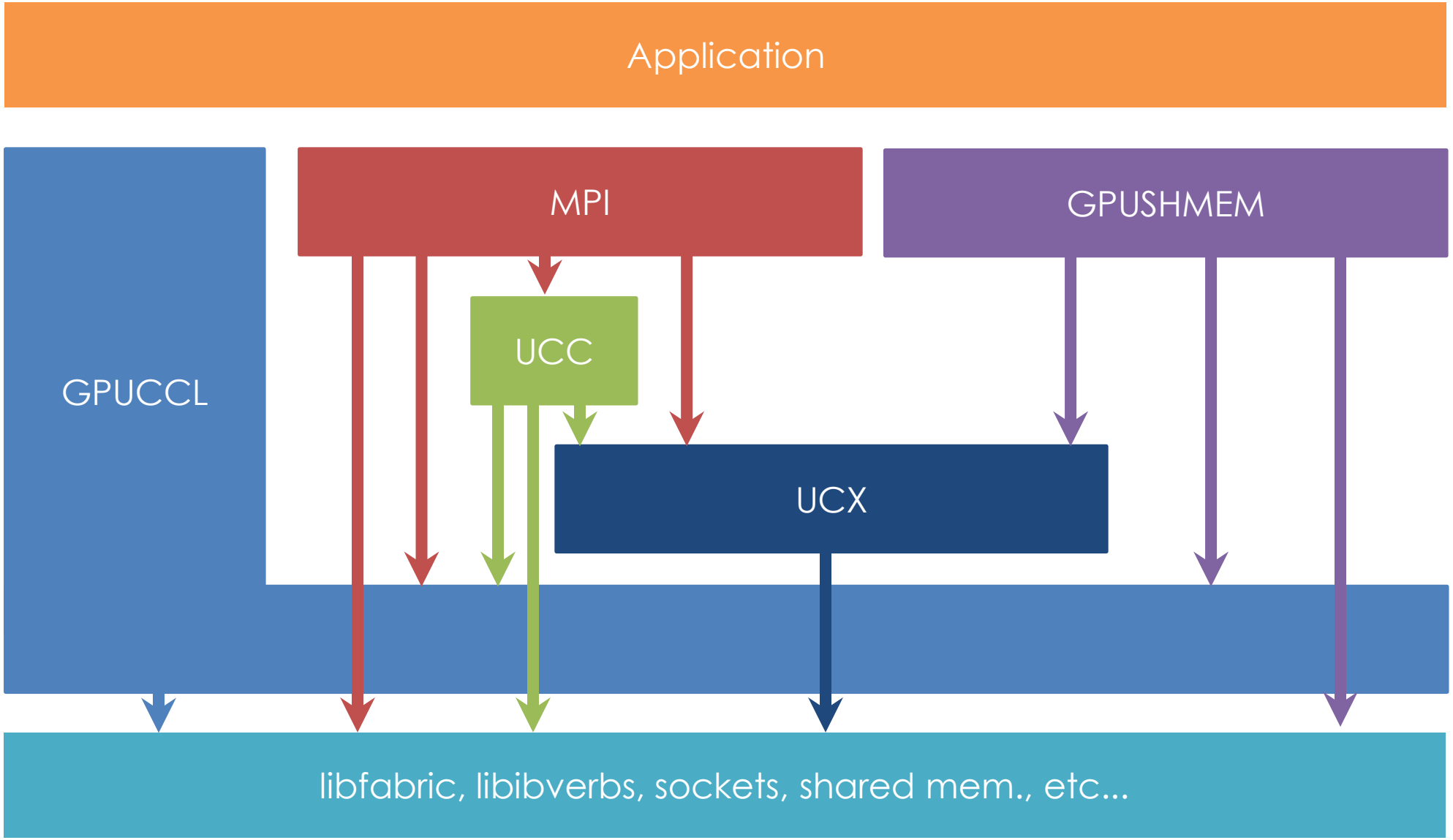
High-Performance Computing

Daniele De Sensi

Software Stack



DISCLAIMER

Most of the following content is CUDA-related, but the same would apply to AMD GPUs
Also, regardless of the specific technology, those techniques (overlap etc…) are general and would apply to different accelerators etc…

Multi GPU Programming in MPI

Slides adapted from slides from:
Lena Oden, Fern Universität in Hagen
Simon Garcia de Gonzalo, Sandia National Laboratories

https://github.com/FZJ-JSC/tutorial-multi-gpu/

Multi-GPU Computing: What you will learn

- CUDA-aware MPI
- Example: Jacobi-Solver
- Under the hood (why you should use CUDA-aware MPI)
  - GPUs in Clusters
  - CUDA Unified Virtual Addressing
  - GPUDirect P2P and GPUDirect RDMA

Message Passing Interface - MPI

- Standard to exchange data between processes via messages
  - Defines API to exchange messages
    - Point to Point: e.g. MPI_Send, MPI_Recv
    - Collectives: e.g. MPI_Reduce, MPI_Allreduce, MPI_Bcast
- Multiple implementations (open source and commercial)
  - Bindings for C/C++, Fortran, Python, …
  - e.g. MPICH, OpenMPI, MVAPICH, IBM Spectrum MPI, Cray MPT, ParaStation MPI, …

Example MPI Program

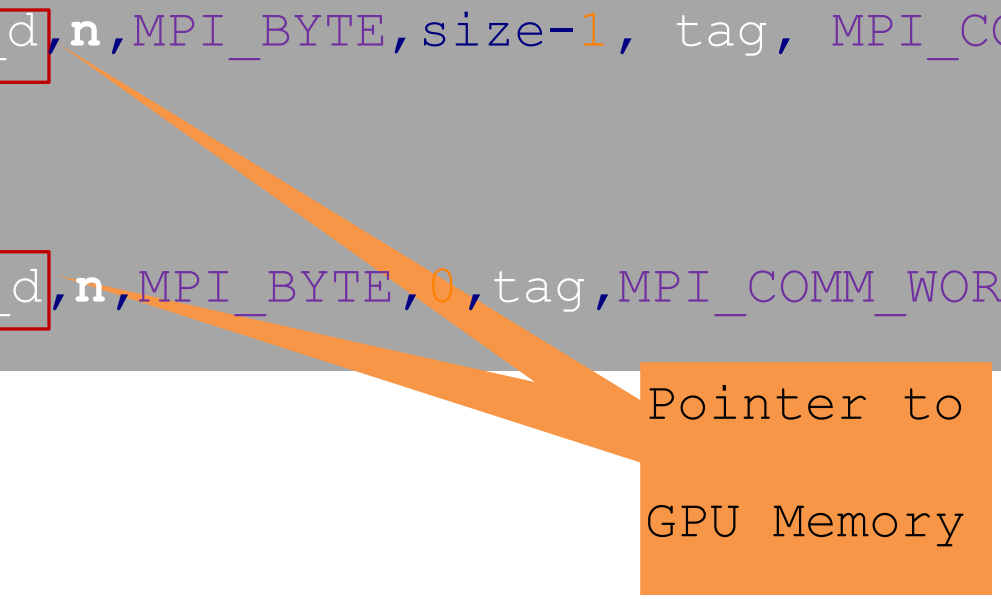
```c
#include <mpi.h>
#include <stdio.h>


int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Finalize();
}
```

```
mpicc -o hello_mpi.out hello_mpi.c
mpirun -n 4 ./hello_mpi.out
```

CUDA-aware MPI

CUDA-aware MPI allows you to use Pointers to GPU-Memory as source and destination

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

//MPI size-1
MPI_Recv(r_buf_d,n,MPI_BYTE,0,tag,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



LAUNCH MPI+CUDA

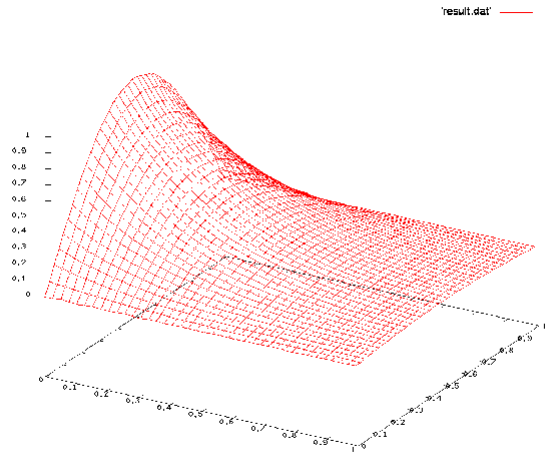
- Launch one process per GPU
- How to use CUDA-aware MPI
  - *MVAPICH*: $MV2_USE_CUDA=1 mpirun -np ${np} ./myapp <args>
  - *Open MPI: CUDA-aware features are enabled per default (using UCX)*
  - *Cray:* MPICH_RDMA_ENABLED_CUDA
  - *IBM Spectrum MPI*: $mpirun **-gpu** -np ${np} ./myapp <args>
  - *ParaStation MPI* (using UCX): $PSP_CUDA=1 mpirun -np ${np} ./myapp <args>

How to compile

```
nvcc -o my_kernel.o $(NVCC_FLAGS) my_kernels.cu -c

mpicc -o my_multiGPUapp -lcudart my_kernel.o my_multiGPUapp.c
```

Example: Jacobi Solver

- Solves the 2D-Laplace Equation on a rectangle
- $\Delta \boldsymbol{u}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{0} \; \forall \; (\boldsymbol{x}, \boldsymbol{y}) \in \Omega \backslash \boldsymbol{\delta}\Omega$

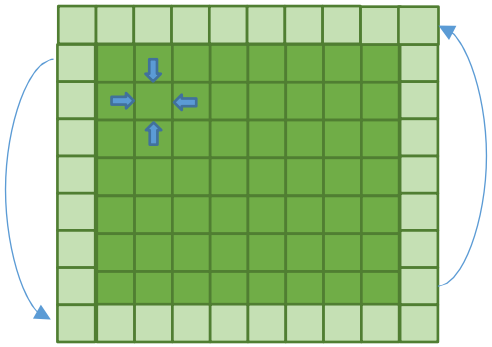


- Dirichlet boundary conditions on left/right boundaries (constant values on boundaries)
- Reflecting boundaries on top and bottom
- Iterative solver: u(t+1) = f(u(t))

Example: Jacobi solver

While not converged do Jacobi step:

```
int iy = blockIdx.y * blockDim.y + threadIdx.y
int ix = blockIdx.x * blockDim.x + threadIdx.x
if (iy < ny-1 && ix < nx-1) {
    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;
}
```

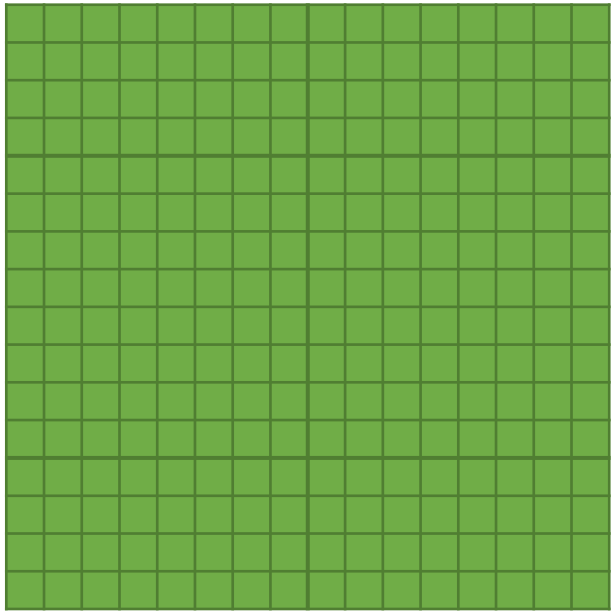


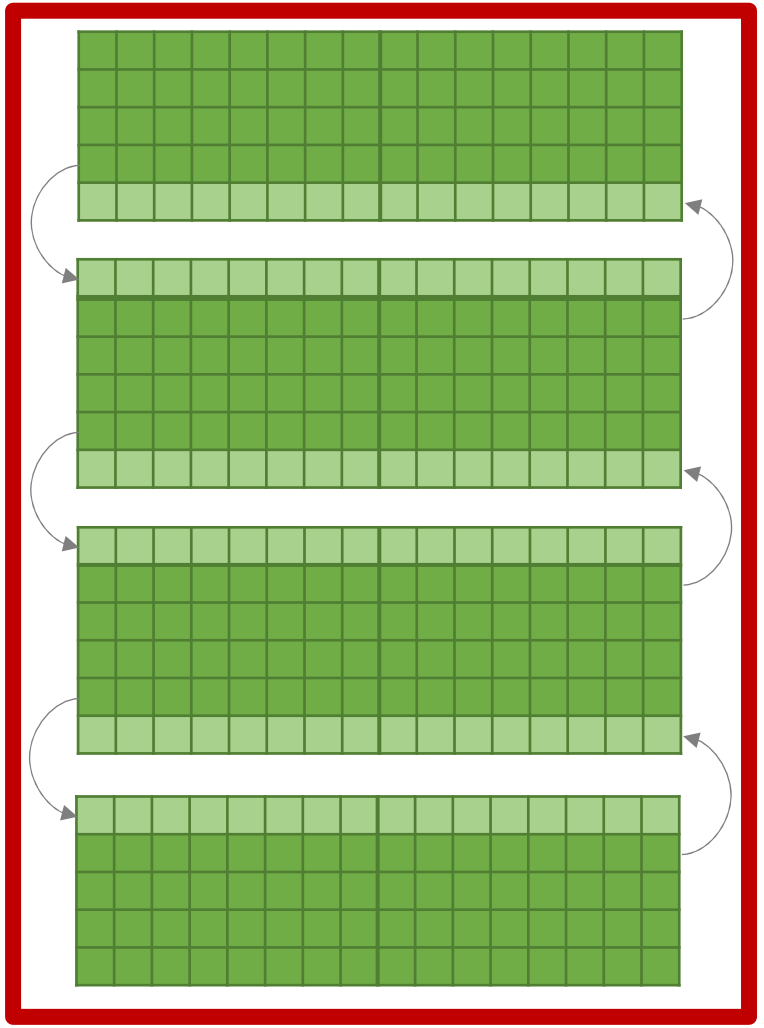
apply boundary Condition

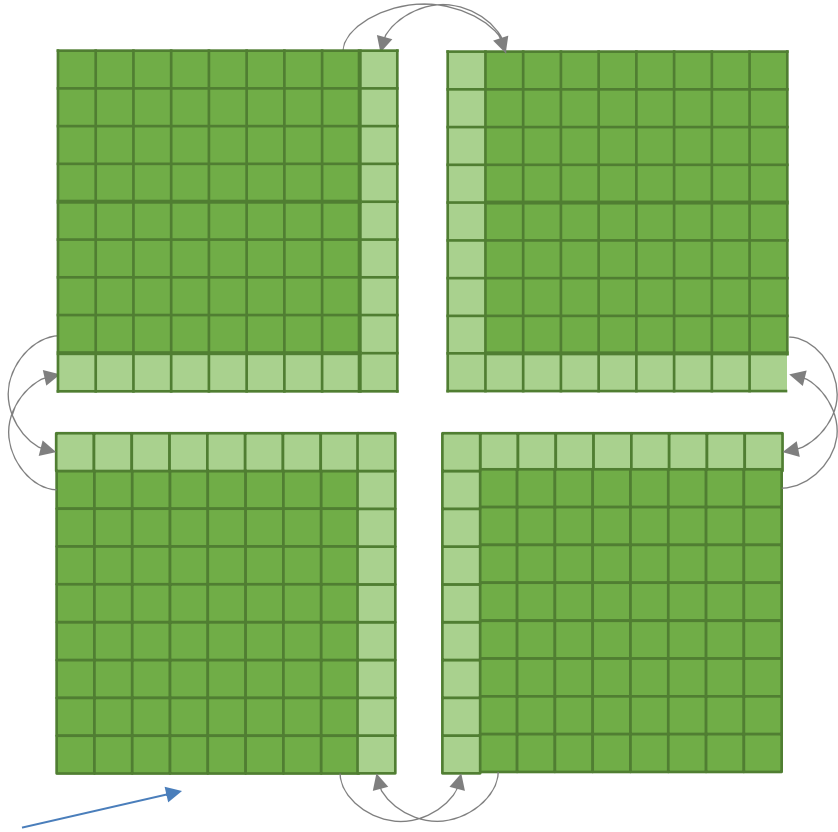
swap a_new and a

next iteration

Domain Decomposition







* Minimize number of neighbors: Communicate to fewer neighbors
* Optimal for latency bound communication,
* Contiguous Transfers

