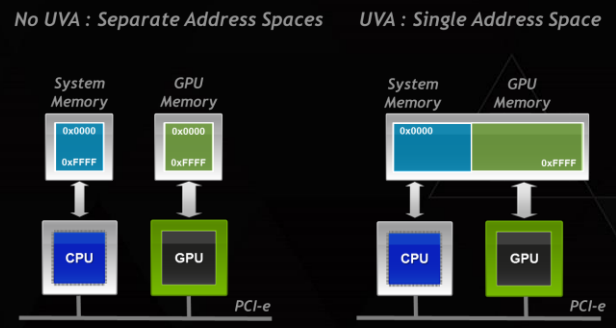
No UVA : Separate Address Spaces



UVA : Single Address Space

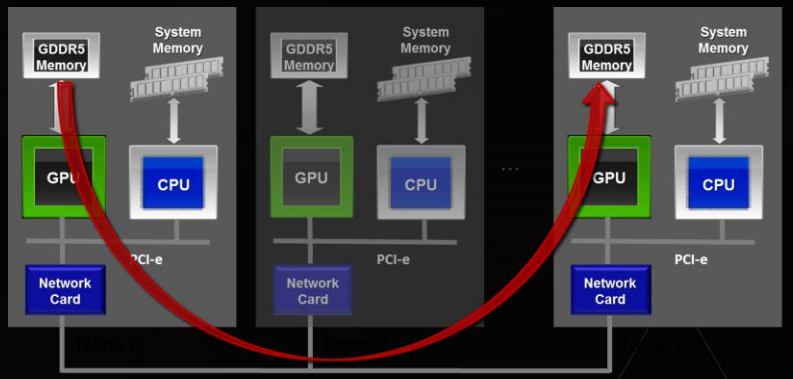


UNIFIED VIRTUAL ADDRESSING



- One address space for all CPU and GPU memory
  - Determine physical memory location from a pointer value
  - Enable libraries to simplify their interfaces (e.g. MPI and cudaMemcpy)
- Supported on devices with compute capability 2.0 for
  - 64-bit applications on Linux and on Windows also TCC mode

MPI+CUDA



With UVA and CUDA-aware MPI

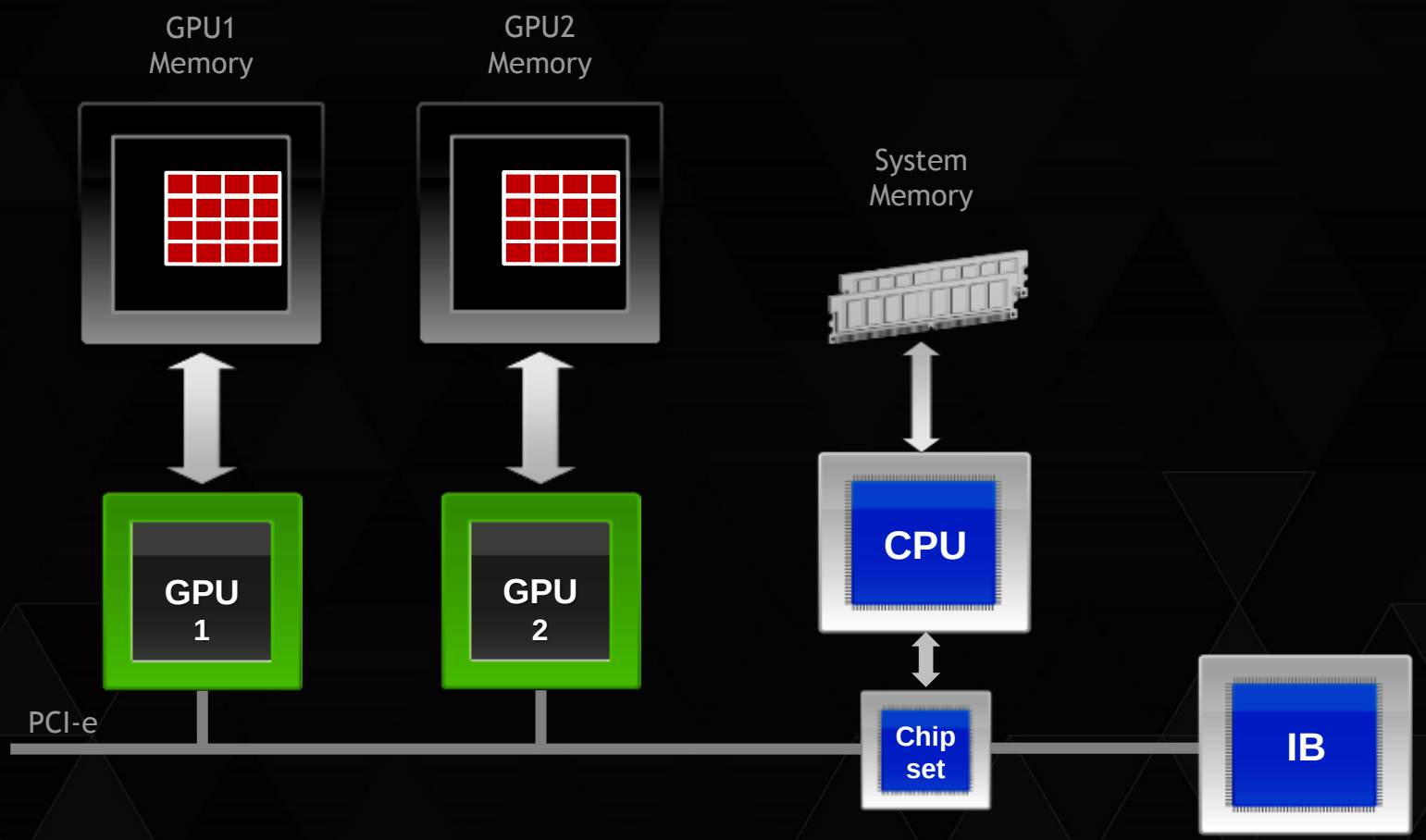
```
//MPI rank 0
MPI_Send(s_buf_d,size,…);


//MPI rank n-1
MPI_Recv(r_buf_d,size,…);
```

No UVA and regular MPI

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,…);
MPI_Send(s_buf_h,size,…);

