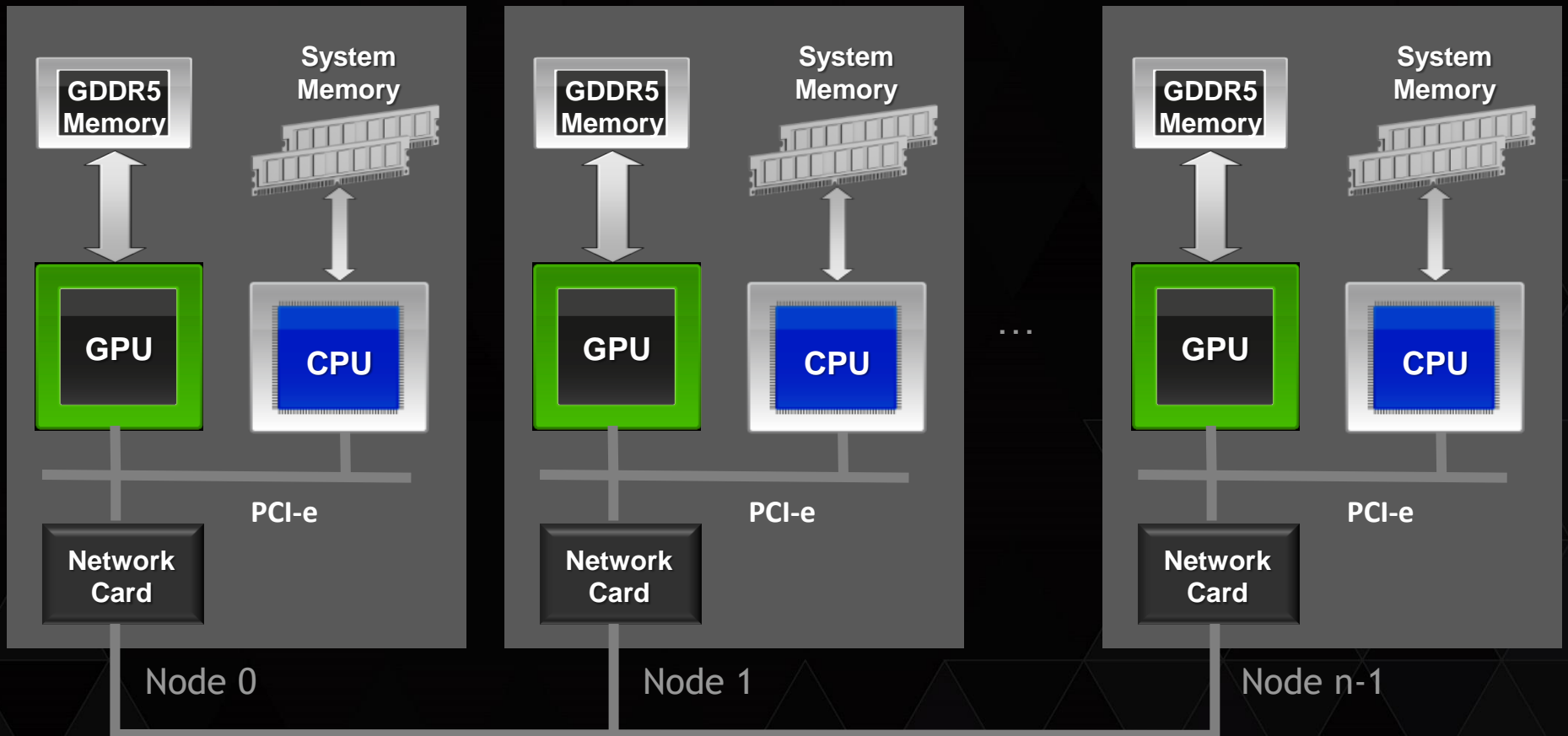


GPU TECHNOLOGY
CONFERENCE

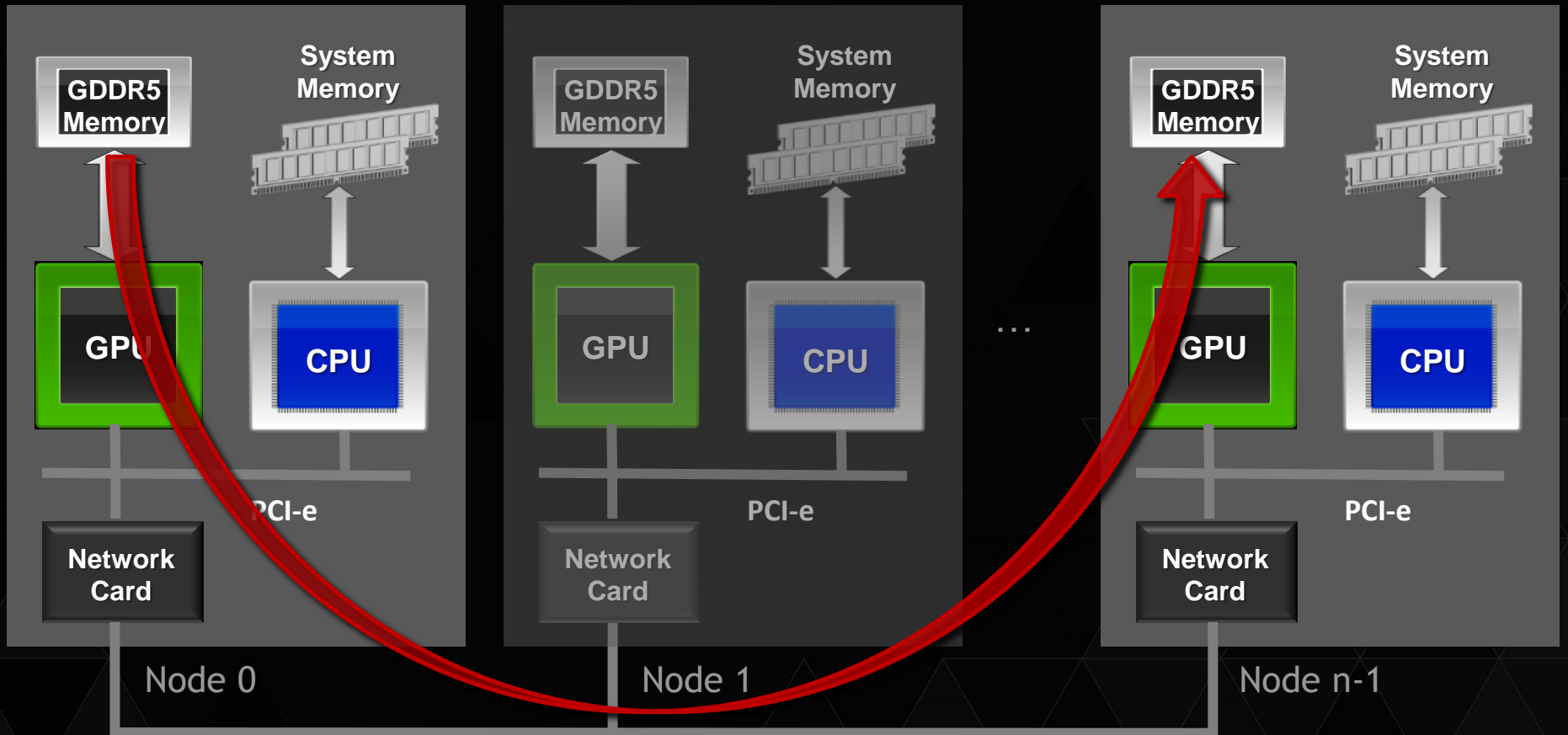
MULTI GPU PROGRAMMING WITH MPI

JIRI KRAUS, NVIDIA

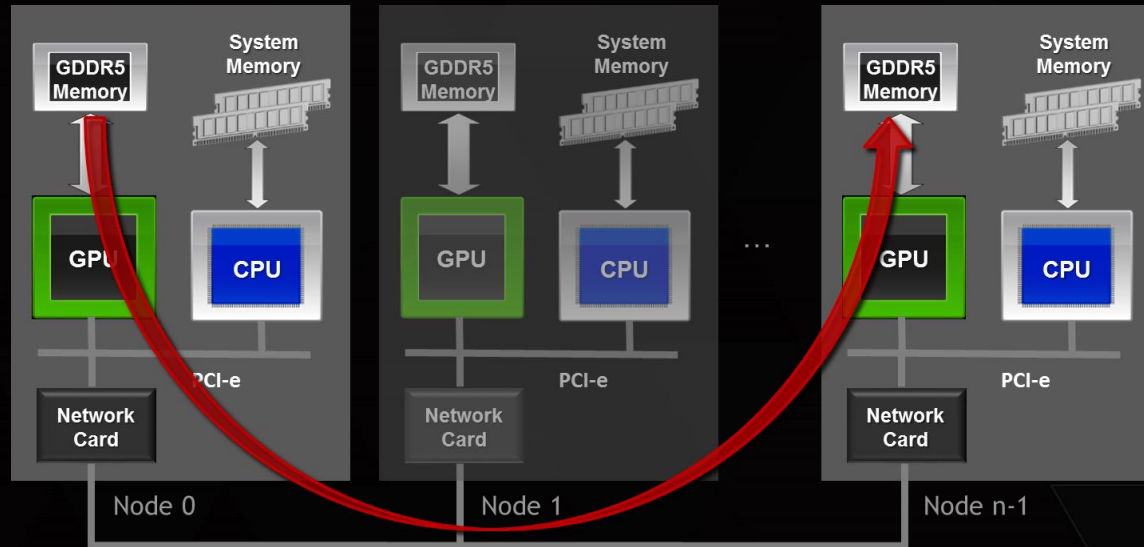
MPI+CUDA



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

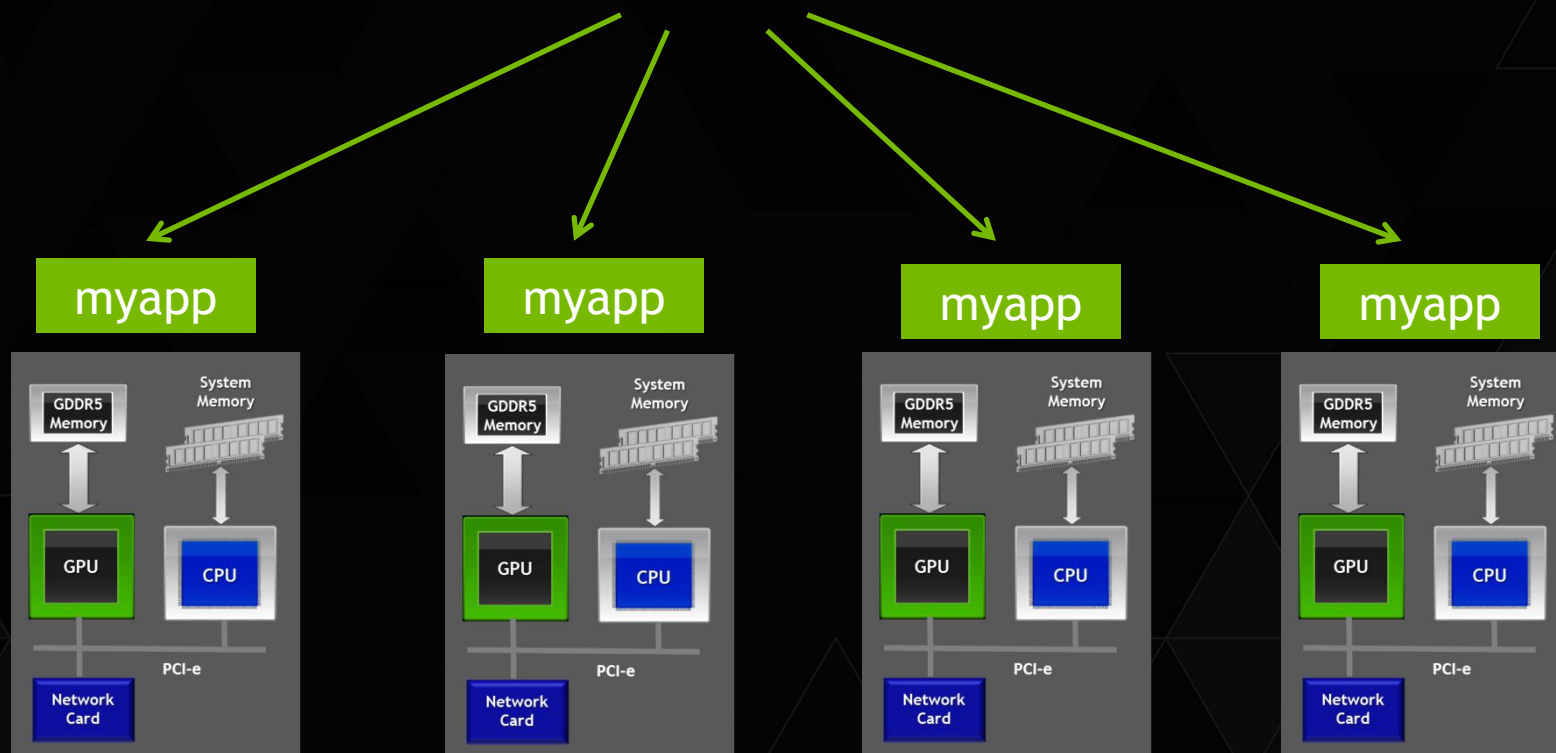
```
#include <mpi.h>

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
```

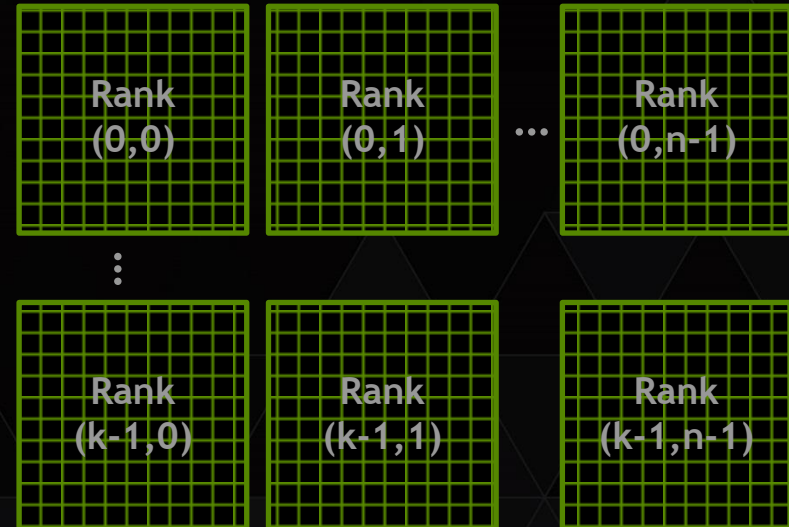
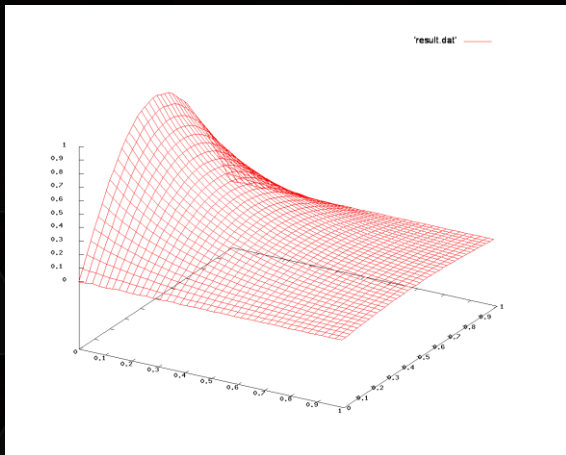
```
$ mpirun -np 4 ./myapp <args>
```



A SIMPLE EXAMPLE

EXAMPLE: JACOBI SOLVER

- ▶ Solves the 2D-Laplace equation on a rectangle
$$\Delta u(x, y) = 0 \quad \forall (x, y) \in \Omega \setminus \delta\Omega$$
 - ▶ Dirichlet boundary conditions (constant values on boundaries)
$$u(x, y) = f(x, y) \in \delta\Omega$$
- ▶ 2D domain decomposition with $n \times k$ domains



EXAMPLE: JACOBI SOLVER - SINGLE GPU

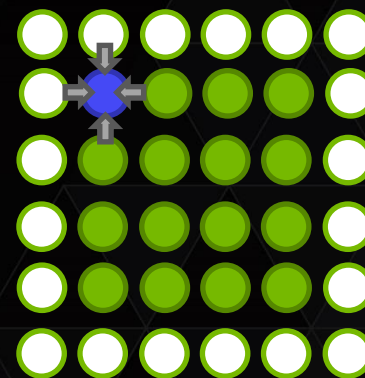
While not converged

▶ 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

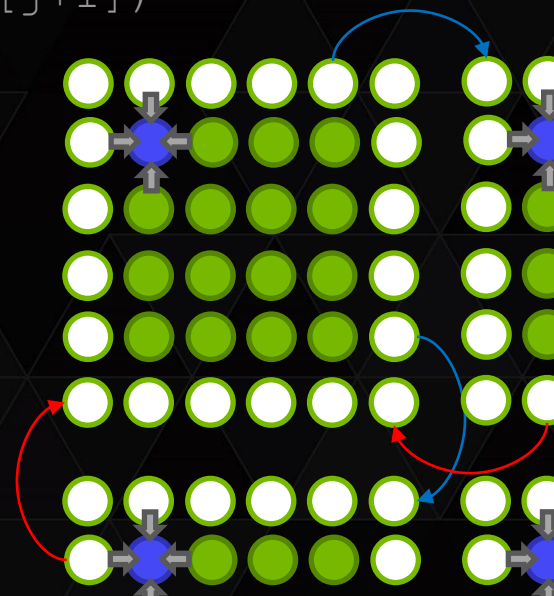
▶ Do Jacobi step:

```
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

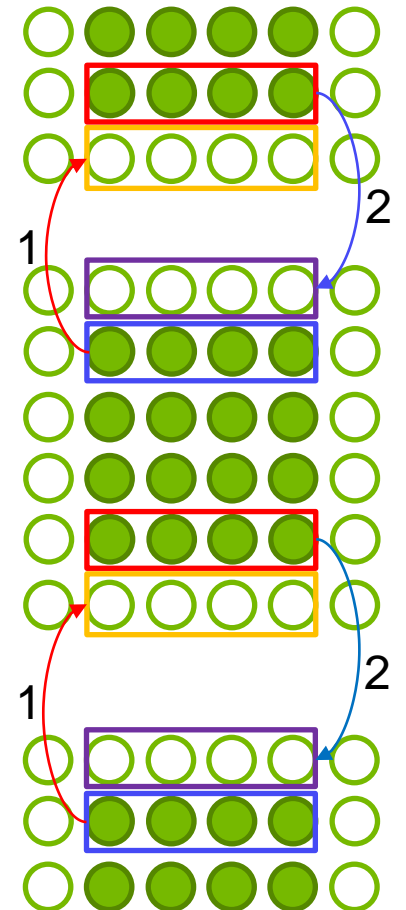
▶ Next iteration



EXAMPLE: JACOBI - TOP/BOTTOM HALO

