PROGRAMMING MULTI-GPU NODES

Steve Abbott, Summit Training Workshop, December 2018

AGENDA

Multi-GPU Programming Models

Multi-GPU Programming with OpenACC and CUDA

SUMMIT NODE

(2) IBM POWER9 + (6) NVIDIA VOLTA V100

MULTI-GPU PROGRAMMING MODELS

MULTI-GPU PROGRAMMING MODELS

Single Thread, Multiple GPUs

- A single thread will change devices as-needed to send data and kernels to different GPUs

Multiple Threads, Multiple GPUs

- Using OpenMP, Pthreads, or similar, each thread can manage its own GPU

Multiple Ranks, Single GPU

- Each rank acts as-if there's just 1 GPU, but multiple ranks per node use all GPUs

Multiple Ranks, Multiple GPUs

- Each rank manages multiple GPUs, multiple ranks/node. Gets complicated quickly!

MULTI-GPU PROGRAMMING MODELS

Trade-offs Between Approaches

Single Thread, Multiple GPUs	Multiple Threads, Multiple GPUs	Multiple Ranks, Single GPU	Multiple Ranks, Multiple GPUs
• Conceptually Simple • Requires additional loops • CPU can become a bottleneck • Remaining CPU cores often underutilized	• Conceptually Very Simple • Set and forget the device numbers • Relies on external Threading API • Can see improved utilization • Watch affinity	• Little to no code changes required • Re-uses existing domain decomposition • Probably already using MPI • Watch affinity	• Easily share data between peer devices • Coordinating between GPUs extremely tricky

MULTI-DEVICE CUDA

CUDA by default exposes all devices, numbered 0 – (N-1), if devices are not all the same, it will reorder the “best” to device 0.

Each device has its own pool of streams.

If you do nothing, *all* work will go to Device #0.

Developer must change the current device explicitly

MULTI-DEVICE OPENACC

OpenACC presents devices numbered 0 – (N-1) for each device type available.

The order of the devices comes from the runtime, almost certainly the same as CUDA

By default all data and work go to the *current device*

Developers must change the current device and maybe the current device type using an API