//MPI rank n-1
MPI_Recv(r_buf_h,size,…);
cudaMemcpy(r_buf_d,r_buf_h,size,…);
```

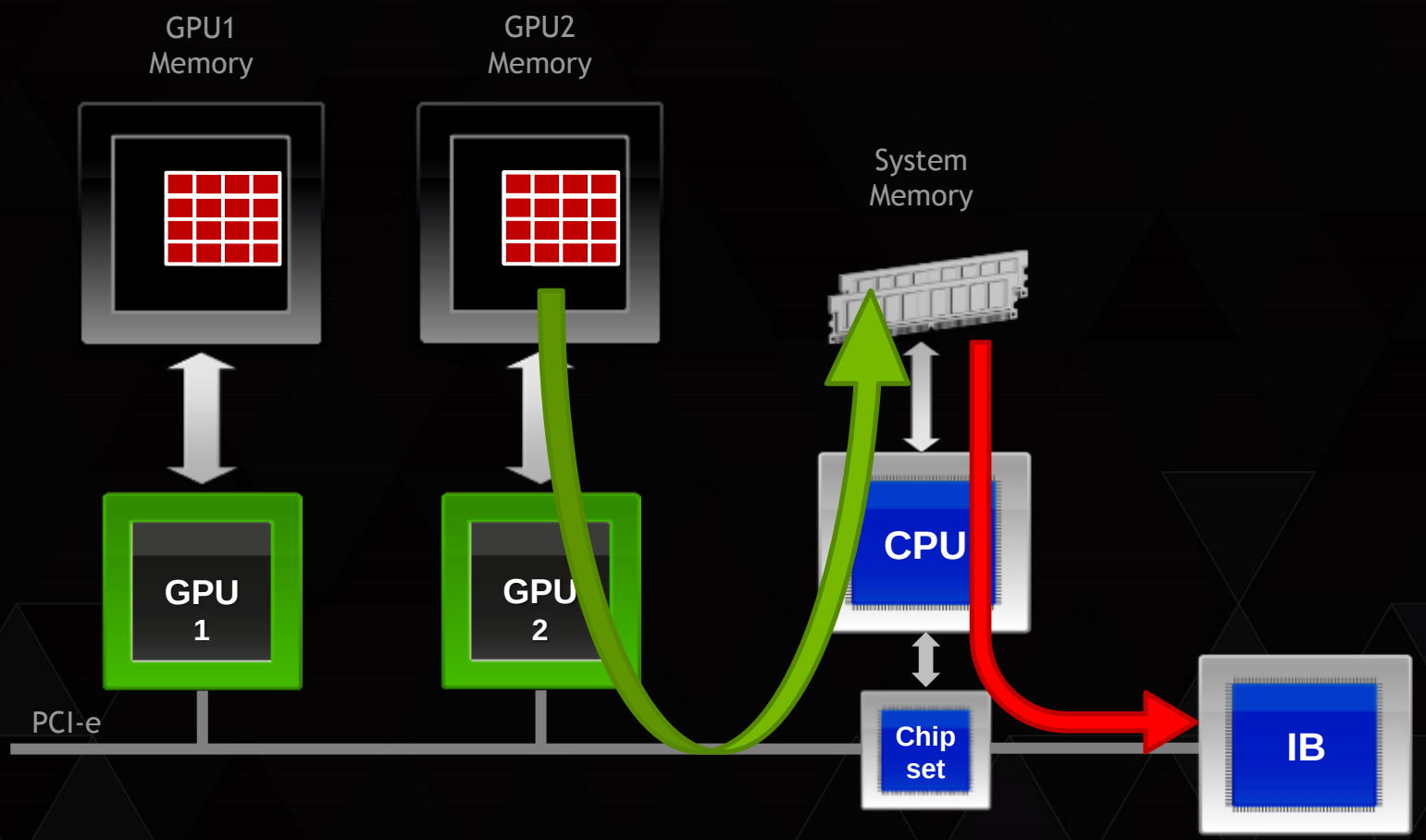
NVIDIA GPUDIRECT™

ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES



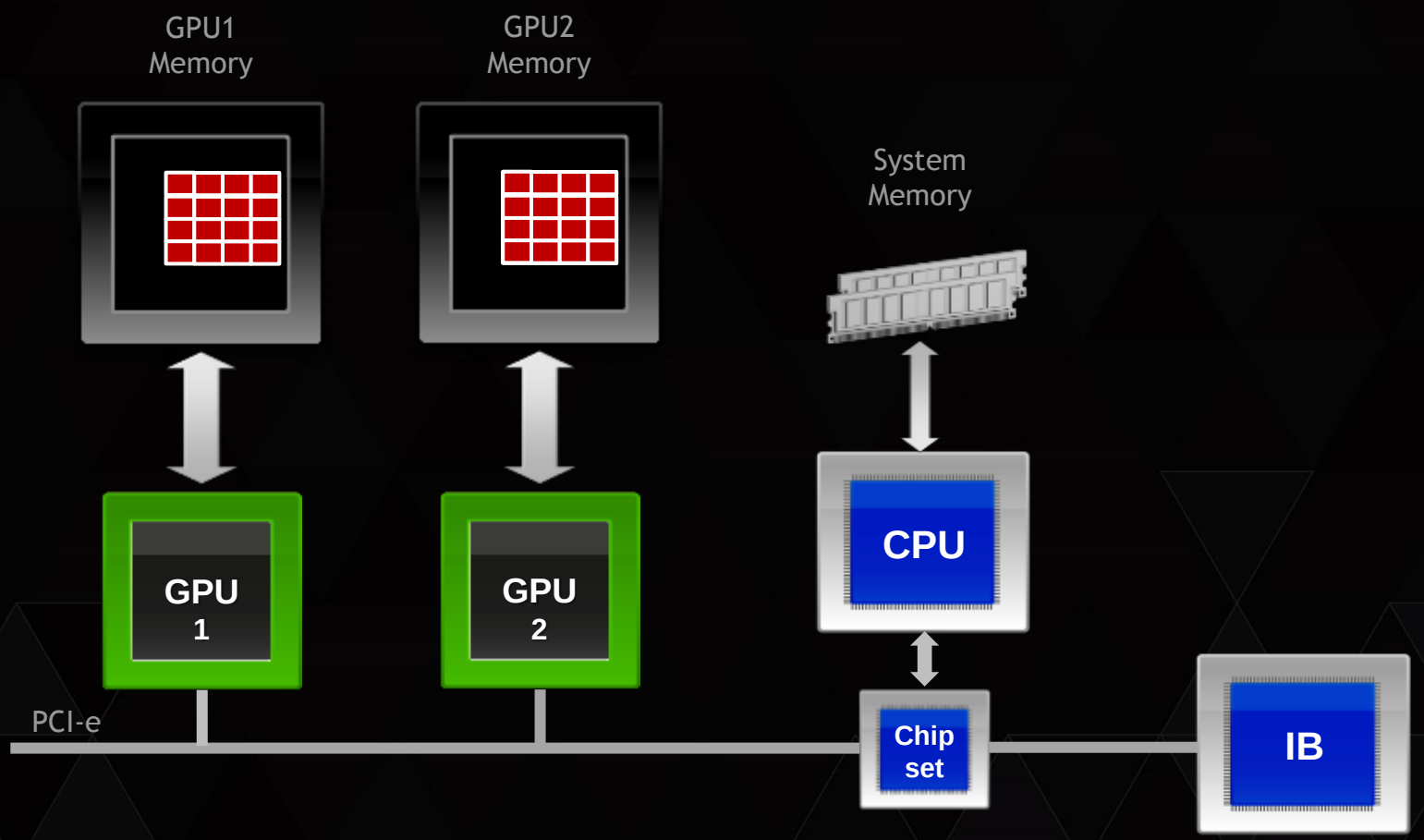
NVIDIA GPUDIRECT™

ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES



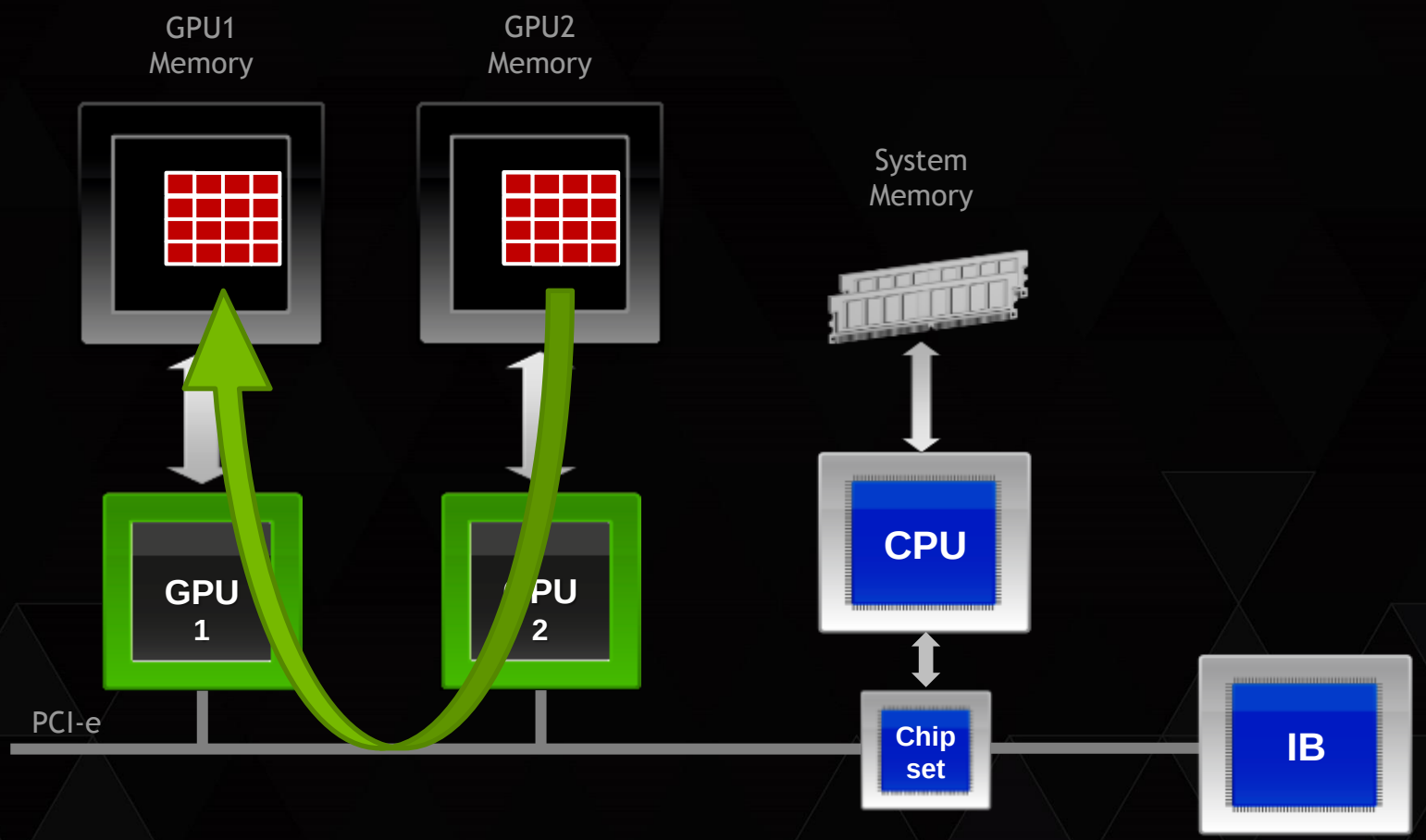
NVIDIA GPUDIRECT™

PEER TO PEER TRANSFERS



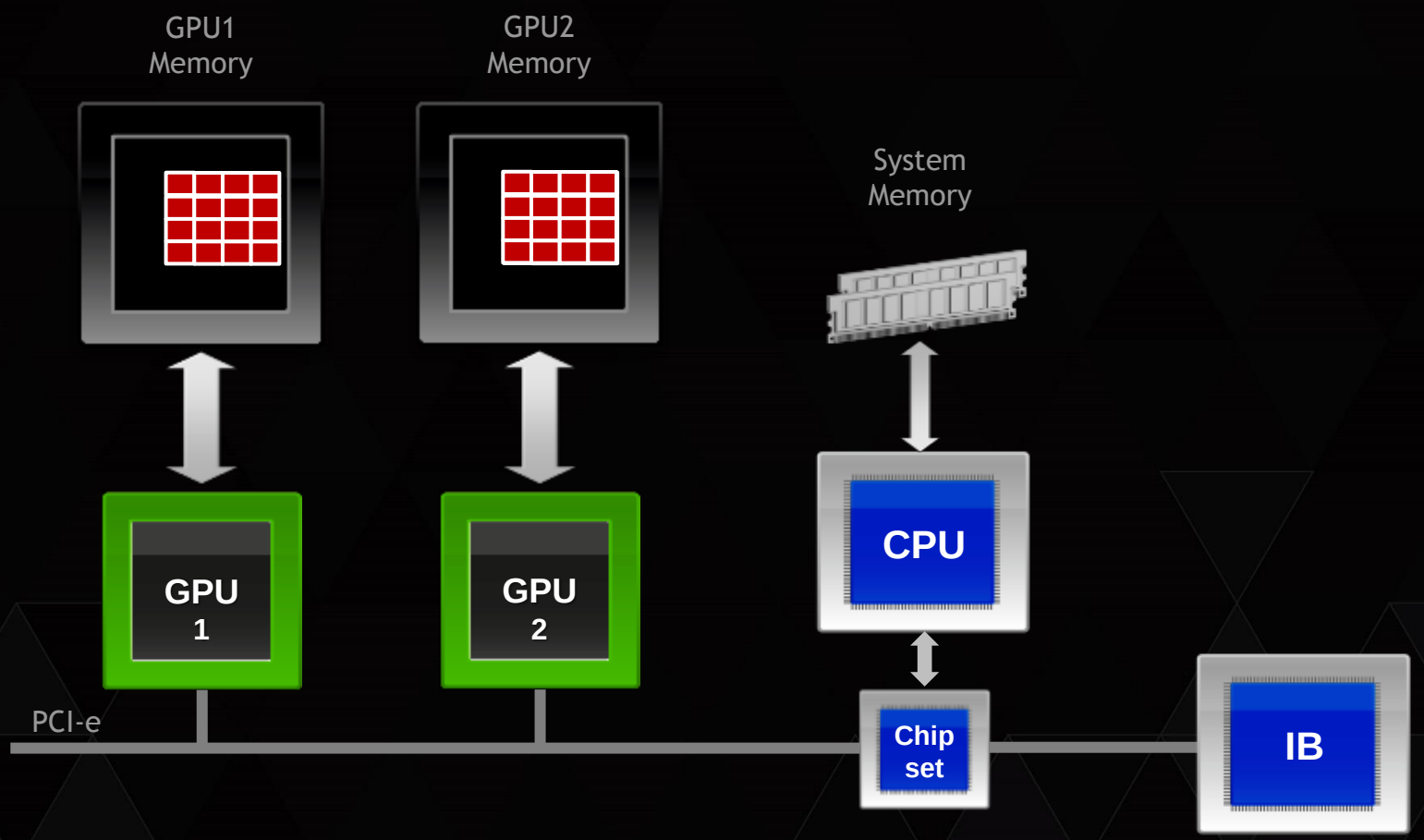
NVIDIA GPUDIRECT™

PEER TO PEER TRANSFERS



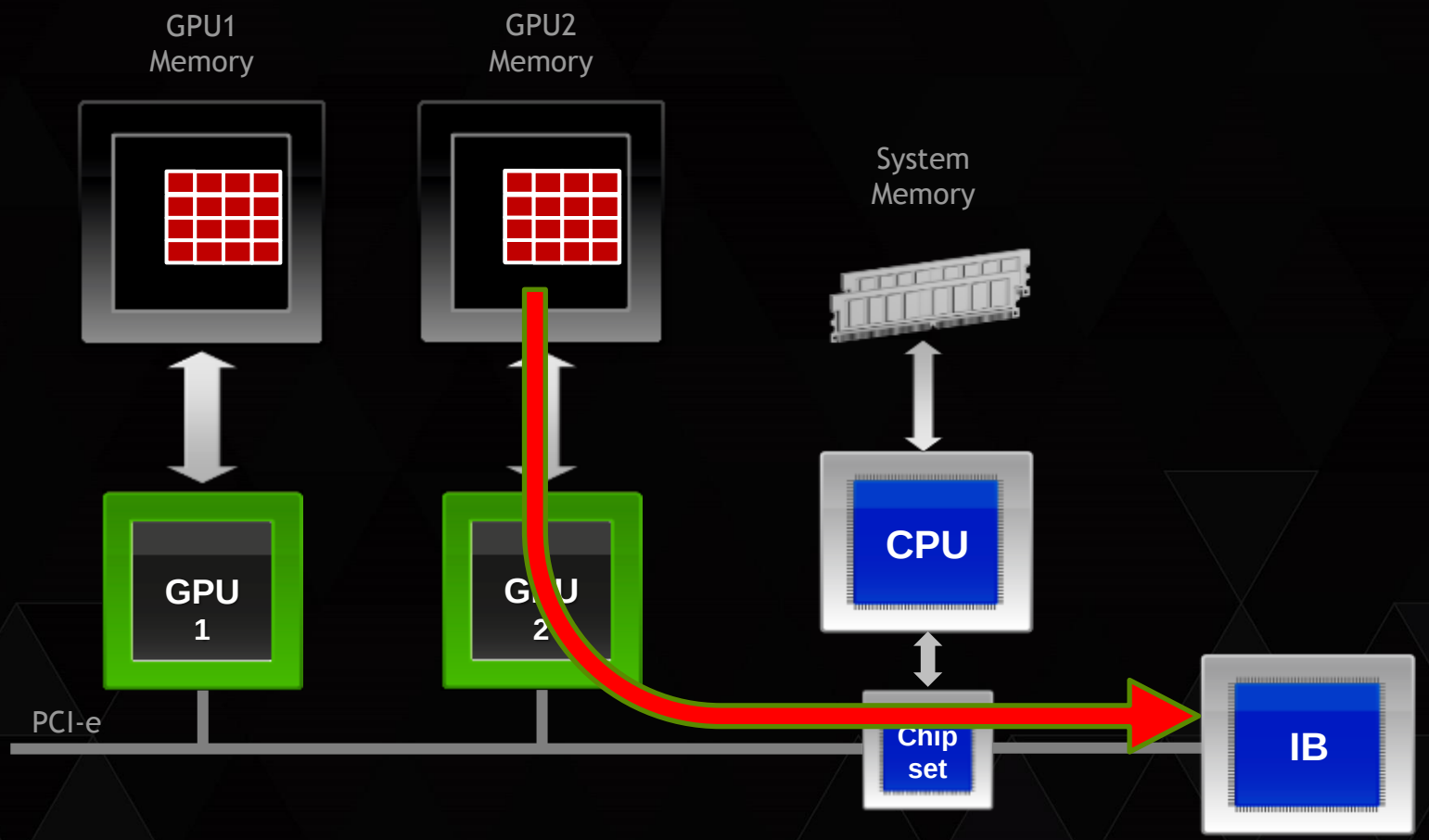
NVIDIA GPUDIRECT™

SUPPORT FOR RDMA



NVIDIA GPUDIRECT™

SUPPORT FOR RDMA



CUDA-AWARE MPI

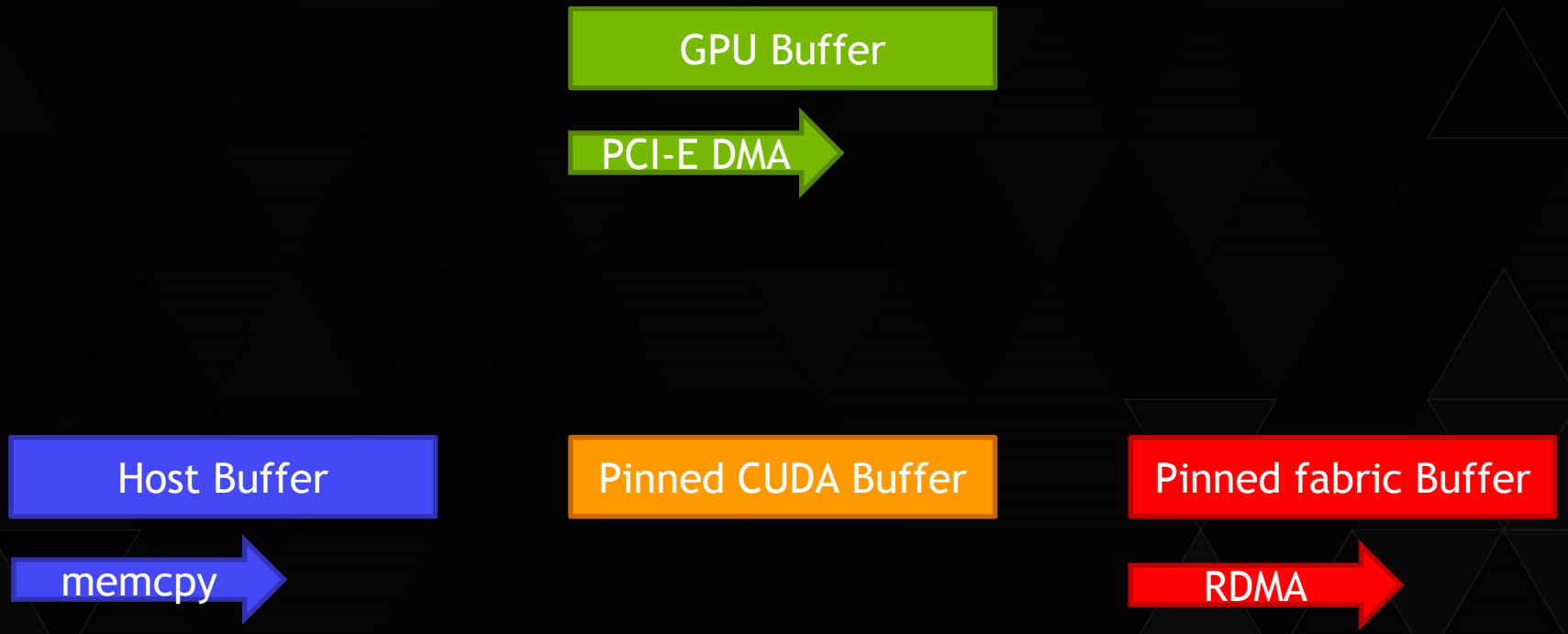
Example:

MPI Rank 0 MPI_Send from GPU Buffer

MPI Rank 1 MPI_Recv to GPU Buffer

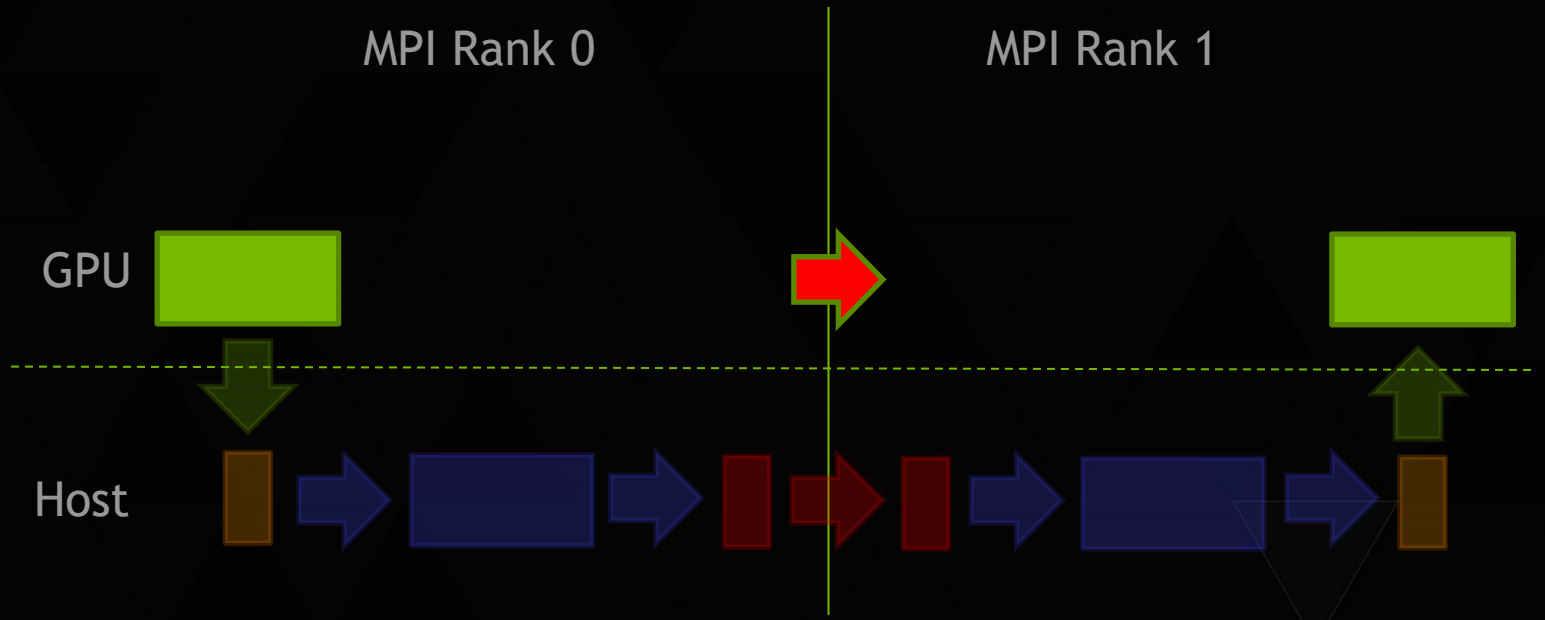
- Show how CUDA+MPI works in principle
  - Depending on the MPI implementation, message size, system setup, … situation might be different
- Two GPUs in two nodes

CUDA-AWARE MPI



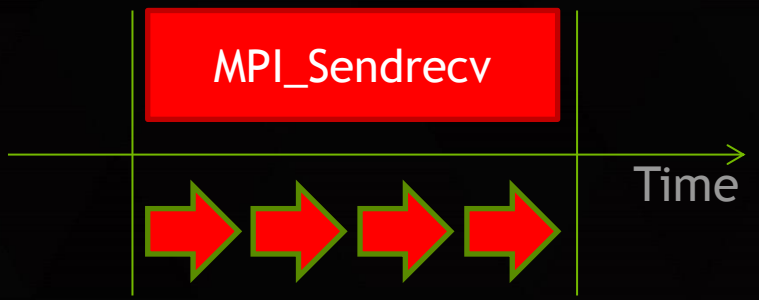
MPI GPU TO REMOTE GPU

SUPPORT FOR RDMA

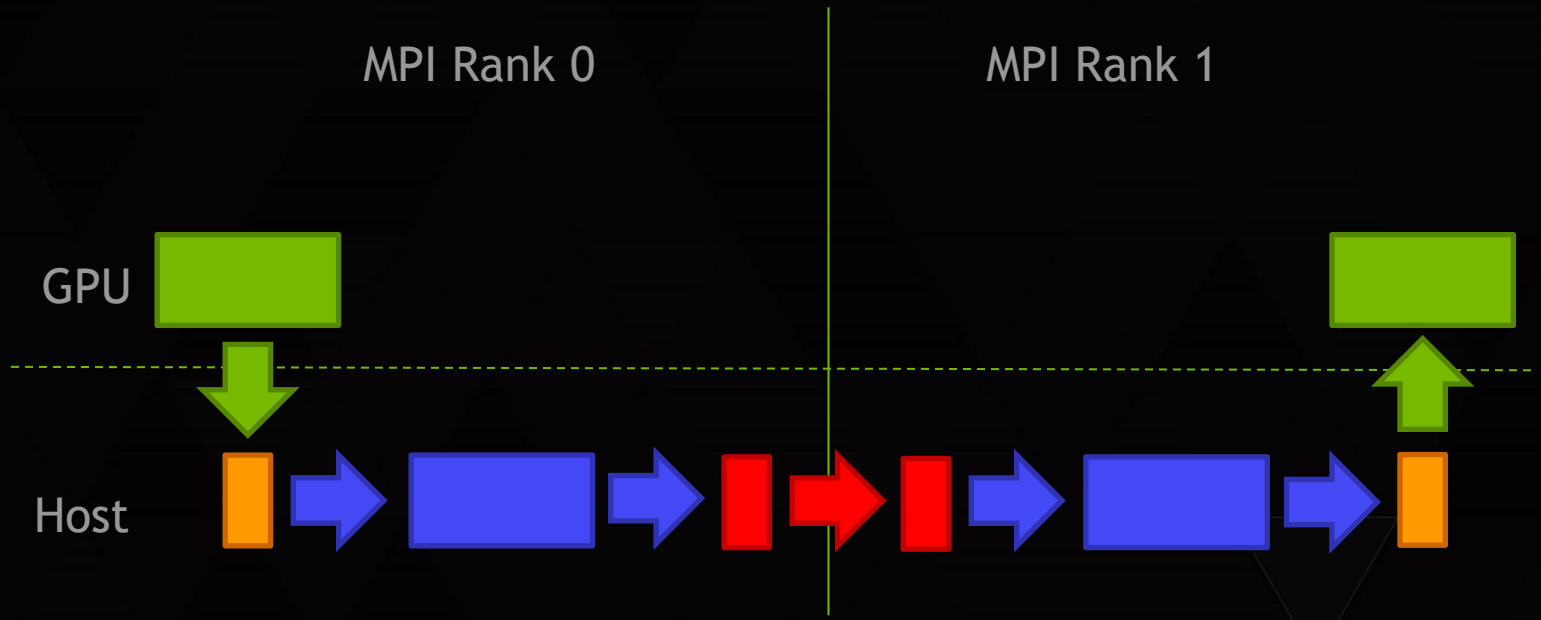



```
MPI_Send(s_buf_d,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MPI GPU TO REMOTE GPU
SUPPORT FOR RDMA