* Minimize surface area/volume ratio: Communicate less data
* Optimal for bandwidth bound communication
* Non-Contiguous Transfers

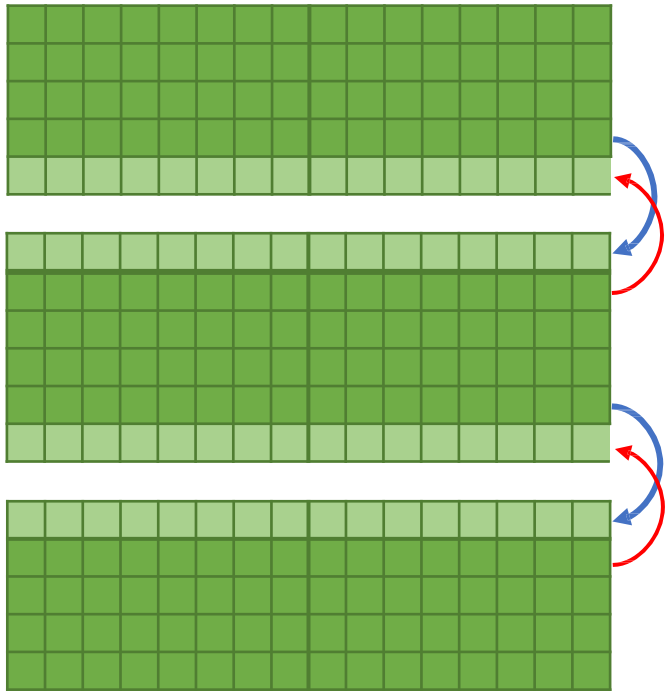
Jacobi example: Top and Bottom Boundaries

bottom neighbor

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

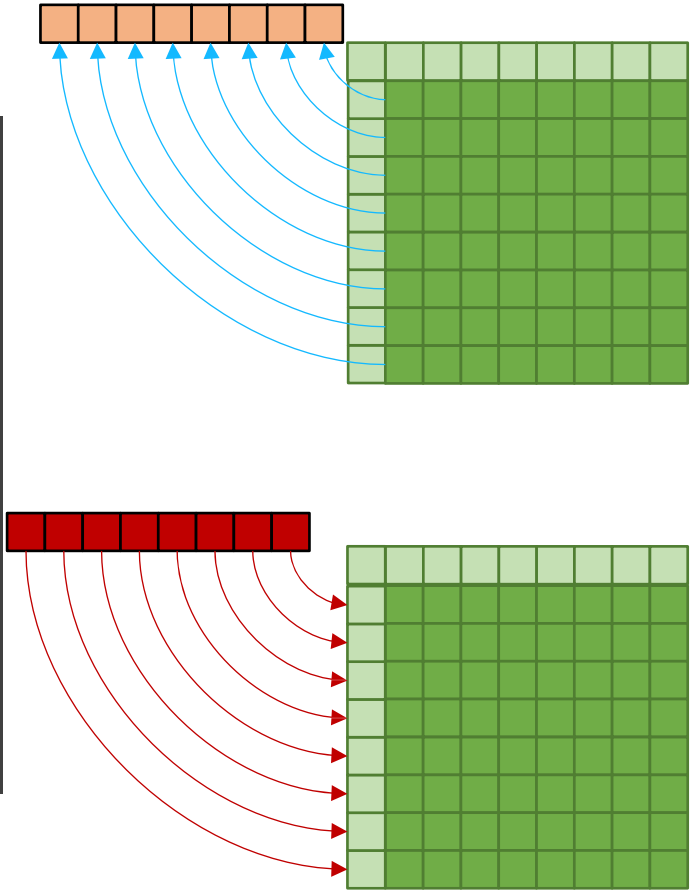
top neighbor

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

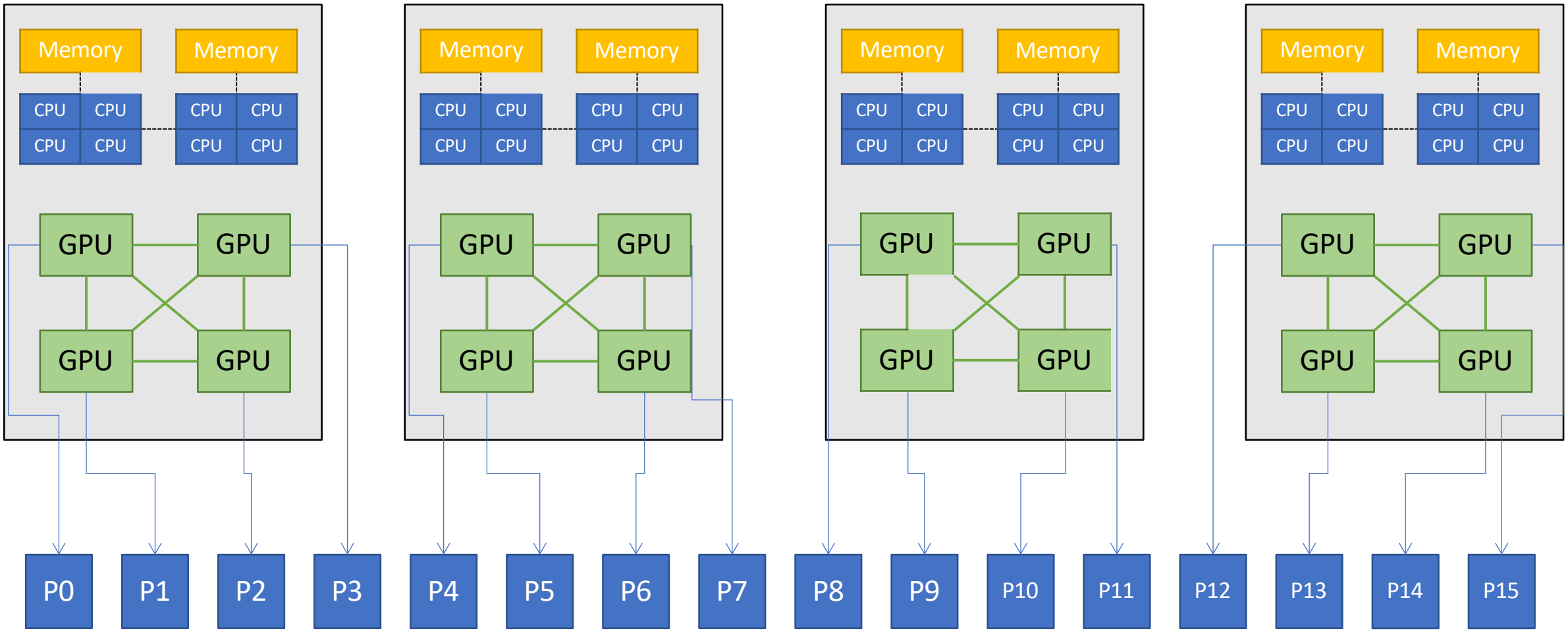


Jacobi: Left (and right) neighbours

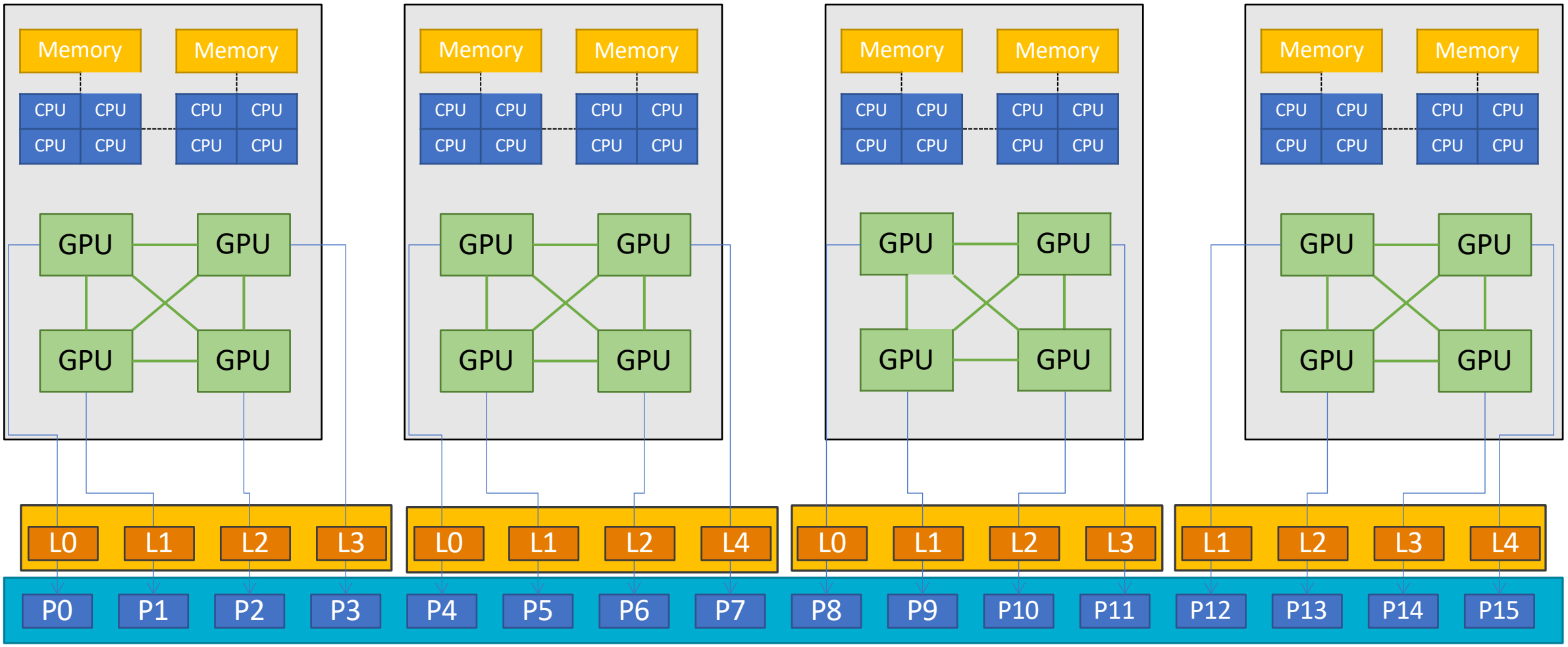
```
//right neighbor omitted
pack<<<gs,bs,0,s>>>(to_left_d, u_new_d, n, m);
cudaStreamSynchronize(s);
MPI_Sendrecv(to_left_d, n-2, MPI_DOUBLE, l_nb, 0,
             from_right_d, n-2, MPI_DOUBLE, r_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE );
unpack<<<gs,bs,0,s>>>(u_new_d, from_right_d, n, m);
cudaStreamSynchronize(s);
```