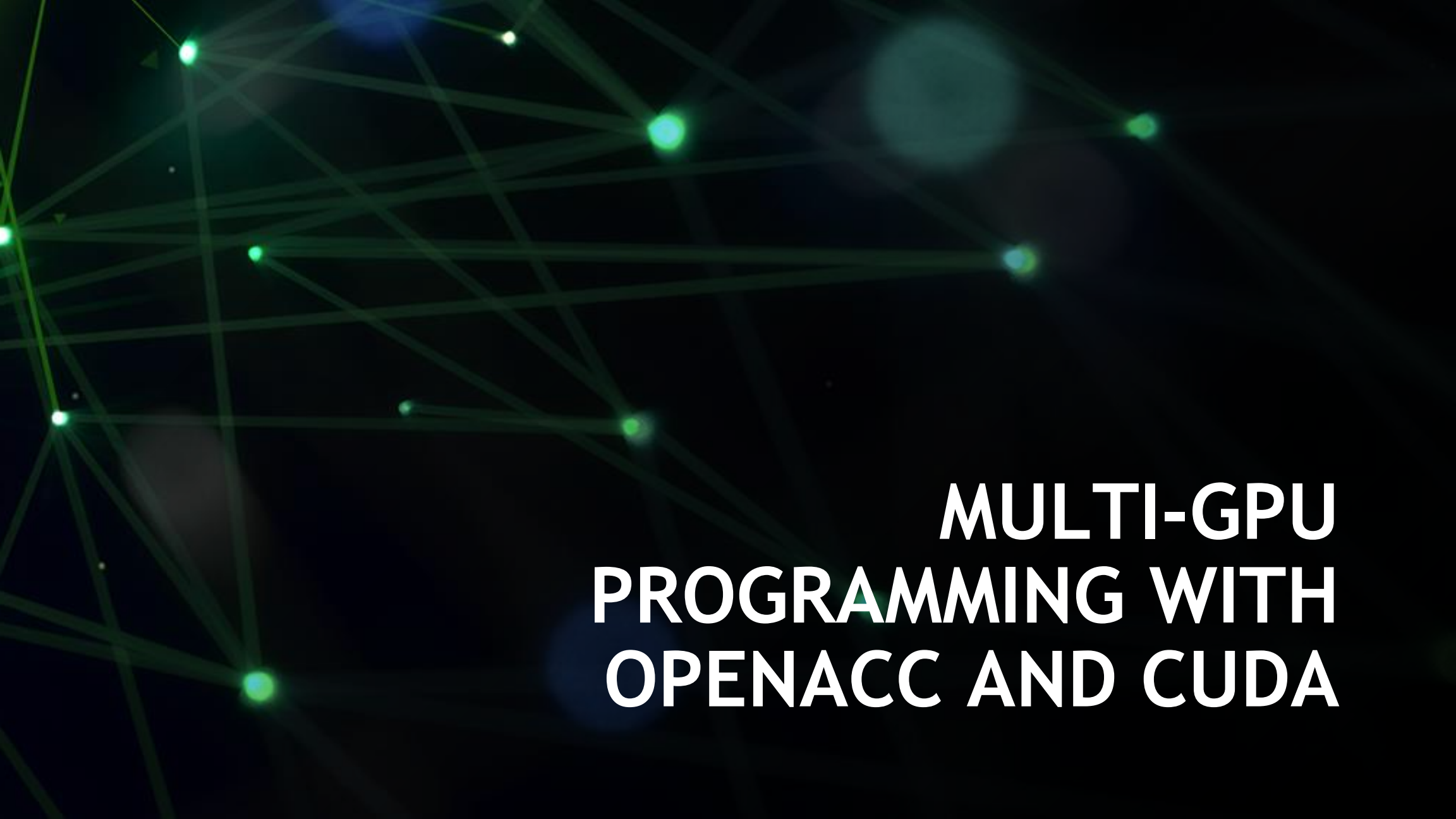
MULTI-DEVICE OPENMP

OpenMP devices numbered 0 – (N-1) for *ALL* devices on the machine, including the host.

The order is determined by the runtime, but devices of the same type are contiguous.

To change the device for data and compute a clause is added to directives.

Device API routines include a devicenum


MULTI-GPU
PROGRAMMING WITH
OPENACC AND CUDA

MULTI-GPU W/ CUDA AND OPENACC

The CUDA and OpenACC approaches are sufficiently similar, that I will demonstrate using OpenACC.

Decoder Ring:

OpenACC	CUDA
acc_get_device_type()	N/A
acc_set_device_type()	N/A
acc_set_device_num()	cudaSetDevice()
acc_get_device_num()	cudaGetDevice()
acc_get_num_devices()	cudaGetDeviceCount()

Multi-Device Pipeline

A Case Study

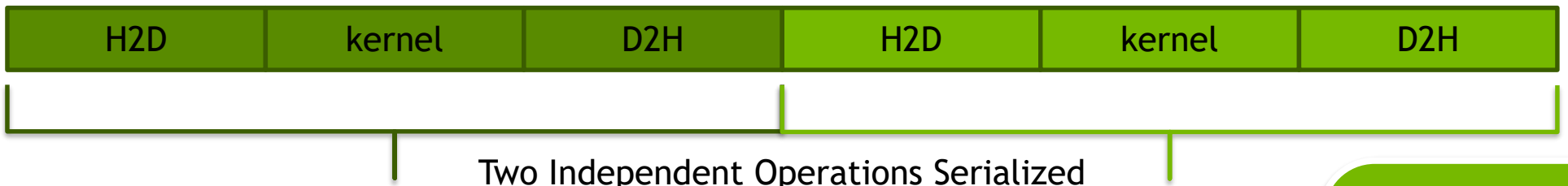
We'll use a simple image filter to demonstrate these techniques.

No inter-GPU communication required

Pipelining: Breaking a large operation into smaller parts so that independent operations can overlap.