```
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,  
            u_new+offset_bottom_boundary, 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_boundary, 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 boundary, 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 boundary, m-2, MPI_DOUBLE, t_nb, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

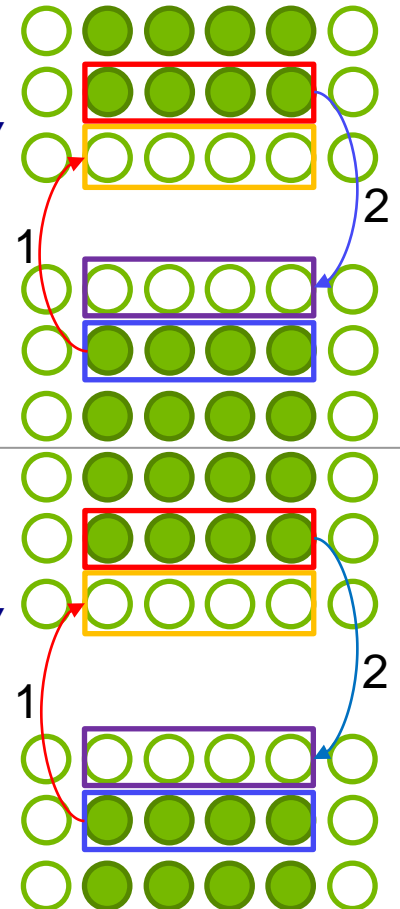
```

CUDA

```

MPI_Sendrecv( u_new_d+offset first row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new_d+offset bottom boundary, 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 boundary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```

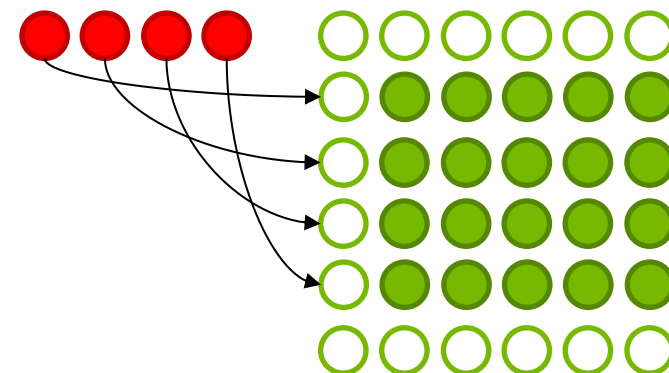
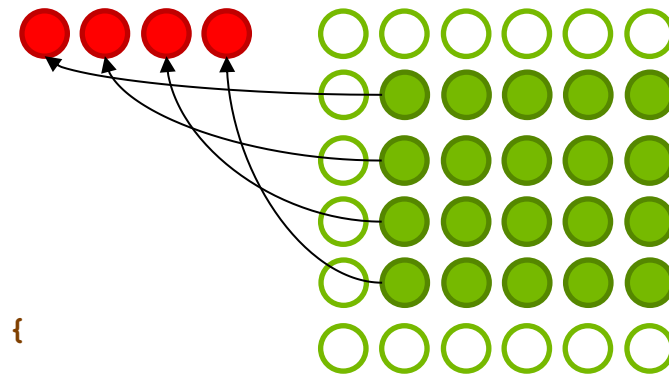


EXAMPLE: JACOBI - LEFT/RIGHT HALO

```
//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];
```

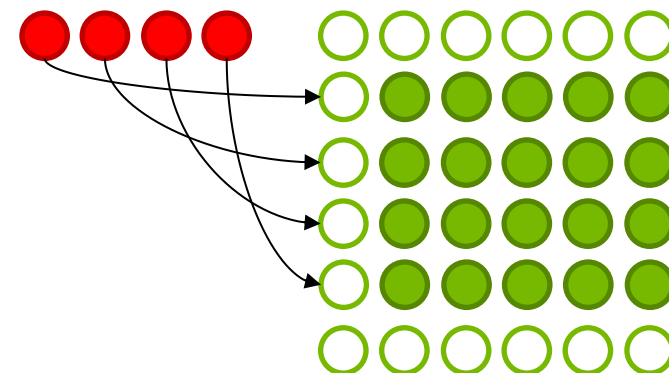
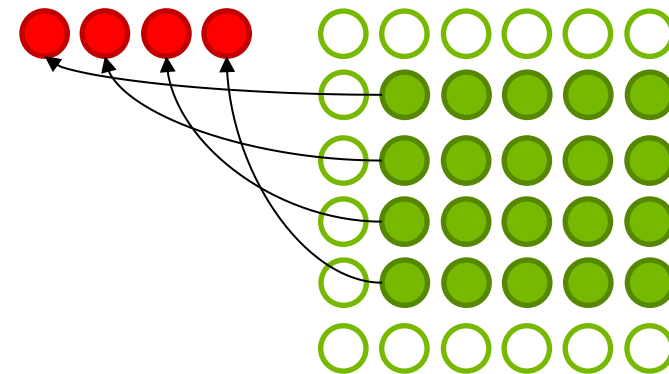


EXAMPLE: JACOBI - LEFT/RIGHT HALO

```
//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>
```

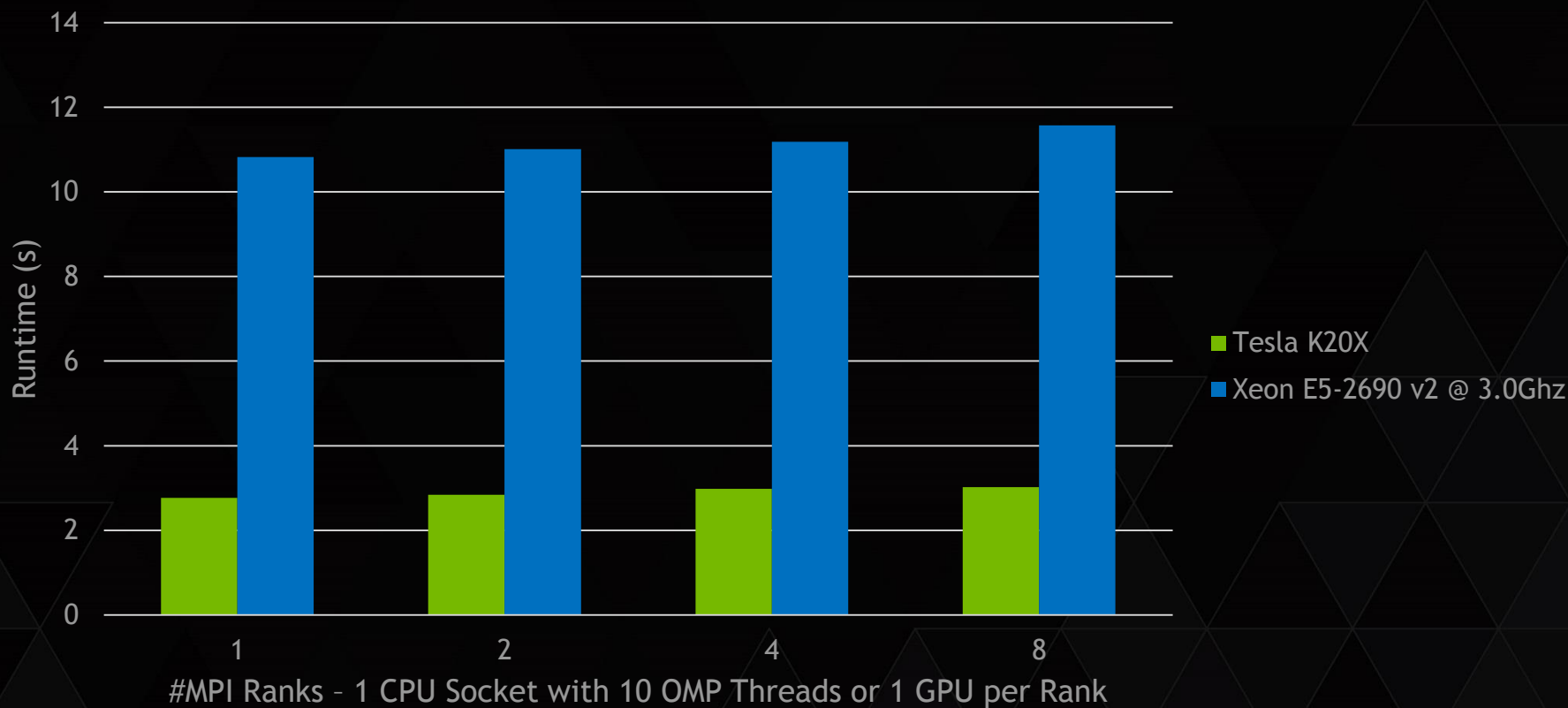
- ▶ **Open MPI:** CUDA-aware features are enabled per default

- ▶ **Cray:** `MPICH_RDMA_ENABLED_CUDA`

- ▶ **IBM Platform MPI:** `PMPI_GPU_AWARE`

# JACOBI RESULTS (1000 STEPS)

MVAPICH2-2.0b FDR IB - weak scaling 4k x 4k per process



# 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_boundary, 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_boundary, 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])
//send to bottom and receive from top - top bottom omitted
```

CUDA

```
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_boundary, 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

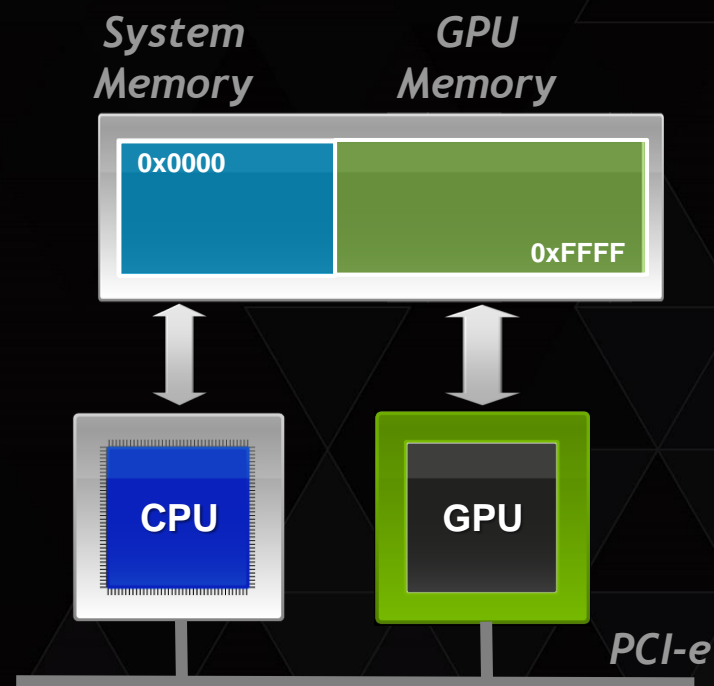
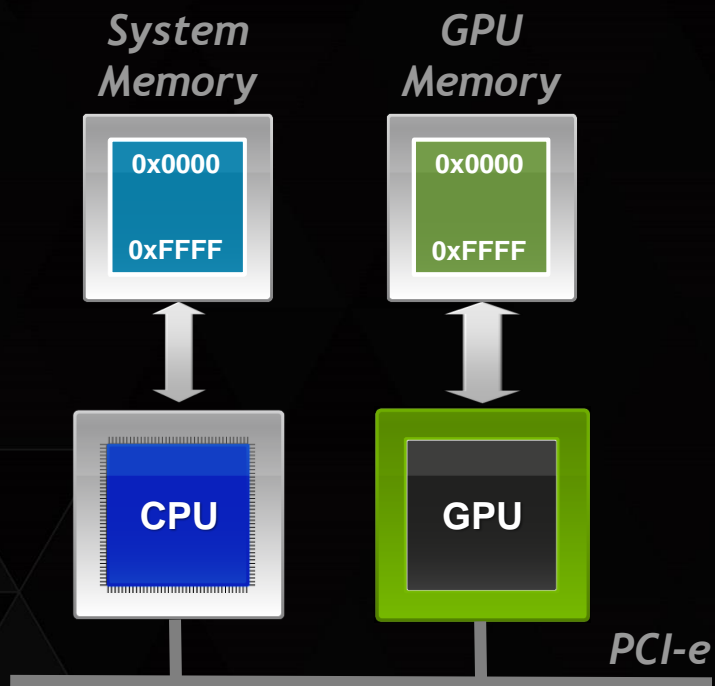
**GPU** TECHNOLOGY  
CONFERENCE

# THE DETAILS

# UNIFIED VIRTUAL ADDRESSING

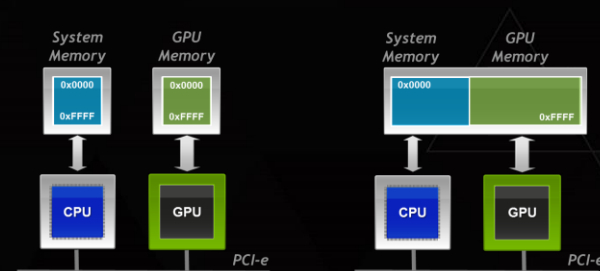
*No UVA : Separate Address Spaces*

*UVA : Single Address Space*



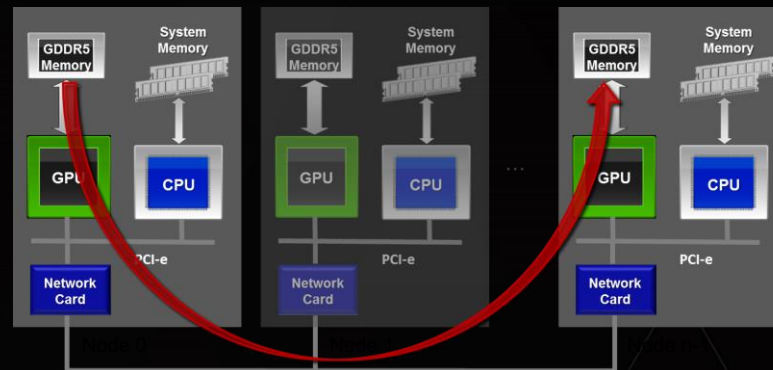
# UNIFIED VIRTUAL ADDRESSING

No UVA : Separate Address Spaces      UVA : Single Address Space



- ▶ 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,...);
```

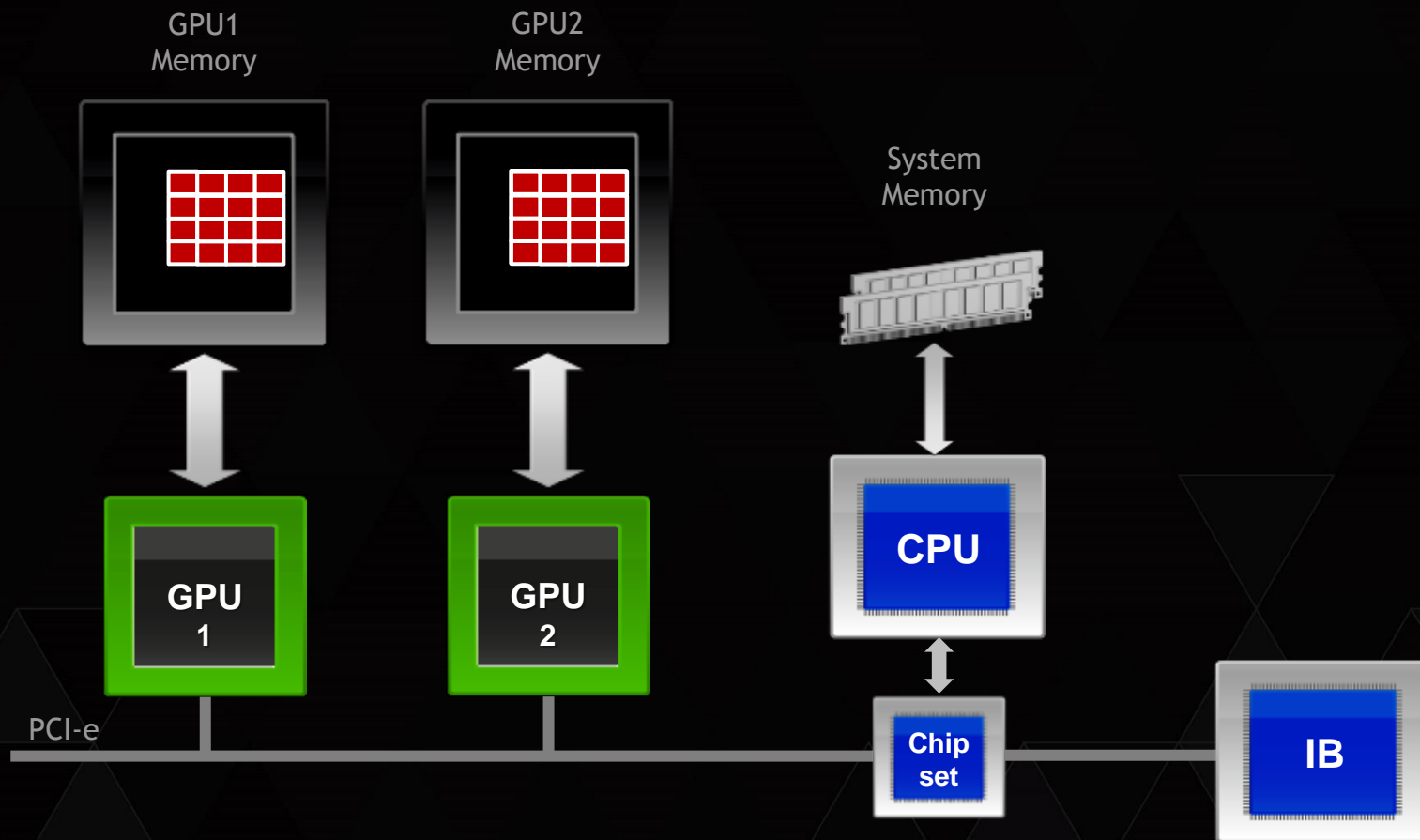
No UVA and regular MPI