Process Mapping on Multi GPU Systems: One GPU per Process



Process Mapping on Multi GPU Systems: One GPU per Process

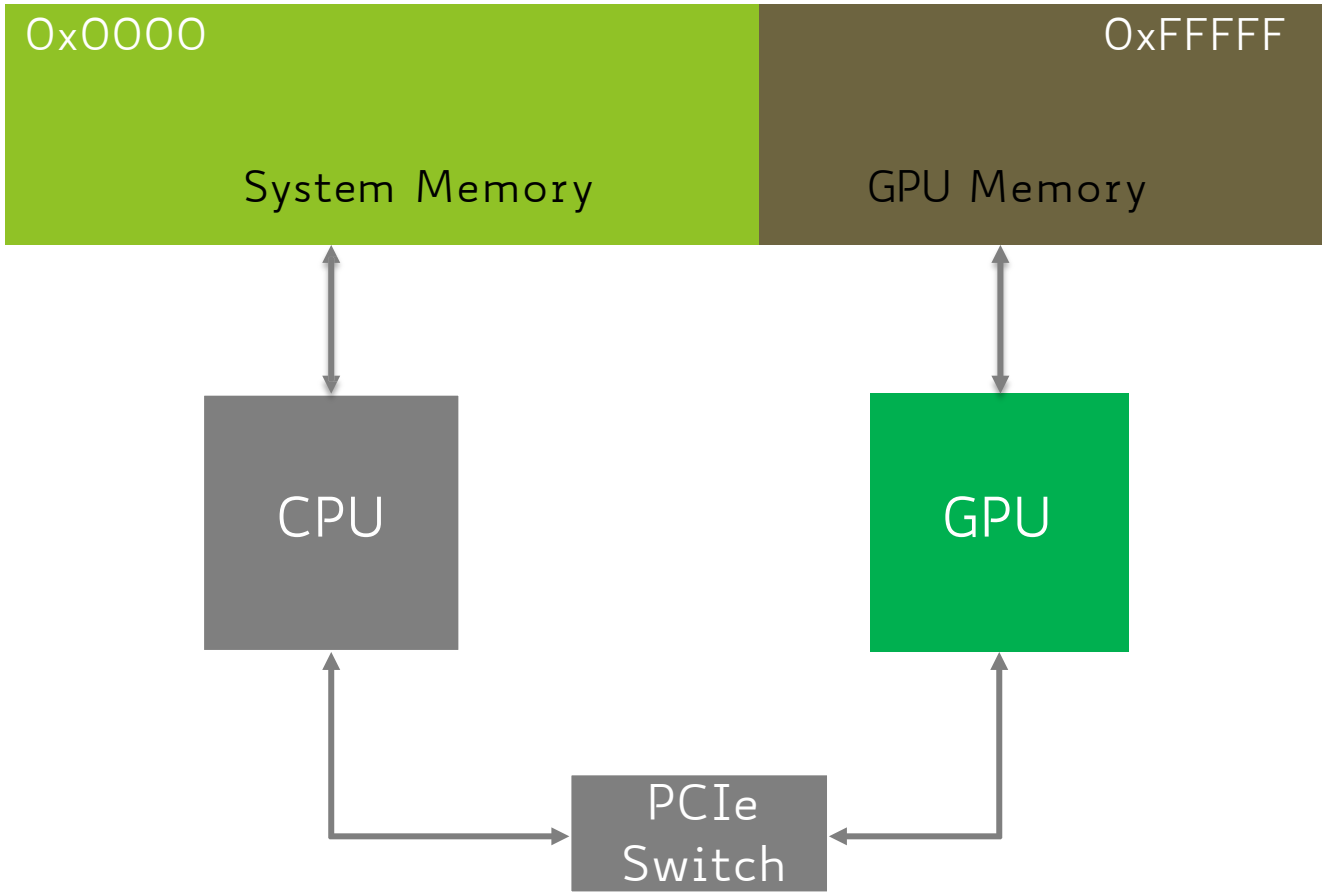


Distribute GPUs to local Nodes

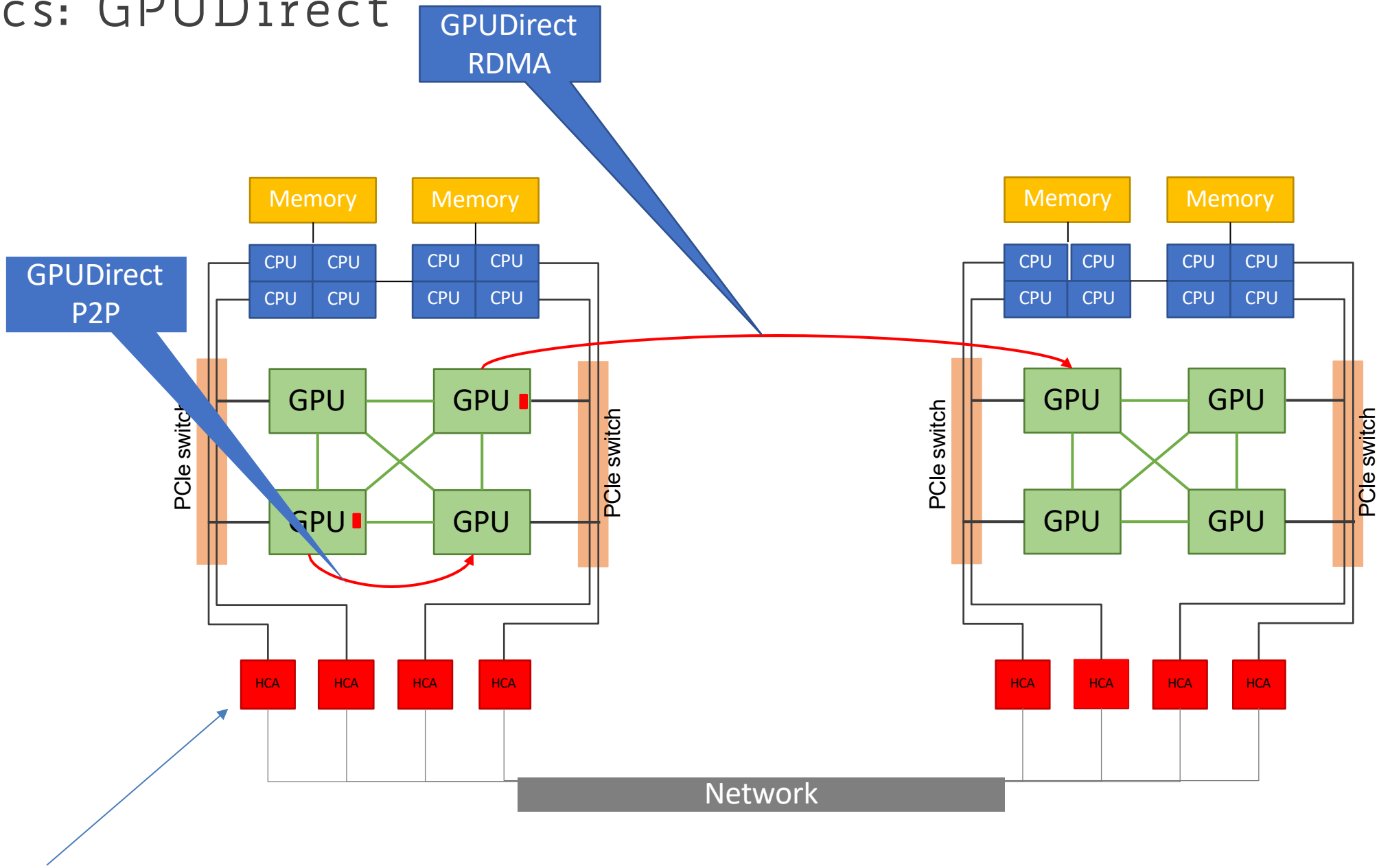
```
MPI_Comm local_comm;
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank,
                    MPI_INFO_NULL, &local_comm);
int local_rank = -1;
MPI_Comm_rank(local_comm,&local_rank);
MPI_Comm_free(&local_comm);


int num_devs = 0;
cudaGetDeviceCount(&num_devs);
cudaSetDevice(local_rank%num_devs);
```

CUDA 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)
- Supported on devices with compute capability 2.0+ for
  - 64-bit applications on Linux and Windows (+TCC)

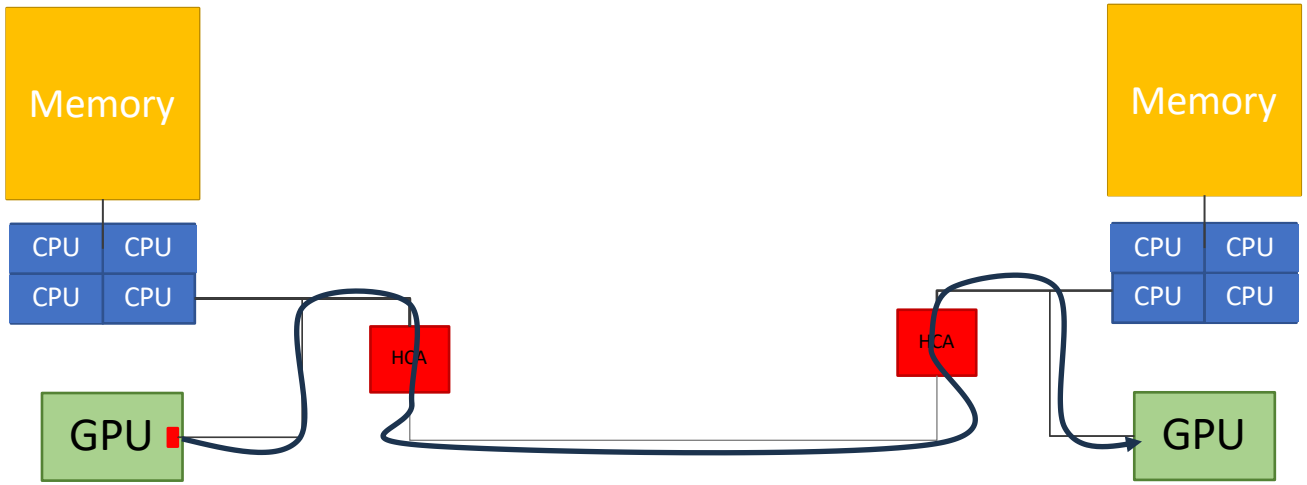


Basics: GPUDirect



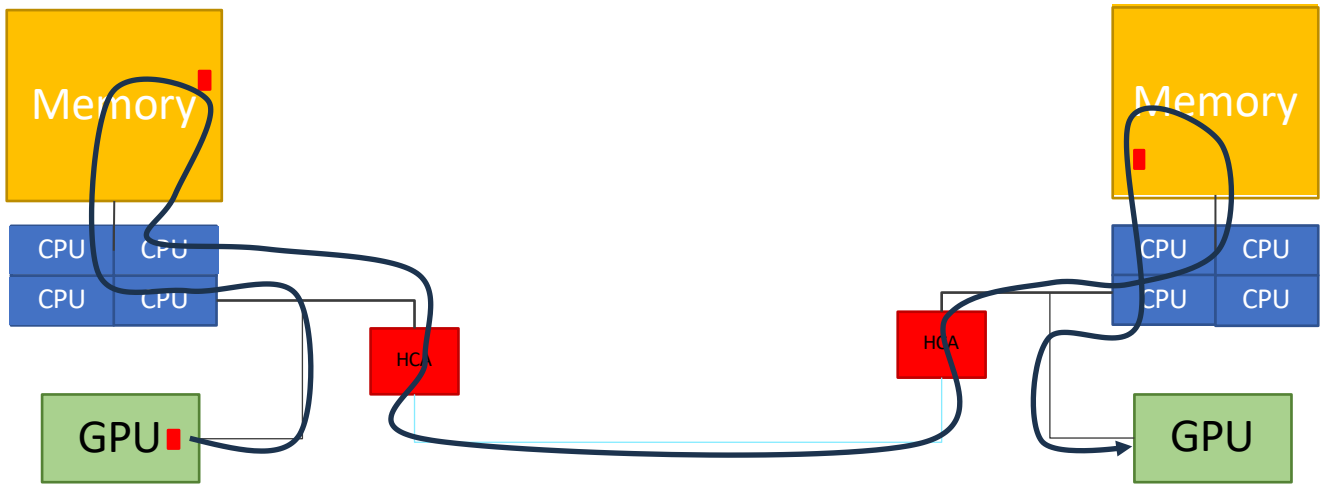
HCA = Host Channel Adapter = NIC (Network Card Interface)

CUDA-aware MPI with GPUDirect RDMA



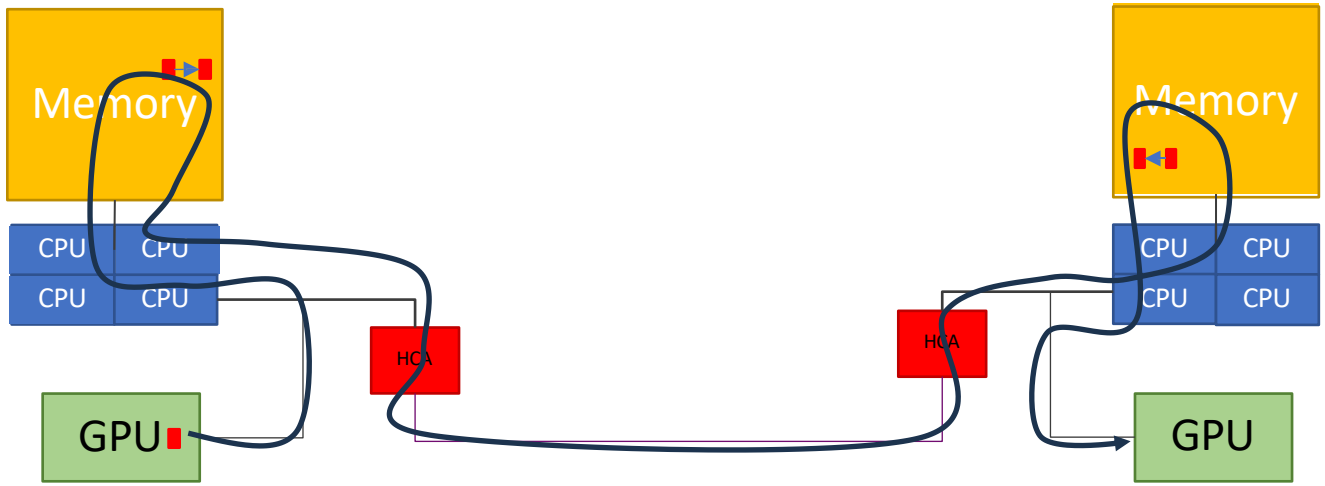

```
MPI_Send(s_buf_d,size,MPI_BYTE,1,tag,MPI_COMM_WORLD);
MPI_Recv(r_buf_d,size,MPI_BYTE,0,tag,MPI_COMM_WORLD,&stat);
```