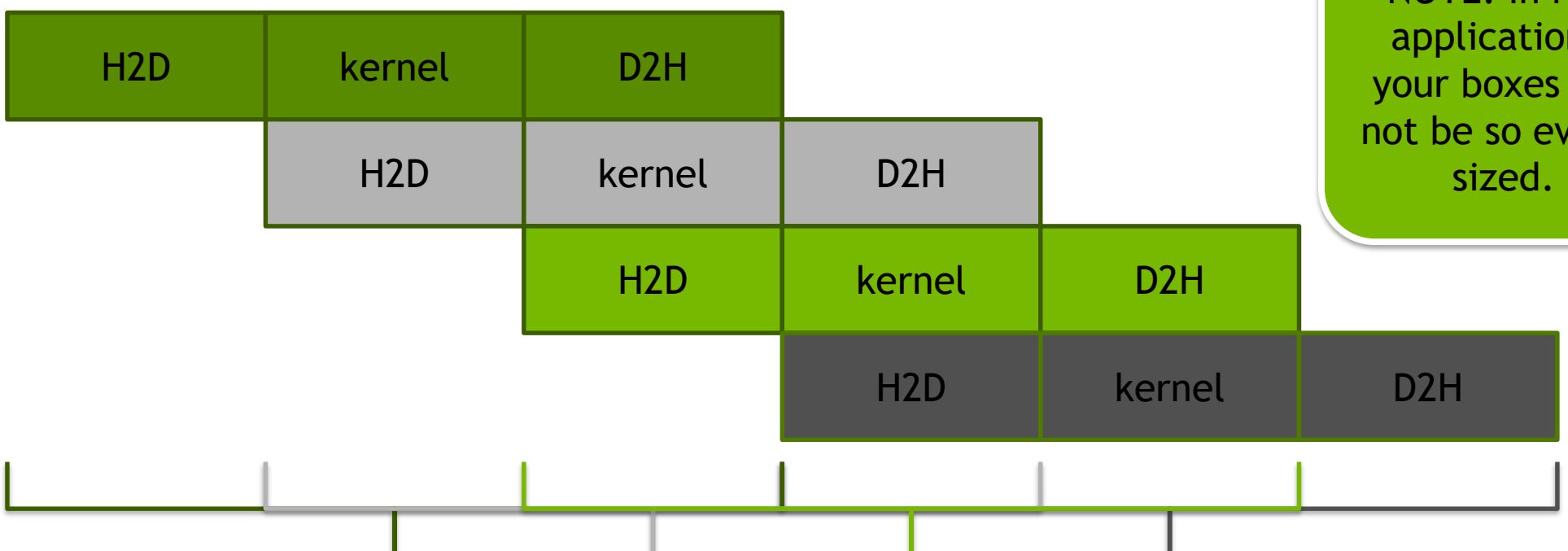
Since each part is independent, they can easily be run on different devices. We will extend the filter to run on more than one device.

Pipelining in a Nutshell



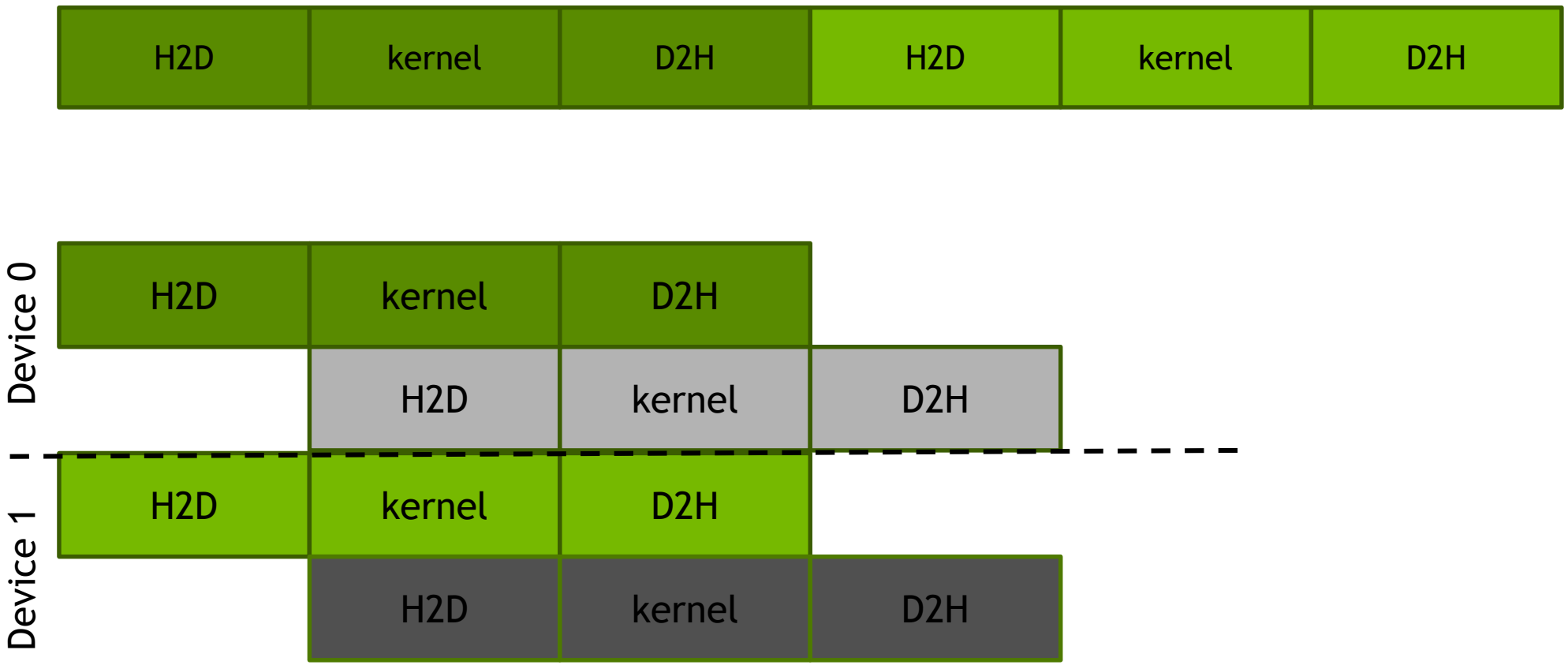
Two Independent Operations Serialized

NOTE: In real applications, your boxes will not be so evenly sized.