```
//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™

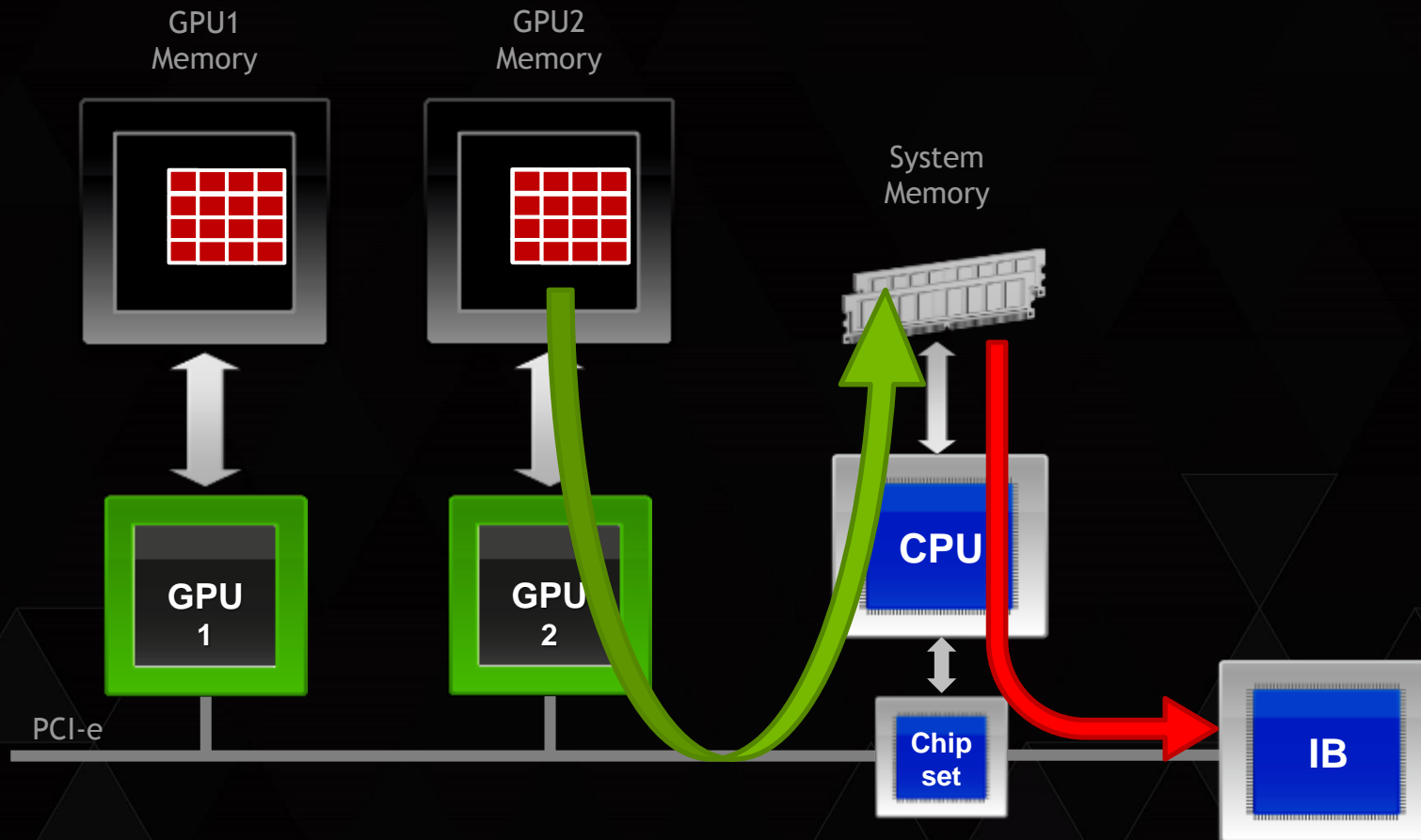
## ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES





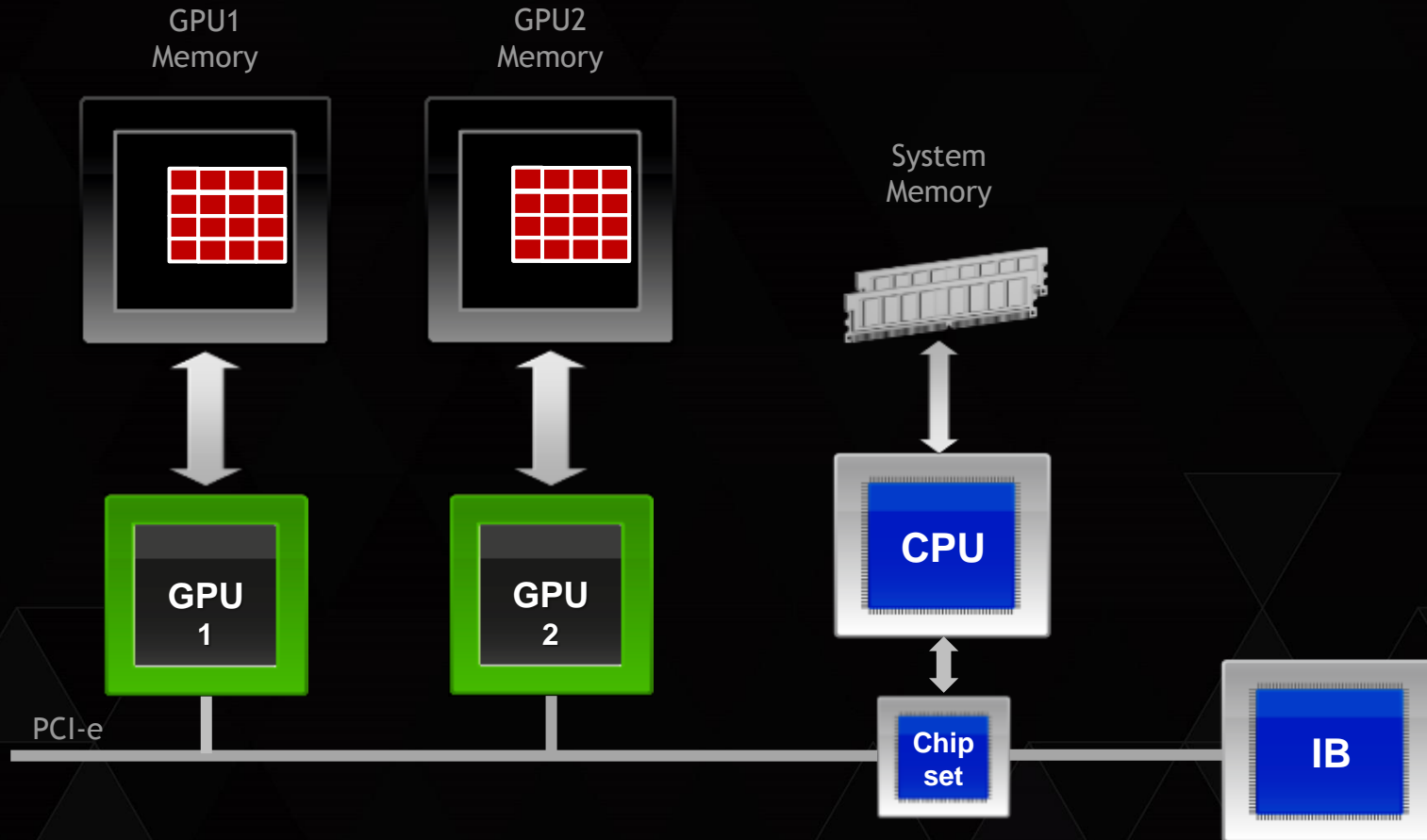
# NVIDIA GPUDIRECT™

## ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES



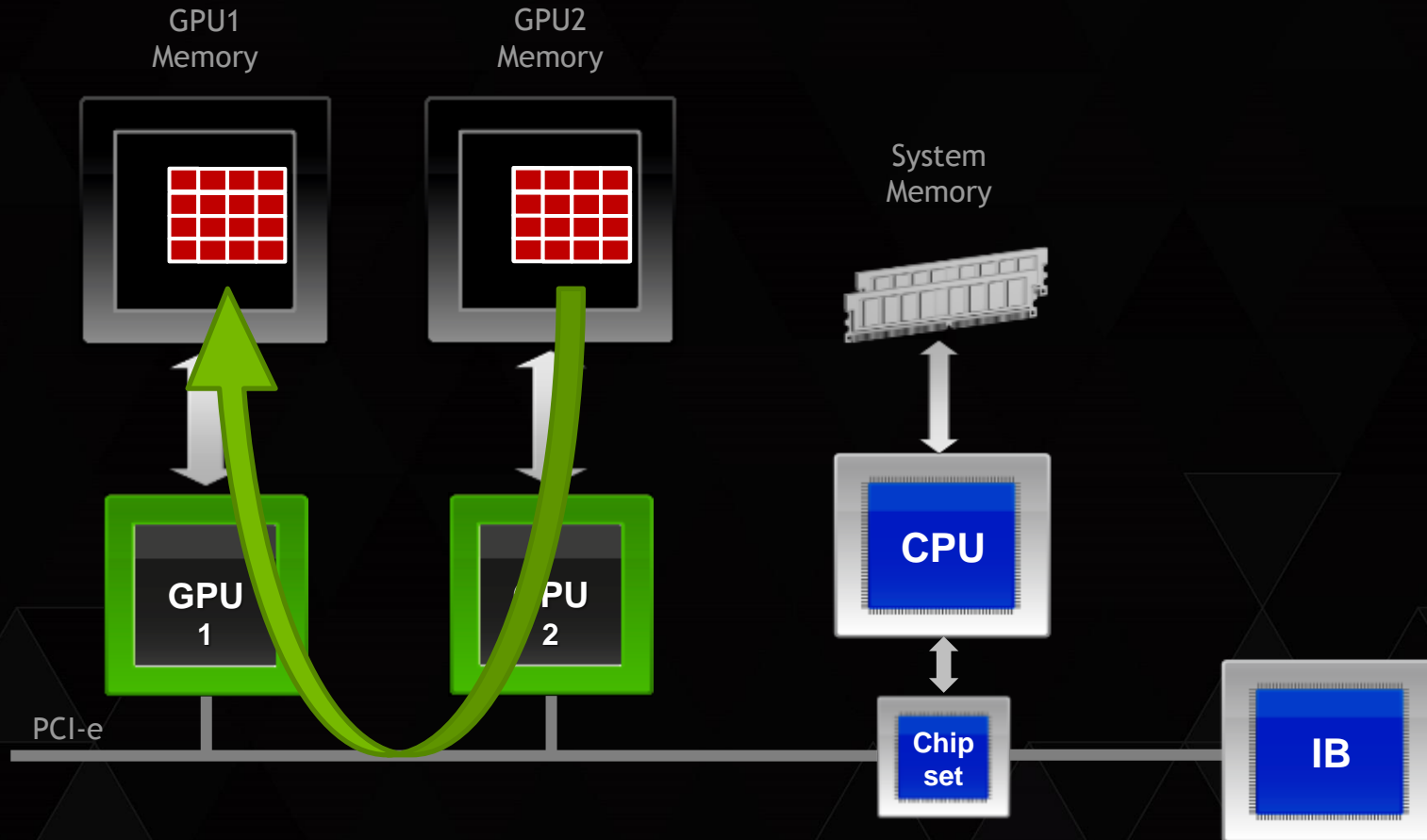
# NVIDIA GPUDIRECT™

## PEER TO PEER TRANSFERS

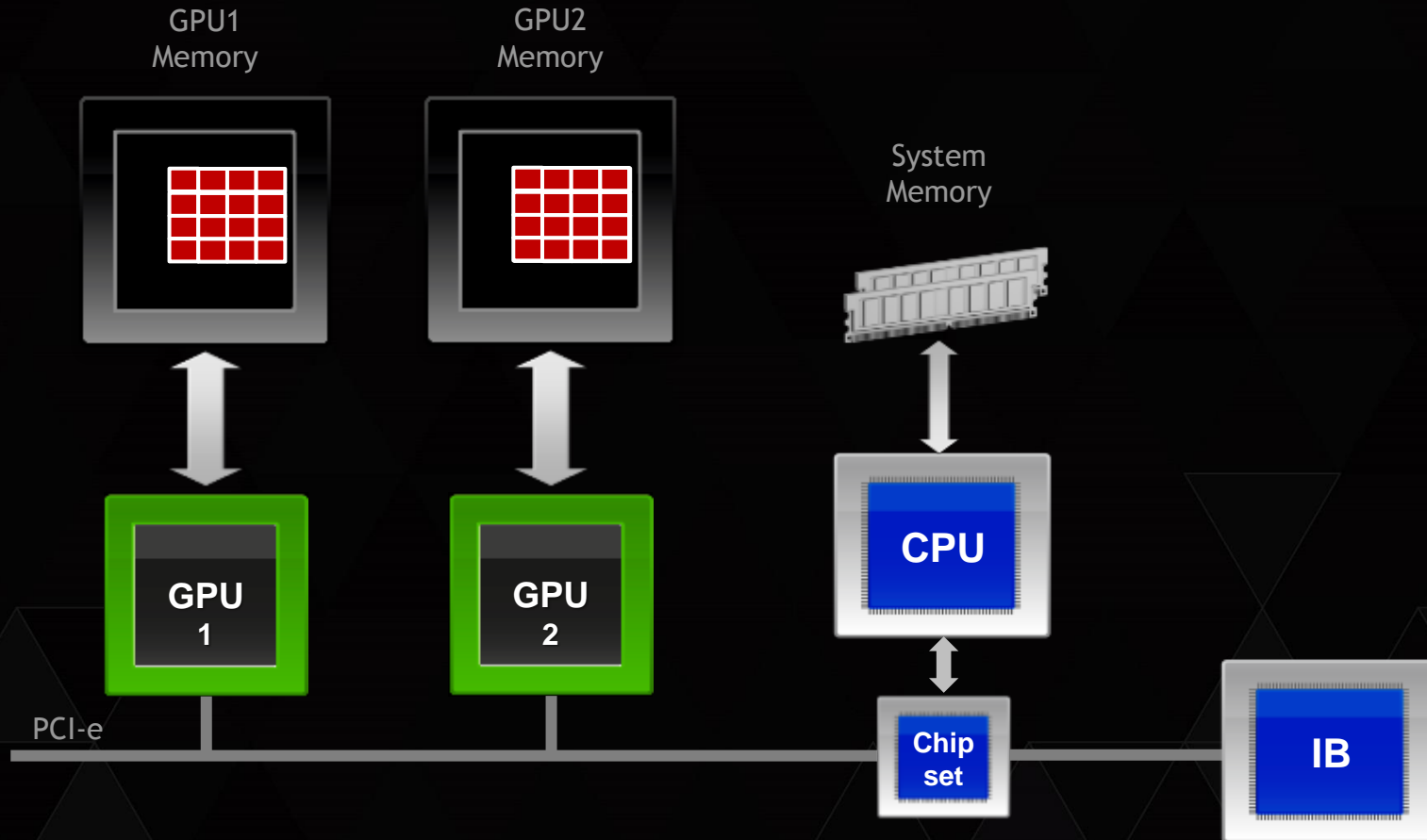


# NVIDIA GPUDIRECT™

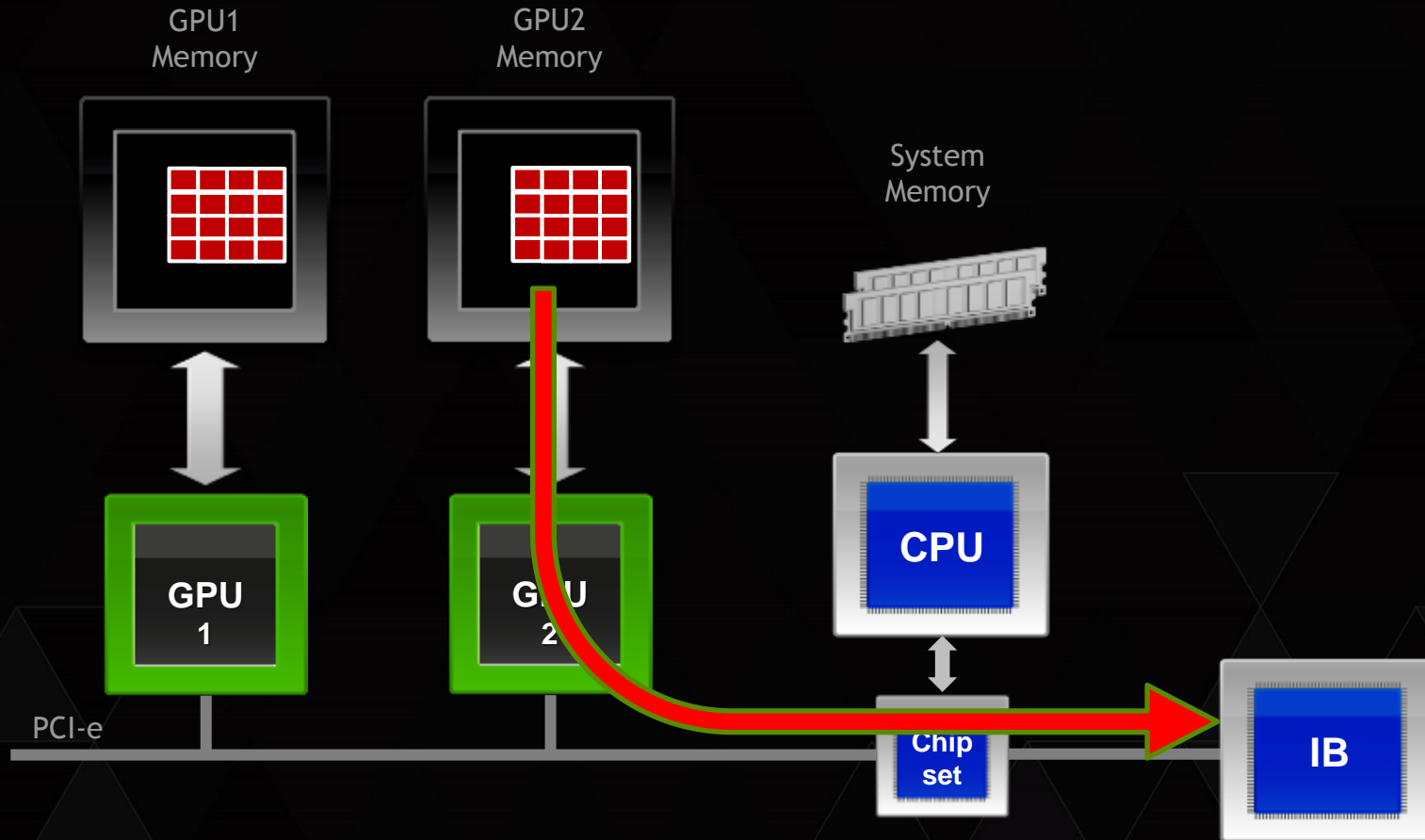
## PEER TO PEER TRANSFERS



# NVIDIA GPUDIRECT™ SUPPORT FOR RDMA



# NVIDIA GPUDIRECT™ SUPPORT FOR RDMA



# CUDA-AWARE MPI

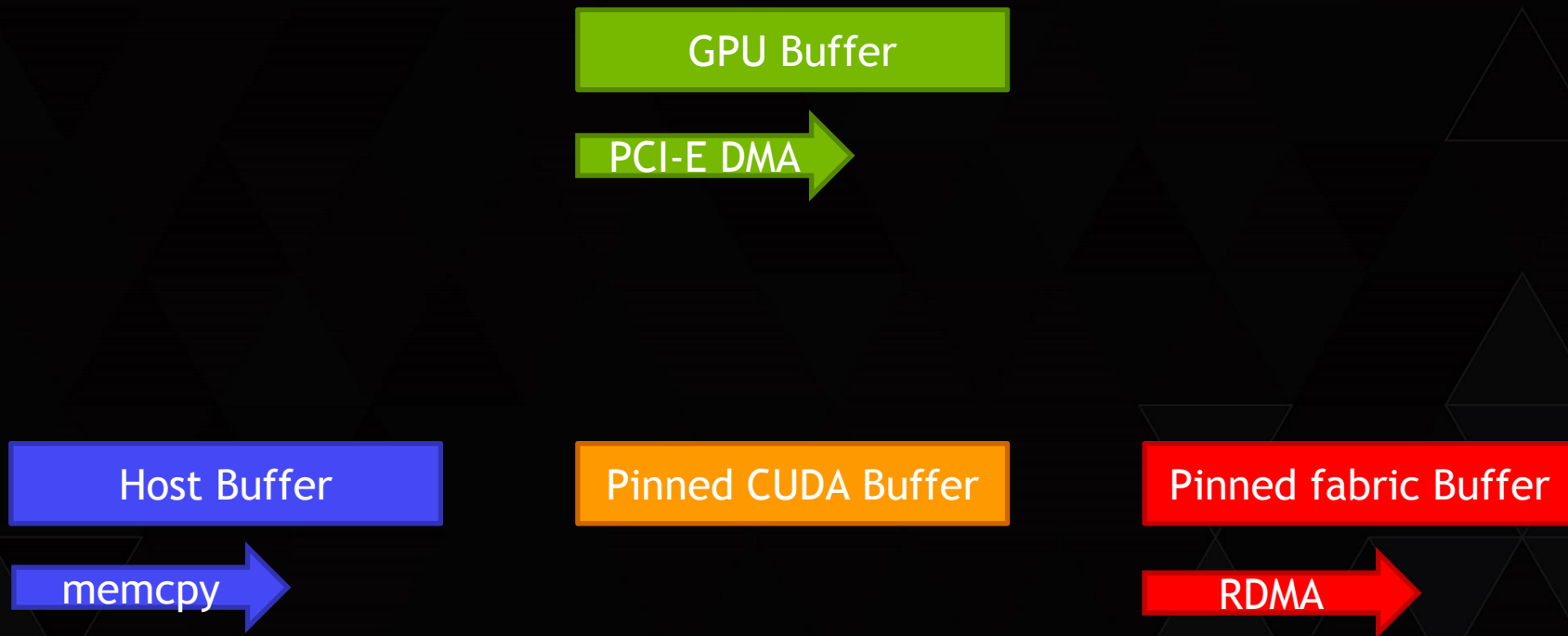
Example:

MPI Rank 0 MPI\_Send from GPU Buffer

MPI Rank 1 MPI\_Recv to GPU Buffer

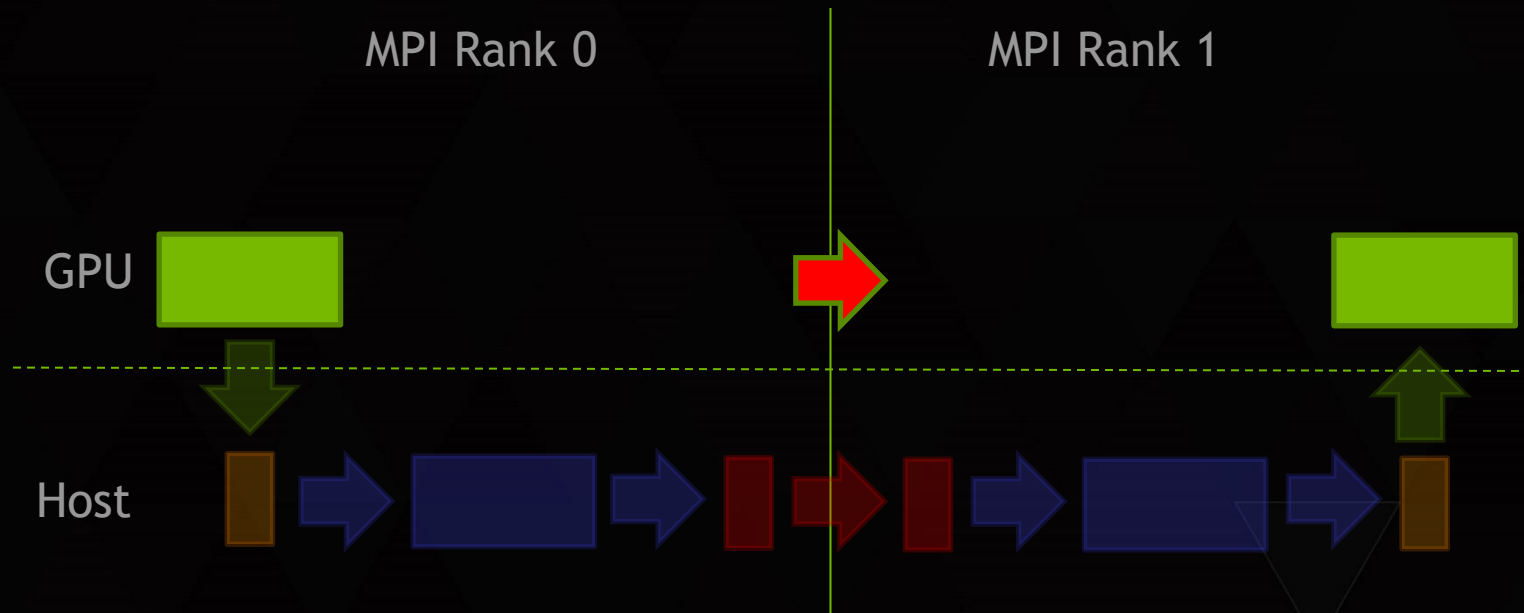
- ▶ Show how CUDA+MPI works in principle
  - ▶ Depending on the MPI implementation, message size, system setup, ... situation might be different
- ▶ Two GPUs in two nodes

# CUDA-AWARE MPI



# MPI GPU TO REMOTE GPU

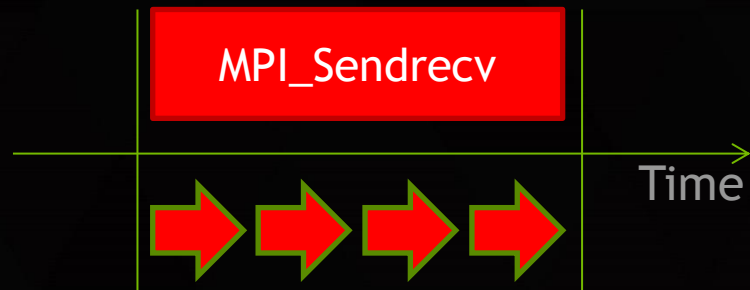
## SUPPORT FOR RDMA



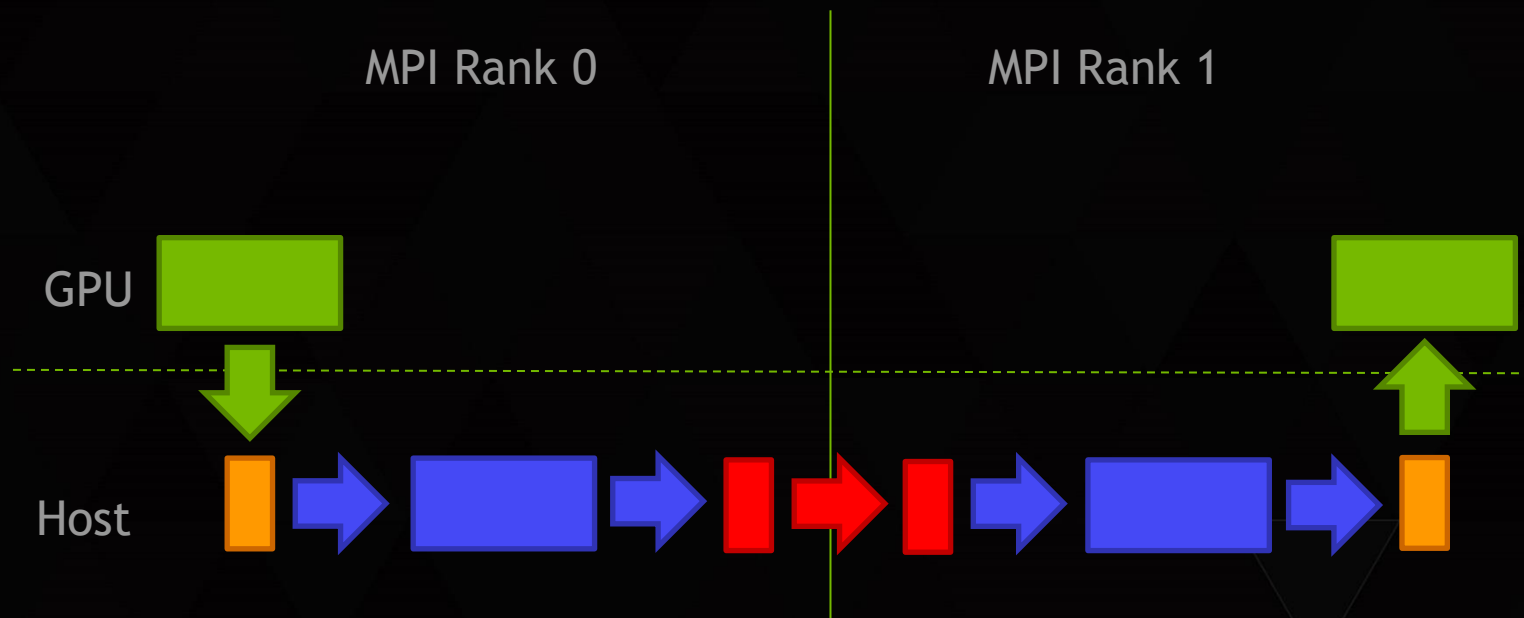
```
MPI_Send(s_buf_d, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
MPI_Recv(r_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
```



# MPI GPU TO REMOTE GPU SUPPORT FOR RDMA



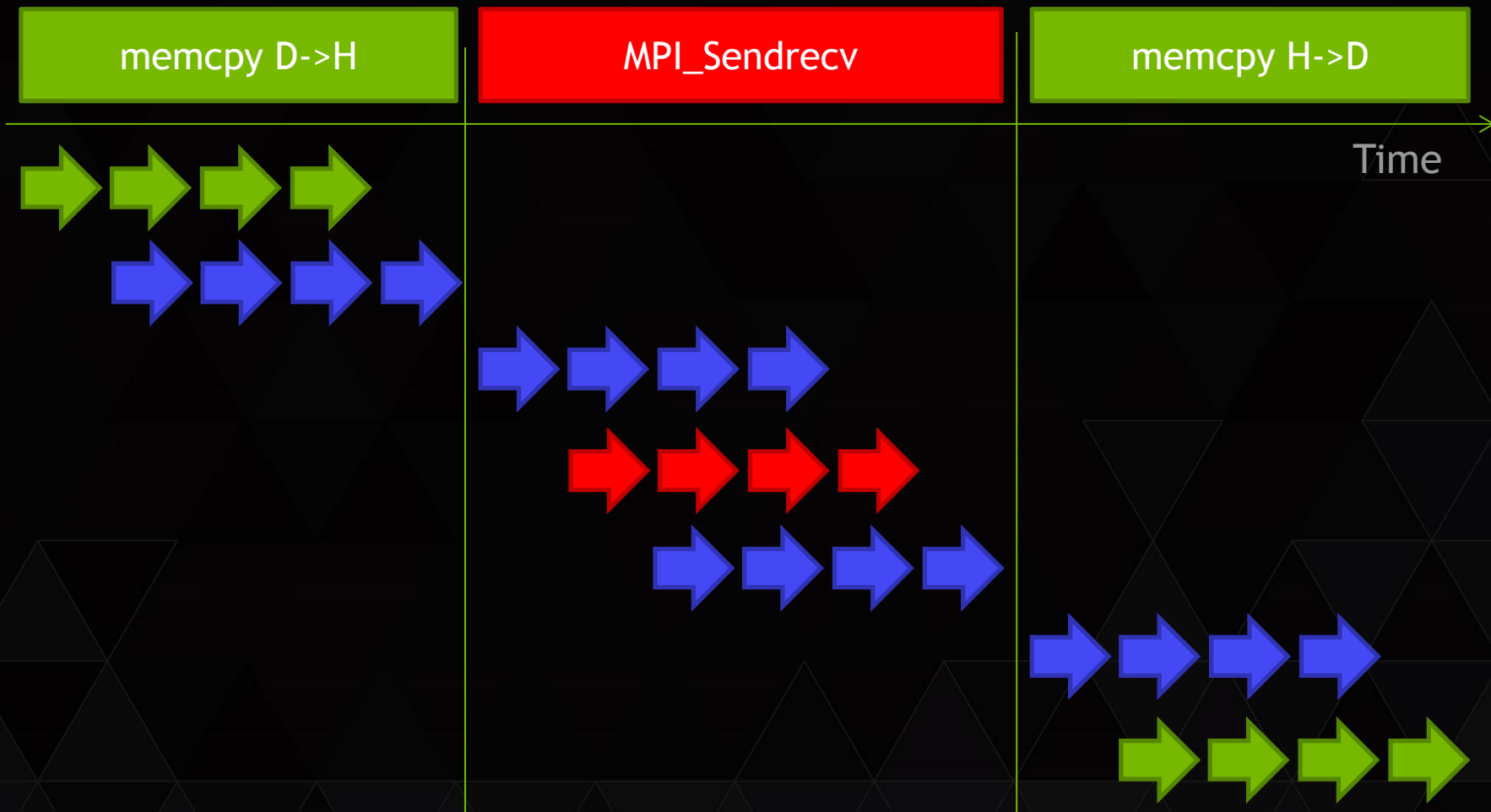
## REGULAR MPI GPU TO REMOTE GPU



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

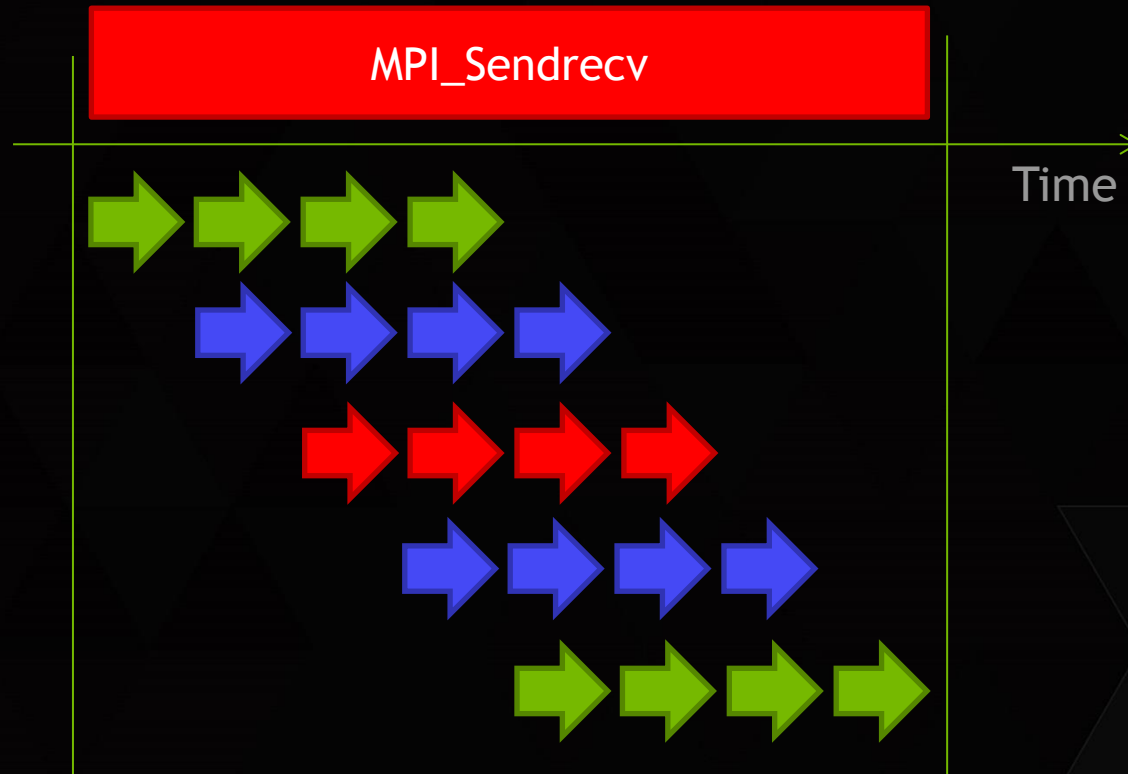
MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