CUDA-aware MPI without GPUDirect RDMA




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

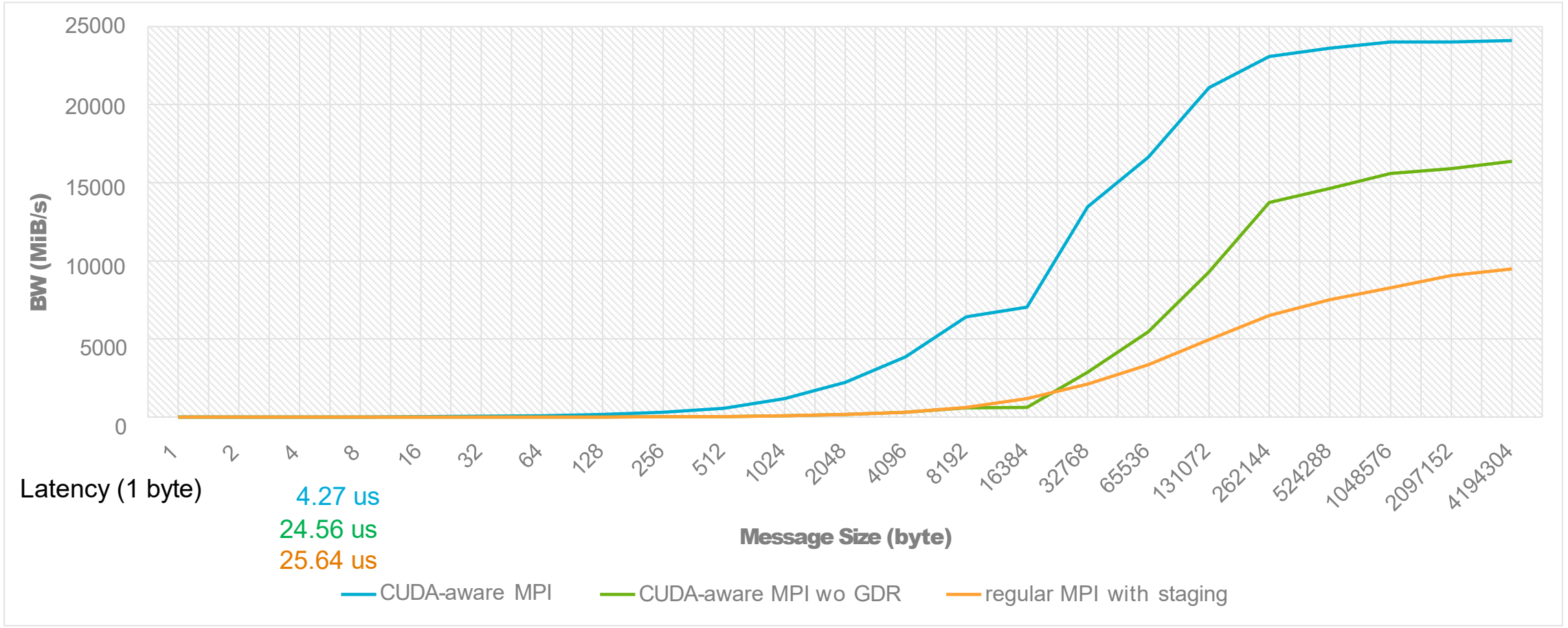
Regular MPI




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

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

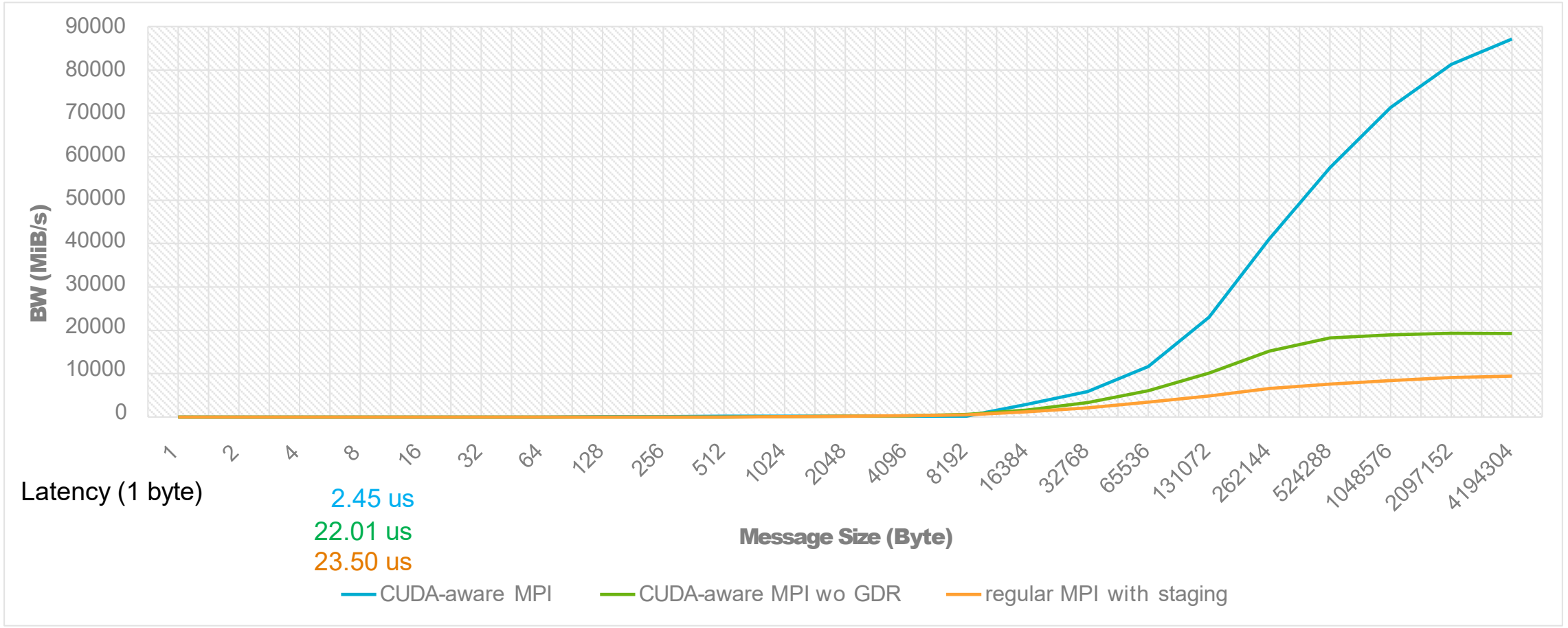
Performance Results GPUDirect RDMA

OpenMPI 4.1.0RC1 + UCX 1.9.0 on JUWELS Booster



Performance Results GPUDirect P2P

OpenMPI 4.1.0RC1 + UCX 1.9.0 on JUWELS-Booster



Summary

- CUDA-aware MPI allows efficient communication for multi-GPU applications
  - Allows MPI-communication operations from GPU memory buffers
  - Simplified programming
  - Use GPUDirect technologies for performance
  - Minimizes data copies
- Most MPI versions have support for CUDA-aware MPI
- Best practice: One process per rank
  - Use local communicator in MPI

Hands-On

https://github.com/FZJ-JSC/tutorial-multi-gpu/

Instructions

- Join the assignment on Github Classroom: https://classroom.github.com/a/D9grIARs
  - At the moment, there is no auto-grading yet, check correctness by yourself
- Access the cluster and clone the repository
- Fill the «TODOs» in the jacobi.cu code
- Run "source /home/guest/init-hpc.sh" to load the MPI, CUDA compilers etc…
- Run «make run» to compile and run the code (by default on 4 GPUs, one GPU per node)

Solution

```
int num_devices = 0;
// TODO: Get the available GPU devices into `num_devices` and use it and the current rank to set the active GPU.
CUDA_RT_CALL(cudaGetDeviceCount(&num_devices));
CUDA_RT_CALL(cudaSetDevice(local_rank%num_devices));
CUDA_RT_CALL(cudaFree(0));
```

```
//TODO: Compute top and bottom neighbor, use reflecting/periodic boundaries. This means rank 0 and rank (size-1)
// exchange data
const int top = rank > 0 ? rank - 1 : (size - 1);
const int bottom = (rank + 1) % size;
```

Solution

```
// TODO: Use MPI_Sendrecv to exchange the data with the top and bottom neighbors
// Use CUDA-aware MPI here, i.e. receive the data directly in a_new on the GPU and send it from there
// without manually copying the data.

// The first newly calculated row ('iy_start') is sent to the top neigbor and the bottom boundary row
// (`iy_end`) is received from the bottom process.
// The last calculated row (`iy_end-1`) is send to the bottom process and the top boundary (`0`) is received from the top
// Don't forget to synchronize the computation on the GPU before starting the data transfer
CUDA_RT_CALL(cudaEventSynchronize(compute_done));
PUSH_RANGE("MPI", 5)
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));
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));
POP_RANGE
```

Performance

```
Jacobi relaxation: 1000 iterations on 16384 x 16384 mesh with norm check every 1 iterations
    0, 31.999022
  100, 0.897983
  200, 0.535685
  300, 0.395651
  400, 0.319039
  500, 0.269961
  600, 0.235510
  700, 0.209829
  800, 0.189854
  900, 0.173818
Num GPUs: 4.
16384x16384: 1 GPU:   6.3126 s, 4 GPUs:   1.9474 s, speedup:     3.24, efficiency:    81.04
```

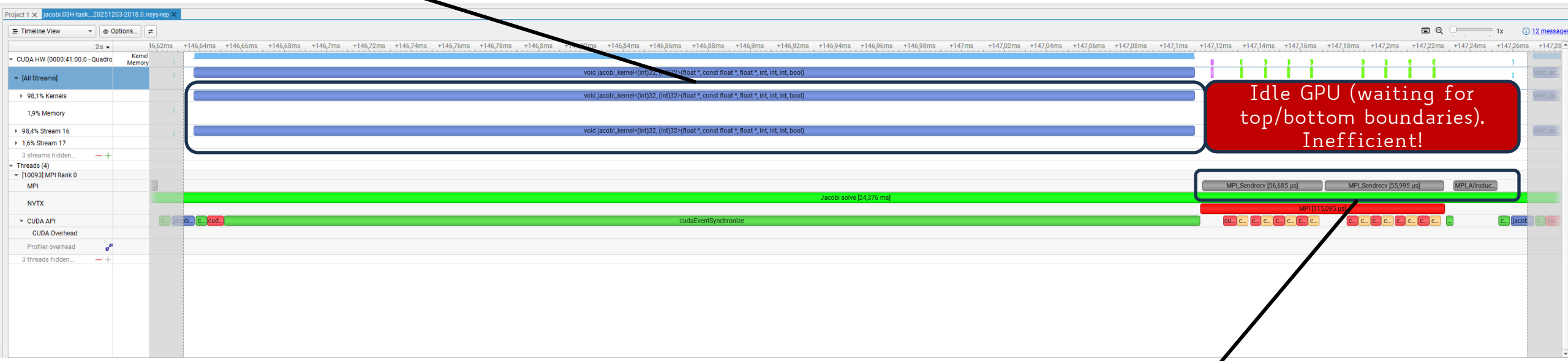
Profiling

Instructions

- Install NVIDIA nsight systems on your laptop
- Run «source /home/guest/init-hpc.sh»
- Run «make profile» to generate nsys profile files
- Copy back the profile files from the cluster to your laptop, and open them with nsight systems

Profile

Kernel Execution

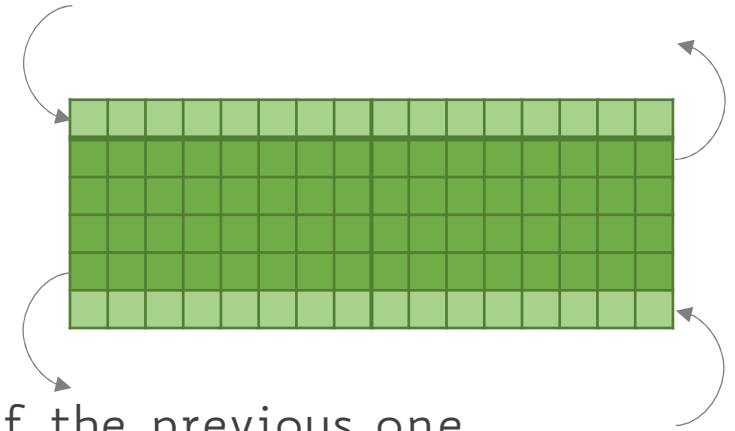


MPI Communication

`assignment-3-mgpu-mpi/profiles/`

Intuition

- Separate computation of top/bottom borders from the inner part
- In this way, we can send/recv immediately the borders while we compute the inner part
- So we would like to have (in parallel):
  compute_top_kernel<<<…>>>
  compute_bottom_kernel<<<…>>>
  compute_inner_kernel<<<…>>>
- Then:
  wait for the top kernel to finish
  sendrecv the top border
  wait for the bottom kernel to finish
  sendrecv the bottom border
- Issues:
  - By default, each kernel won't start before the termination of the previous one
  - cudaDeviceSynchronize waits for all the running kernels to finish (not only for the top/bottom one



CUDA Streams

(Let's forget about Multi-GPU for a moment and let's get back to a single GPU scenario)

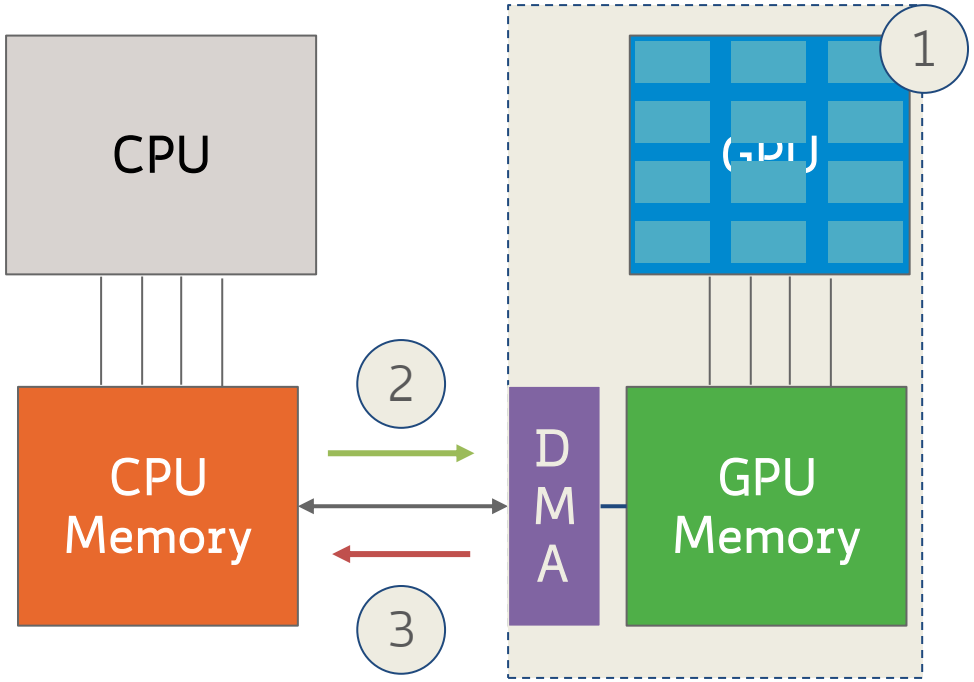
Slide courtesy: Tiziano De Matteis (Vrijie Univ. Amsterdam)

Parallelism In System Architecture

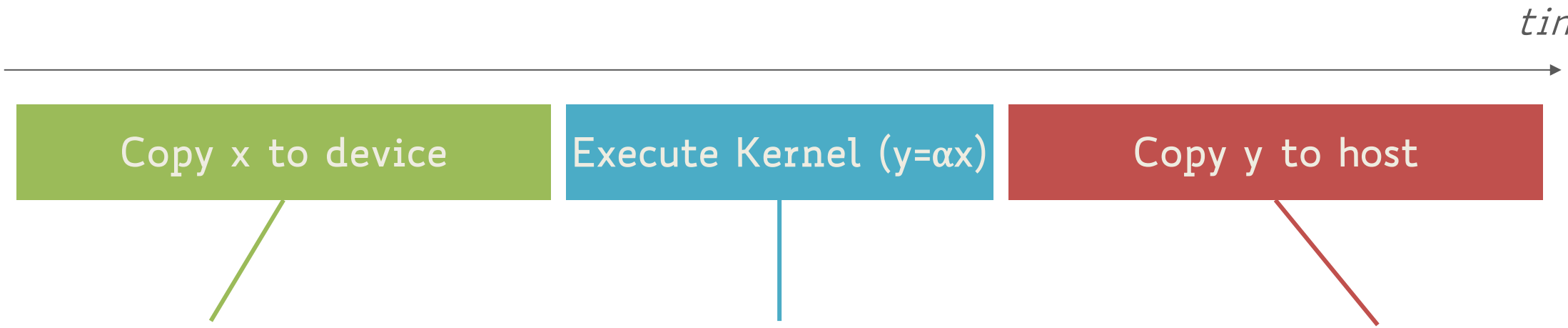
Typical GPU based systems can do multiple things at the same time:

1. Execute a grid on the GPU
2. Copy data from the host to the device
3. Copy data from the device to the host

```
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
Run time limit on kernels:                     No
```



Vector Scaling System Utilization



GPU is idle, and we are not copying to the host

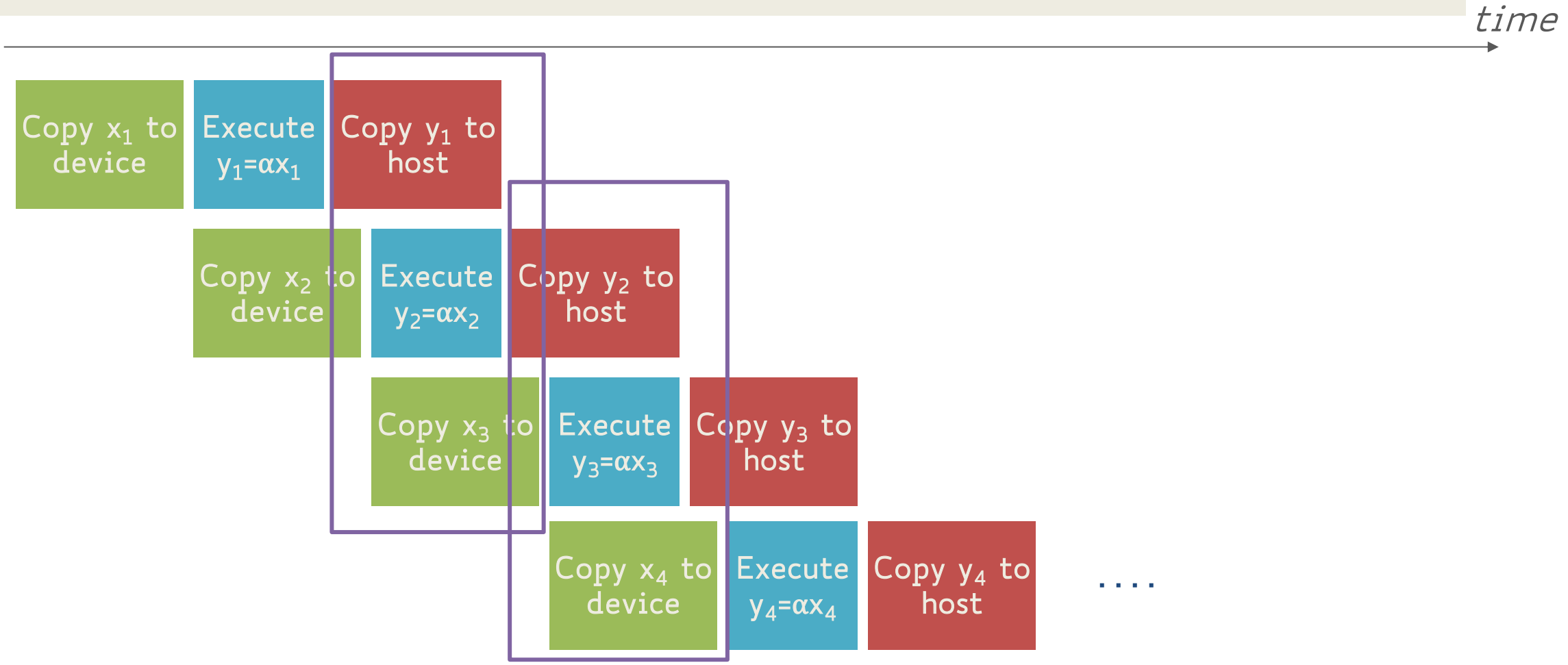
Not utilizing copy from the host and copy to the host

GPU is idle, and we are not copying from the host

$$T = T_{H2D} + T_{COMP} + T_{D2H}$$

Q: how to leverage system parallelism?

PIPELINING



Q: before the execution time was given by $T_{H2D} + T_{COMP} + T_{D2H}$. Assuming no extra overhead is introduced with pipelining, what is the now the execution time?

A: $T_{H2D} + (1/4)*T_{COMP} + (1/4)*T_{D2H}$

CUDA Streams and Asynchronous Memcpy

CUDA **Streams** support managed concurrent execution of CUDA API functions (e.g., kernel execution and Memcpy-s). They are FIFO command queues:

- Operations (tasks) in the same stream are serialized
- Operations (tasks) in different streams are executed in parallel

If no stream is specified, tasks go in the **default stream.** Streams allow you to implement a sort of **task parallelism**.

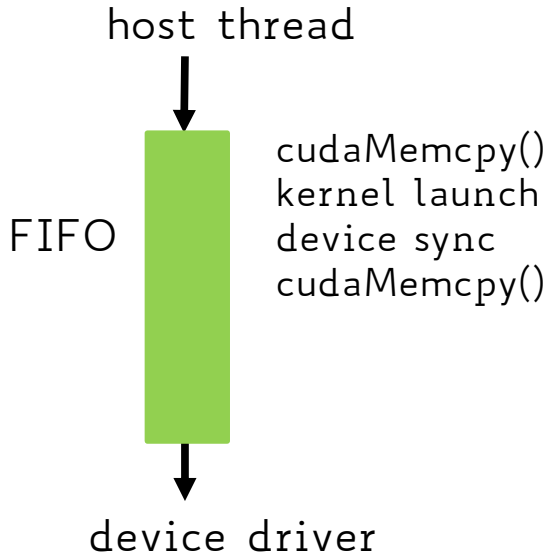
Out-of-order execution for tasks from different streams

To overlap memory copies with host execution, we need to use **asynchronous memory copies. Pinned memory is necessary for performance reasons.**

(Check PMC slides on why/how pinned memory is used)

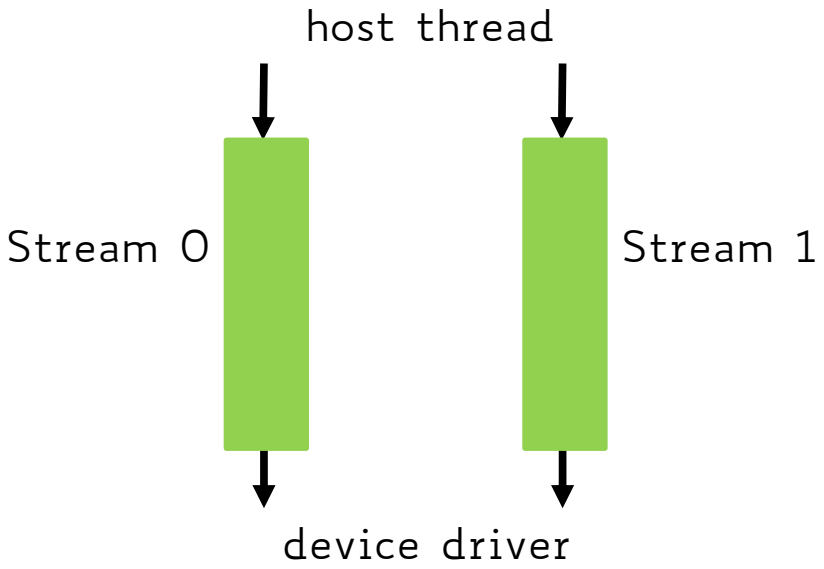
Streams

- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.



Streams

- To allow concurrent copying and kernel execution, use multiple queues, called "streams"



CUDA Streams and Asynchronous Memcpy API

Streams have type cudaStream_t. To create/destroy/syncronize a stream:

```
cudaError_t cudaStreamCreate(cudaStream_t* pStream);
cudaError_t cudaStreamDestroy(cudaStream_t Stream);
cudaError_t cudaStreamSynchronize(cudaStream_t Stream);
```

To create an asynchronous copy in a stream:

```
cudaError_t cudaMemcpyAsync( void* dst, const void* src, size_t count,
                             cudaMemcpyKind kind, cudaStream_t stream = 0)
```

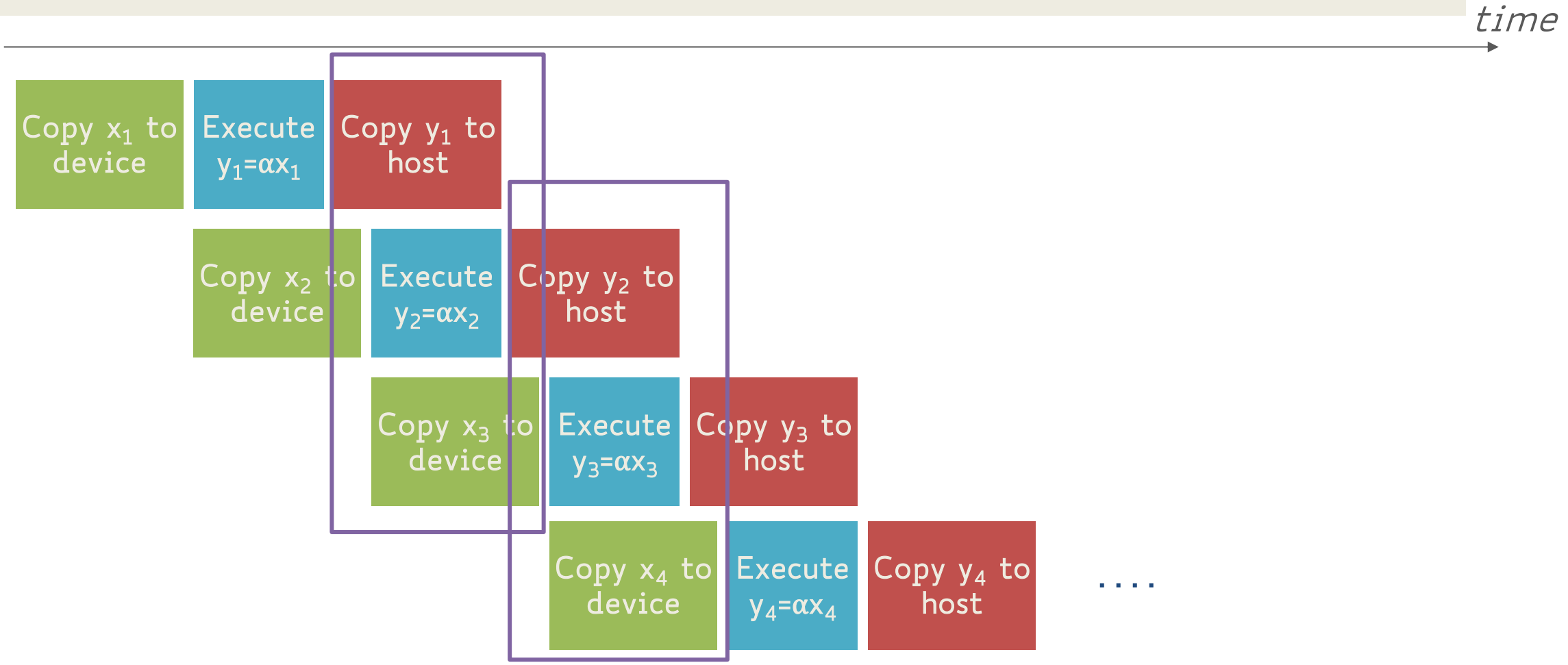
To memset on a stream:

cudaMemsetAsync: same as cudaMemset, with the addition of a stream parameter

To launch a grid on a specific stream:

```
my_kernel<<<grid_dim, block_dim, 0, stream>>>(...)
```

PIPELINING



Q: before the execution time was given by $T_{H2D} + T_{COMP} + T_{D2H}$. Assuming no extra overhead is introduced with pipelining, what is the now the execution time?

A: $T_{H2D} + (1/4)*T_{COMP} + (1/4)*T_{D2H}$

The Code

```
// Kernel code Unchanged ...

int main(int argc, char **argv){
  float *x_h, *y_,*x_d, *y_d;
  // Allocate pinned and device memory ...
```

```
}
```

The Code

```
// Kernel code Unchanged ...

int main(int argc, char **argv){
  float *x_h, *y_,*x_d, *y_d;
  // Allocate pinned and device memory ...

  const unsigned int num_streams = 32;
  cudaStream_t streams[num_streams];
  for (int i = 0; i < num_streams; i++)
    cudaStreamCreate(&streams[i]);









}
```

The Code

```
// Kernel code Unchanged ...

int main(int argc, char **argv){
  float *x_h, *y_,*x_d, *y_d;
  // Allocate pinned and device memory ...

  const unsigned int num_streams = 32;
  cudaStream_t streams[num_streams];
  for (int i = 0; i < num_streams; i++)
    cudaStreamCreate(&streams[i]);

  unsigned int num_segments = num_streams;
  unsigned int segment_size = (N + num_segments - 1) / num_segments;









}
```

The Code

```
// Kernel code Unchanged ...

int main(int argc, char **argv){
  float *x_h, *y_,*x_d, *y_d;
  // Allocate pinned and device memory ...

  const unsigned int num_streams = 32;
  cudaStream_t streams[num_streams];
  for (int i = 0; i < num_streams; i++)
    cudaStreamCreate(&streams[i]);

  unsigned int num_segments = num_streams;
  unsigned int segment_size = (N + num_segments - 1) / num_segments;

  for (int i = 0; i < num_segments; i++) {
```

```
}
```

The Code

```
// Kernel code Unchanged ...

int main(int argc, char **argv){
  float *x_h, *y_,*x_d, *y_d;
  // Allocate pinned and device memory ...

  const unsigned int num_streams = 32;
  cudaStream_t streams[num_streams];
  for (int i = 0; i < num_streams; i++)
    cudaStreamCreate(&streams[i]);

  unsigned int num_segments = num_streams;
  unsigned int segment_size = (N + num_segments - 1) / num_segments;

  for (int i = 0; i < num_segments; i++) {
    unsigned int start = i * segment_size;
    unsigned int end = (start + segment_size) < N ? start + segment_size : N;
    unsigned int this_segment_size = end - start;




}
```

The Code

```
// Kernel code Unchanged ...

int main(int argc, char **argv){
  float *x_h, *y_,*x_d, *y_d;
  // Allocate pinned and device memory ...

  const unsigned int num_streams = 32;
  cudaStream_t streams[num_streams];
  for (int i = 0; i < num_streams; i++)
    cudaStreamCreate(&streams[i]);

  unsigned int num_segments = num_streams;
  unsigned int segment_size = (N + num_segments - 1) / num_segments;

  for (int i = 0; i < num_segments; i++) {
    unsigned int start = i * segment_size;
    unsigned int end = (start + segment_size) < N ? start + segment_size : N;
    unsigned int this_segment_size = end - start;

    // memcpy async
    cudaMemcpyAsync(x_d + start, x_h + start, this_segment_size * sizeof(float), cudaMemcpyHostToDevice, streams[i]);

    scal<<<(segment_size + 128-1)/128, 128, 0, streams[i]>>>(this_segment_size, alpha, x, y);

    // memcpy async
    cudaMemcpyAsync(y_h + start, y_d + start, this_segment_size * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);
  }


}
```

The Code

```
// Kernel code Unchanged ...

int main(int argc, char **argv){
  float *x_h, *y_,*x_d, *y_d;
  // Allocate pinned and device memory ...

  const unsigned int num_streams = 32;
  cudaStream_t streams[num_streams];
  for (int i = 0; i < num_streams; i++)
    cudaStreamCreate(&streams[i]);

  unsigned int num_segments = num_streams;
  unsigned int segment_size = (N + num_segments - 1) / num_segments;

  for (int i = 0; i < num_segments; i++) {
    unsigned int start = i * segment_size;
    unsigned int end = (start + segment_size) < N ? start + segment_size : N;
    unsigned int this_segment_size = end - start;

    // memcpy async
    cudaMemcpyAsync(x_d + start, x_h + start, this_segment_size * sizeof(float), cudaMemcpyHostToDevice, streams[i]);

    scal<<<(segment_size + 128-1)/128, 128, 0, streams[i]>>>(this_segment_size, alpha, x, y);

    // memcpy async
    cudaMemcpyAsync(y_h + start, y_d + start, this_segment_size * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);
  }
  cudaDeviceSynchronize();
  // Free memory, destroy streams...
}
```

Overlapping Communication and Computation

How small the segments should be?