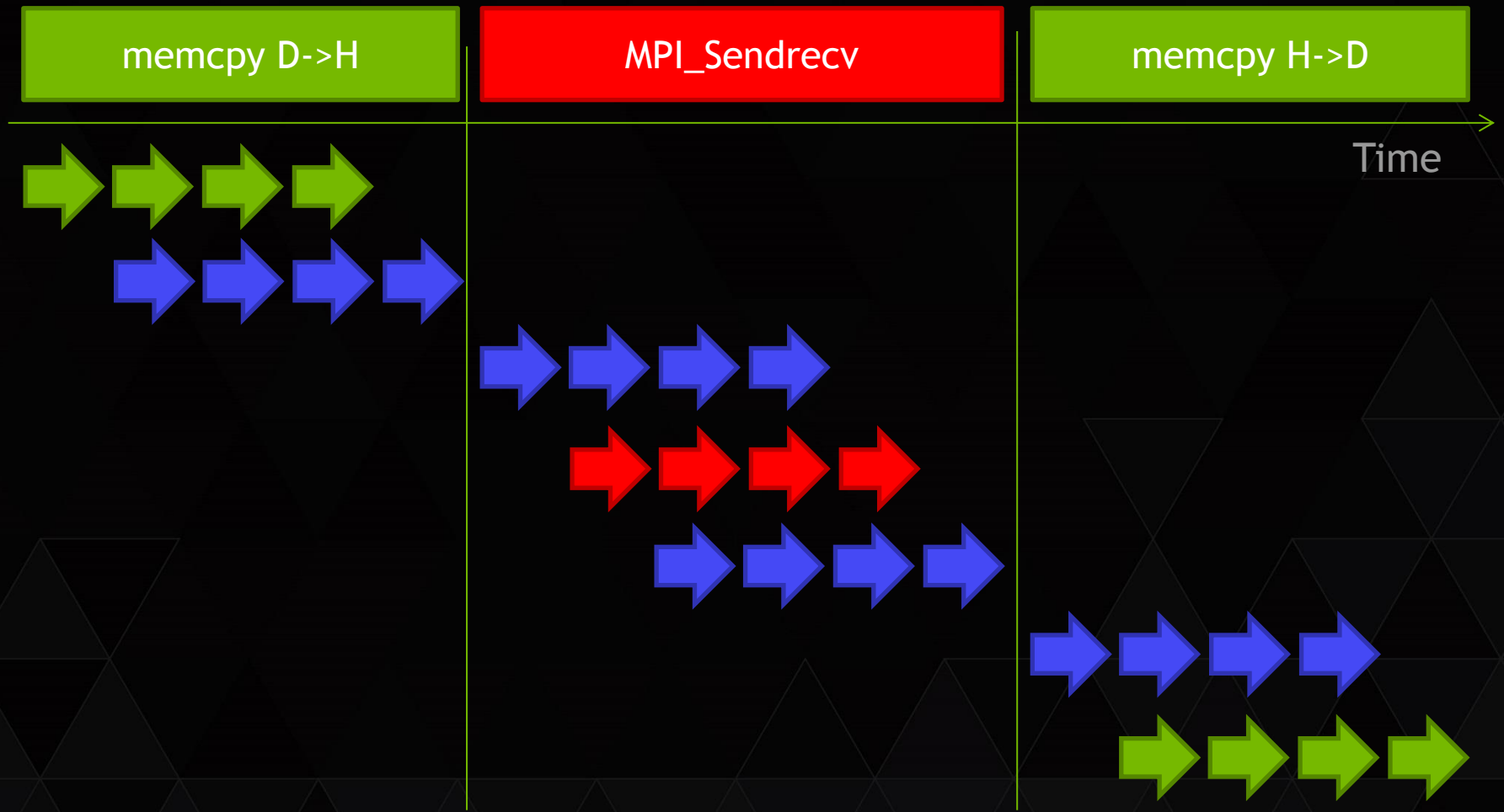


REGULAR MPI GPU TO REMOTE GPU

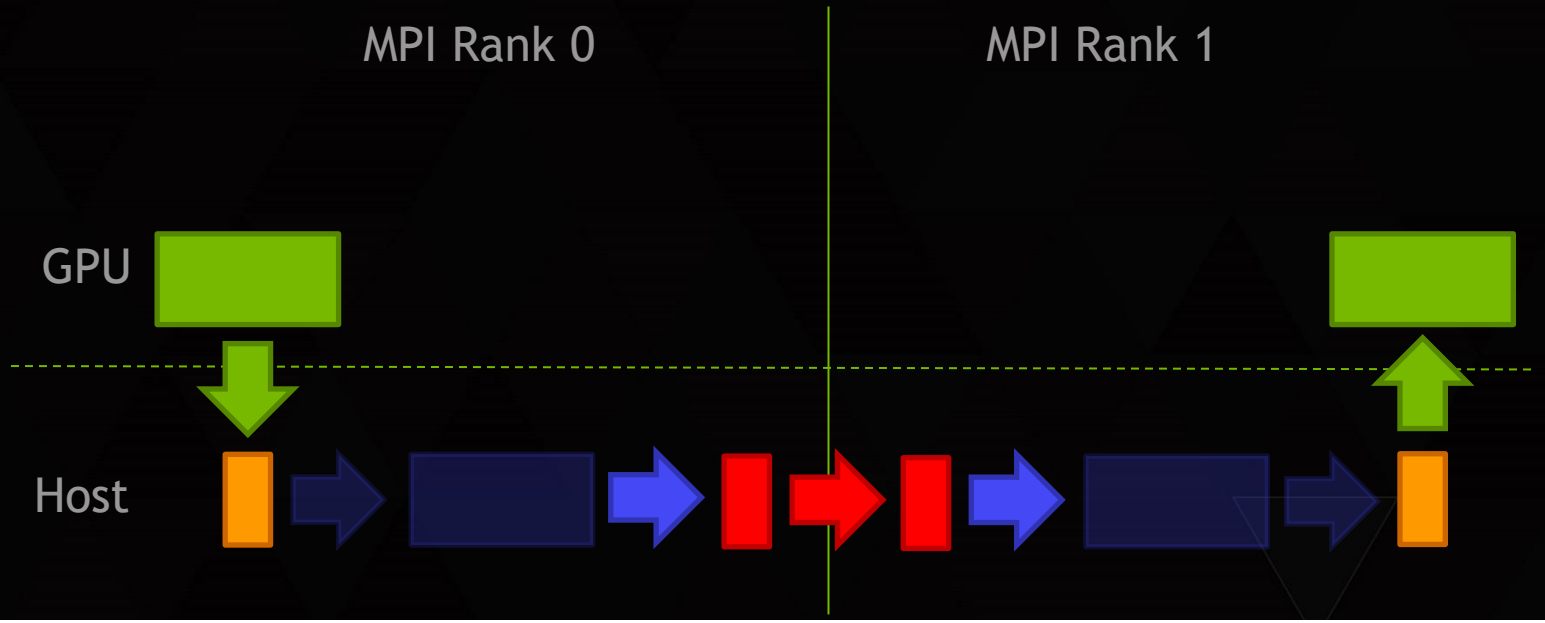



```
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

REGULAR MPI GPU TO REMOTE GPU

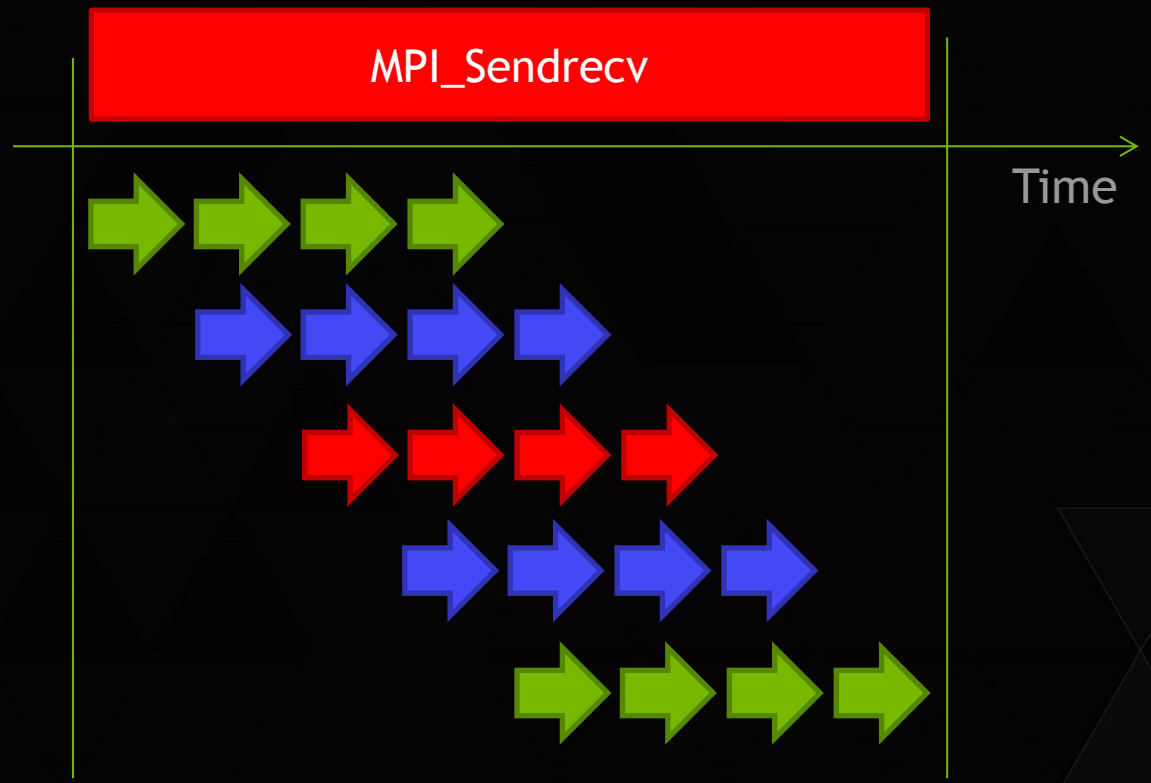


MPI GPU TO REMOTE GPU
WITHOUT GPUDIRECT

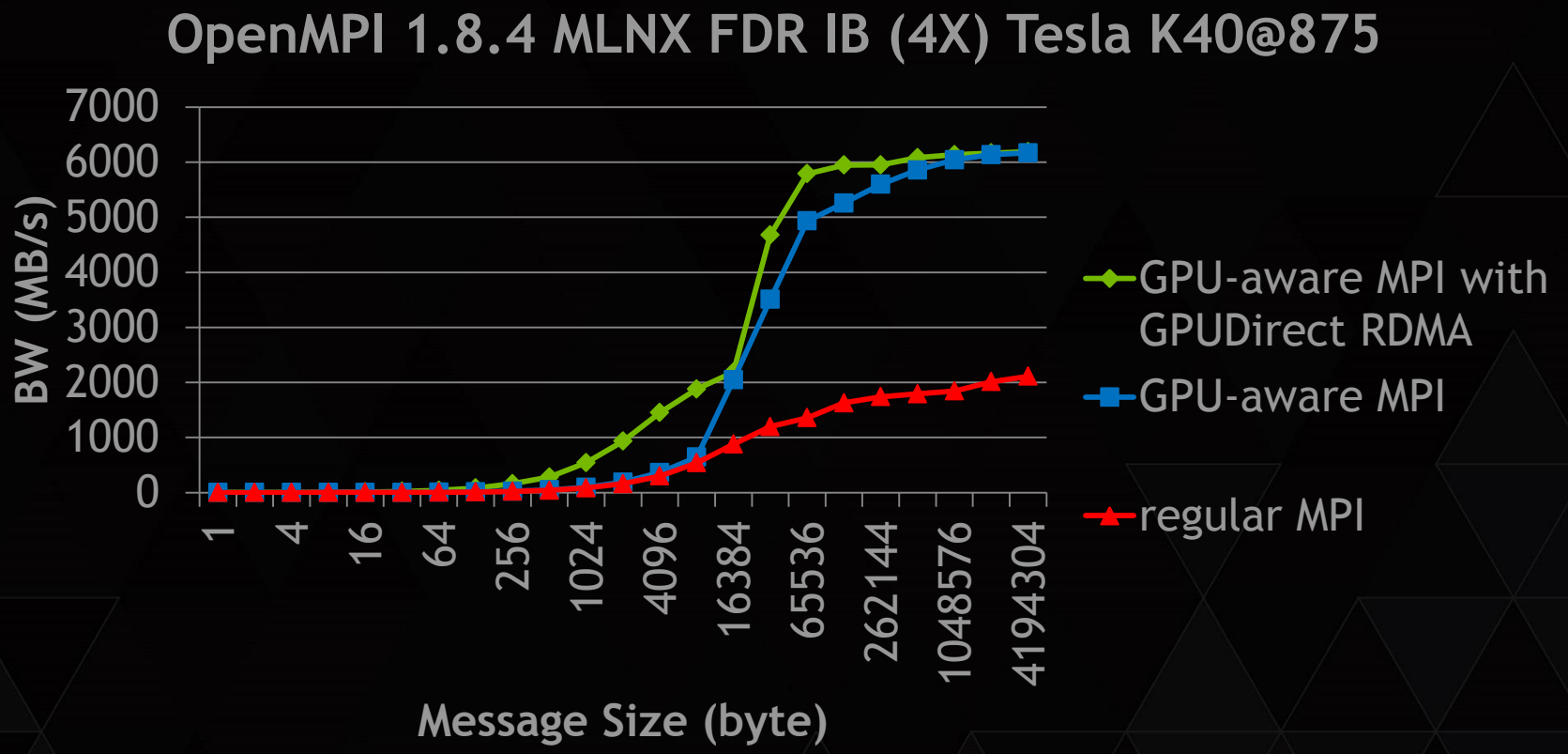



```
MPI_Send(s_buf_h,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MPI GPU TO REMOTE GPU
WITHOUT GPUDIRECT