Overlapping Copying and Computation

Multi-device Pipelining in a Nutshell



Pipelined Code

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                 copyin(filter)
{
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:blocksize*step]) async(block%3)
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(block%3)
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]     = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step]) async(block%3)
}
#pragma acc wait
}
```

Cycle between 3 async queues by blocks.

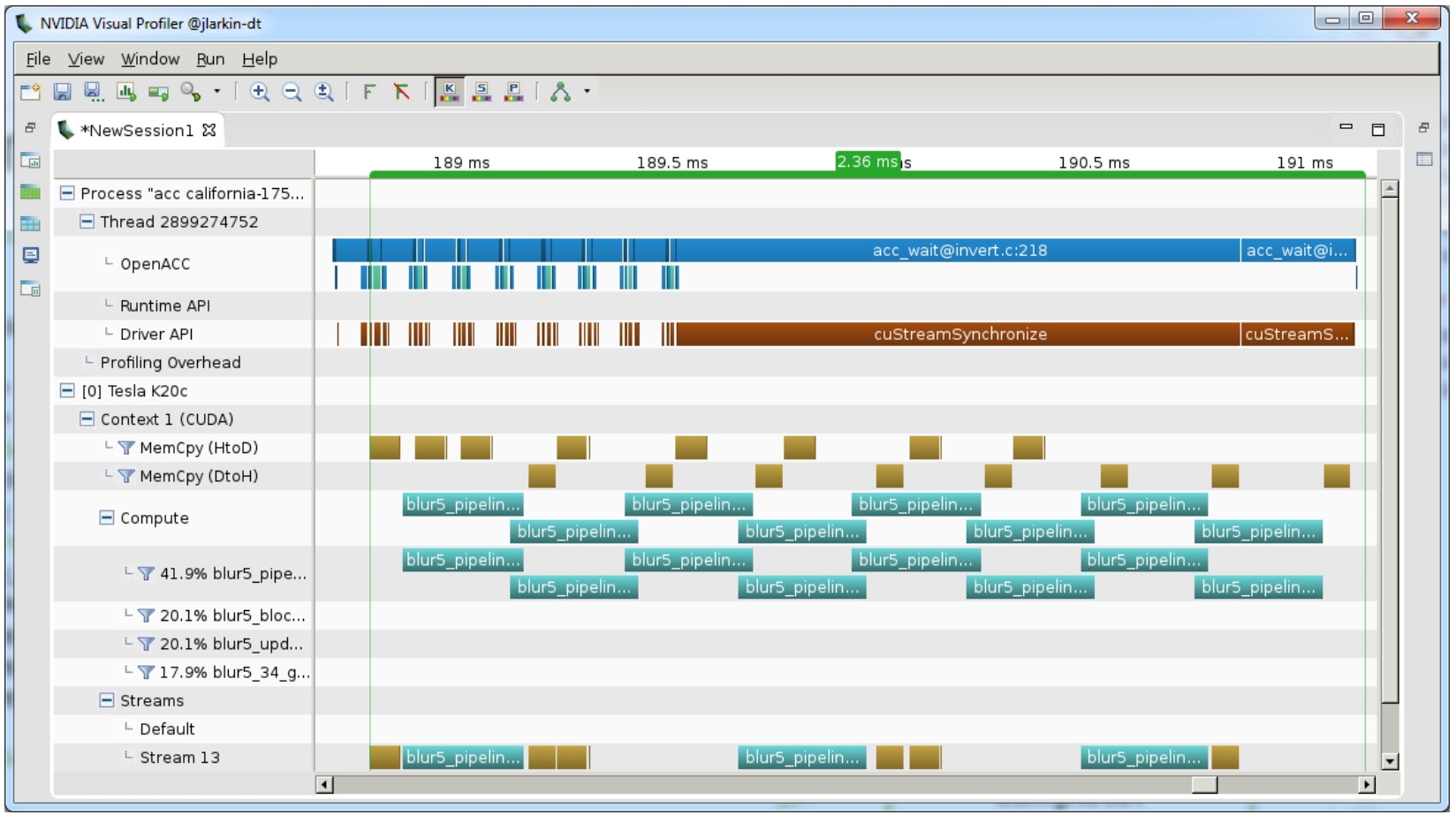
Pipelined Code

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                 copyin(filter)
{
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:blocksize*step]) async(block%3)
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(block%3)
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]     = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step]) async(block%3)
}
#pragma acc wait
}
```

Cycle between 3 async queues by blocks.