- There are filling/emptying phases, slowest stage dominates
- We could be tempted to make very fine grained segments, but:
  - At a certain point the compute will be too small to fully utilize the GPU
  - You have overheads (kernel launching time, DMA setup, …)

So, have "**moderately**" sized blocks

We can use pinned memory and streams to overlap communication and computation

- This requires changing the code
- Be careful with the use of pinned memory
- Properly size the segments

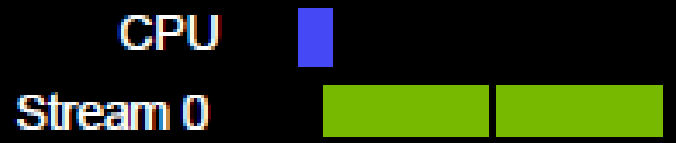
DEFAULT STREAM

- Unless otherwise specified all calls are placed into a default stream
  - Often referred to as “Stream 0”
- Stream 0 has special synchronization rules
  - Synchronous with all streams
    - Operations in stream 0 cannot overlap other streams
- Exception: Streams with non-blocking flag set
  - `cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking)`
  - Use to get concurrency with libraries out of your control (e.g. MPI)

KERNEL CONCURRENCY

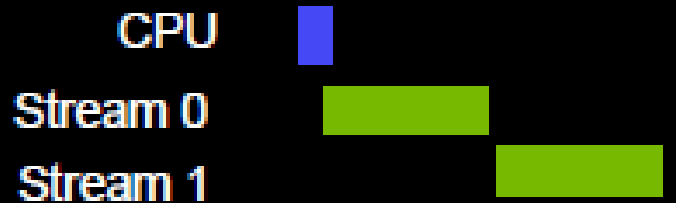
- Assume foo only utilizes 50% of the GPU
- Default stream

```
foo<<<blocks,threads>>>();
foo<<<blocks,threads>>>();
```



- Default & user streams

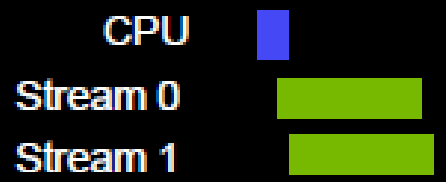
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
- Default & user streams

```
cudaStream_t stream1;
cudaStreamCreateWithFlags(&stream1,cudaStreamNonBlocking);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```

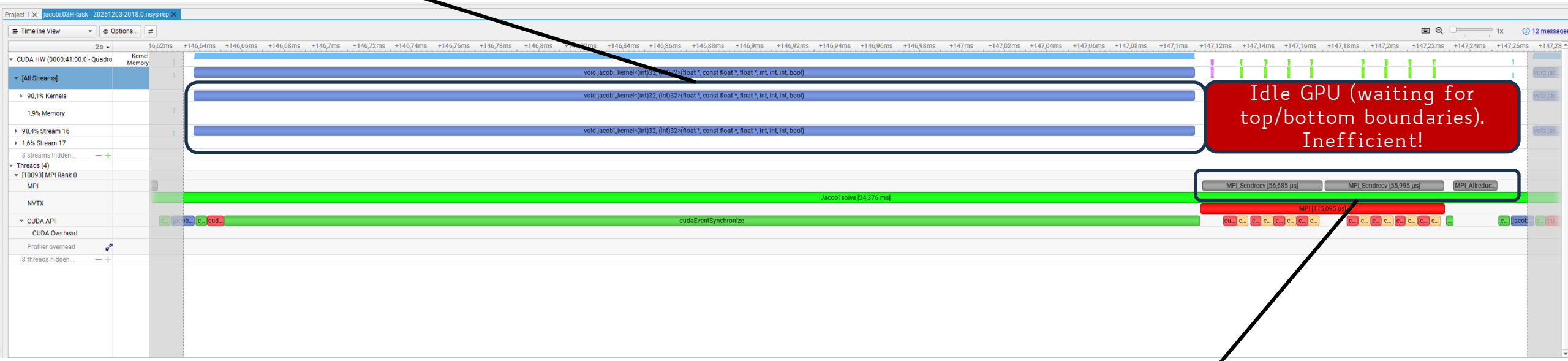


Back to Profiling and Multi GPU

https://github.com/FZJ-JSC/tutorial-multi-gpu/
Slide courtesy: Simon Garcia de Gonzalo, Jiri Kraus

Profile

Kernel Execution

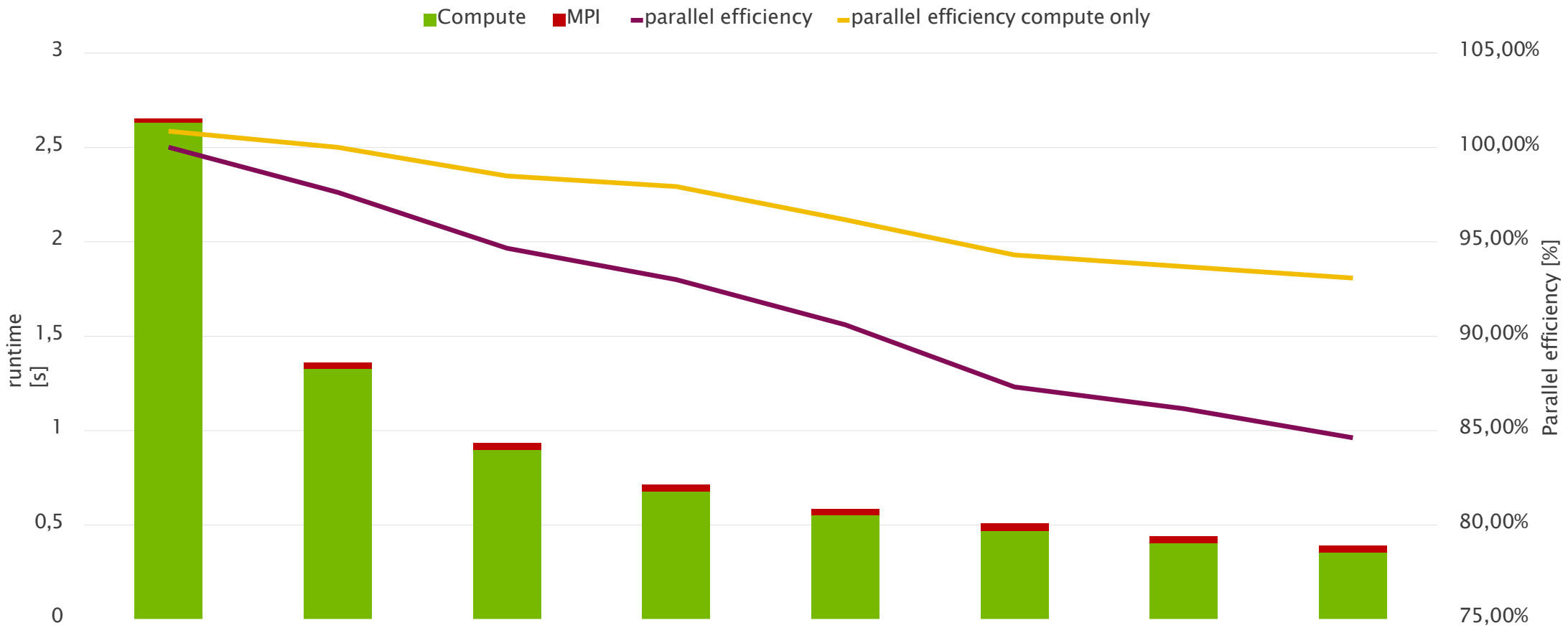


MPI Communication

`assignment-3-mgpu-mpi/profiles/`

Expectations

ParaStationMPI 5.4.10-1 – 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

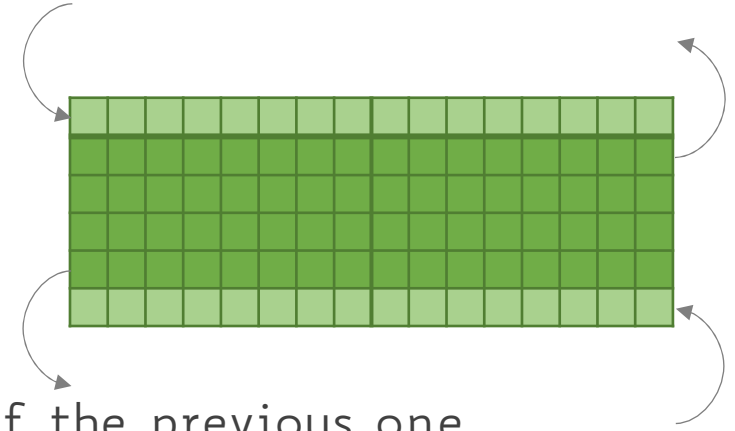
Intuition

- Separate computation of top/bottom borders from the inner part
- In this way, we can send/recv immediately the borders while we compute the inner part
- So we would like to have (in parallel):

```
compute_top_kernel<<<...>>>
compute_bottom_kernel<<<...>>>
compute_inner_kernel<<<...>>>
```

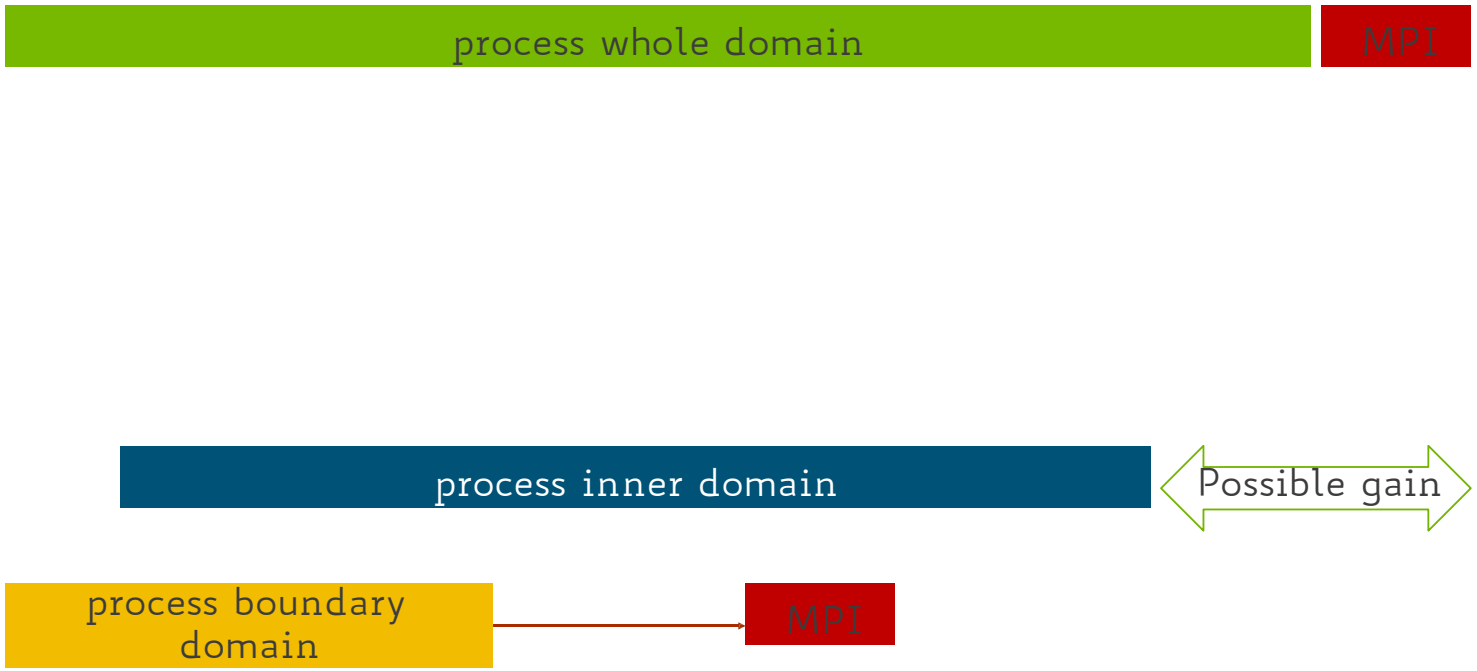
- Then:

```
wait for the top kernel to finish
sendrecv the top border
wait for the bottom kernel to finish
sendrecv the bottom border
```



- Issues:
  - By default, each kernel won't start before the termination of the previous one
  - cudaDeviceSynchronize waits for all the running kernels to finish (not only for the top/bottom one

Communication + Computation Overlap



MPI Communication + Computation Overlap

```
launch_jacobi_kernel(a_new,    a,  l2_norm_d,  iy_start,   (iy_start + 1),  nx, push_top_stream);
launch_jacobi_kernel(a_new,    a,  l2_norm_d,  (iy_end -  1), iy_end, nx,  push_bottom_stream);
launch_jacobi_kernel(a_new,    a,  l2_norm_d,  (iy_start  + 1), (iy_end -  1), nx, compute_stream);


const int top = rank > 0 ? rank - 1 : (size - 1);
const int bottom = (rank + 1) % size;