# REGULAR MPI GPU TO REMOTE GPU



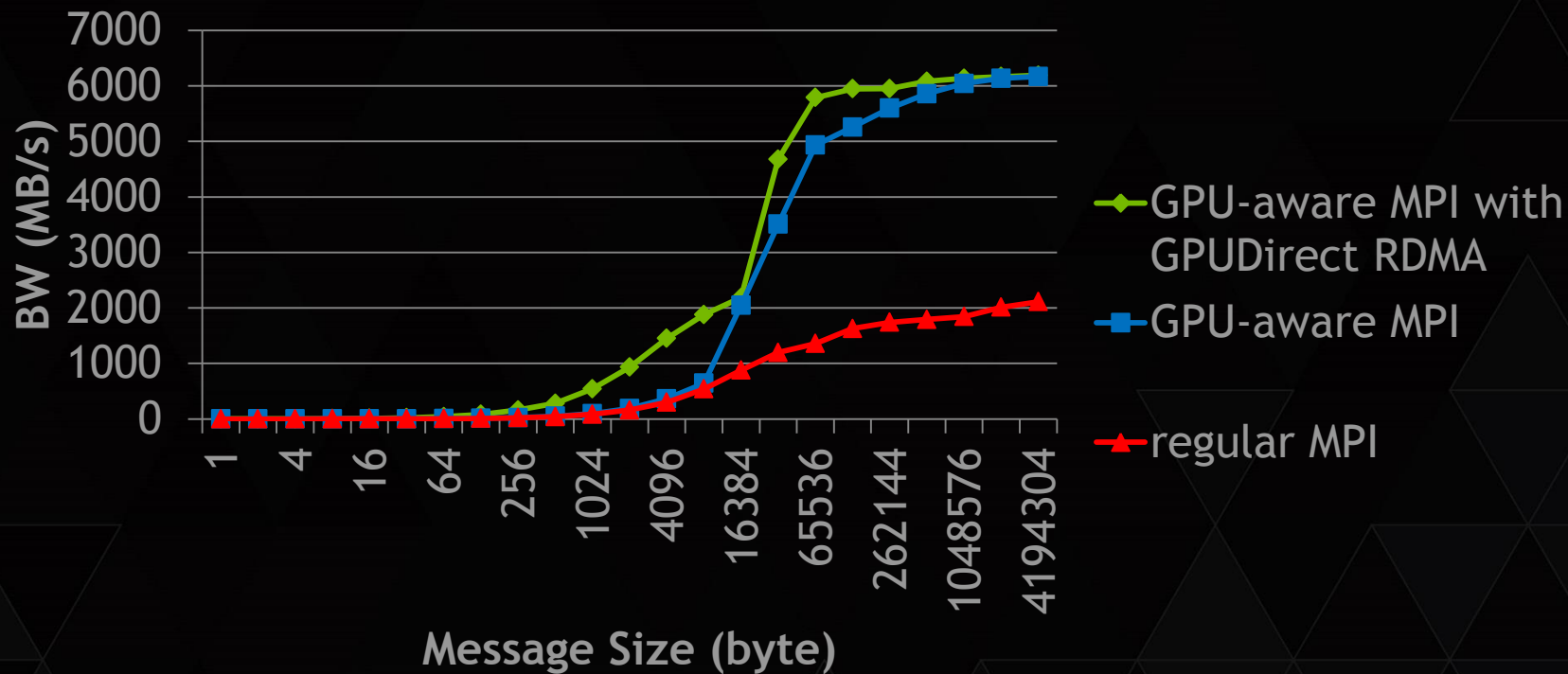


# MPI GPU TO REMOTE GPU WITHOUT GPUDIRECT



# PERFORMANCE RESULTS TWO NODES

OpenMPI 1.8.4 MLNX FDR IB (4X) Tesla K40@875



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



- GPU parallelizable part
- CPU parallel part
- Serial part

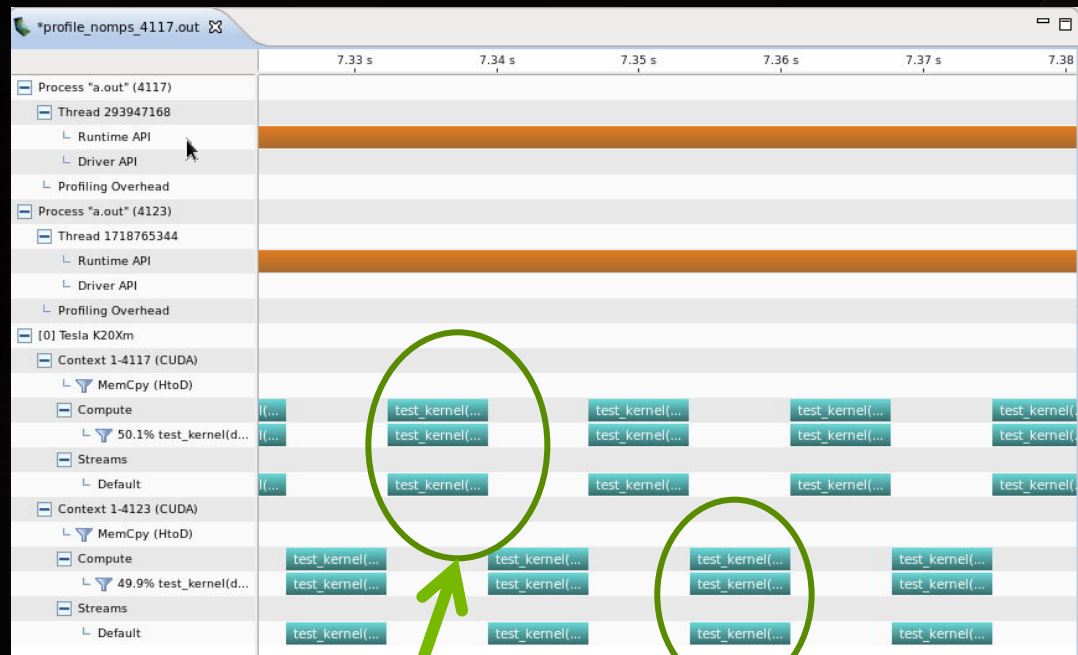
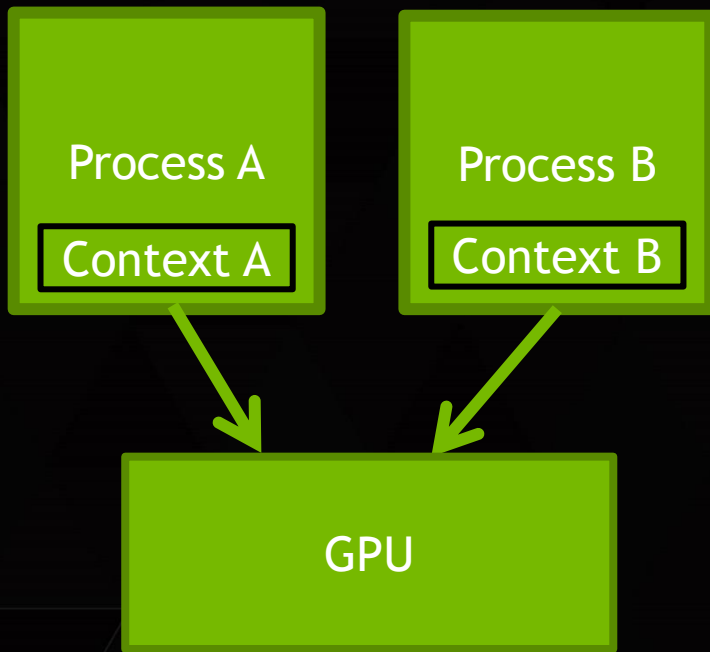


With Hyper-Q/MPS  
Available in K20, K40, K80

Multicore CPU only

GPU accelerated CPU

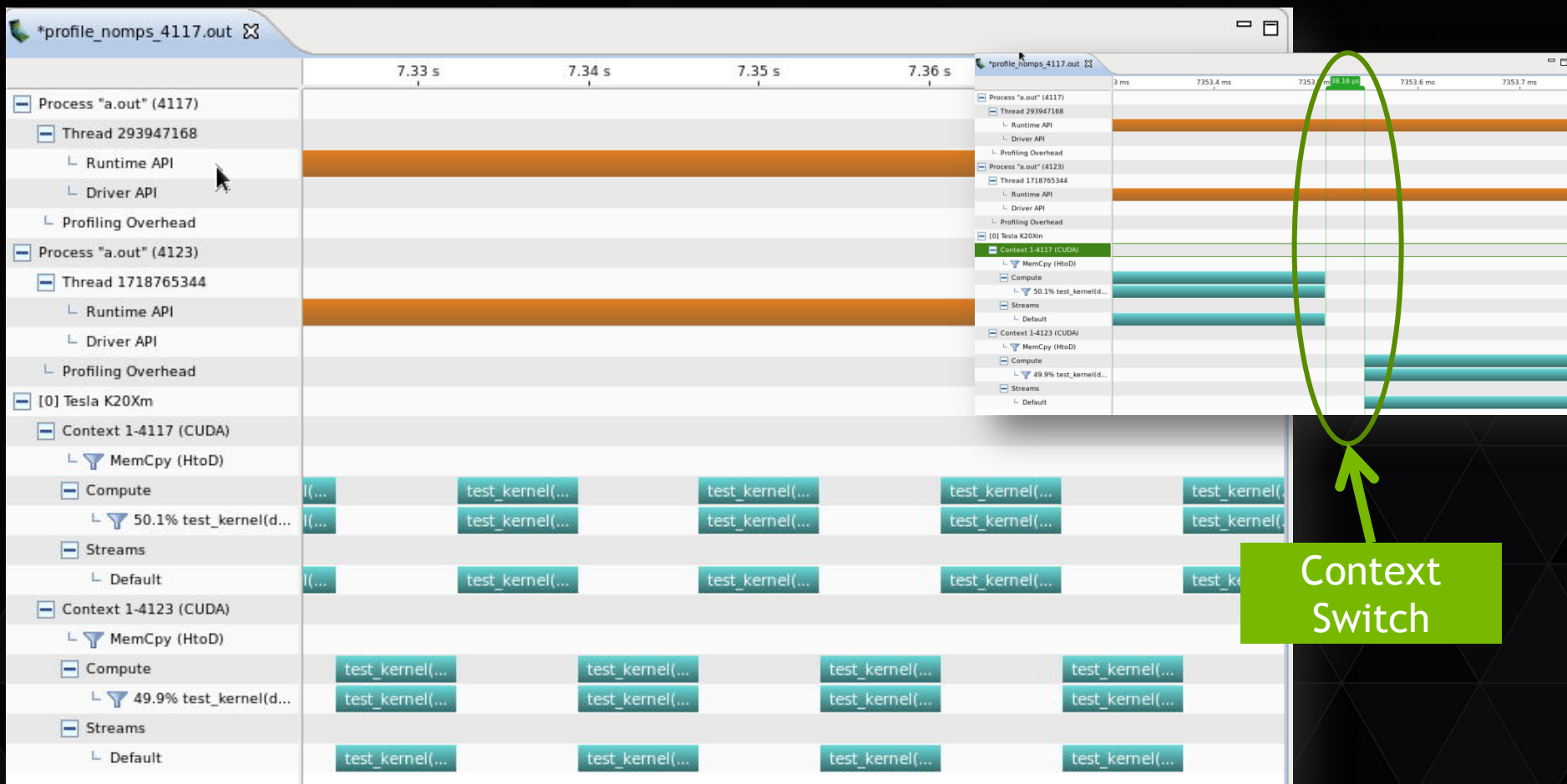
# PROCESSES SHARING GPU WITHOUT MPS: NO OVERLAP



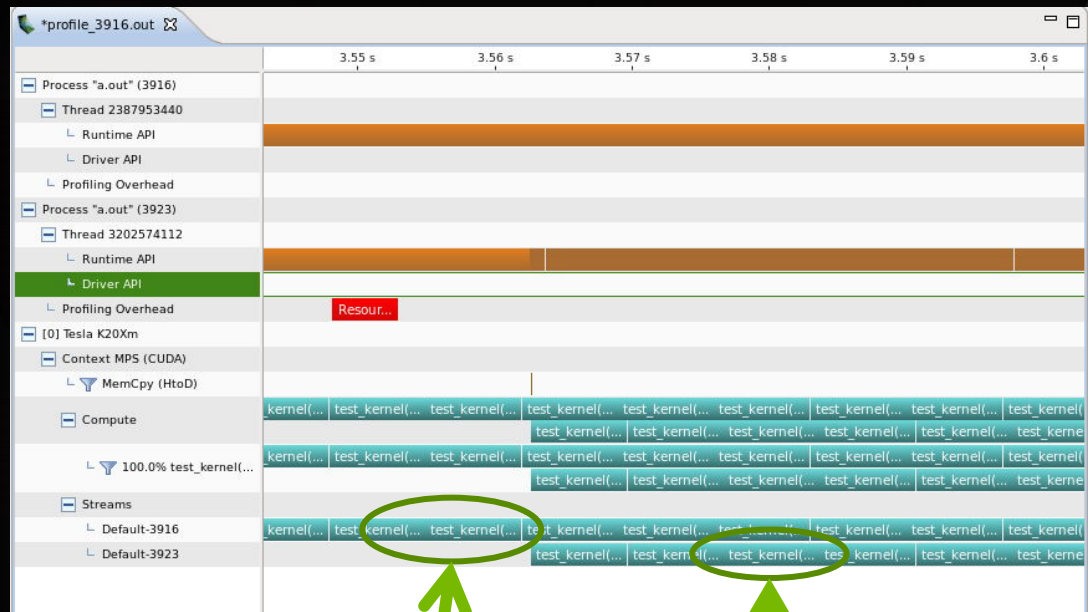
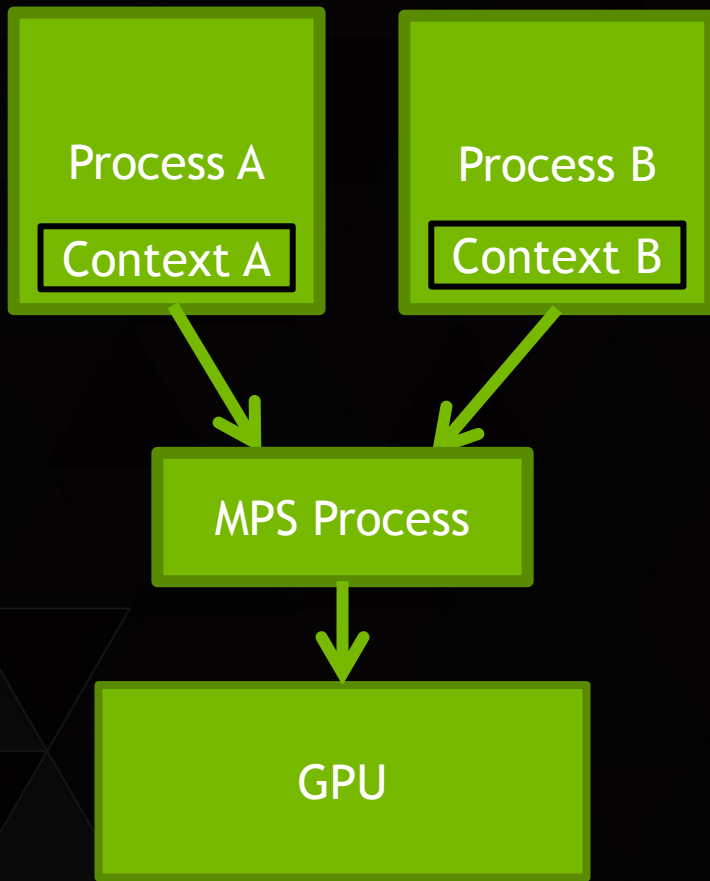
Process A

Process B

# GPU SHARING WITHOUT MPS



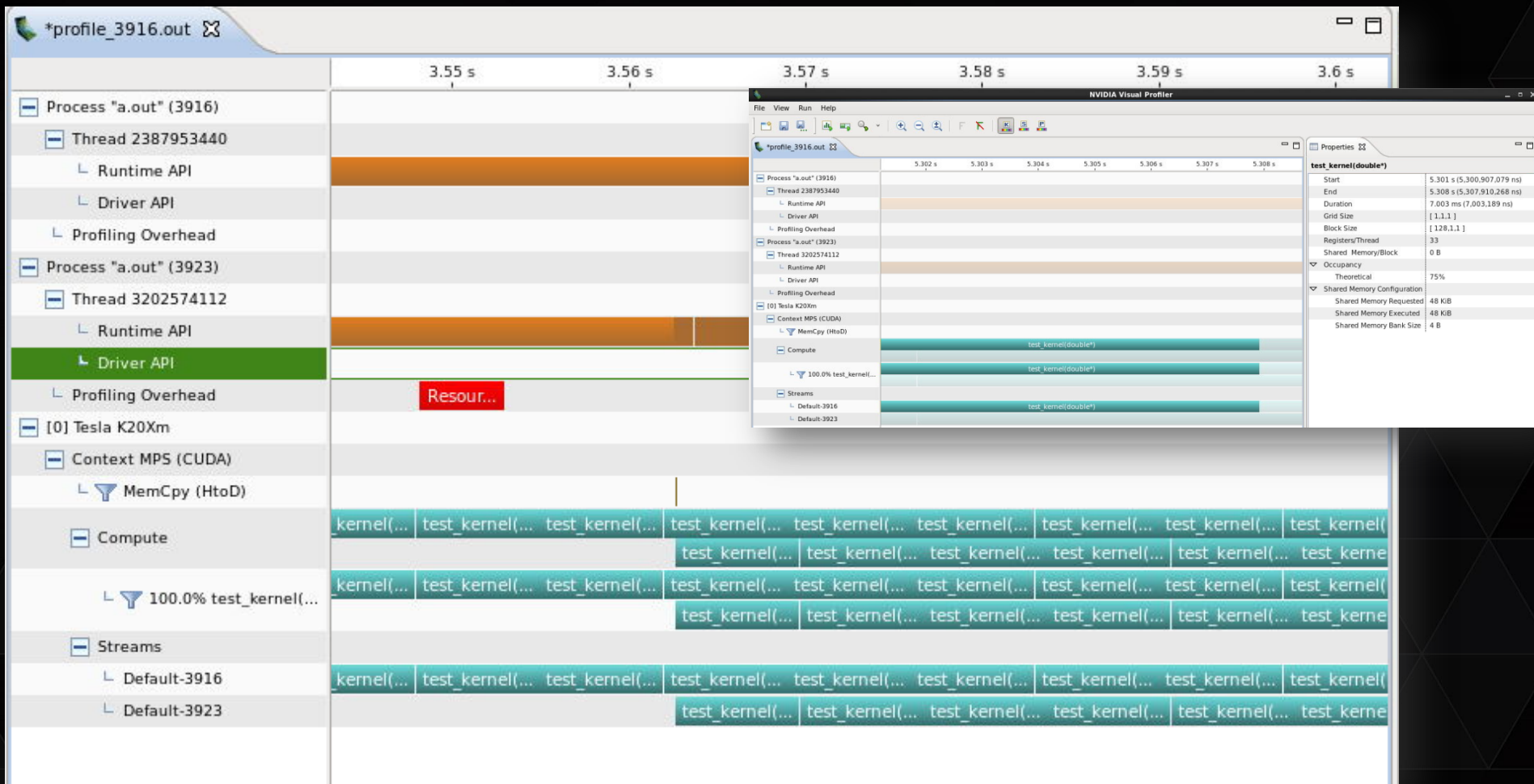
# PROCESSES SHARING GPU WITH MPS: MAXIMUM OVERLAP