Wait for all blocks to complete.

NVPROF Timeline of Pipeline



Extending to multiple devices

Create 1 OpenMP thread on the CPU per-device. This is not strictly necessary, but simplifies the code.

Within each thread, set the device number.

Divide the blocks as evenly as possible among the CPU threads.

Multi-GPU Pipelined Code (OpenMP)

```
#pragma omp parallel num_threads(acc_get_num_devices(acc_device_default))
  {
    acc_set_device_num(omp_get_thread_num(),acc_device_default);
    int queue = 1;
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
  {
#pragma omp for schedule(static)
  for ( long blocky = 0; blocky < nblocks; blocky++) {
    // For data copies we need to include the ghost zones for the filter
    long starty = MAX(0,blocky * blocksize - filtersize/2);
    long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:(endy-starty)*step]) async(queue)
    starty = blocky * blocksize;
    endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(queue)
    for ( long y = starty; y < endy; y++ ) { for ( long x = 0; x < w; x++ ) {
        <filter code removed for space>
    }}
#pragma acc update self(out[starty*step:blocksize*step]) async(queue)
    queue = (queue%3)+1;
  }
#pragma acc wait
  }
  }
```

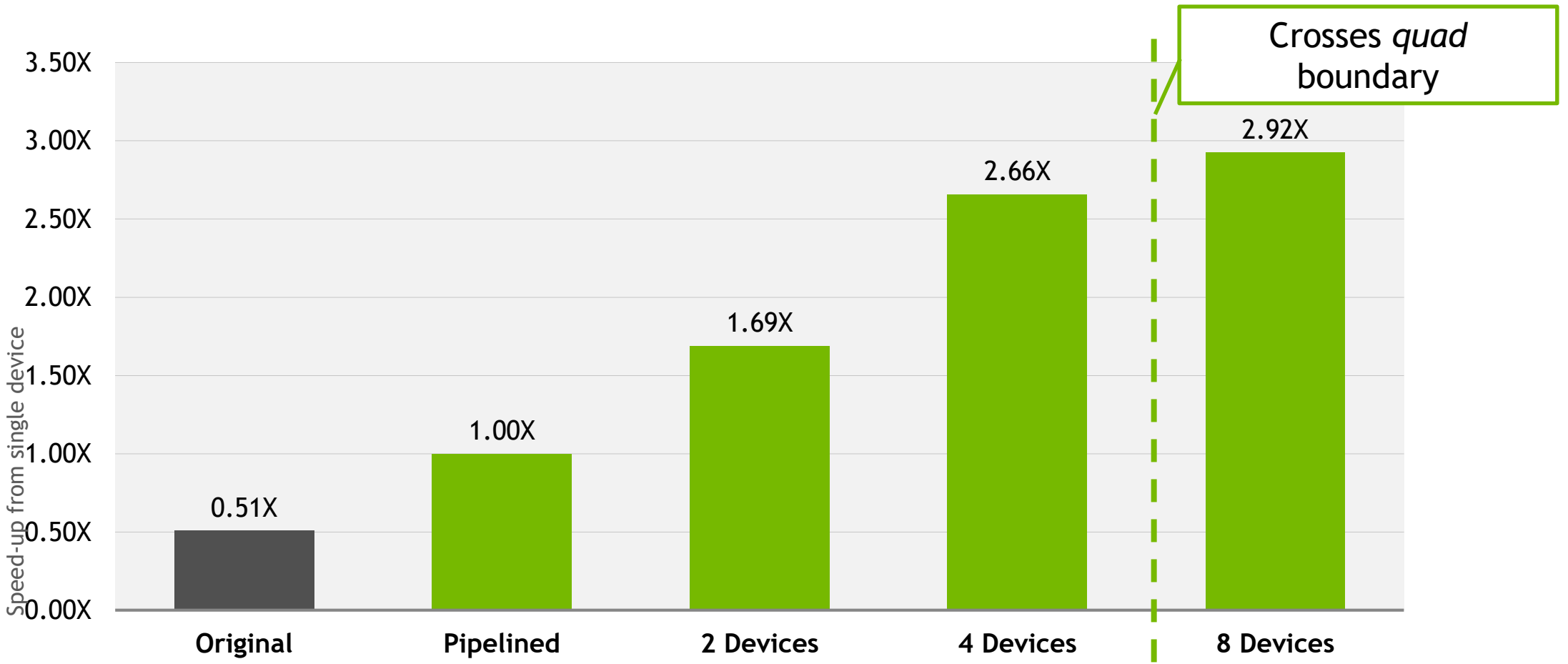
Spawn 1 thread per device.