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));
```

Race Conditions with Streams

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

    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, iy_end, nx, calculate_norm,
                         compute_stream);
```

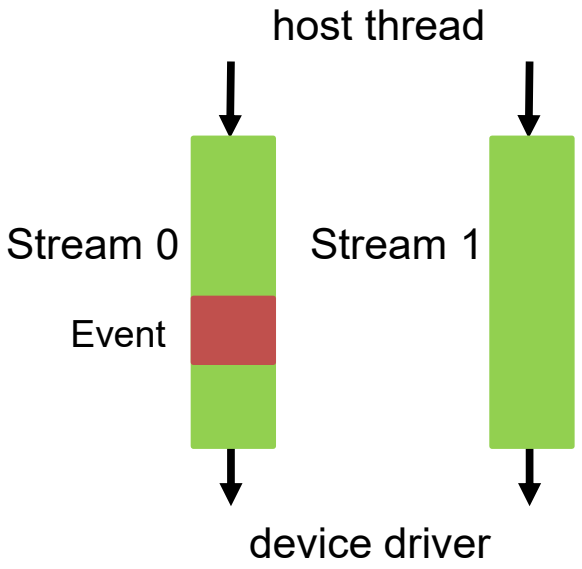
- At the beginning of every iteration, l2_norm_d is set to 0. This is done with a cudaMemsetAsync
- This operation must be run on a specific stream (compute_stream in this case)
- If there are other kernels enqueued on other streams (e.g., those for computing top/bottom boundaries), those might start before this cudaMemset is completed, and would work on a wrong l2_norm_d value
- Thus, we must ensure that those kernels running on other streams, do not start before the completion of this cudaMemsetAsync

CUDA Events

(Let's forget about Multi-GPU for a moment and let's get back to a single GPU scenario)

CUDA Events

- CUDA "events" allow the host thread to query and synchronize with individual streams
- By checking these event I can know what have been processed so far from the stream queue



CUDA Events

- Events in CUDA can be used to capture the time instance a GPU operation completes.
- Events allow fine-grained timing of GPU operations.
- Events are represented as instances of the **cudaEvent_t** type and they can be:
  - **Created**:
    cudaError_t cudaEventCreate (cudaEvent_t *event)
    or
    cudaEventCreateWithFlags(&ev, **cudaEventDisableTiming**)
    - Disable timing to increase performance
  - **Destroyed**:
    cudaError_t cudaEventDestroy (cudaEvent_t event)
  - **Captured (i.e., enqueued on the stream queue)**:
    cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream=0)
  - **Waited for:**
    cudaError_t cudaEventSynchronize (cudaEvent_t event)

CUDA Events

- Once two events have been triggered, (e.g. After cudaEventSynchronize, cudaDeviceSynchronize, cudaStreamDestroy, etc...), the duration between two events can be calculated via:

```
cudaError_t cudaEventElapsedTime(
                float *ms,          // Storage for elapsed time
                                    //   in msec units (OUT)
                cudaEvent_t start,// Start event identifier (IN)
                cudaEvent_t end); // End event identifier (IN)
```

CUDA Events

E.g., timing host-to-device memory copy:

```
cudaEvent_t startT, endT;
float duration;
cudaEventCreate (&startT);
cudaEventCreate (&endT);

for (dataSize = 0; dataSize <= MAXDATASIZE; dataSize += step)
  {
    cudaEventRecord (startT, str);
    for (i = 0; i < iter; i++)
      {
        cudaMemcpyAsync (d_data, h_data, sizeof (int) * dataSize, ↩
            cudaMemcpyHostToDevice, str);
      }
    cudaEventRecord (endT, str);
    cudaEventSynchronize (endT);
    cudaEventElapsedTime (&duration, startT, endT);
    printf ("%i %f\n", (int) (dataSize * sizeof (int)), duration / ↩
        iter);
  }
```

Testing for different data block sizes

Testing multiple times

Synchronization using Events

- cudaEventSynchronize(cudaEvent_t event)
  - Host wait for the events to occur (i.e., all the preceding operations on the stream completed)
- cudaEventQuery(cudaEvent_t event)
  - Host checks if the event occurred (without blocking)
- cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event, unsigned int flags = 0)
  - The subsequent work submitted to the stream "stream" must wait until the event "event" occurs
  - Does not block the host!

- ATTENTION: do not call cudaEventSynchronize (it is blocking!) before enqueueing that event with cudaEventRecord

Synchronization using Events - Recap

Synchronization between host and device through:

- cudaDeviceSynchronize
- cudaStreamSynchronize
- cudaEventSyncrhonize

Synchronization between streams:

- cudaStreamWaitEvent

Recap

- Starting point: having to wait for all the tile to be computed before sending/receiving is inefficient
- Let's separate the computation in three kernels: 1 for bottom boundary, 1 for top boundary, one for inner rows, so that we can overlap computation and communication
- Let's put these three kernels on three different streams
- However, we need to pay attention to race conditions, we might need to synchronize events between different streams (e.g., setting l2_norm to 0 before any kernel can start)

Back to Comp. Comm. Overlap: Hands-On

https://github.com/FZJ-JSC/tutorial-multi-gpu/
Slide courtesy: Simon Garcia de Gonzalo, Jiri Kraus

Instructions

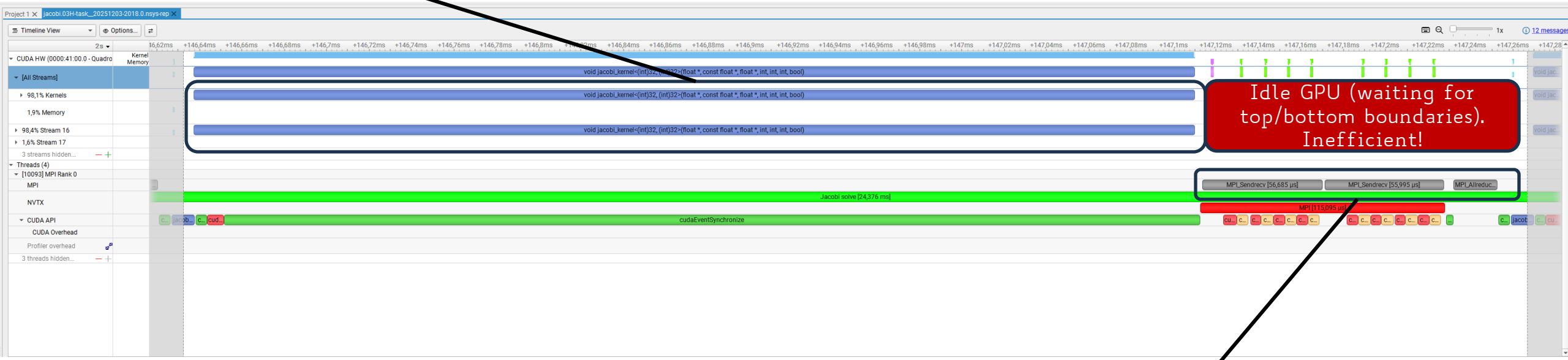
- Join the assignment on Github Classroom: https://classroom.github.com/a/K8voN6eF
  - At the moment, there is no auto-grading yet, check correctness by yourself
- Access the cluster and clone the repository
- Fill the «TODOs» in the jacobi.cu code
- Run "source /home/guest/init-hpc.sh" to load the MPI, CUDA compilers etc…
- Run «make run» to compile and run the code (by default on 4 GPUs, one GPU per node)

Performance

```
Jacobi relaxation: 1000 iterations on 16384 x 16384 mesh with norm check every 1 iterations
    0, 31.999020
  100, 0.897983
  200, 0.535685
  300, 0.395651
  400, 0.319039
  500, 0.269961
  600, 0.235510
  700, 0.209829
  800, 0.189854
  900, 0.173818
Num GPUs: 4.
16384x16384: 1 GPU:   5.9741 s, 4 GPUs:   1.5338 s, speedup:     3.90, efficiency:    97.38
```

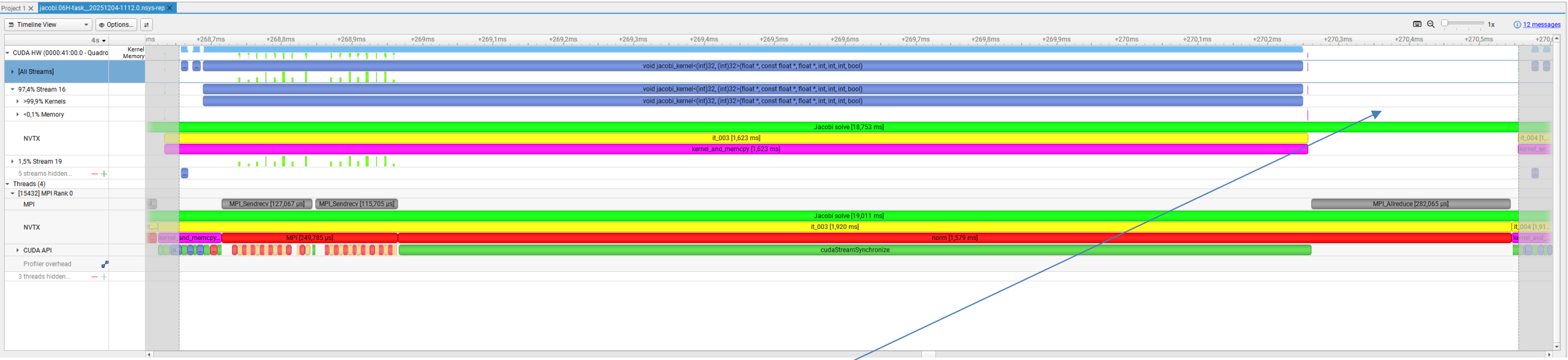
Profile (Before, no overlap)

Kernel Execution



MPI Communication

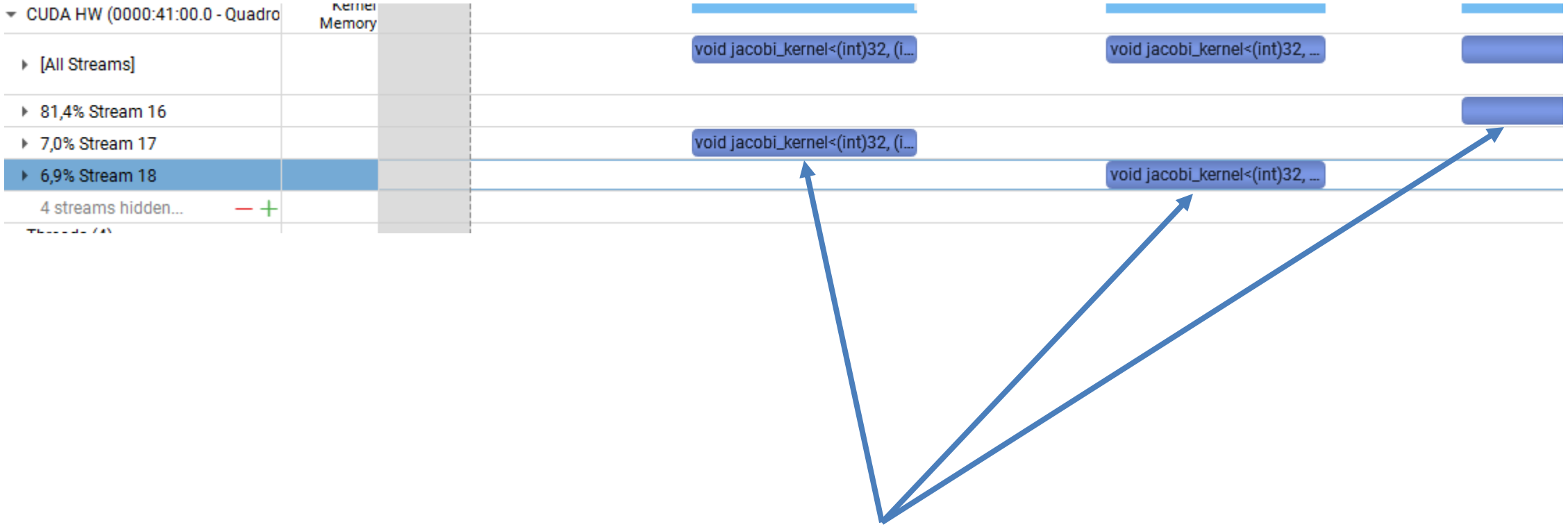
Profile (Now, with overlap)



Less Idle Time than before

`assignment-4-mgpu-mpi-overlap/profiles/streams`

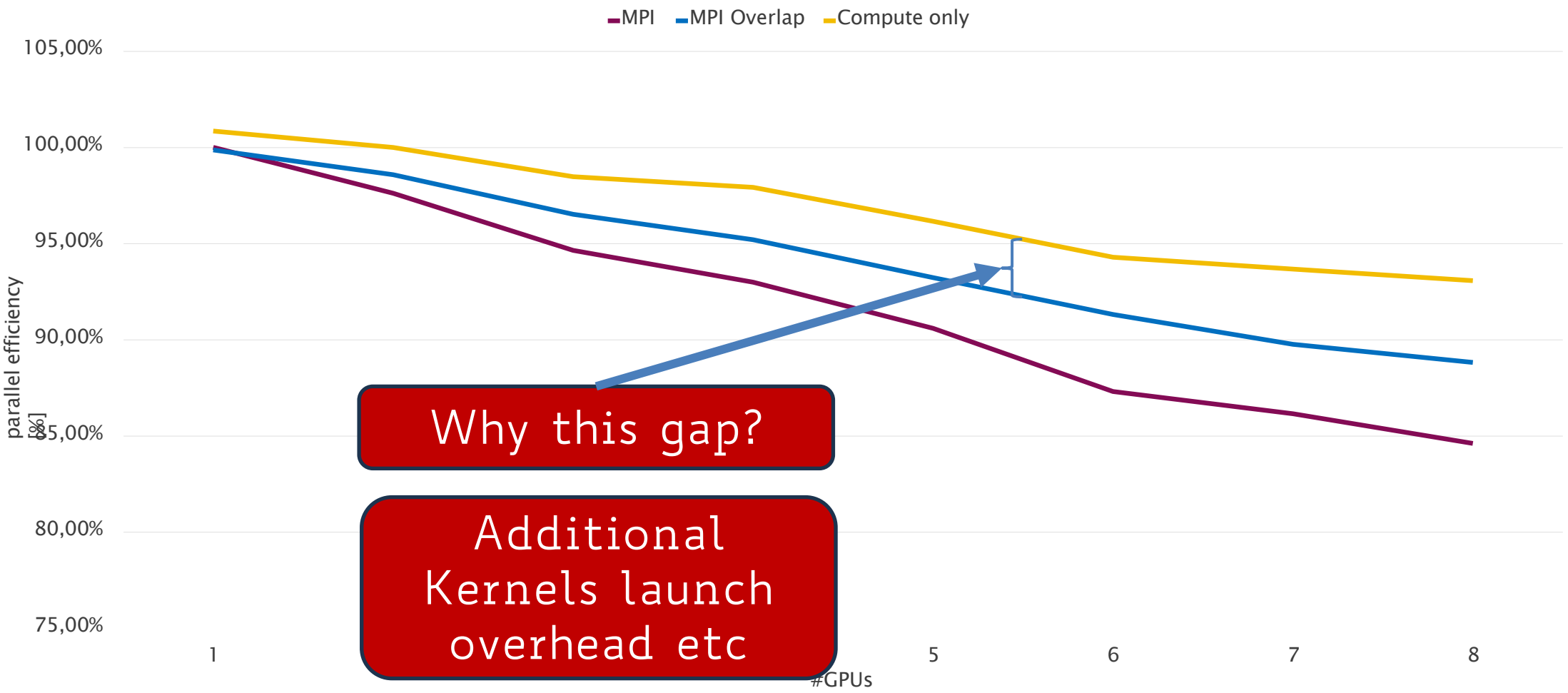
Profile (Now, with overlap)