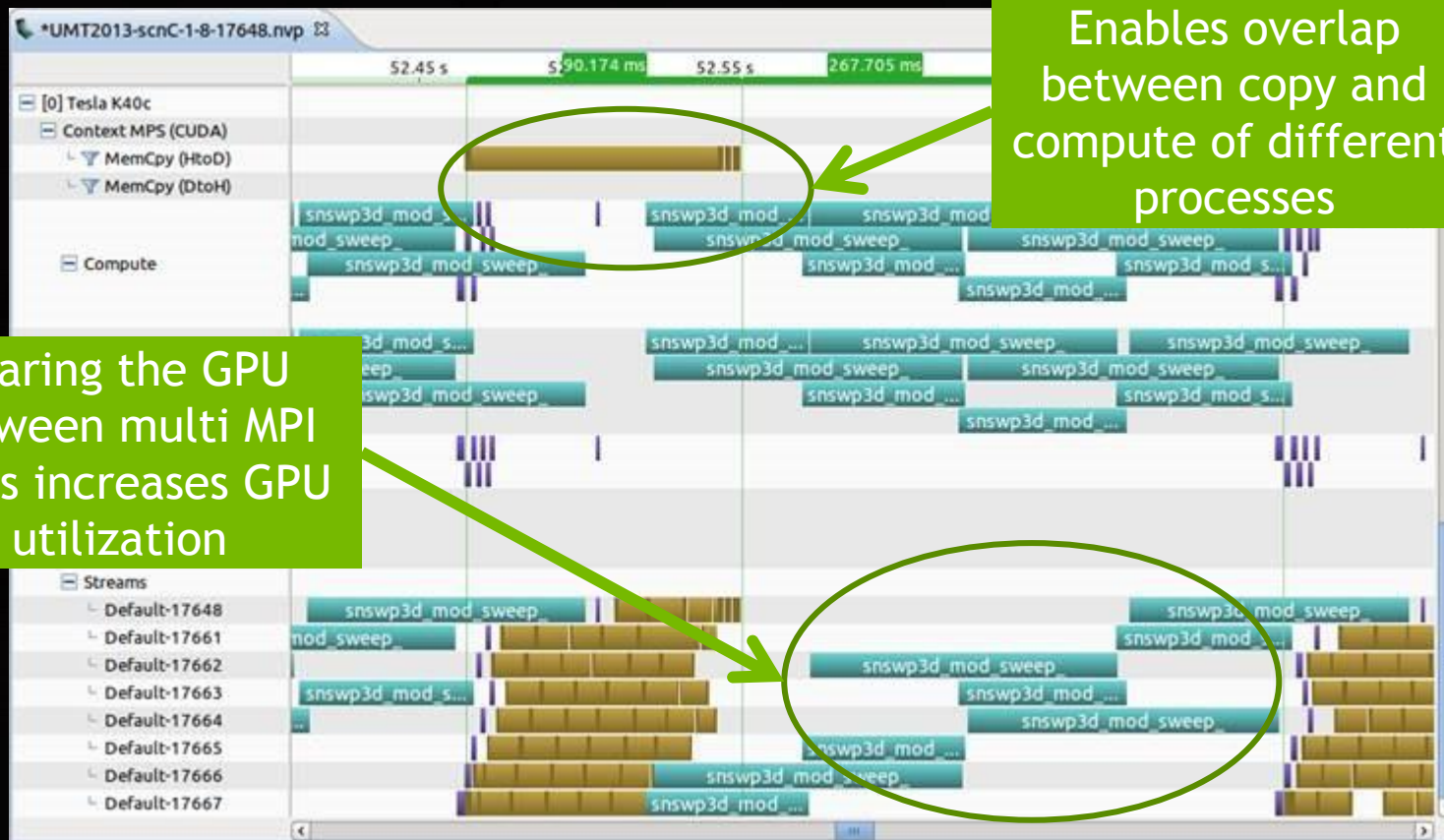
Kernels from Process A

Kernels from Process B

# GPU SHARING WITH MPS



# CASE STUDY: HYPER-Q/MPS FOR UMT

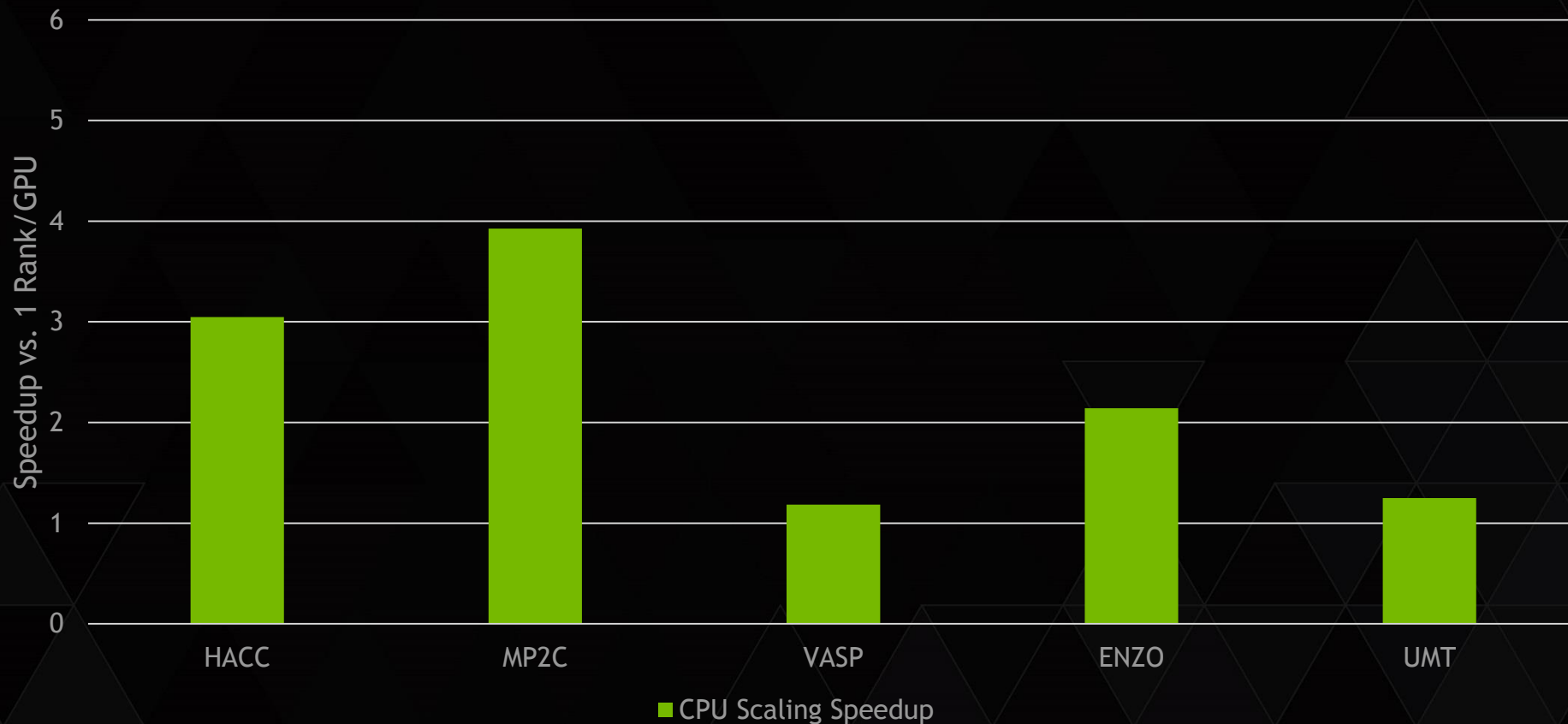


Enables overlap between copy and compute of different processes

Sharing the GPU between multi MPI ranks increases GPU utilization

# HYPER-Q/MPS CASE STUDIES

CPU Scaling Speedup



# HYPER-Q/MPS CASE STUDIES

Additional speedup with MPS





# 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
```

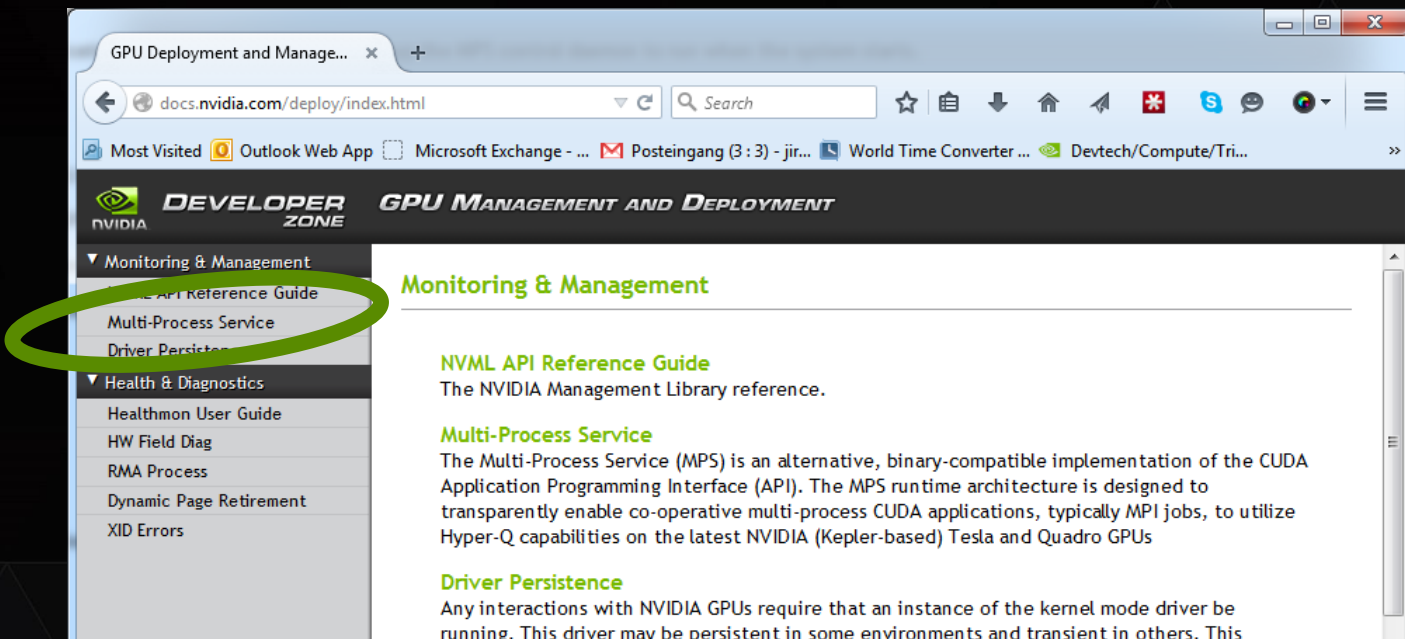
- On Cray XK/XC systems

```
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 your application does not benefit you want to avoid starting MPS.



The screenshot shows a web browser window displaying the NVIDIA Developer Zone page for GPU Management and Deployment. The URL is docs.nvidia.com/deploy/index.html. The page features a navigation menu on the left with the following items: Monitoring & Management, NVML API Reference Guide, Multi-Process Service, Driver Persistence, Health & Diagnostics, Healthmon User Guide, HW Field Diag, RMA Process, Dynamic Page Retirement, and XID Errors. The 'Multi-Process Service' link is highlighted with a green circle. The main content area is titled 'Monitoring & Management' and contains three sections: 'NVML API Reference Guide' (The NVIDIA Management Library reference.), 'Multi-Process Service' (The Multi-Process Service (MPS) is an alternative, binary-compatible implementation of the CUDA Application Programming Interface (API). The MPS runtime architecture is designed to transparently enable co-operative multi-process CUDA applications, typically MPI jobs, to utilize Hyper-Q capabilities on the latest NVIDIA (Kepler-based) Tesla and Quadro GPUs), and 'Driver Persistence' (Any interactions with NVIDIA GPUs require that an instance of the kernel mode driver be running. This driver may be persistent in some environments and transient in others. This).

**GPU** TECHNOLOGY  
CONFERENCE

# 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

```
jkraus@ivb114:~/workspace/qwiklabs/Multi-GPU-MPI/task3
[jkraus@ivb114 task3]$ mpirun -np 2 cuda-memcheck --log-file laplace2d.%q{OMPI COMM WORLD RANK}.log --save laplace2d.%q{OMPI COMM WORLD RANK}.memcheck ./laplace2d
Jacobi relaxation Calculation: 2048 x 2048 mesh
Calculate reference solution and time serial execution.
call to cuMemcpyDtoHAsync returned error 719: Launch failed (often invalid pointer dereference)
call to cuMemcpyDtoHAsync returned error 719: Launch failed (often invalid pointer dereference)

Primary job terminated normally, but 1 process returned
a non-zero exit code.. Per user-direction, the job has been aborted.

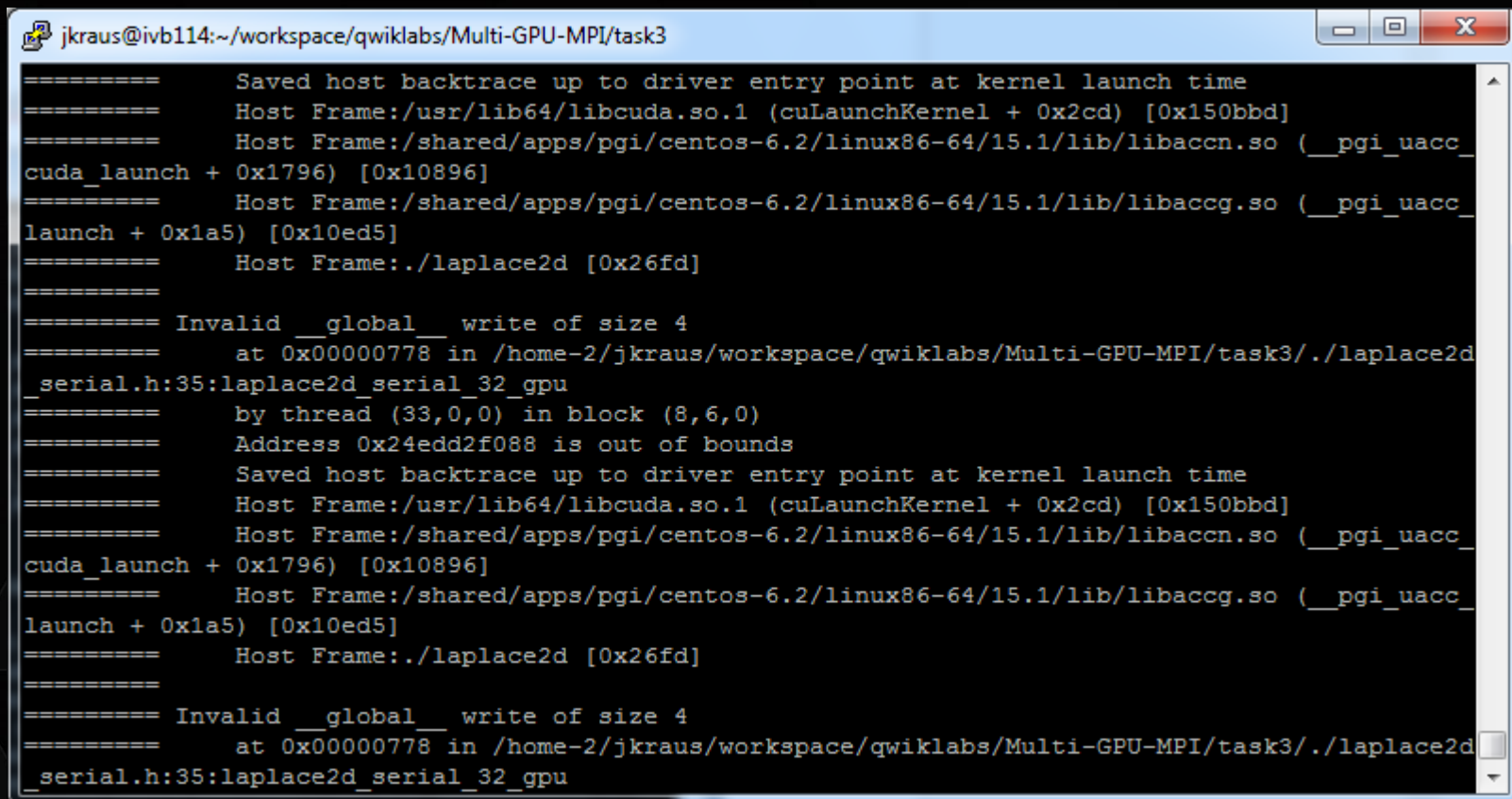
mpirun detected that one or more processes exited with non-zero status, thus causing
the job to be terminated. The first process to do so was:

 Process name: [[42894,1],0]
 Exit code: 1

[jkraus@ivb114 task3]$ ls laplace2d.*.log laplace2d.*.memcheck
laplace2d.0.log laplace2d.0.memcheck laplace2d.1.log laplace2d.1.memcheck
[jkraus@ivb114 task3]$
```

# MEMORY CHECKING WITH CUDA-MEMCHECK

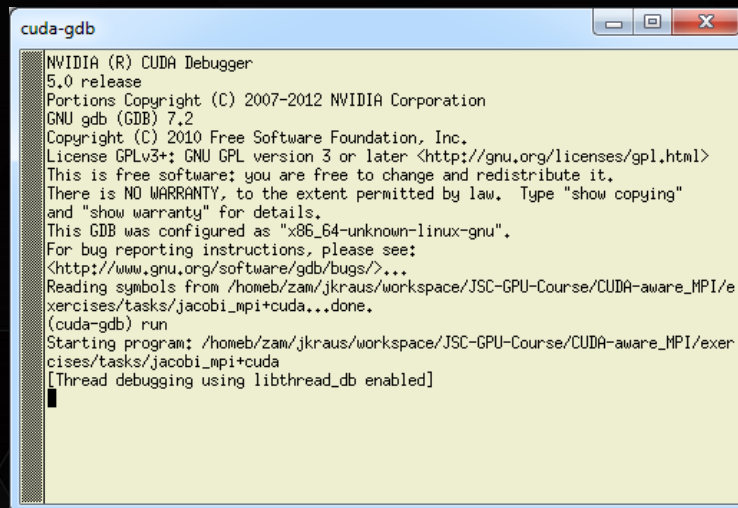
Read outputfiles with `cuda-memcheck --read`