Set the device number per-thread.

Divide the work among threads.

Wait for each device in its thread.

Multi-GPU Pipelined Performance



Source: PGI 17.3, NVIDIA Tesla P100 (DGX-1)

OpenACC with MPI

Domain decomposition is performed using MPI ranks

Each rank should set its own device

- Maybe acc_set_device_num
- Maybe handled by environment variable (CUDA_VISIBLE_DEVICES)

GPU affinity can be handled by standard MPI task placement

Multiple MPI Ranks/GPU (using MPS) can work in place of OpenACC work queues/CUDA Streams

Setting a device by local rank

```
// This is not portable to other MPI libraries
 char *comm_local_rank = getenv("OMPI_COMM_WORLD_LOCAL_RANK");
 int local_rank = atoi(comm_local_rank);
 char *comm_local_size = getenv("OMPI_COMM_WORLD_LOCAL_SIZE");
 int local_size = atoi(comm_local_size);
 int num_devices = acc_get_num_devices(acc_device_nvidia);
#pragma acc set device_num(local_rank%num_devices) \
               device_type(acc_device_nvidia)
```

Determine a unique ID for each rank on the same node.

Use this unique ID to select a device per rank.

You may also try using MPI_Comm_split_type() using MPI_COMM_TYPE_SHARED or OMPI_COMM_TYPE_SOCKET.

In the end, you need to understand how jsrun/mpirun is placing your ranks.

MPI Image Filter (pseudocode)

```
if (rank == 0 ) read_image();
// Distribute the image to all ranks
MPI_Scatterv(image);

MPI_Barrier(); // Ensures all ranks line up for timing
omp_get_wtime();
blur_filter(); // Contains OpenACC filter
MPI_Barrier(); // Ensures all ranks complete before timing
omp_get_wtime();

MPI_Gatherv(out);
if (rank == 0 ) write_image();

$ jsrun –n 6 –a 1 –c 1 –g 1 ...
```

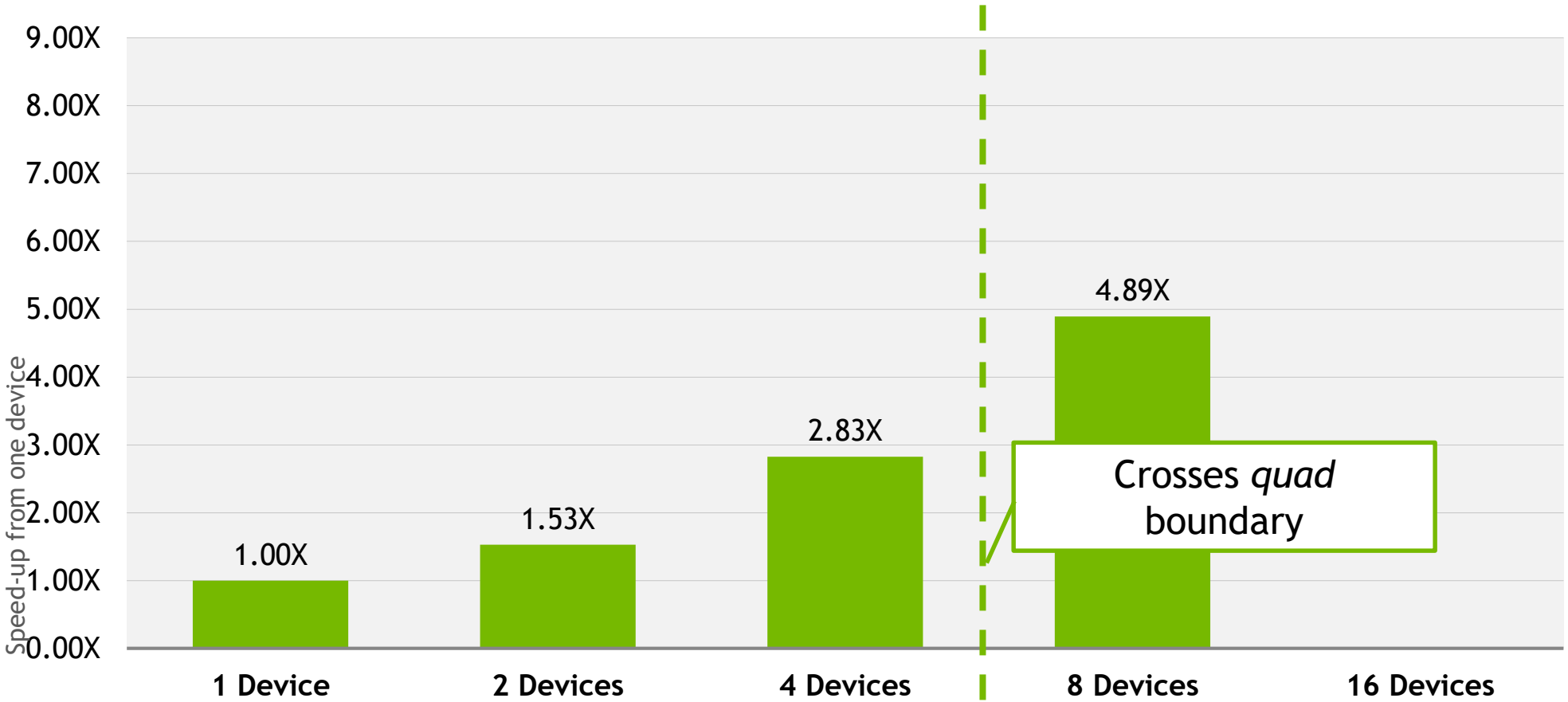
Decompose image across processes (ranks)