PERFORMANCE RESULTS TWO NODES

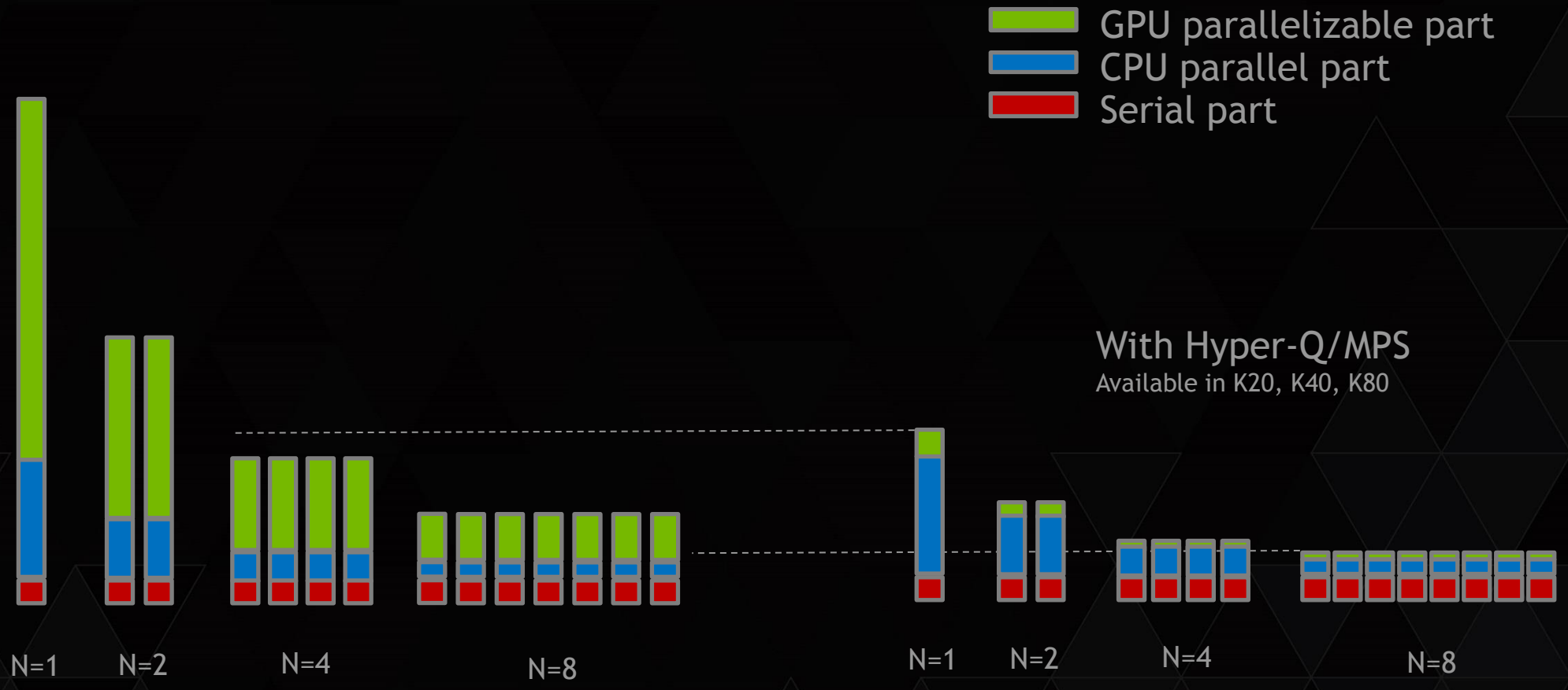


Latency (1 byte)	19.79 us	17.97 us	5.70 us

MULTI PROCESS SERVICE (MPS) FOR MPI APPLICATIONS

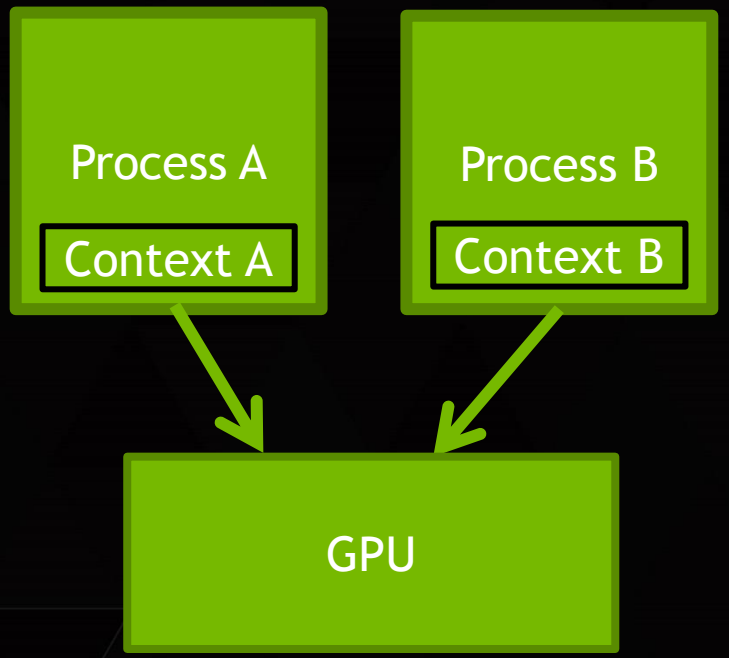
GPU ACCELERATION OF LEGACY MPI APPS

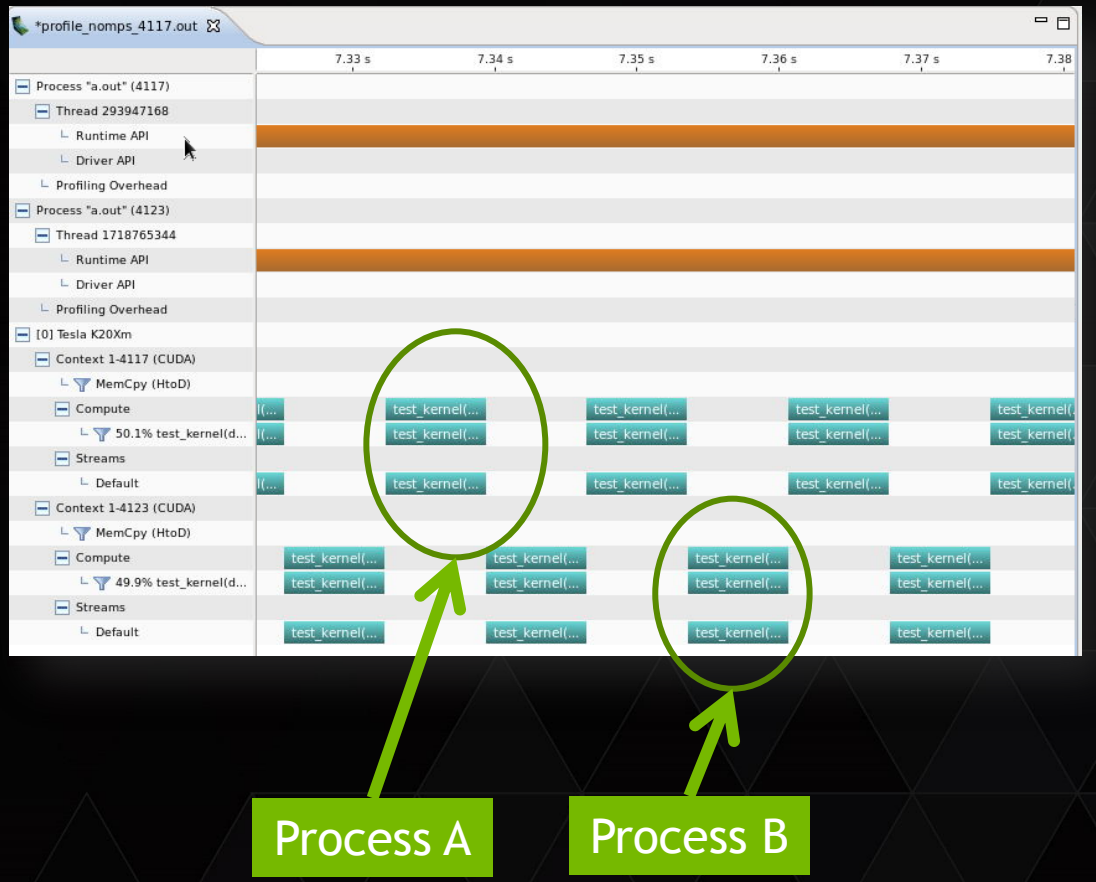
- Typical legacy application
  - MPI parallel
  - Single or few threads per MPI rank (e.g. OpenMP)
- Running with multiple MPI ranks per node
- GPU acceleration in phases
  - Proof of concept prototype, ..
  - Great speedup at kernel level
- Application performance misses expectations



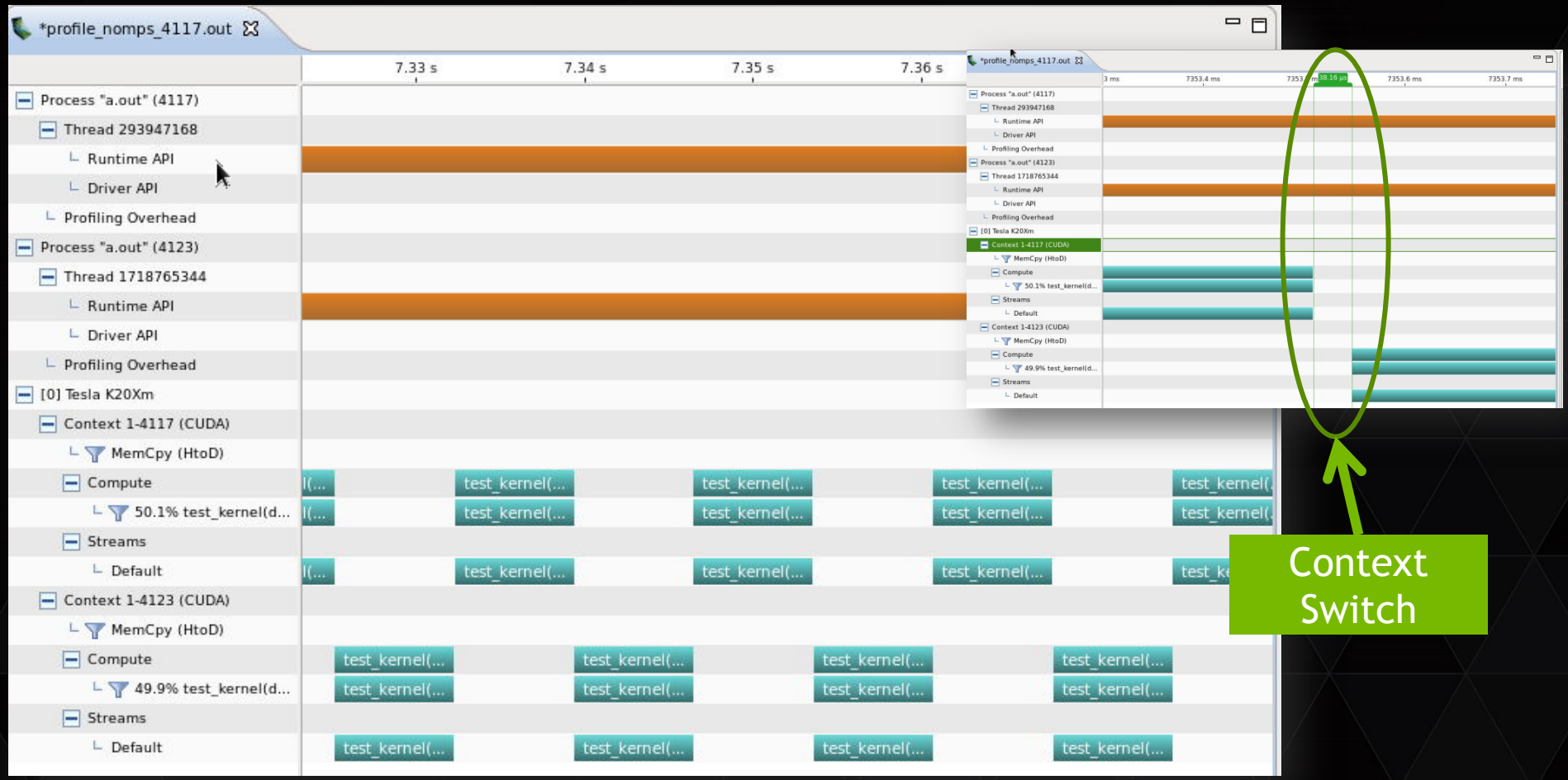
PROCESSES SHARING GPU WITHOUT MPS:

NO OVERLAP

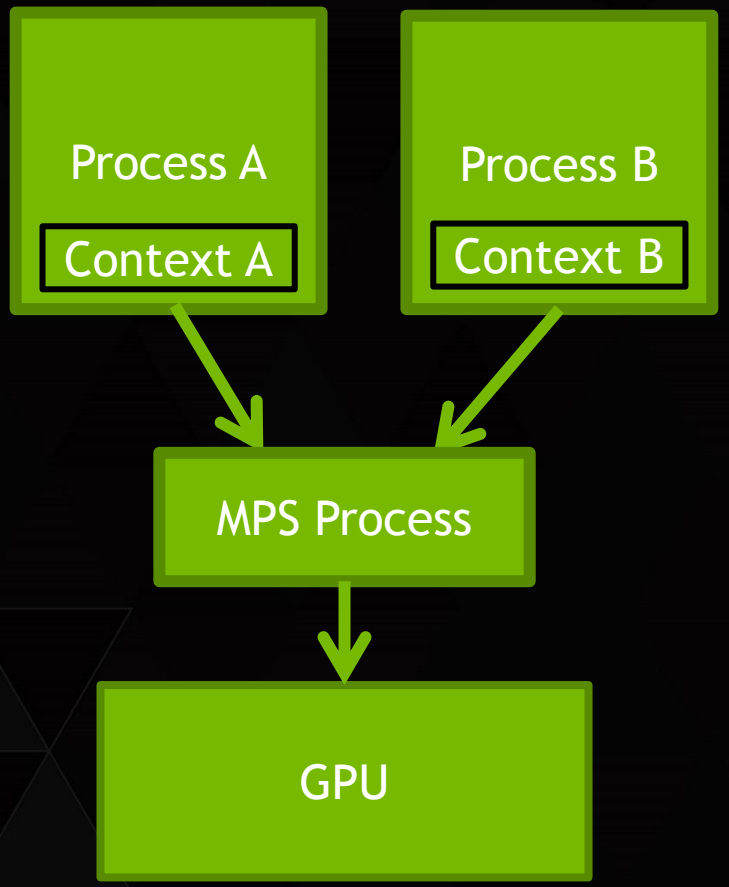


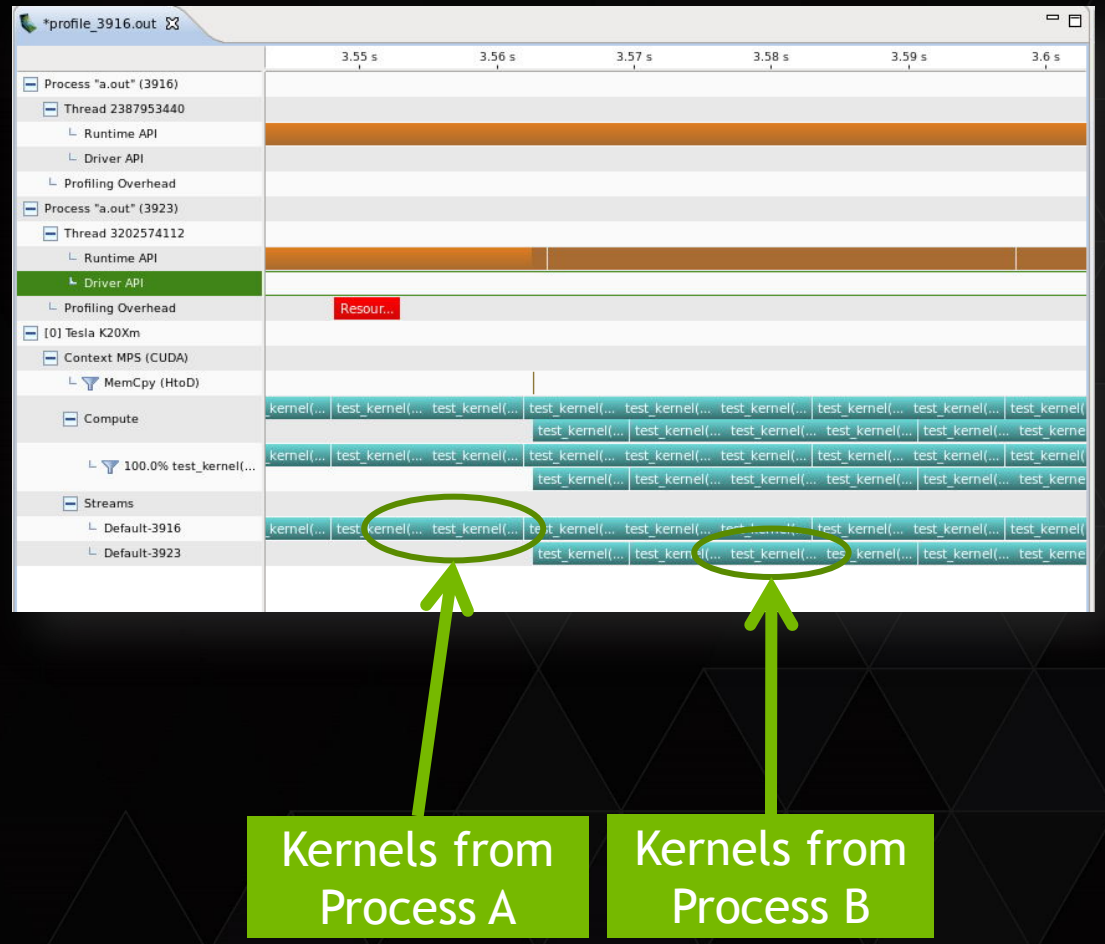


GPU SHARING WITHOUT MPS

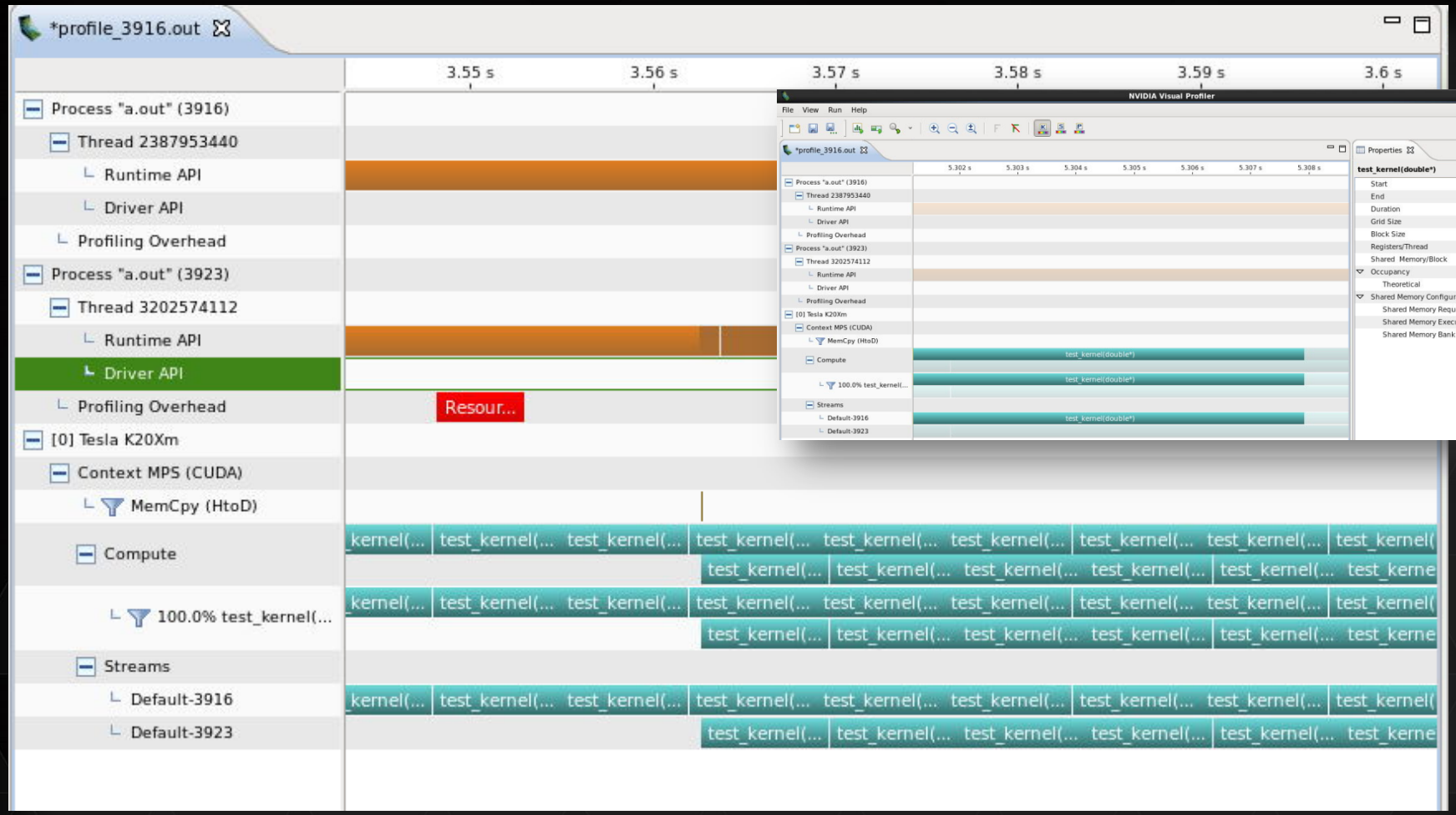


PROCESSES SHARING GPU WITH MPS:
MAXIMUM OVERLAP

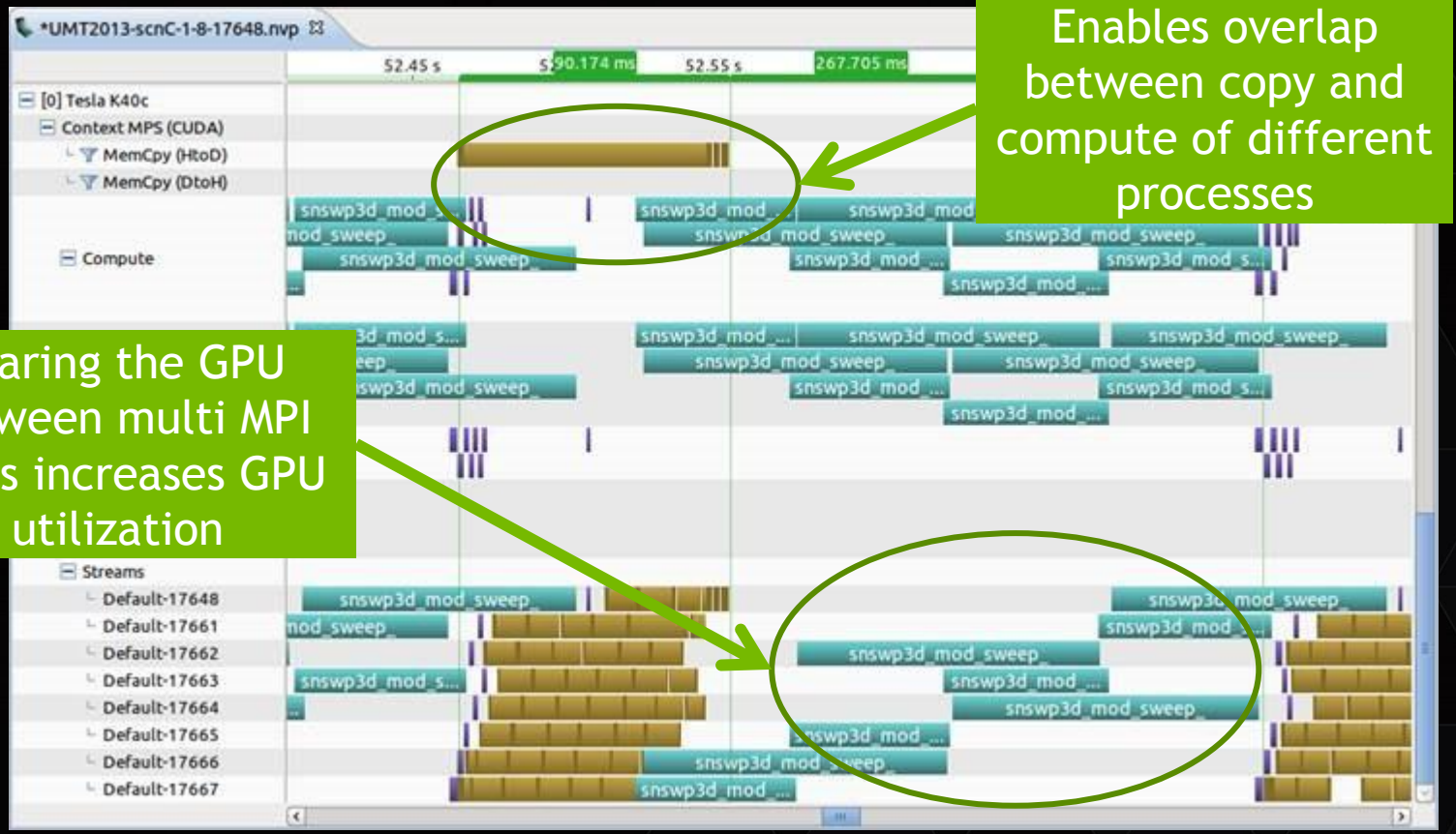




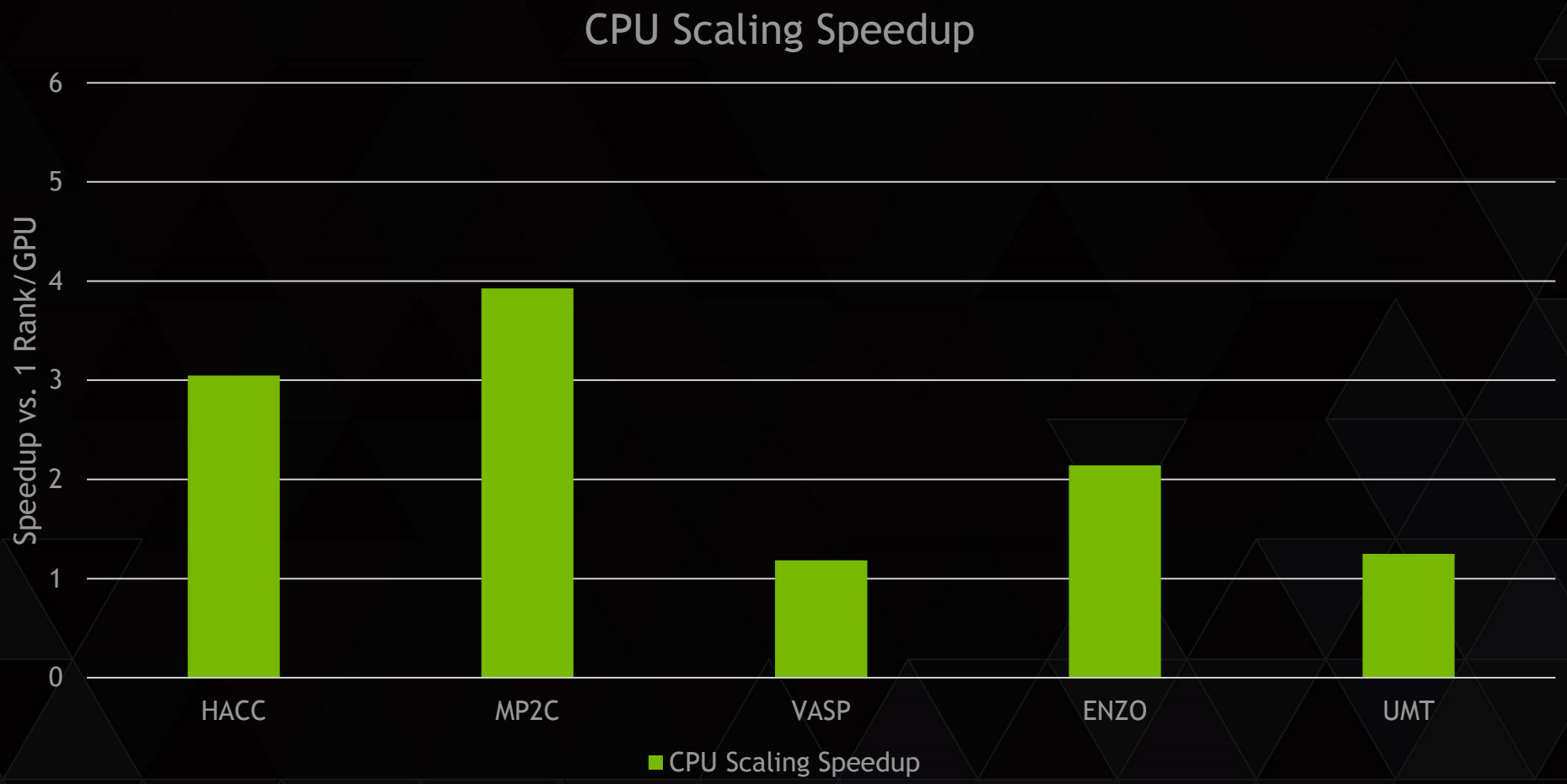
GPU SHARING WITH MPS



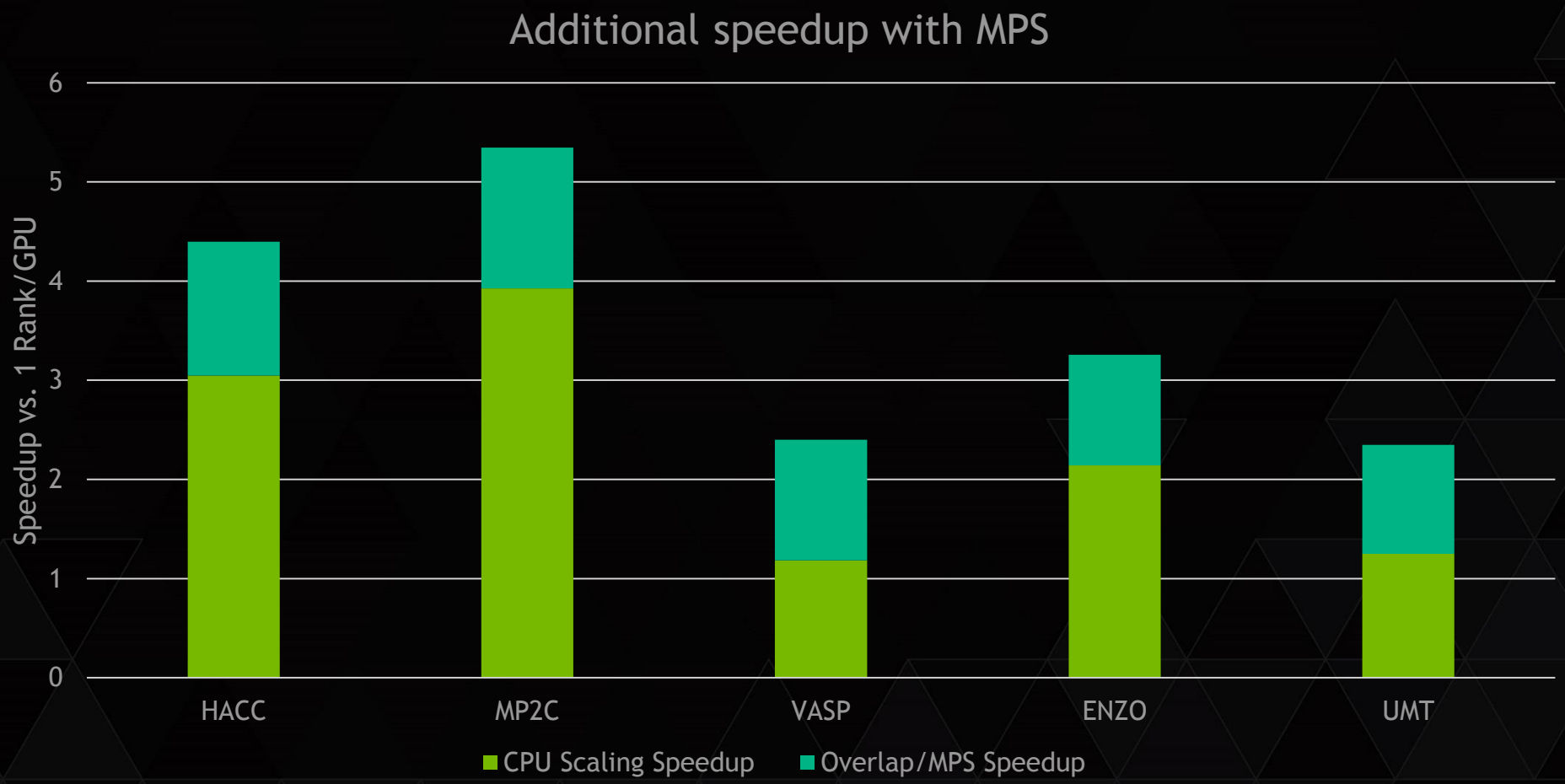
CASE STUDY: HYPER-Q/MPS FOR UMT



HYPER-Q/MPS CASE STUDIES



HYPER-Q/MPS CASE STUDIES



USING MPS

- No application modifications necessary
- Not limited to MPI applications
- MPS control daemon
  - Spawn MPS server upon CUDA application startup

- Typical setup

```
export CUDA_VISIBLE_DEVICES=0
nvidia-smi       -c EXCLUSIVE_PROCESS
nvidia-cuda-mps-control -d
```

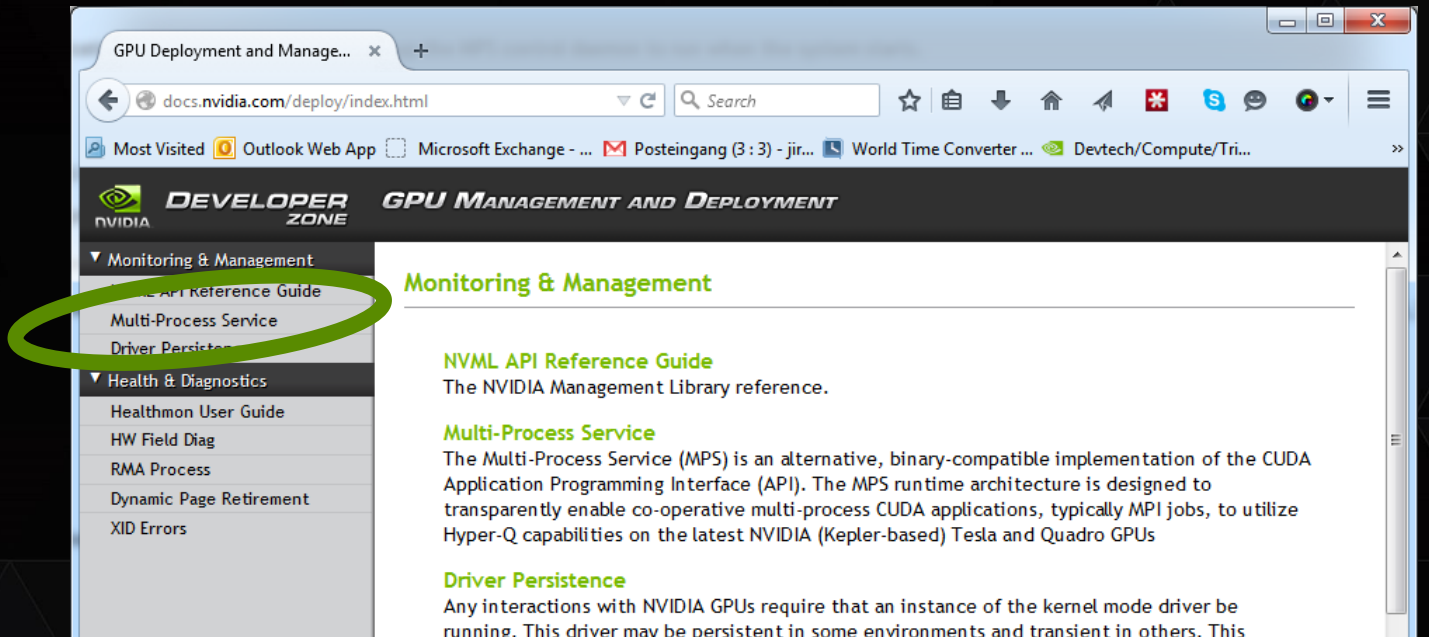
- On Cray XK/XC systems

```
export CRAY_CUDA_MPS=1
```

New with CUDA 7:
MPS support on
MULTI-GPU systems

MPS SUMMARY

- Easy path to get GPU acceleration for legacy applications
- Enables overlapping of memory copies and compute between different MPI ranks
- Remark: MPS adds some overhead so if you application does not benefit you want to avoid starting MPS.



DEBUGGING AND PROFILING

TOOLS FOR MPI+CUDA APPLICATIONS

- Memory Checking `cuda-memcheck`
- Debugging `cuda-gdb`
- Profiling `nvprof` and NVIDIA Visual Profiler

MEMORY CHECKING WITH CUDA-MEMCHECK

- Cuda-memcheck is a functional correctness checking suite similar to the valgrind memcheck tool
- Can be used in a MPI environment

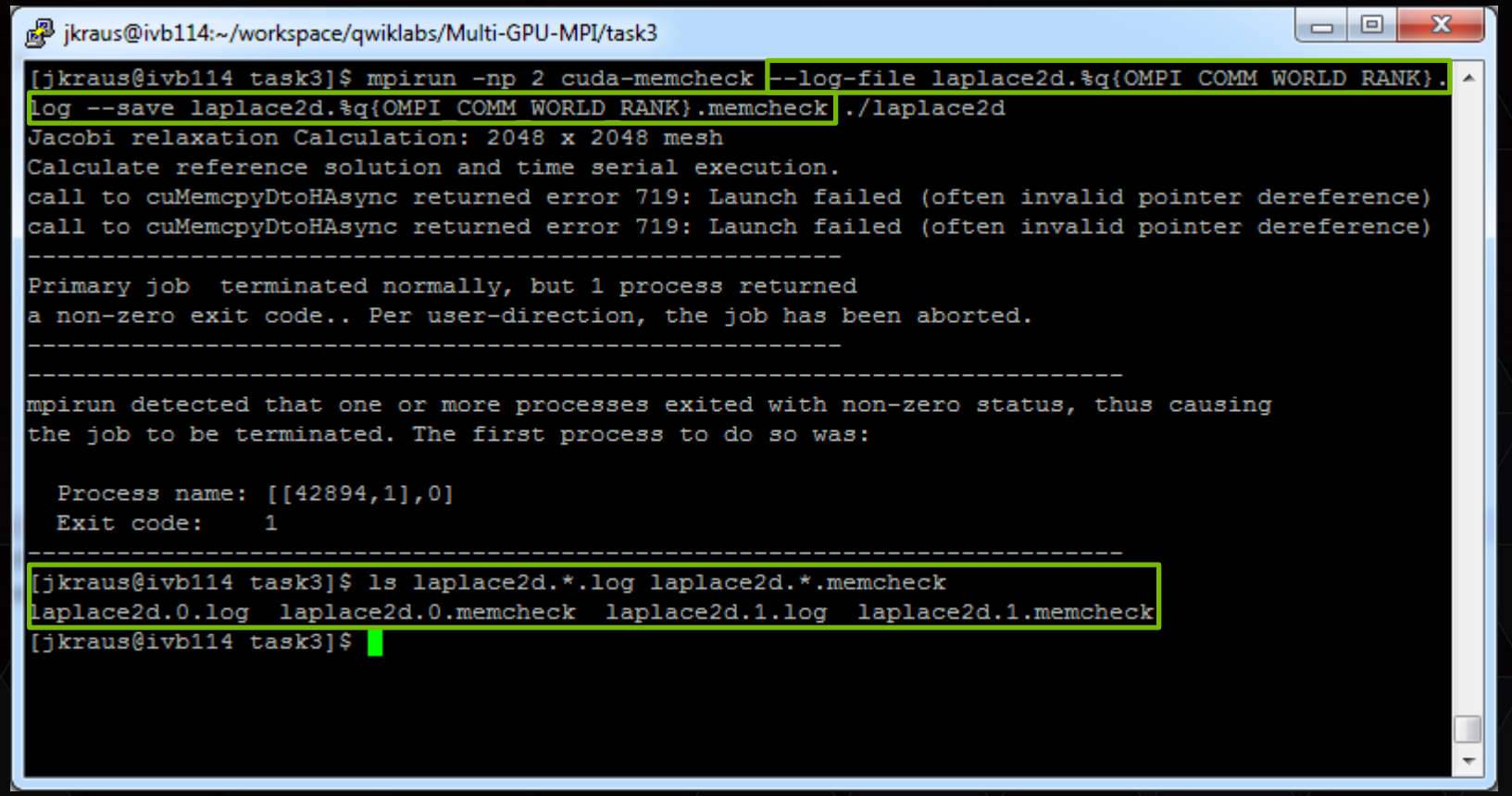
```
mpiexec -np 2 cuda-memcheck ./myapp <args>
```

- Problem: output of different processes is interleaved
  - Use save or log-file command line options

```
mpirun -np 2 cuda-memcheck
  --log-file name.%q{OMPI_COMM_WORLD_RANK}.log
  --save name.%q{OMPI_COMM_WORLD_RANK}.memcheck
 ./myapp <args>
```