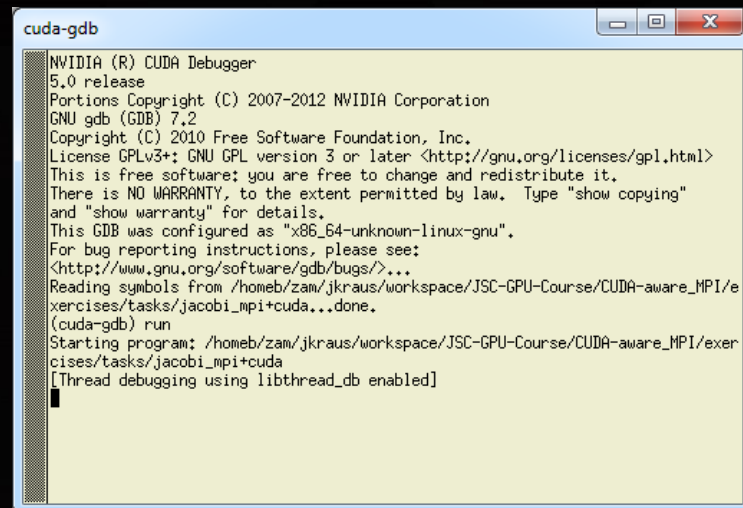
```
jkraus@ivb114:~/workspace/qwiklabs/Multi-GPU-MPI/task3
=====
Saved host backtrace up to driver entry point at kernel launch time
=====
Host Frame:/usr/lib64/libcuda.so.1 (cuLaunchKernel + 0x2cd) [0x150bbd]
=====
Host Frame:/shared/apps/pgi/centos-6.2/linux86-64/15.1/lib/libaccn.so (__pgi_uacc
cuda_launch + 0x1796) [0x10896]
=====
Host Frame:/shared/apps/pgi/centos-6.2/linux86-64/15.1/lib/libaccg.so (__pgi_uacc
launch + 0x1a5) [0x10ed5]
=====
Host Frame:./laplace2d [0x26fd]
=====
Invalid __global__ write of size 4
=====
at 0x00000778 in /home-2/jkraus/workspace/qwiklabs/Multi-GPU-MPI/task3/./laplace2d
_serial.h:35:laplace2d_serial_32_gpu
=====
by thread (33,0,0) in block (8,6,0)
=====
Address 0x24edd2f088 is out of bounds
=====
Saved host backtrace up to driver entry point at kernel launch time
=====
Host Frame:/usr/lib64/libcuda.so.1 (cuLaunchKernel + 0x2cd) [0x150bbd]
=====
Host Frame:/shared/apps/pgi/centos-6.2/linux86-64/15.1/lib/libaccn.so (__pgi_uacc
cuda_launch + 0x1796) [0x10896]
=====
Host Frame:/shared/apps/pgi/centos-6.2/linux86-64/15.1/lib/libaccg.so (__pgi_uacc
launch + 0x1a5) [0x10ed5]
=====
Host Frame:./laplace2d [0x26fd]
=====
Invalid __global__ write of size 4
=====
at 0x00000778 in /home-2/jkraus/workspace/qwiklabs/Multi-GPU-MPI/task3/./laplace2d
_serial.h:35:laplace2d_serial_32_gpu
```

# 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>
```



```
cuda-gdb
NVIDIA (R) CUDA Debugger
5.0 release
Portions Copyright (C) 2007-2012 NVIDIA Corporation
GNU gdb (GDB) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /homeb/zam/jkraus/workspace/JSC-GPU-Course/CUDA-aware_MPI/e
xercises/tasks/jacobi_mpi+cuda...done.
(cuda-gdb) run
Starting program: /homeb/zam/jkraus/workspace/JSC-GPU-Course/CUDA-aware_MPI/exe
rcises/tasks/jacobi_mpi+cuda
[Thread debugging using libthread_db enabled]
█
```



```
cuda-gdb
NVIDIA (R) CUDA Debugger
5.0 release
Portions Copyright (C) 2007-2012 NVIDIA Corporation
GNU gdb (GDB) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /homeb/zam/jkraus/workspace/JSC-GPU-Course/CUDA-aware_MPI/e
xercises/tasks/jacobi_mpi+cuda...done.
(cuda-gdb) run
Starting program: /homeb/zam/jkraus/workspace/JSC-GPU-Course/CUDA-aware_MPI/exe
rcises/tasks/jacobi_mpi+cuda
[Thread debugging using libthread_db enabled]
█
```


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

```
jkraus@sb077:~/workspace/Jacobi/main/bin
Iteration: 700 - Residue: 0.306564
Iteration: 800 - Residue: 0.306564
Iteration: 900 - Residue: 0.306564
Stopped after 1000 iterations with residue 0.306564
Total Jacobi run time: 0.8700 sec.
Average per-process communication time: 0.2765 sec.
Measured lattice updates: 4.81 GLU/s (total), 1.20 GLU/s (per process)
Measured FLOPS: 24.06 GFLOPS (total), 6.01 GFLOPS (per process)
Measured device bandwidth: 230.95 GB/s
[jkraus@sb077 bin]$ CUDA_DEVICE_WAIT_BEFORE_EXCEPTION=1 ./jacobi
aware_mpi_async -t 2 2 -d 1024 1024
Topology size: 2 x 2
Local domain size (current node): 1024
Global domain size (all nodes): 2048
Starting Jacobi run with 4 processes
sb077: The application encountered an error and
can now attach a debugger to the process.
sb077: The application encountered an error and
can now attach a debugger to the process.
sb077: The application encountered an error and
can now attach a debugger to the process.
sb077: The application encountered an error and
can now attach a debugger to the process.
Reading symbols from /usr/lib64/libnes-rdmav2.so...(no debugging symbols found)...done.
Loaded symbols for /usr/lib64/libnes-rdmav2.so
Reading symbols from /usr/lib64/libmlx4-rdmav2.so...(no debugging symbols found)...done.
Loaded symbols for /usr/lib64/libmlx4-rdmav2.so
Reading symbols from /usr/lib64/libipathverbs-rdmav2.so...(no debugging symbols found)...done.
Loaded symbols for /usr/lib64/libipathverbs-rdmav2.so
0x000007f5ba011fa01 in clock_gettime ()
$1 = 1
CUDA Exception: Device Illegal Address
The exception was triggered in device 3.
Program received signal CUDA_EXCEPTION_10, Device Illegal Address.
[Switching focus to CUDA kernel 0, grid 8, block (6,36,0), thread (0,6,0), device 3, sm 0, warp
13, lane 0]
0x00000000018e1ce8 in JacobiComputeKernel<<<(64,64,1),(16,16,1)>>> (size=..., startmod=...,
endmod=..., oldBlock=0x2300200000, newBlock=0x2300b20000, devResidue=0x2301340000,
stride=1024) at Device.cu:150
150 AtomicMax<real>(devResidue, rabs(newVal - oldBlock[memIdx]));
(cuda-gdb) bt
#0 0x00000000018e1ce8 in JacobiComputeKernel<<<(64,64,1),(16,16,1)>>> (size=...,
startmod=..., endmod=..., oldBlock=0x2300200000, newBlock=0x2300b20000,
devResidue=0x2301340000, stride=1024) at Device.cu:150
(cuda-gdb)
```

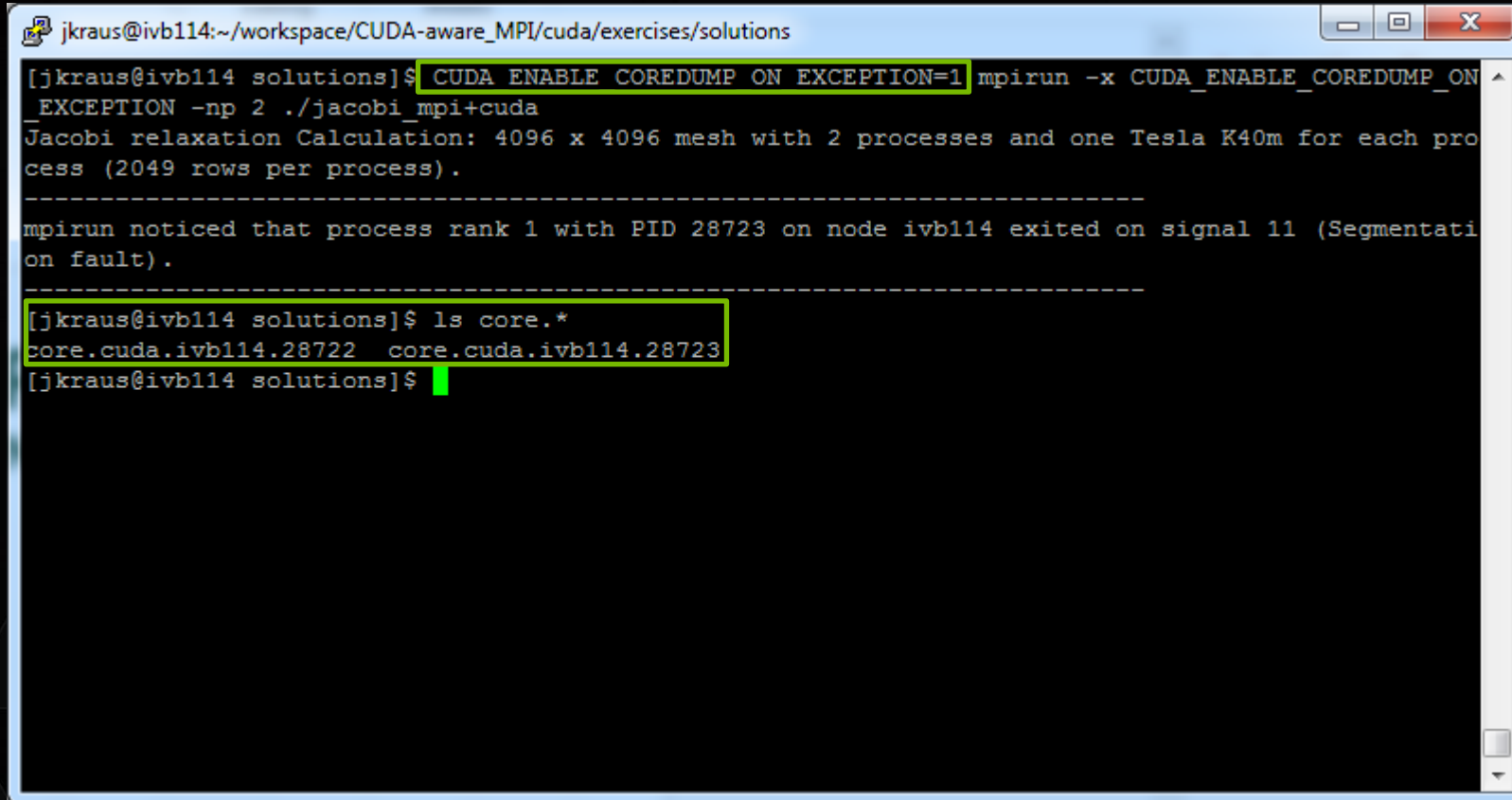
DEBUGGING MPI+CUDA APPLICATIONS

- ▶ 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



```
jkraus@ivb114:~/workspace/CUDA-aware_MPI/cuda/exercises/solutions
[jkraus@ivb114 solutions]$ CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1 mpirun -x CUDA_ENABLE_COREDUMP_ON_EXCEPTION -np 2 ./jacobi_mpi+cuda
Jacobi relaxation Calculation: 4096 x 4096 mesh with 2 processes and one Tesla K40m for each process (2049 rows per process).
-----
mpirun noticed that process rank 1 with PID 28723 on node ivb114 exited on signal 11 (Segmentation fault).
-----
[jkraus@ivb114 solutions]$ ls core.*
core.cuda.ivb114.28722 core.cuda.ivb114.28723
[jkraus@ivb114 solutions]$
```

DEBUGGING MPI+CUDA APPLICATIONS

CUDA_ENABLE_COREDUMP_ON_EXCEPTION

```
jkraus@ivb114:~/workspace/CUDA-aware_MPI/cuda/exercises/solutions
NVIDIA (R) CUDA Debugger
7.0 release
Portions Copyright (C) 2007-2014 NVIDIA Corporation
GNU gdb (GDB) 7.6.2
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(cuda-gdb) target cudacore core.cuda.ivb114.28722
Opening GPU coredump: core.cuda.ivb114.28722
[New Thread 28742]

CUDA Exception: Device Illegal Address
The exception was triggered in device 0.
[Current focus set to CUDA kernel 0, grid 1, block (107,0,0), thread (0,12,0), device 0, sm 12,
warp 6, lane 0]
#0  0x0000000001c02ac0 in jacobi_kernel<<<(257,129,1),(16,16,1)>>> (u_d=0x23048a0000,
    unew_d=0x23068c0000, n=2049, m=4096, residue_d=0x23088e0000) at jacobi_cuda_kernel.cu:43
43      residue = fabsf(unew_d[j *m+ i]-u_d[j *m+ i]);
(cuda-gdb)
```

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)

Stacks		
Threads	CUDA Threads	Function
1	0	main (prefix.cu:193)
1	0	cudasummer (prefix.cu:143)
1	0	prefixsum (prefix.cu:105)
1	512	zarro (prefix.cu:89)
1	480	zarro (prefix.cu:90)

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
if (x < length)
    out[x] = in[x];
__syncthreads();
for (int i = 1; i < BLOCK_SIZE; i += 1)
    if (threadIdx.x + 1 < BLOCK_SIZE && x + 1 < length)
```

On this line:
1 Process: rank 0
1 Thread (Process 0): #2
512 GPU threads:
<<<(0,0),(0,0)>> ... <<<(7,0),(63,0,0)>> (512 threads)