Receive final parts from all ranks.

Launch with good GPU/process affinity

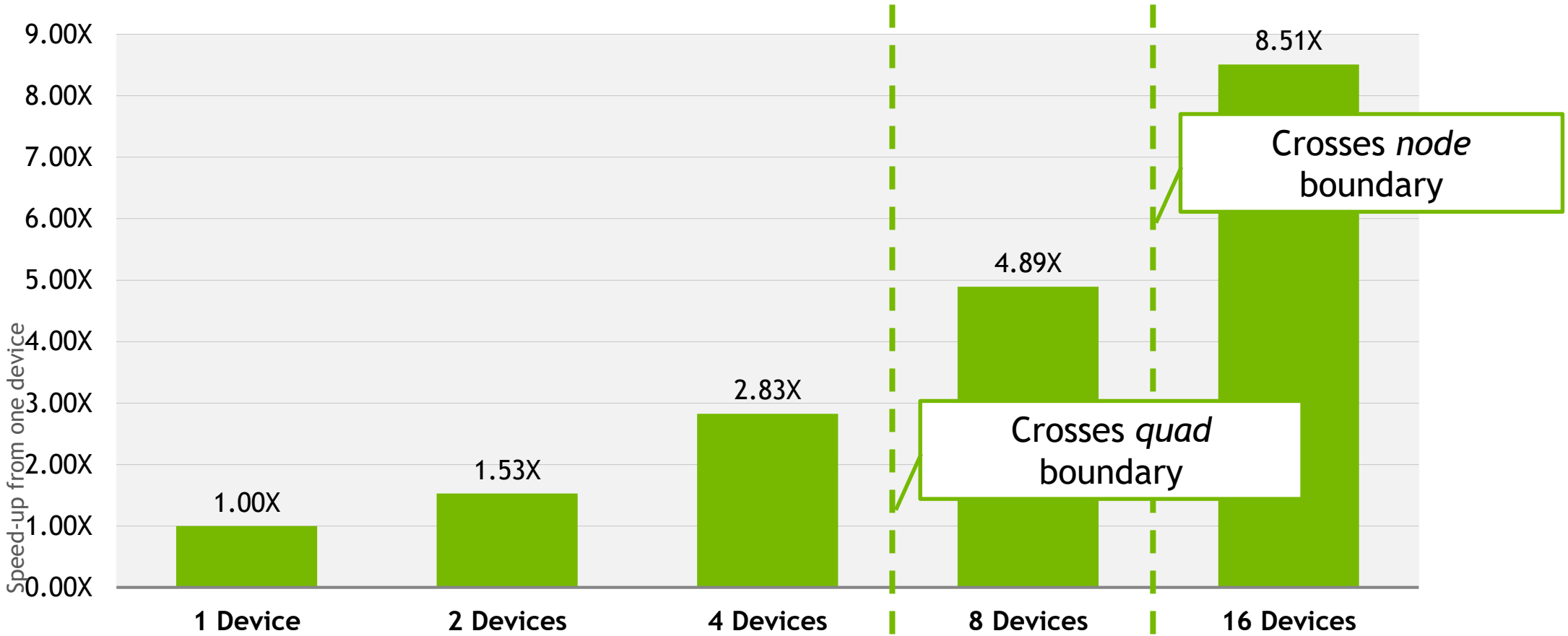
There’s a variety of ways to do MPI decomposition, this is what I used for this particular example.