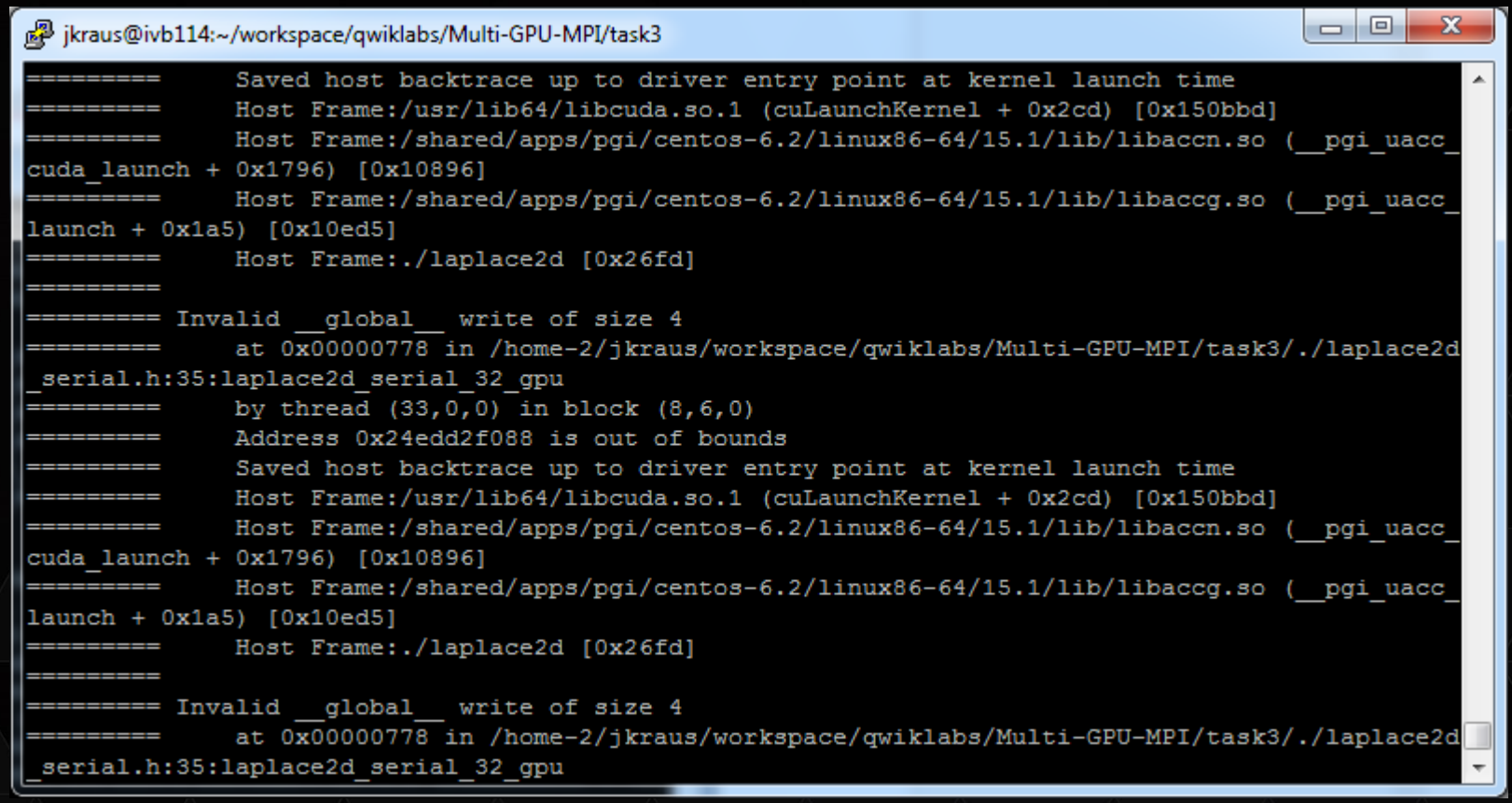
  - OpenMPI: `OMPI_COMM_WORLD_RANK`
  - MVAPICH2: `MV2_COMM_WORLD_RANK`

MEMORY CHECKING WITH CUDA-MEMCHECK



MEMORY CHECKING WITH CUDA-MEMCHECK

Read outputfiles with `cuda-memcheck --read`

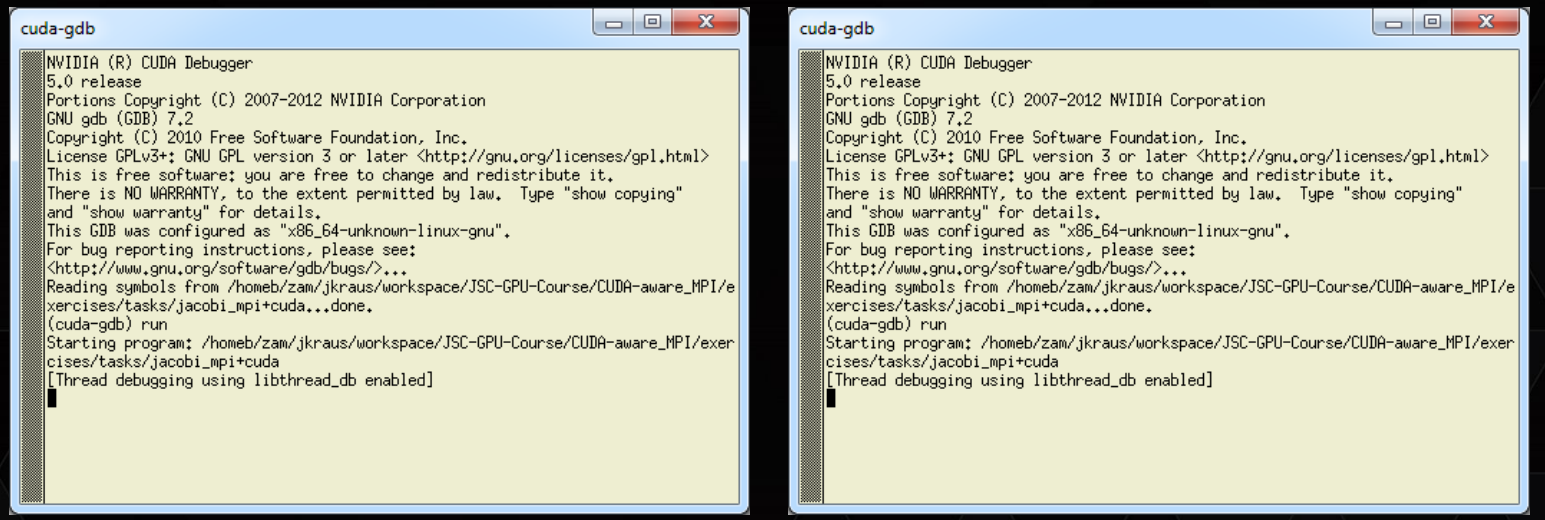


DEBUGGING MPI+CUDA APPLICATIONS

USING CUDA-GDB WITH MPI APPLICATIONS

- You can use cuda-gdb just like gdb with the same tricks
- For smaller applications, just launch xterms and cuda-gdb

```
> mpiexec -x -np 2 xterm -e cuda-gdb ./myapp <args>
```



DEBUGGING MPI+CUDA APPLICATIONS

CUDA-GDB ATTACH

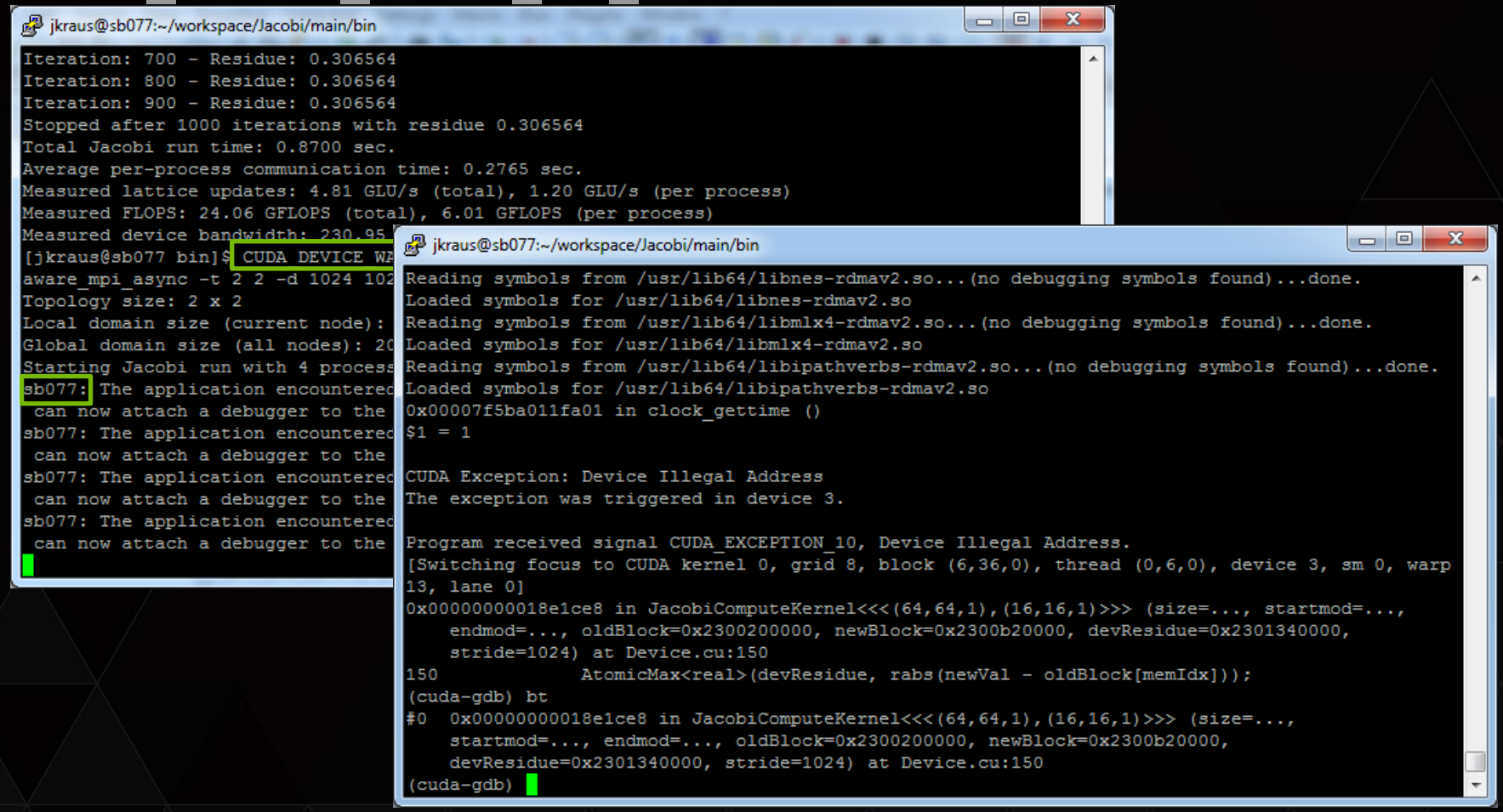
- CUDA 5.0 and forward have the ability to attach to a running process

```
if ( rank == 0 )  {
    int i=0;
    printf("rank %d: pid %d on %s ready for attach\n.", rank, getpid(),name);
    while (0 == i) {
        sleep(5);
    }
}
```

```
> mpiexec -np 2 ./jacobi_mpi+cuda
Jacobi relaxation Calculation: 4096 x 4096 mesh with 2 processes and one Tesla M2070 for each process (2049 rows per process).
rank 0: pid 30034 on judge107 ready for attach
> ssh judge107
jkraus@judge107:~> cuda-gdb --pid 30034
```

DEBUGGING MPI+CUDA APPLICATIONS

CUDA_DEVICE_WAITS_ON_EXCEPTION



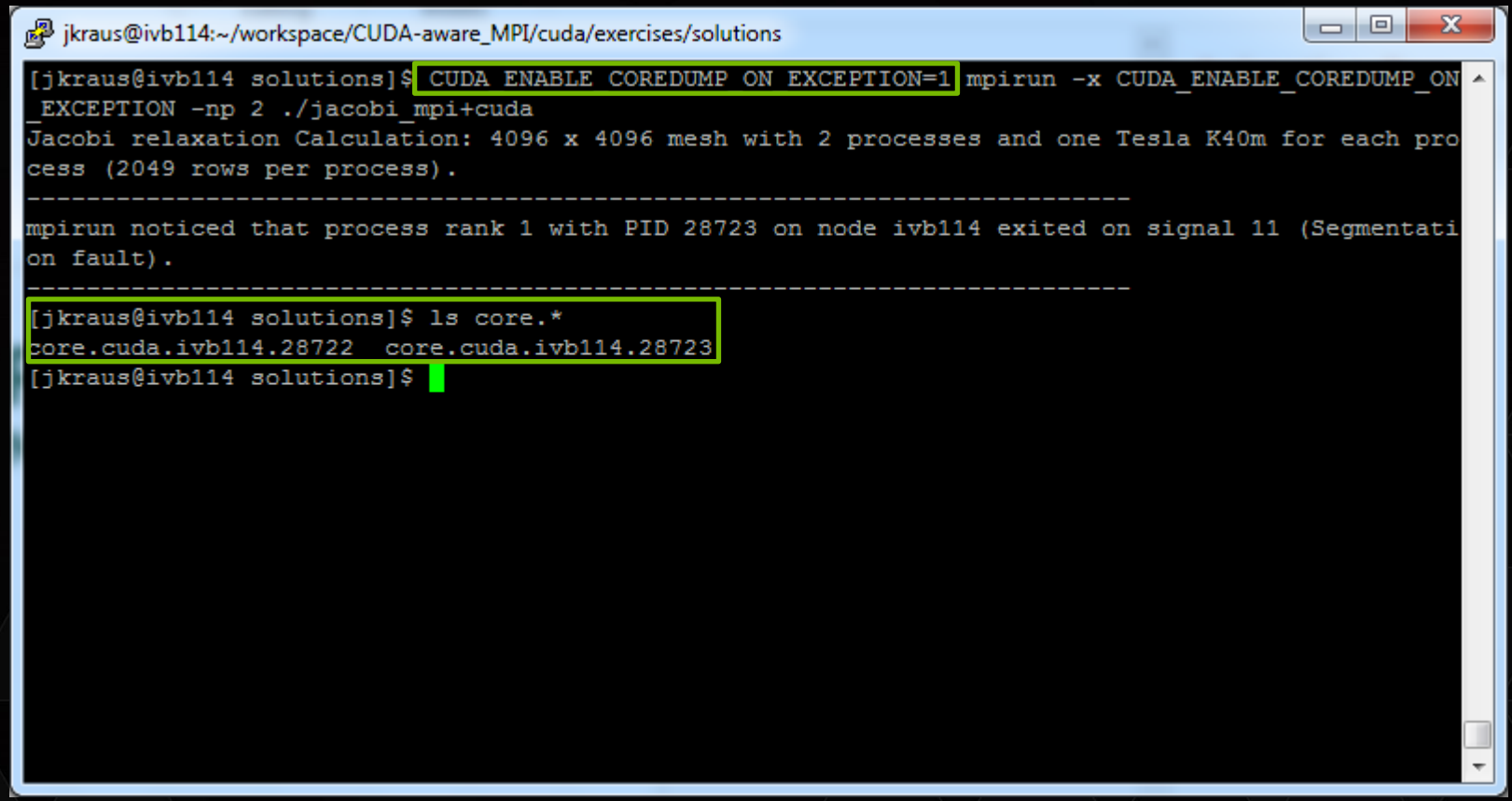
DEBUGGING MPI+CUDA APPLICATIONS

- With CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1 core dumps are generated in case of an exception:
  - Can be used for offline debugging
  - Helpful if live debugging is not possible, e.g. too many nodes needed to reproduce
- CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION
  - Enable/Disable CPU part of core dump (enabled by default)
- CUDA_COREDUMP_FILE
  - Specify name of core dump file
- Open GPU: `(cuda-gdb) target cudacore core.cuda`
- Open CPU+GPU: `(cuda-gdb) target core core.cpu core.cuda`