3 different streams

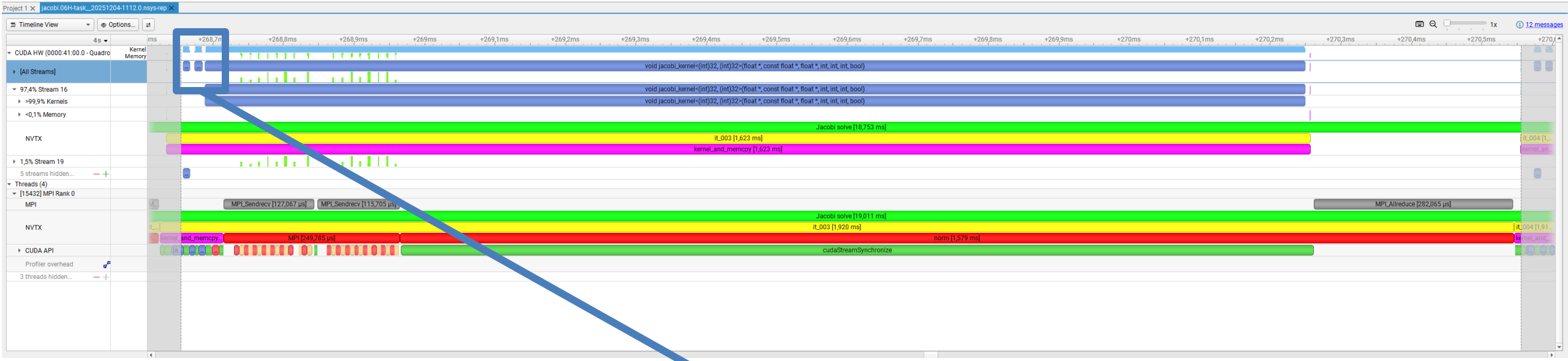
Communication + Computation Overlap

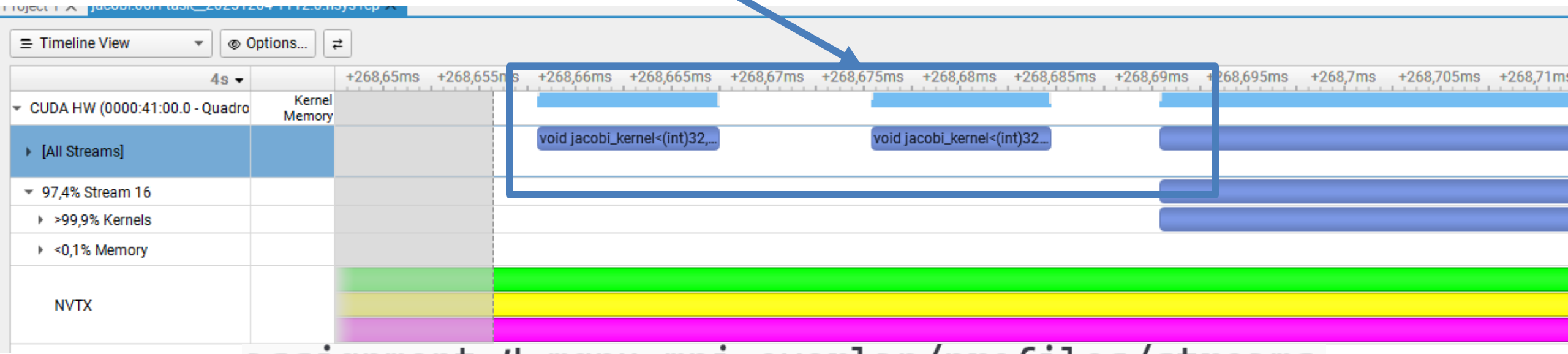
ParaStationMPI 5.4.10-1 – JUWELS Booster – NVIDIA A100 40 GB – Jacobi on 17408x17408



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

Profile (Now, with overlap)





assignment-4-mgpu-mpi-overlap/profiles/streams

MPI Communication + Computation Overlap

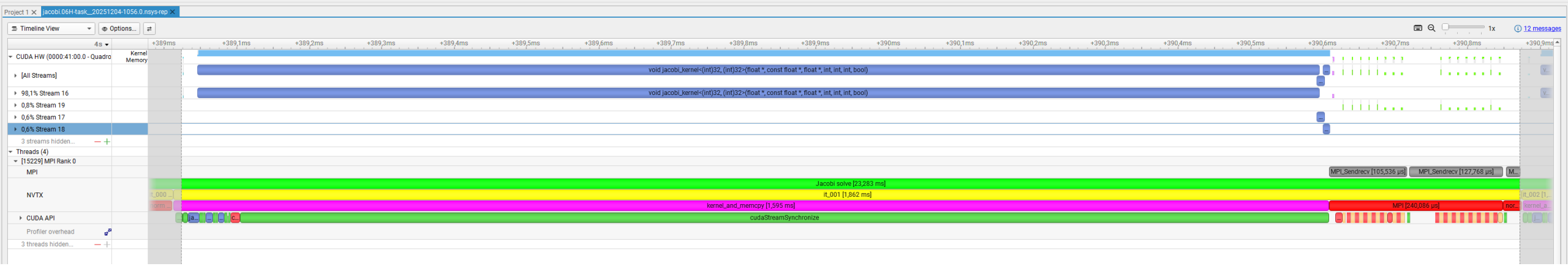
```
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_start + 1), (iy_end - 1), nx, compute_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, push_top_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1), iy_end, nx, push_bottom_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_start + 1), (iy_end - 1), nx, compute_stream);

const int top = rank > 0 ? rank - 1 : (size - 1);
const int bottom = (rank + 1) % size;


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));
```

Performance

```
Jacobi relaxation: 1000 iterations on 16384 x 16384 mesh with norm check every 1 iterations
    0, 31.999020
  100, 0.897983
  200, 0.535685
  300, 0.395651
  400, 0.319039
  500, 0.269961
  600, 0.235510
  700, 0.209829
  800, 0.189854
  900, 0.173818
Num GPUs: 4.
16384x16384: 1 GPU:   5.9750 s, 4 GPUs:   1.7216 s, speedup:     3.47, efficiency:    86.7
```

Why lower speedup/efficiency?

Profiling



Even if kernel on different streams can go in parallel, the first kernel to be launched is eating up all the resources and the two smaller kernels can't start before it is over, falling back to the non-overlapping case

`assignment-4-mgpu-mpi-overlap/profiles/streams_inner_first`

High Priority Streams

Improve scalability with high priority streams (available on CC 3.5+)

```
cudaStreamCreateWithPriority ( cudaStream_t* pStream, unsigned int flags, int priority )
```

- If something else is running with a lower priority on a different stream, hold that and run this first, and then resume it when the stuff on this stream completes

MPI Communication + Computation Overlap

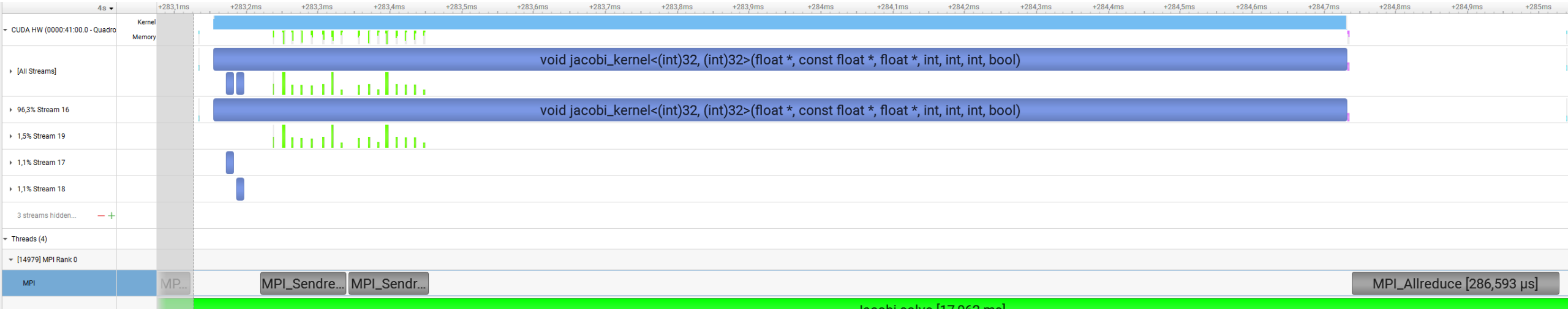
with high priority streams

```
int leastPriority = 0;
int greatestPriority = leastPriority;
CUDA_RT_CALL(cudaDeviceGetStreamPriorityRange(&leastPriority,  &greatestPriority));


cudaStream_t compute_stream;
cudaStream_t push_top_stream;
cudaStream_t push_bottom_stream;


CUDA_RT_CALL(cudaStreamCreateWithPriority(&compute_stream,  cudaStreamDefault,  leastPriority));
CUDA_RT_CALL(cudaStreamCreateWithPriority(&push_top_stream,  cudaStreamDefault,  greatestPriority));
CUDA_RT_CALL(cudaStreamCreateWithPriority(&push_bottom_stream,  cudaStreamDefault,  greatestPriority));
```

Performance

```
Jacobi relaxation: 1000 iterations on 16384 x 16384 mesh with norm check every 1 iterations
    0, 31.999022
  100, 0.897983
  200, 0.535685
  300, 0.395651
  400, 0.319039
  500, 0.269961
  600, 0.235510
  700, 0.209829
  800, 0.189854
  900, 0.173818
Num GPUs: 4.
16384x16384: 1 GPU:   6.0214 s, 4 GPUs:   1.5456 s, speedup:     3.90, efficiency:    97.40
```

Profiling

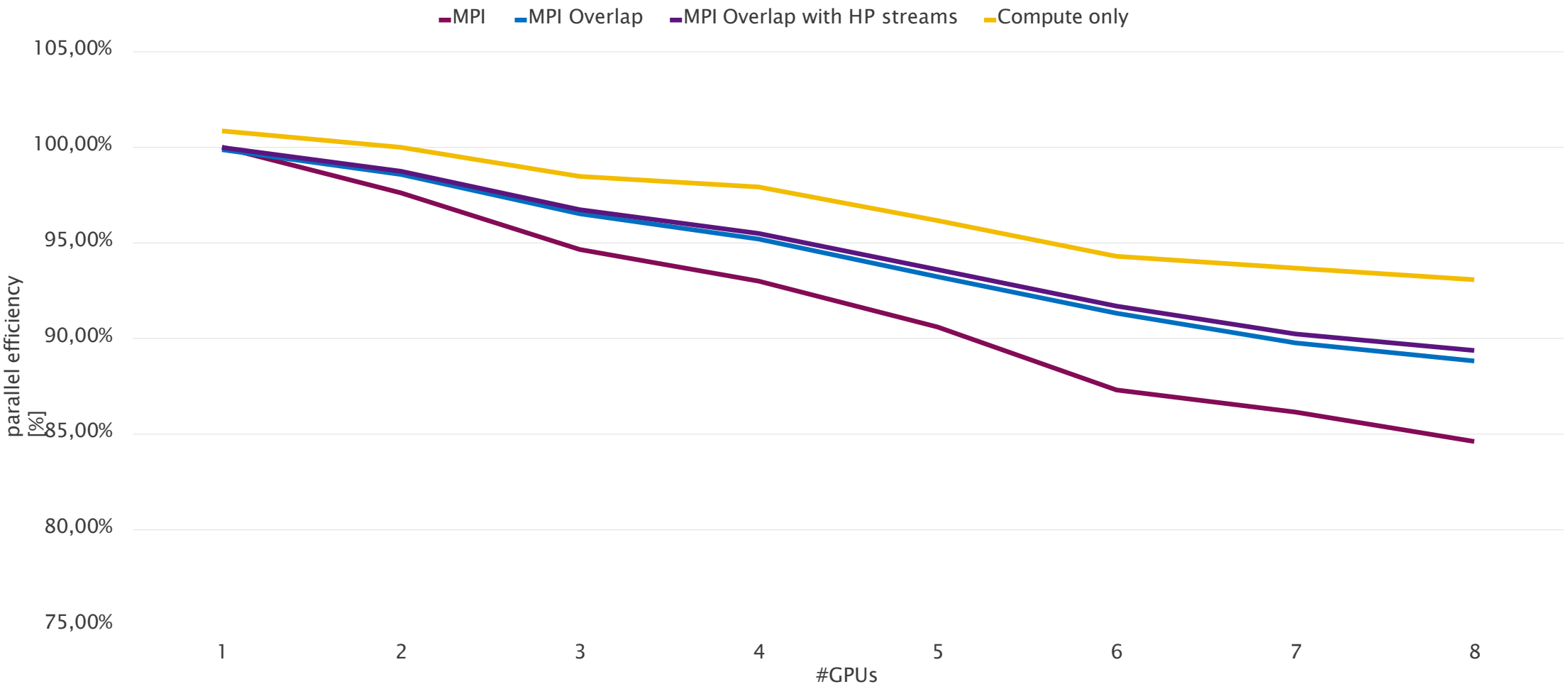


Now the three kernels, in three different streams, run «in parallel» (the smaller kernels might preempt the larger kernel) with the sendrecv

`assignment-4-mgpu-mpi-overlap/profiles/streams_inner_first_prio/`

Communication + Computation Overlap

ParaStationMPI 5.4.10-1 – 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