Variable Name	Value
x	0
out	0x100800
length	500
in	0x100000
i	1

Type: @register int



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}
```

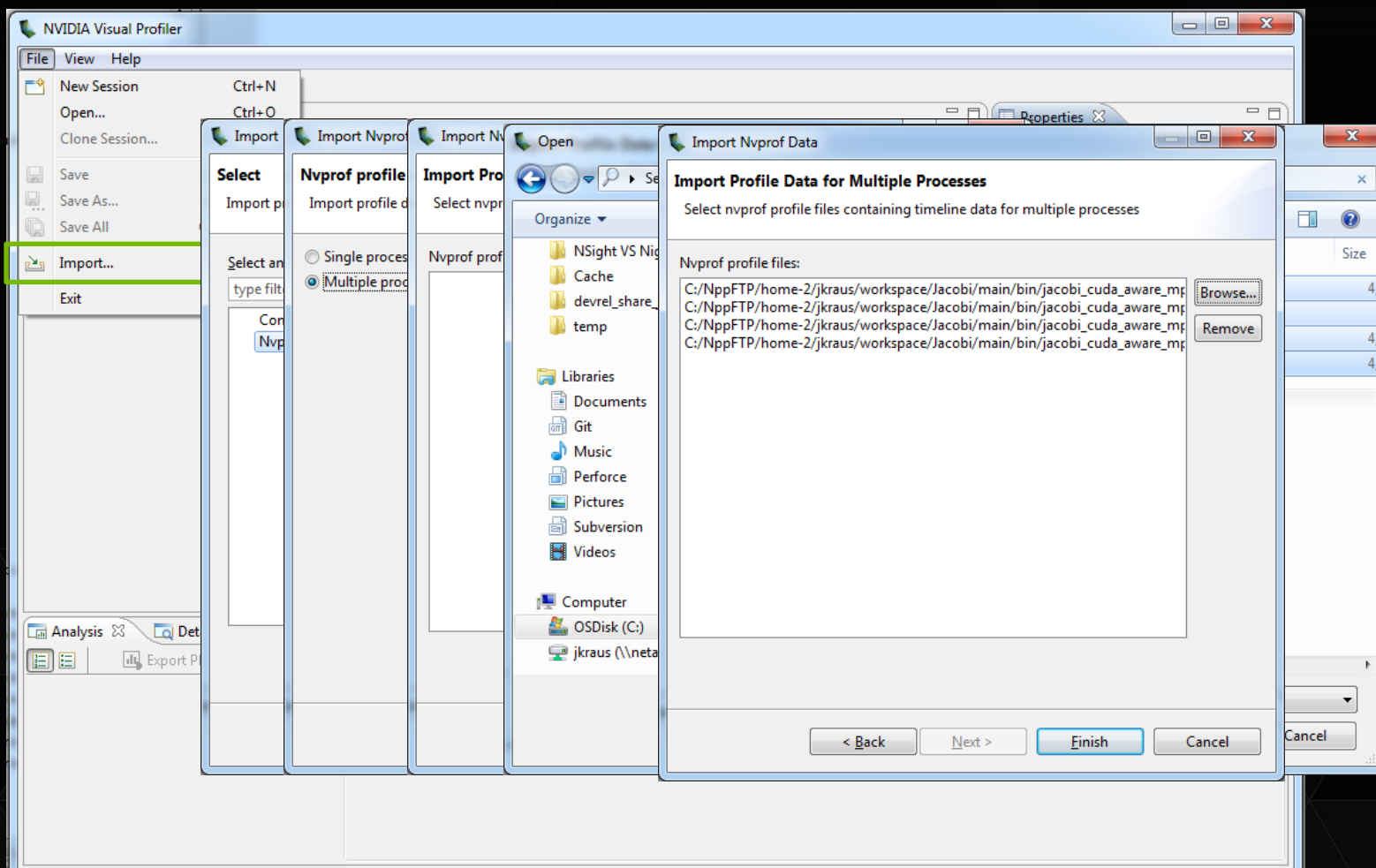
- ▶ Collect profile data on all processes that run on a node

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

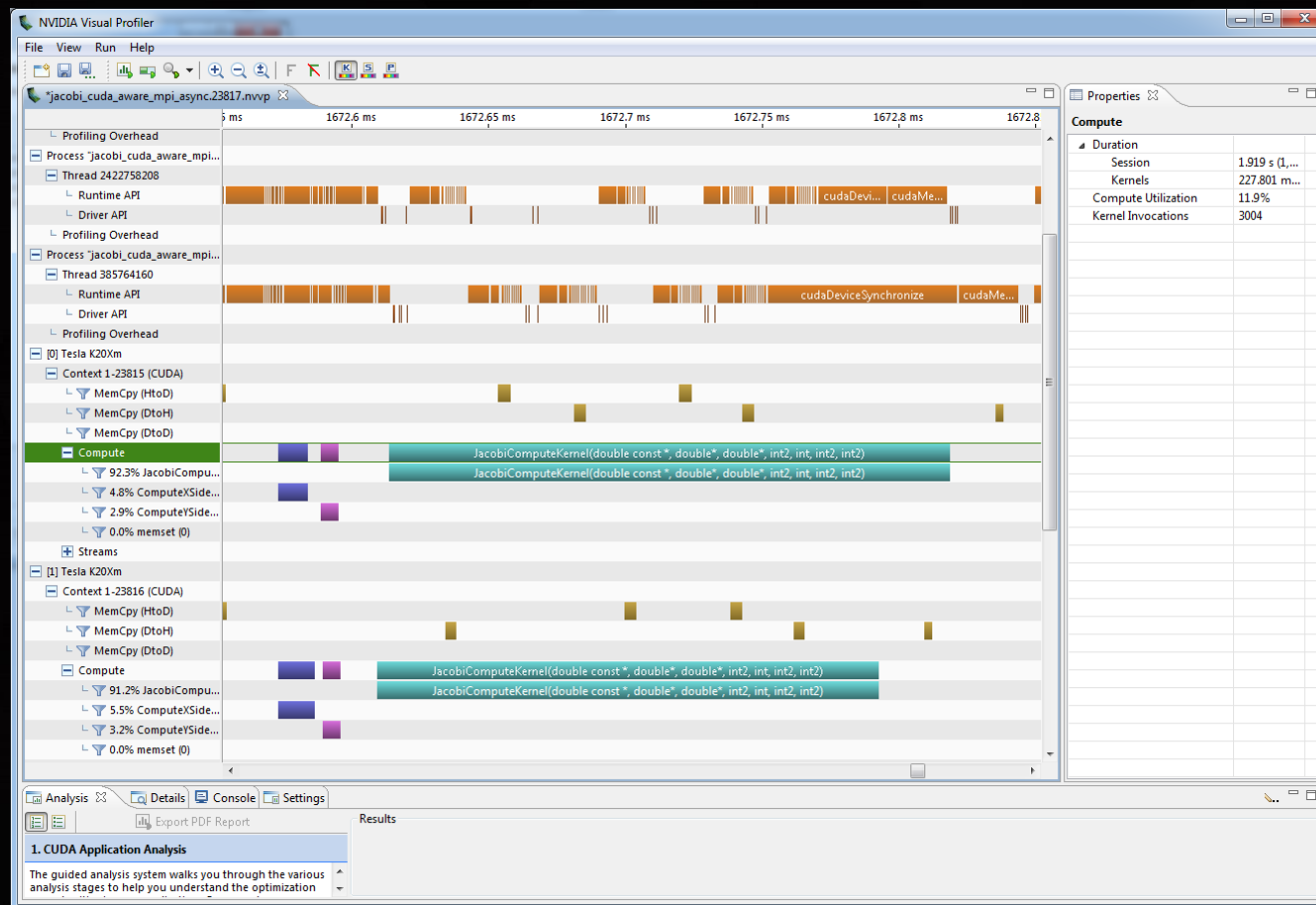
- ▶ **OpenMPI:** OMPI_COMM_WORLD_RANK

- ▶ **MVAPICH2:** MV2_COMM_WORLD_RANK

PROFILING MPI+CUDA APPLICATIONS USING NVPROF+NVVP



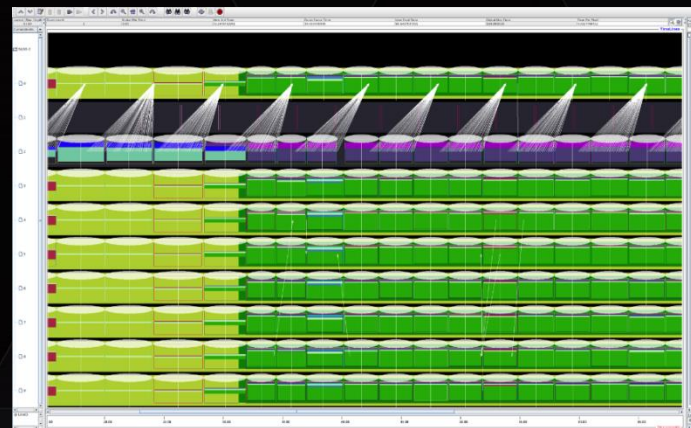
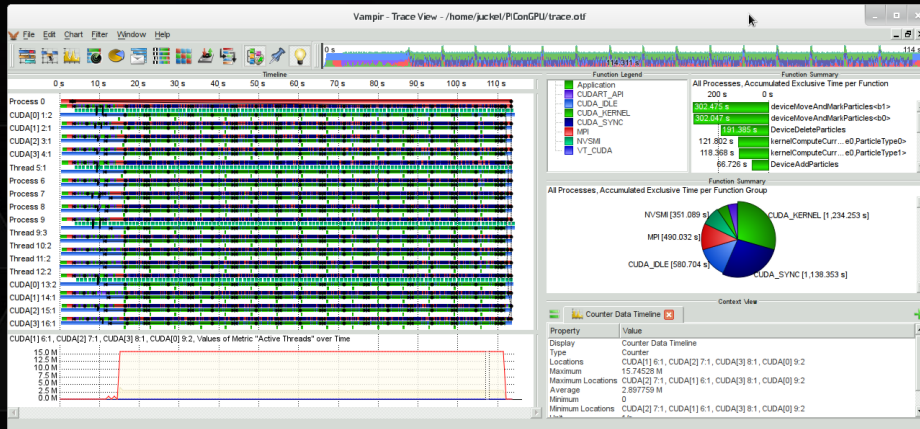
PROFILING MPI+CUDA APPLICATIONS USING NVPROF+NVVP



PROFILING MPI+CUDA APPLICATIONS

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



GPU TECHNOLOGY
CONFERENCE

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);  
}
```

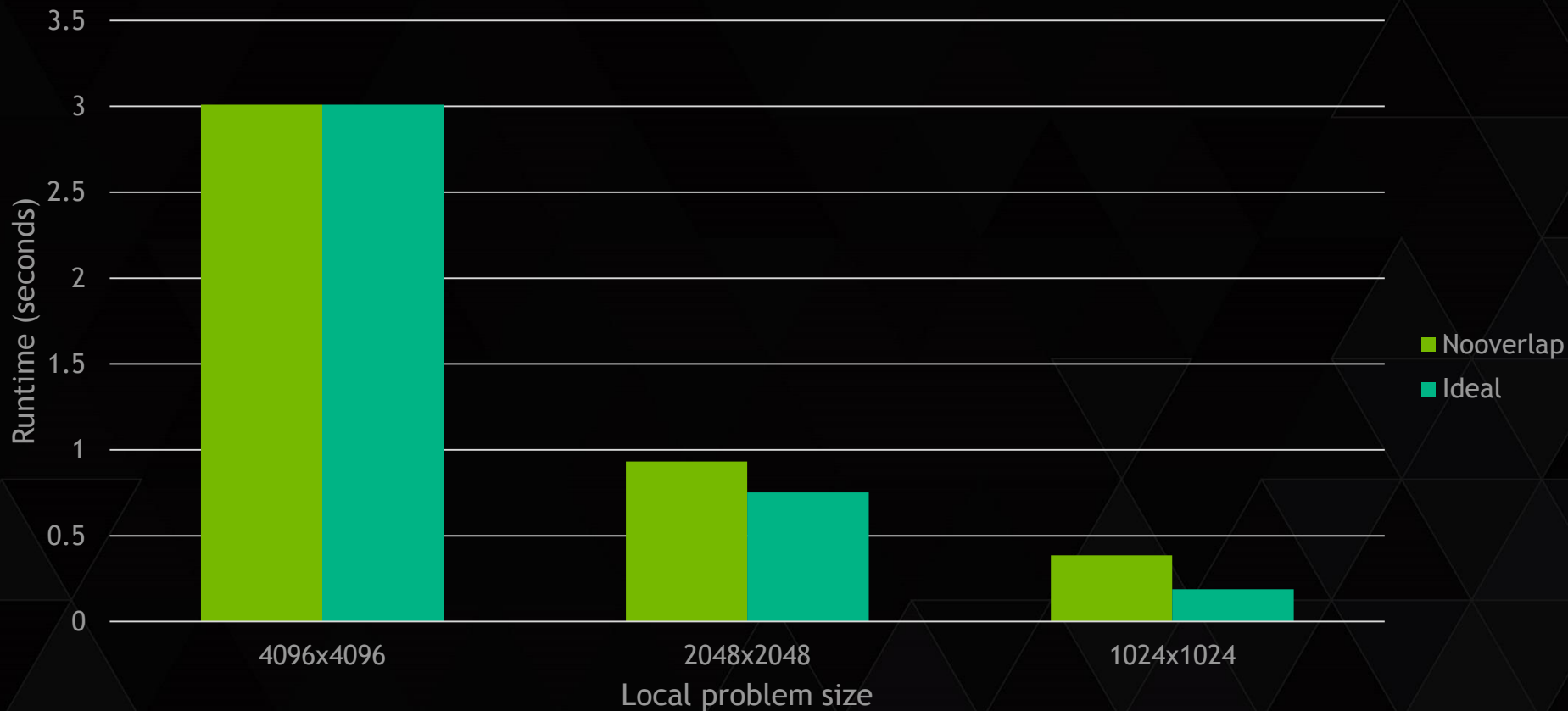
NONE-BLOCKING

```
MPI_Request t_b_req[4];  
#pragma acc host_data use_device ( u_new ) {  
  MPI_Irecv(u_new+offset_top_bondary, m-2, MPI_DOUBLE, MPI_COMM_WORLD, t_b_req);  
  MPI_Irecv(u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, MPI_COMM_WORLD, t_b_req+1);  
  MPI_Isend(u_new+offset_last_row, m-2, MPI_DOUBLE, MPI_COMM_WORLD, t_b_req+2);  
  MPI_Isend(u_new+offset_first_row, m-2, MPI_DOUBLE, 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

MVAPICH2 2.0b - 8 Tesla K20X - FDR IB

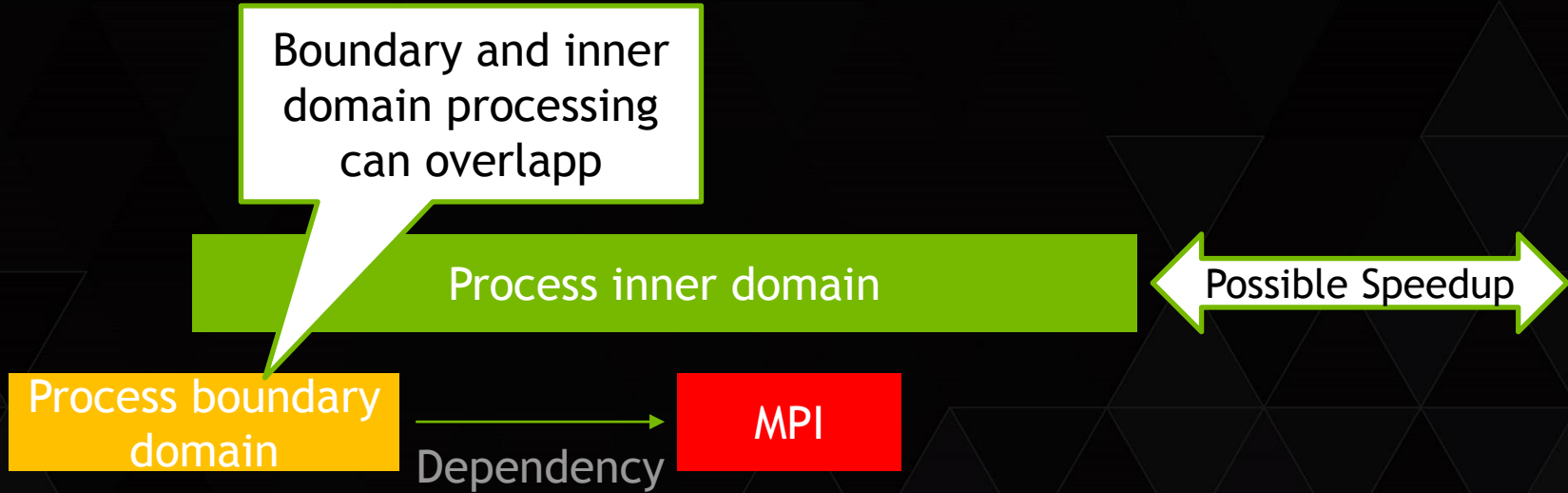


COMMUNICATION + COMPUTATION OVERLAP

No Overlapp



Overlapp



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, from_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];

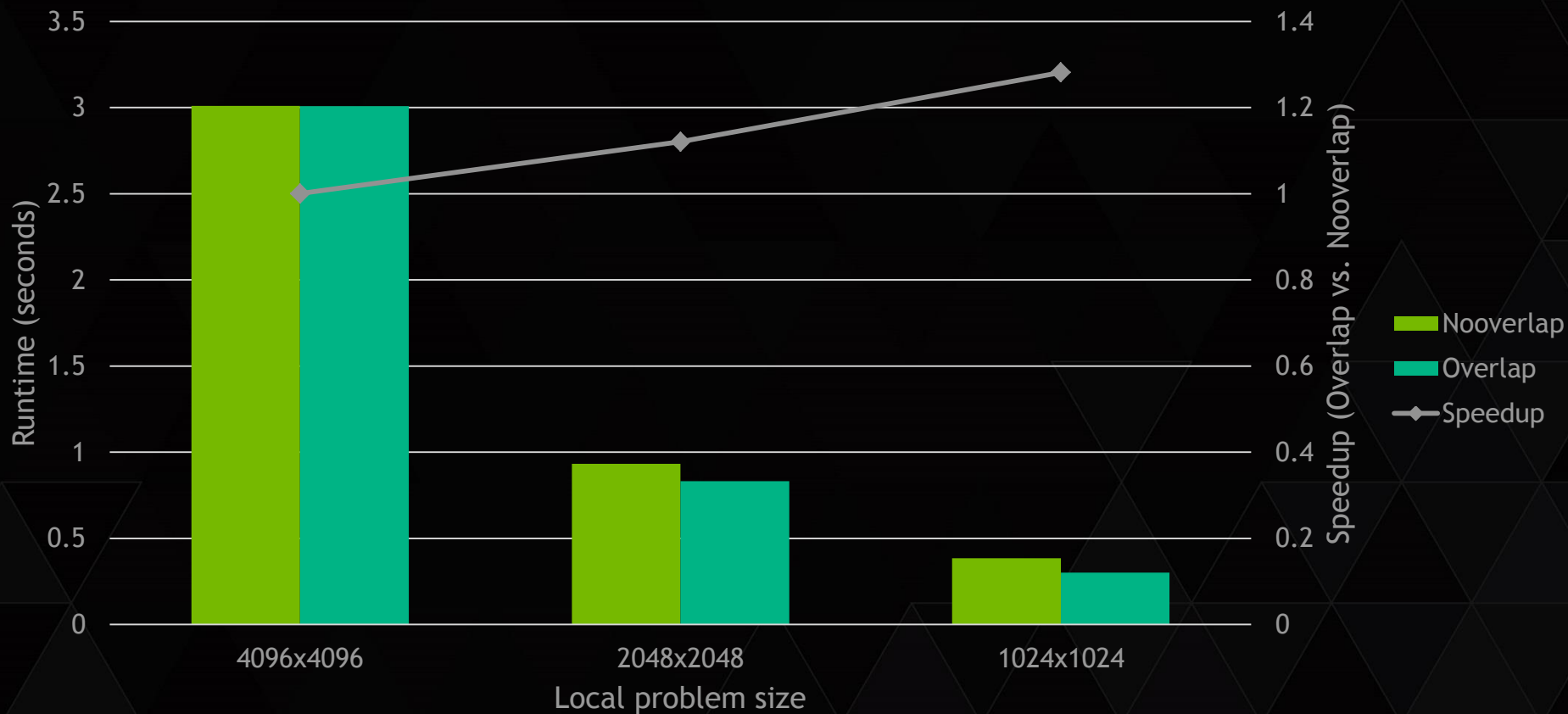
    //Exchange halo with left, right, top and bottom neighbor

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
```

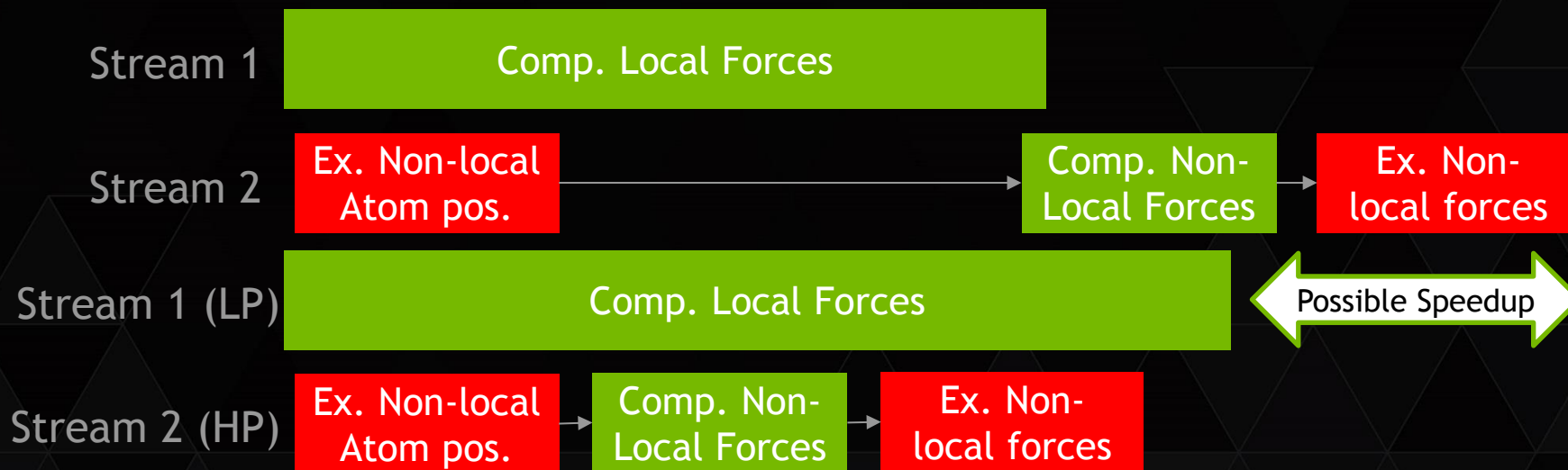
COMMUNICATION + COMPUTATION OVERLAP

MVAPICH2 2.0b - 8 Tesla K20X - FDR IB



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
- ▶ Multiple 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)