New with CUDA 7

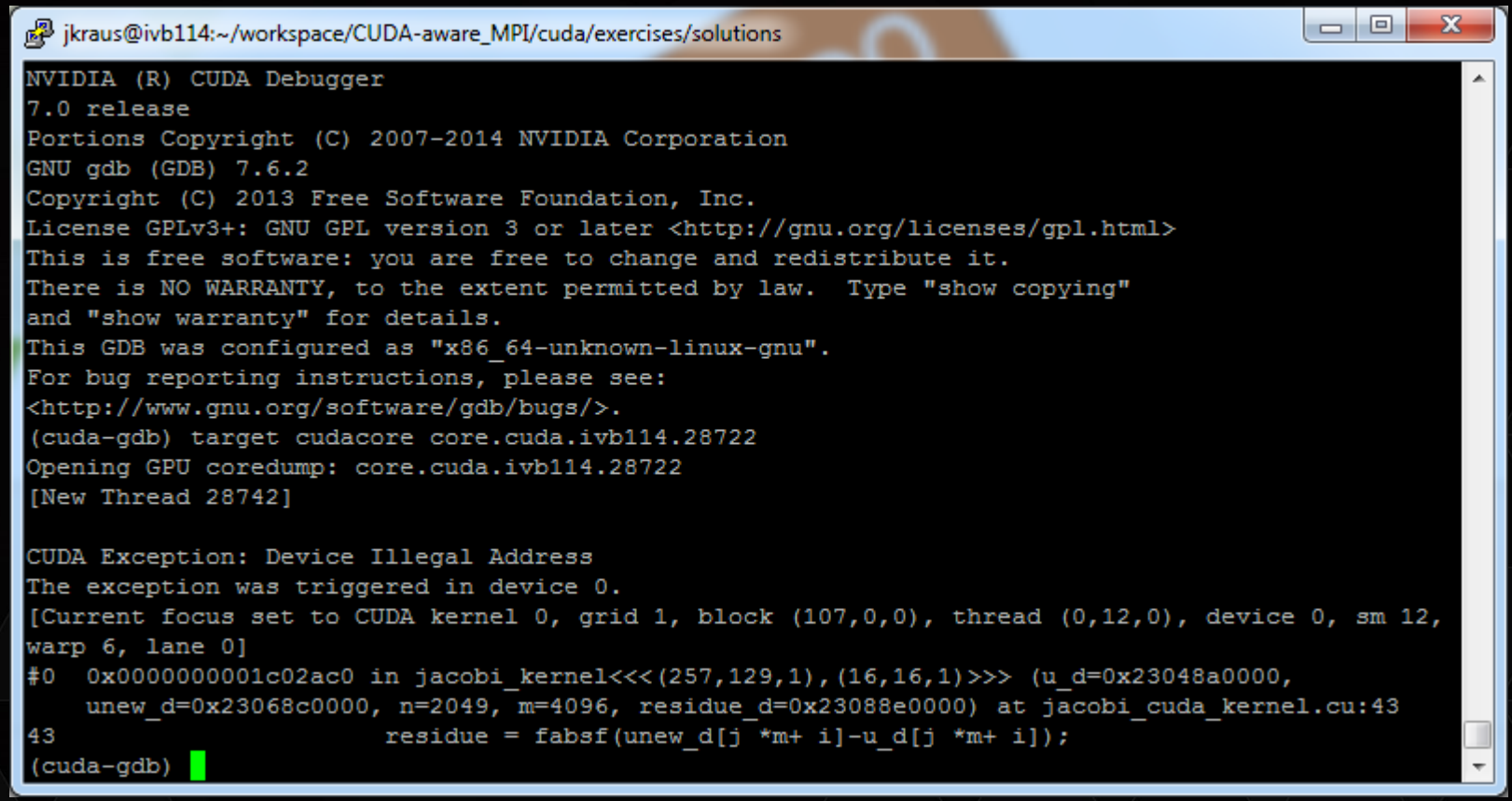
DEBUGGING MPI+CUDA APPLICATIONS

CUDA_ENABLE_COREDUMP_ON_EXCEPTION



DEBUGGING MPI+CUDA APPLICATIONS

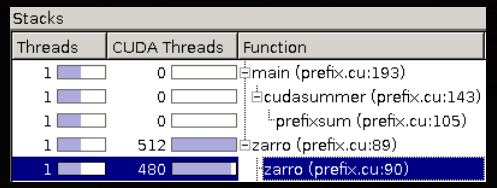
CUDA_ENABLE_COREDUMP_ON_EXCEPTION

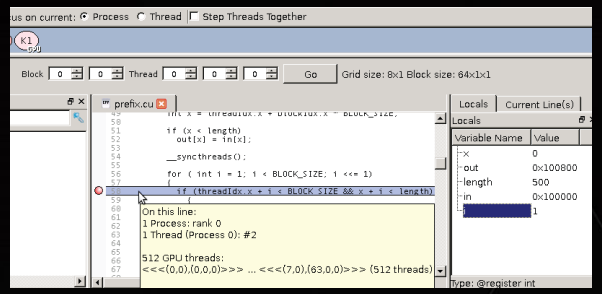


DEBUGGING MPI+CUDA APPLICATIONS

THIRD PARTY TOOLS

- Allinea DDT debugger
- Totalview
- S5417 - Three Ways to Debug Parallel CUDA Applications: Interactive, Batch, and Corefile – (Thursday 03/19, 17:00 – 17:25, Room 212A)





PROFILING MPI+CUDA APPLICATIONS

USING NVPROF+NVVP

3 Usage modes:

- Embed MPI rank in output filename

```
mpirun -np 2 nvprof --output-profile
profile.out.%q{OMPI_COMM_WORLD_RANK}
```

- Only save the textual output

```
mpirun -np 2 nvprof --log-file profile.out.
%q{OMPI_COMM_WORLD_RANK}
```

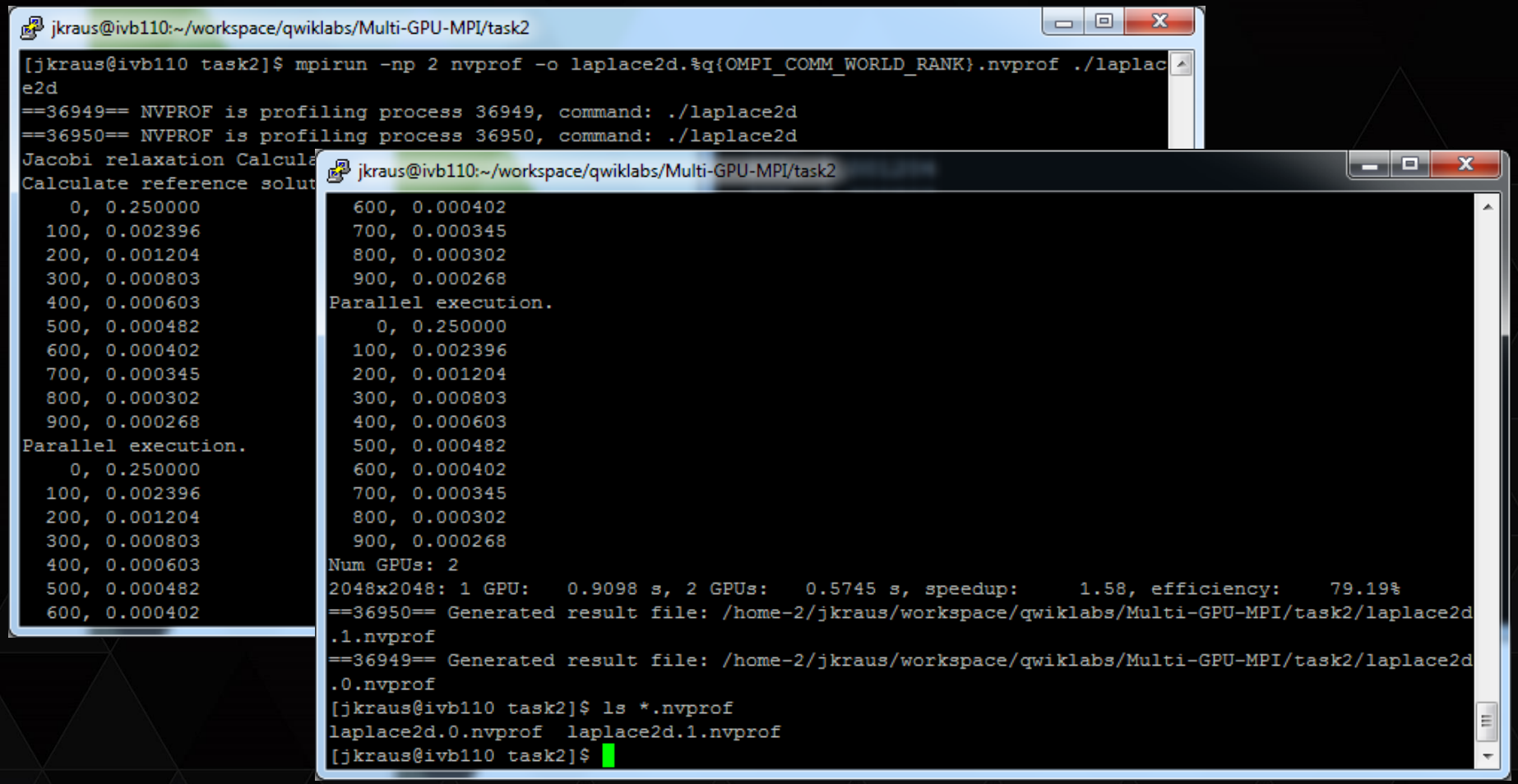
- Collect profile data on all processes that run on a node

```
nvprof --profile-all-processes -o profile.out.%h.%p
```

- OpenMPI: `OMPI_COMM_WORLD_RANK`
- MVAPICH2: `MV2_COMM_WORLD_RANK`

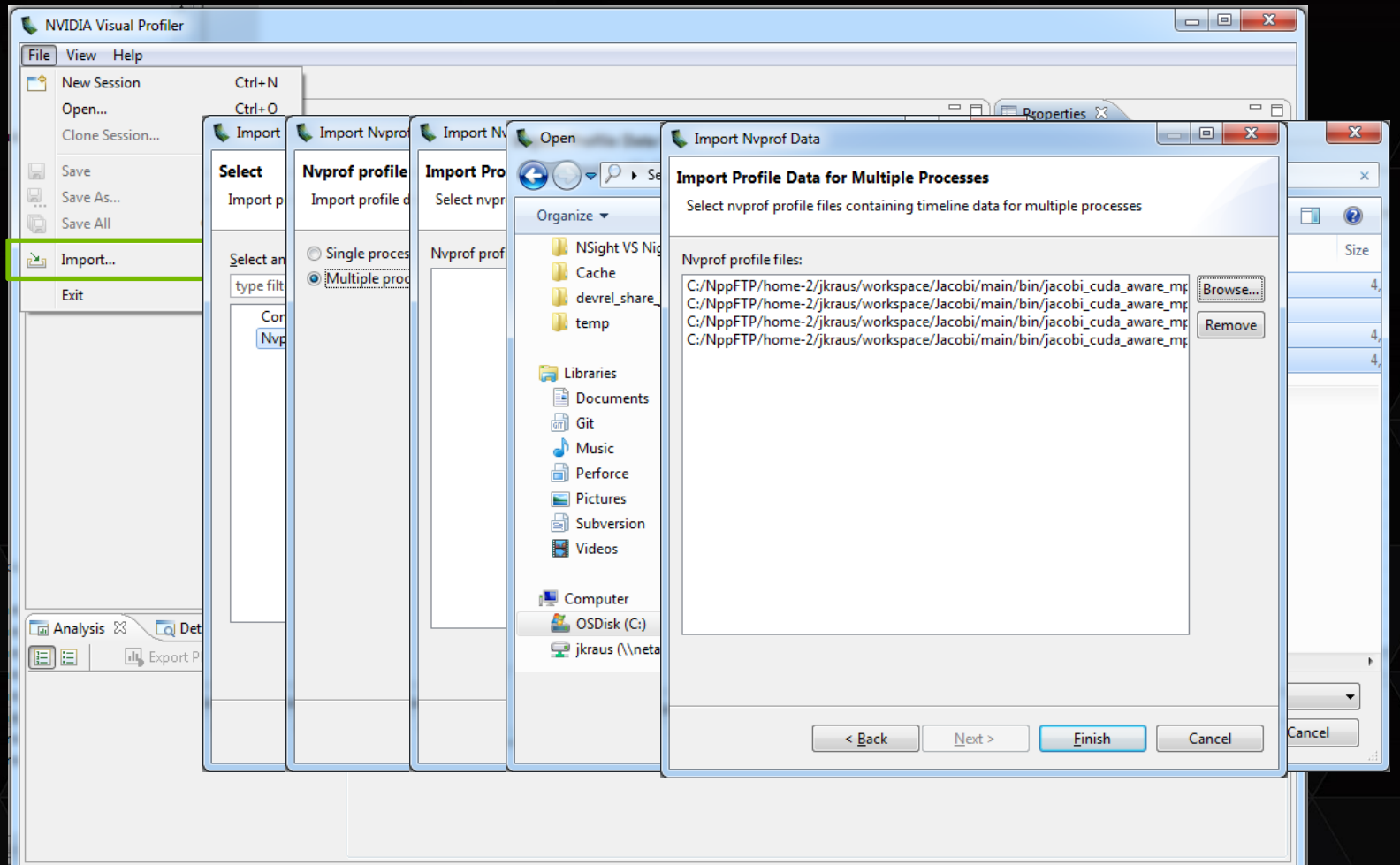
PROFILING MPI+CUDA APPLICATIONS

USING NVPROF+NVVP



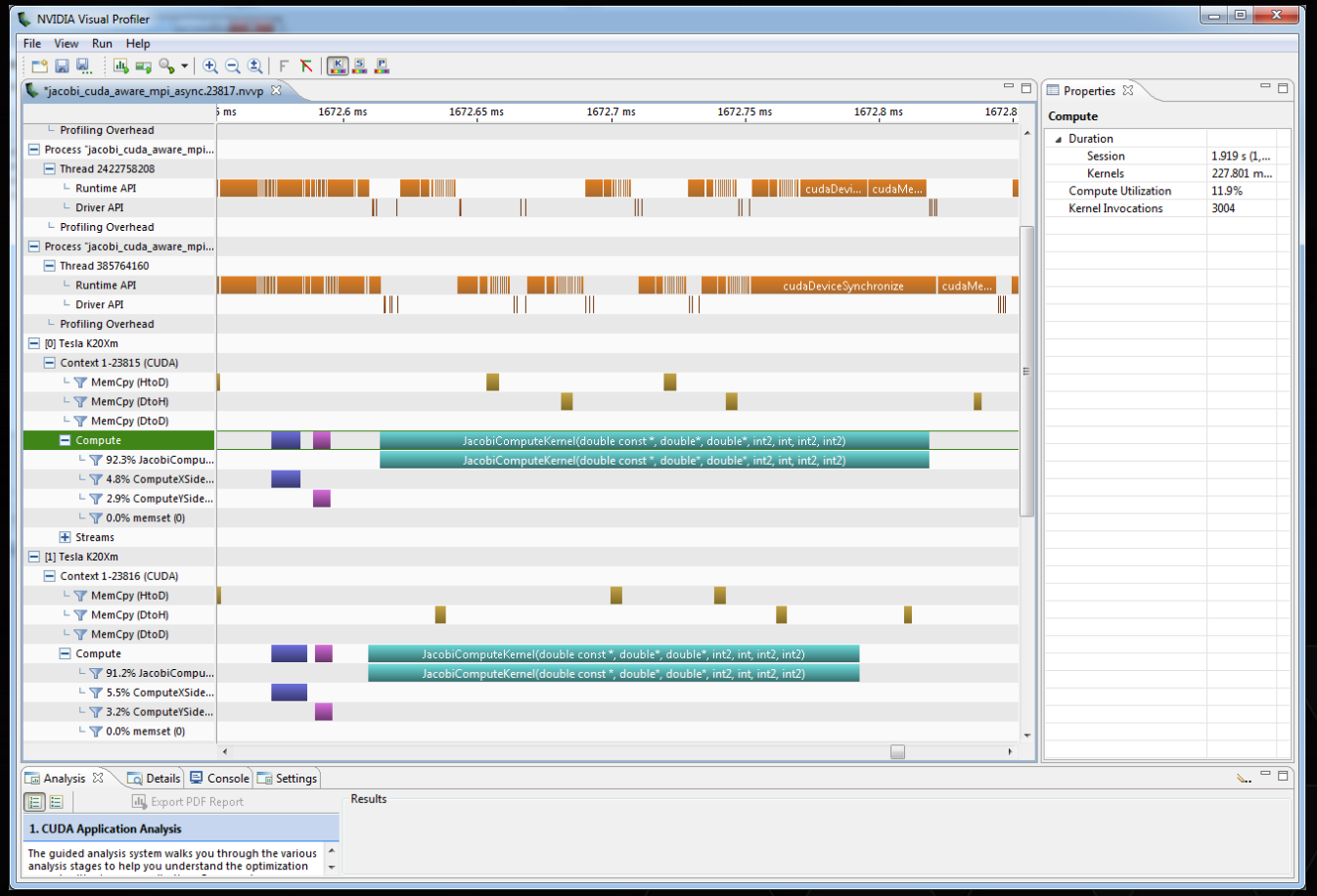
PROFILING MPI+CUDA APPLICATIONS

USING NVPROF+NVVP



PROFILING MPI+CUDA APPLICATIONS

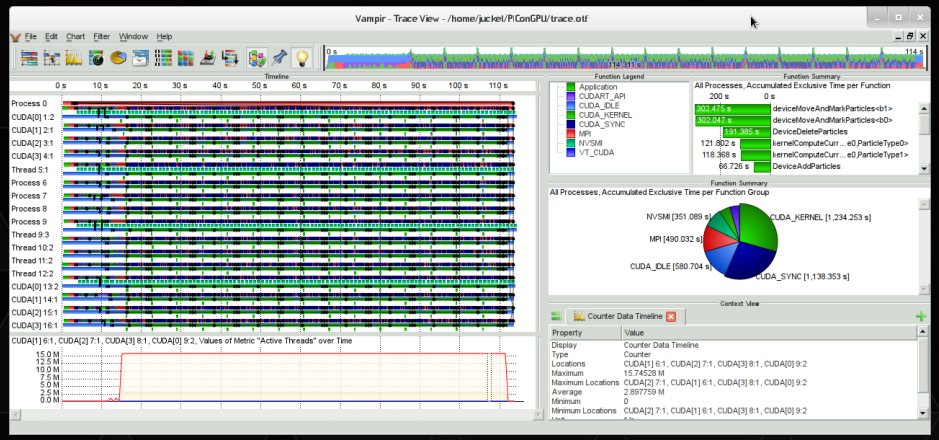
USING NVPROF+NVVP



PROFILING MPI+CUDA APPLICATIONS

THIRD PARTY TOOLS

- Multiple parallel profiling tools are CUDA aware
  - Score-P
  - Vampir
  - Tau
- These tools are good for discovering MPI issues as well as basic CUDA performance inhibitors