Multi-GPU Pipelined Performance (MPI)



Source: PGI 17.3, NVIDIA Tesla P100 (DGX-1), Communication Excluded

Multi-GPU Pipelined Performance (MPI)



Source: PGI 17.3, NVIDIA Tesla P100 (DGX-1), Communication Excluded

MULTI-DEVICE CUDA

Same Pattern, Different API

```
#pragma omp parallel
{
  cudaSetDevice(idx);
  #pragma omp for
  for ( int b=0; b < nblocks; b++)
  {
    cudaMemcpyAsync(…, streams[b%3]);
    blur_kernel <<<griddim, blockdim,
                  0, streams[b%3]>>>();
    cudaMemcpyAsync(…, streams[b%3]);
  }

  cudaDeviceSynchronize();
}
```

```
MPI_Comm_rank(local_comm, &local_rank);

cudaSetDevice(local_rank);

for ( int b=0; b < nblocks; b++)
{
  cudaMemcpyAsync(…, streams[b%3]);
  blur_kernel <<<griddim, blockdim,
                  0, streams[b%3]>>>();
  cudaMemcpyAsync(…, streams[b%3]);
}

cudaDeviceSynchronize();
```

MULTI-DEVICE OPENMP 4.5

Same Pattern, Different API

```
#pragma omp parallel num_threads(num_dev)
{
  #pragma omp for
  for ( int b=0; b < nblocks; b++)
  {
    #pragma omp target update map(to:…) \
            device(dev) depend(inout:A) \
            nowait
    #pragma omp target teams distribute \
            parallel for simd device(dev) \
            depend(inout:A)
    for(…) { … }
    #pragma omp target update map(from:…) \
            device(dev) depend(inout:A) \
            nowait
  }
  #pragma omp taskwait
}
```

```
MPI_Comm_rank(local_comm, &local_rank);
int dev = local_rank;

for ( int b=0; b < nblocks; b++)
{
  #pragma omp target update map(to:…) \
          device(dev) depend(inout:A) \
          nowait
  #pragma omp target teams distribute \
          parallel for simd device(dev) \
          depend(inout:A)
  for(…) { … }
  #pragma omp target update map(from:…) \
          device(dev) depend(inout:A) \
          nowait
}
#pragma omp taskwait
```

Multi-GPU Approaches

Choosing an approach

Single-Threaded, Multiple-GPUs – Requires additional loops to manage devices, likely undesirable.

Multi-Threaded, Multiple-GPUs – Very convenient set-and-forget the device. Could possibly conflict with existing threading.

Multiple-Ranks, Single-GPU each – Probably the simplest if you already have MPI, he decomposition is done. Must get your MPI placement correct

Multiple-Ranks, Multiple-GPUs – Can allow all GPUs to share common data structures. Only do this is you absolutely need to, difficult to get right.


CLOSING SUMMARY

MULTI-GPU APPROACHES

Choosing an approach

Single-Threaded, Multiple-GPUs – Requires additional loops to manage devices, likely undesirable.

Multi-Threaded, Multiple-GPUs – Very convenient set-and-forget the device. Could possibly conflict with existing threading.

Multiple-Ranks, Single-GPU each – Probably the simplest if you already have MPI, he decomposition is done. Must get your MPI placement correct

Multiple-Ranks, Multiple-GPUs – Can allow all GPUs to share common data structures. Only do this is you absolutely need to, difficult to get right.

GPU TO GPU COMMUNICATION

- CUDA aware MPI functionally portable
  - OpenACC/MP interoperable
  - Performance may vary between on/off node, socket, HW support for GPU Direct
  - WARNING: Unified memory support varies wildly between implementations!
- Single-process, multi-GPU
  - Enable peer access for straight forward on-node transfers
- Multi-process, single-gpu
  - Pass CUDA IPC handles for on-node copies
- Combine for more flexibility/complexity!

ESSENTIAL TOOLS AND TRICK

- Pick on-node layout with OLCF jsrun visualizer
  - https://jsrunvisualizer.olcf.ornl.gov/index.html
- Select MPI/GPU interaction with jsrun --smpiargs
  - “-gpu” for CUDA aware, “off” for pure GPU without MPI
- Profile MPI and NVLinks with nvprof
- Good performance will require experimentation!


NVIDIA