ADVANCED MPI ON GPUS

BEST PRACTICE: USE NONE-BLOCKING MPI

BLOCKING

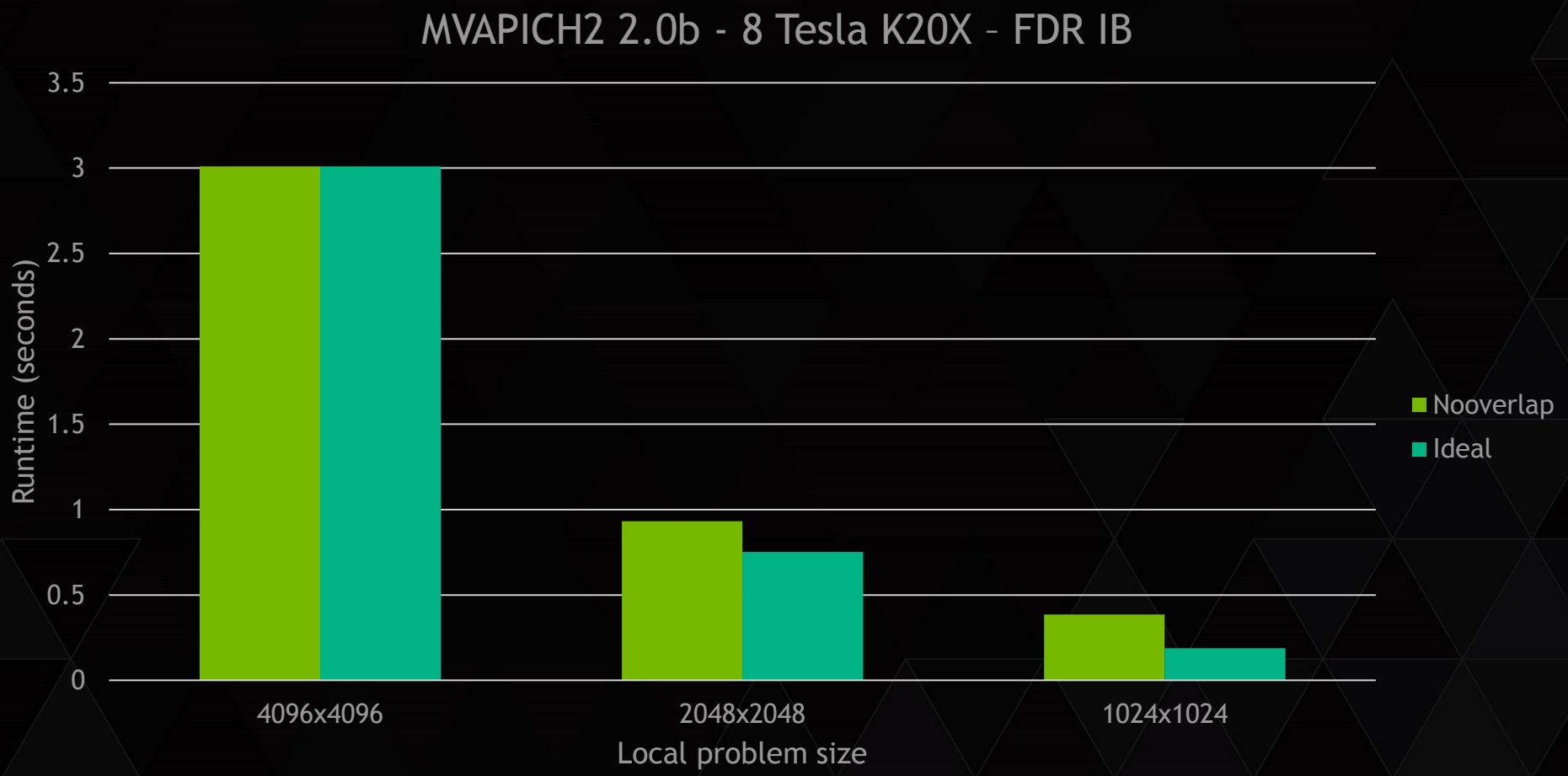
```
#pragma acc host_data use_device ( u_new ) {
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

NONE-BLOCKING

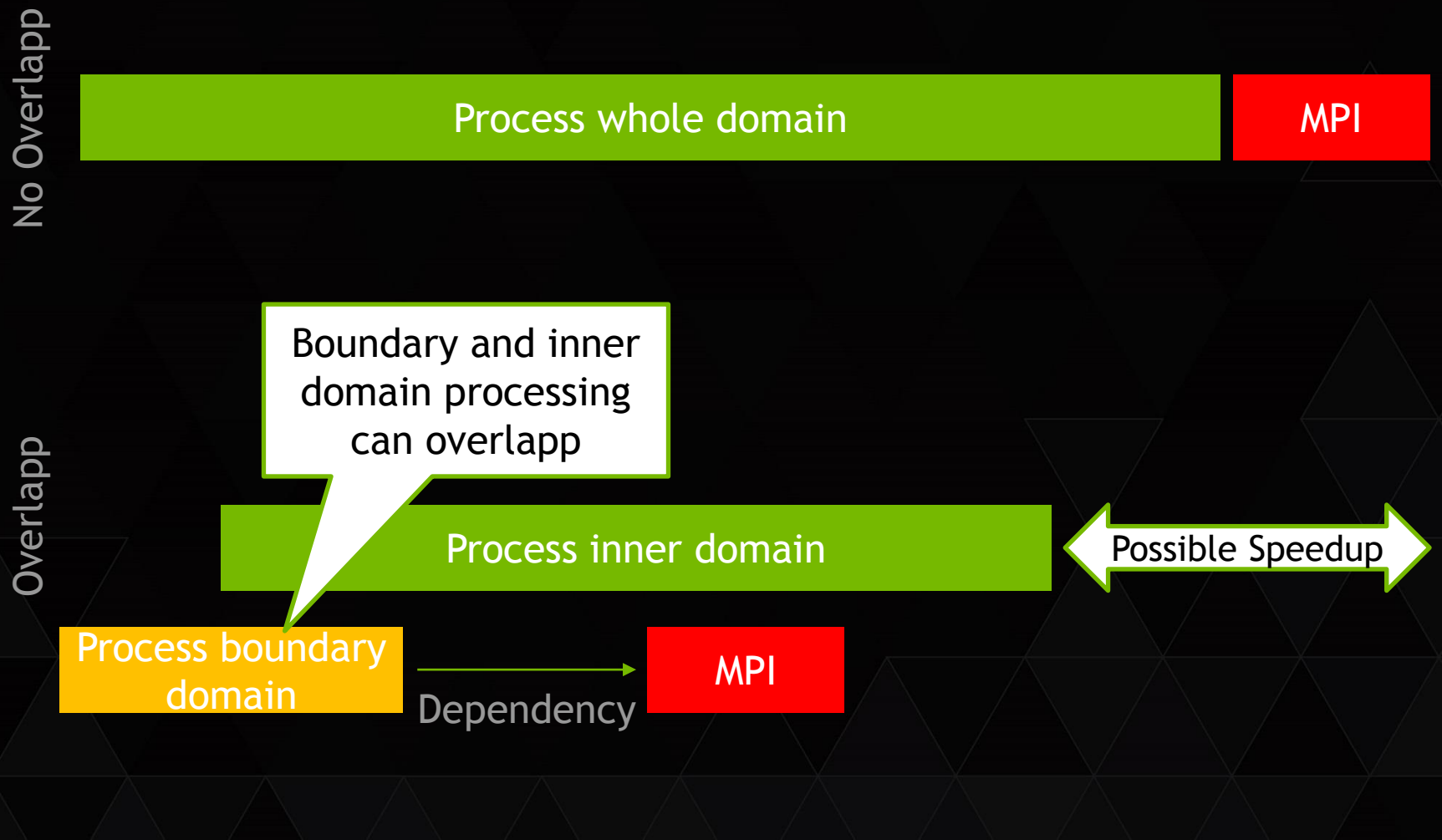
```
MPI_Request t_b_req[4];
#pragma acc host_data use_device ( u_new ) {
  MPI_Irecv(u_new+offset_top_bondary,m-2,M                    _req);
  MPI_Irecv(u_new+offset_bottom_bondar                    LD,t_b_req+1);
  MPI_Isend(u_new+offset_last_row,m-2,M                    WORLD,t_b_req+2);
  MPI_Isend(u_new+offset_first_row,m-2,                    1,MPI_COMM_WORLD,t_b_req+3);
}
MPI_Waitall(4, t_b_req, MPI_STATUSES_IGNORE);
```

Gives MPI more opportunities to build efficient piplines

COMMUNICATION + COMPUTATION OVERLAP



COMMUNICATION + COMPUTATION OVERLAP



COMMUNICATION + COMPUTATION OVERLAP

OpenACC

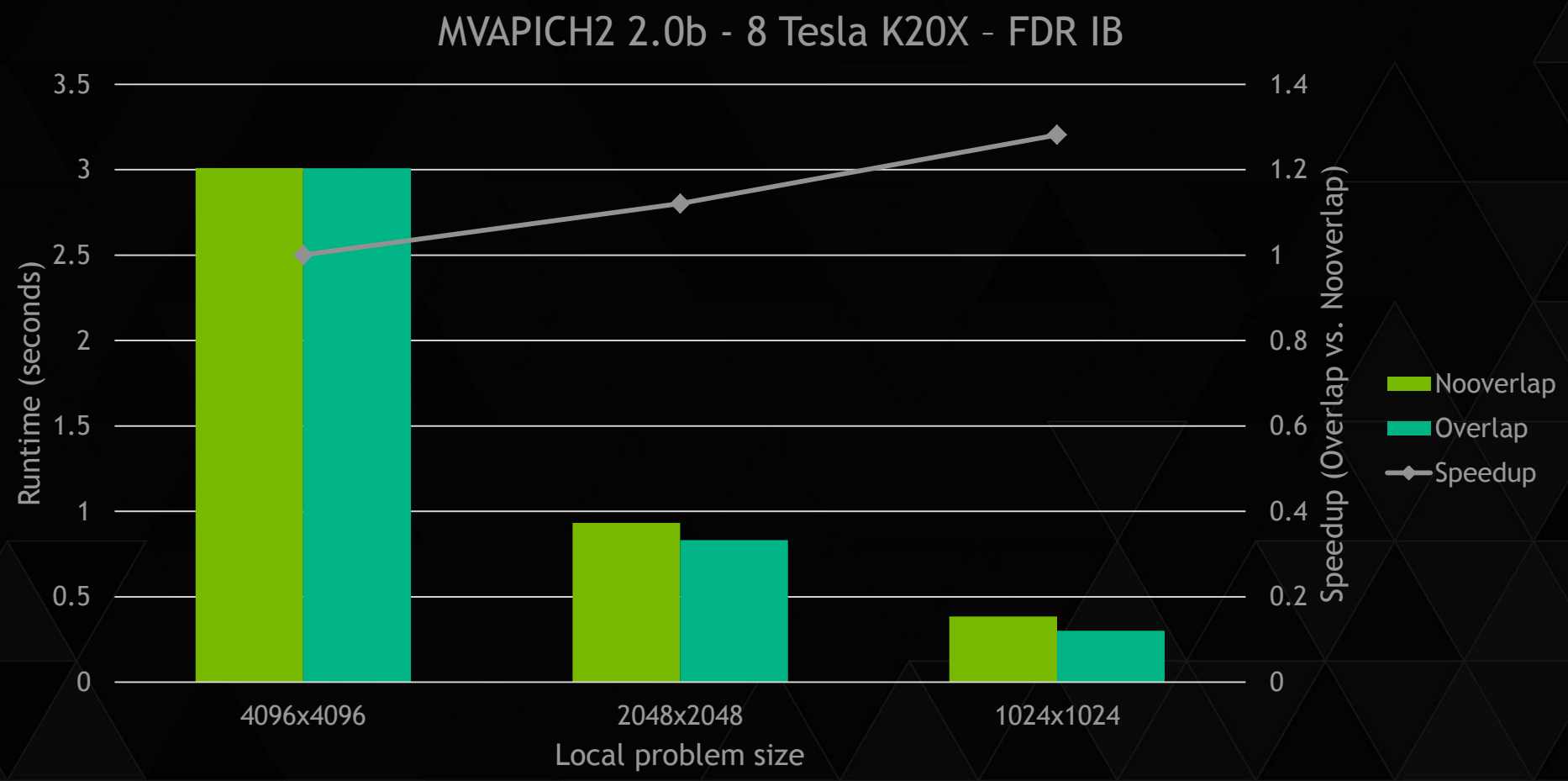
```
#pragma acc parallel loop present ( u_new, u, to_left, to_right ) async(1)
for ( ... )
      //Process boundary and pack to_left and to_right
#pragma acc parallel loop present ( u_new, u ) async(2)
for ( ... )
      //Process inner domain
#pragma acc wait(1)                  //wait for boundary
MPI_Request req[8];
#pragma acc host_data use_device ( from_left, to_left, form_right, to_right, u_new ) {
  //Exchange halo with left, right, top and bottom neighbor
}
MPI_Waitall(8, req, MPI_STATUSES_IGNORE);
#pragma acc parallel loop present ( u_new, from_left, from_right )
for ( ... )
      //unpack from_left and from_right
#pragma acc wait                     //wait for iteration to finish
```

COMMUNICATION + COMPUTATION OVERLAP

CUDA

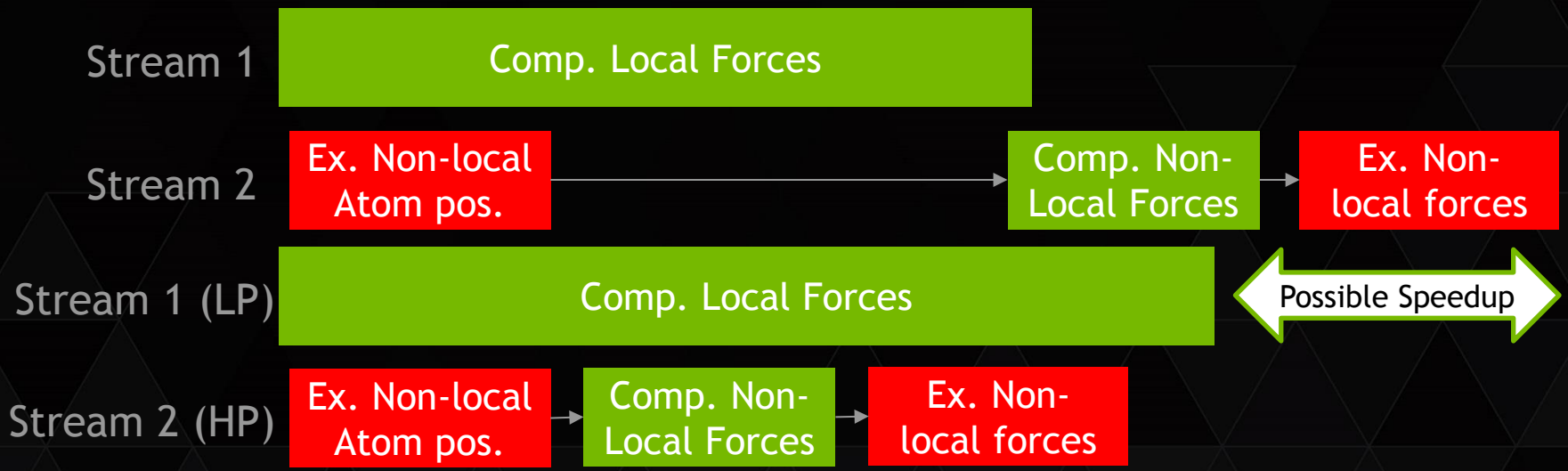
```
process_boundary_and_pack<<<gs_b,bs_b,0,s1>>>(u_new_d,u_d,to_left_d,to_right_d,n,m);


process_inner_domain<<<gs_id,bs_id,0,s2>>>(u_new_d, u_d,to_left_d,to_right_d,n,m);


cudaStreamSynchronize(s1);        //wait for boundary
MPI_Request req[8];

  //Exchange halo with left, right, top and bottom neighbor

MPI_Waitall(8, req, MPI_STATUSES_IGNORE);
unpack<<<gs_s,bs_s>>>(u_new_d, from_left_d, from_right_d, n, m);


cudaDeviceSynchronize();          //wait for iteration to finish
```

COMMUNICATION + COMPUTATION OVERLAP



HIGH PRIORITY STREAMS

- Improve scalability with high priority streams
  - cudaStreamCreateWithPriority
- Use-case: MD Simulations



MPI AND UNIFIED MEMORY

- Unified Memory support for CUDA-aware MPI needs explicit support from the MPI implementation:
  - Check with your MPI implementation of choice for their support
  - OpenMPI 1.8.5 supports unified memory
- Unified Memory and regular MPI
  - Require unmanaged staging buffer
    - Regular MPI has no knowledge of managed memory
    - CUDA 6 managed memory does not play well with RDMA protocols

COMMUNICATION + COMPUTATION OVERLAP

TIPS AND TRICKS

- CUDA-aware MPI might use the default stream
  - Allocate stream with the non-blocking flag (cudaStreamNonBlocking)
- In case of multiple kernels for boundary handling the kernel processing the inner domain might sneak in
  - Use single stream or events for inter stream dependencies via cudaStreamWaitEvent (#pragma acc wait async) – disables overlapping of boundary and inner domain kernels
  - Use high priority streams for boundary handling kernels – allows overlapping of boundary and inner domain kernels

HANDLING MULTI GPU NODES

- Multi GPU nodes and GPU-affinity:
  - Use local rank:

```
int local_rank = //determine local rank
int num_devices = 0;
cudaGetDeviceCount(&num_devices);
cudaSetDevice(local_rank % num_devices);
```

  - Use exclusive process mode

HANDLING MULTI GPU NODES

- How to determine local rank:
  - Rely on process placement (with one rank per GPU)

```
int rank = 0;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
int num_devices = 0;
cudaGetDeviceCount(&num_devices); // num_devices == ranks per node
int local_rank = rank % num_devices;
```

  - Use environment variables provided by MPI launcher
    - e.g for OpenMPI

```
int local_rank = atoi(getenv("OMPI_COMM_WORLD_LOCAL_RANK"));
```

    - e.g. For MVPAICH2

```
int local_rank = atoi(getenv("MV2_COMM_WORLD_LOCAL_RANK"));
```

CONCLUSIONS

- Using MPI as abstraction layer for Multi GPU programming allows multi GPU programs to scale beyond a single node
  - CUDA-aware MPI delivers ease of use, reduced network latency and increased bandwidth
- All NVIDIA tools are usable and third party tools are available
- Multipe CUDA-aware MPI implementations available
  - OpenMPI, MVAPICH2, Cray, IBM Platform MPI

CONCLUSIONS

- S5461 - Latest Advances in MVAPICH2 MPI Library for NVIDIA GPU Clusters with InfiniBand – (Thursday 03/19, 17:00 - 17:50, Room 212B)
- S5146 - Data Movement Options for Scalable GPU Cluster Communication – (Thursday 03/19, 14:30 – 14:55, Room 210D)
- S5417 - Three Ways to Debug Parallel CUDA Applications: Interactive, Batch, and Corefile – (Thursday 03/19, 17:00 – 17:25, Room 212A)
- S5470 - Enabling Efficient Use of UPC and OpenSHMEM PGAS Models on GPU Clusters – (Thursday 03/19, 10:00 - 10:25, Room 212B)
- S5426 - Lesson Learned Using GPU Direct over RDMA on Production Heterogeneous Clusters – (Friday 03/20, 09:00 - 09:25, Room 212B)
- S5169 - Maximizing Scalability Performance in HOOMD-blue by Exploiting GPUDirect® RDMA on Green500 Supercomputer – (Friday 03/20, 10:30 – 10:55, Room 212B)