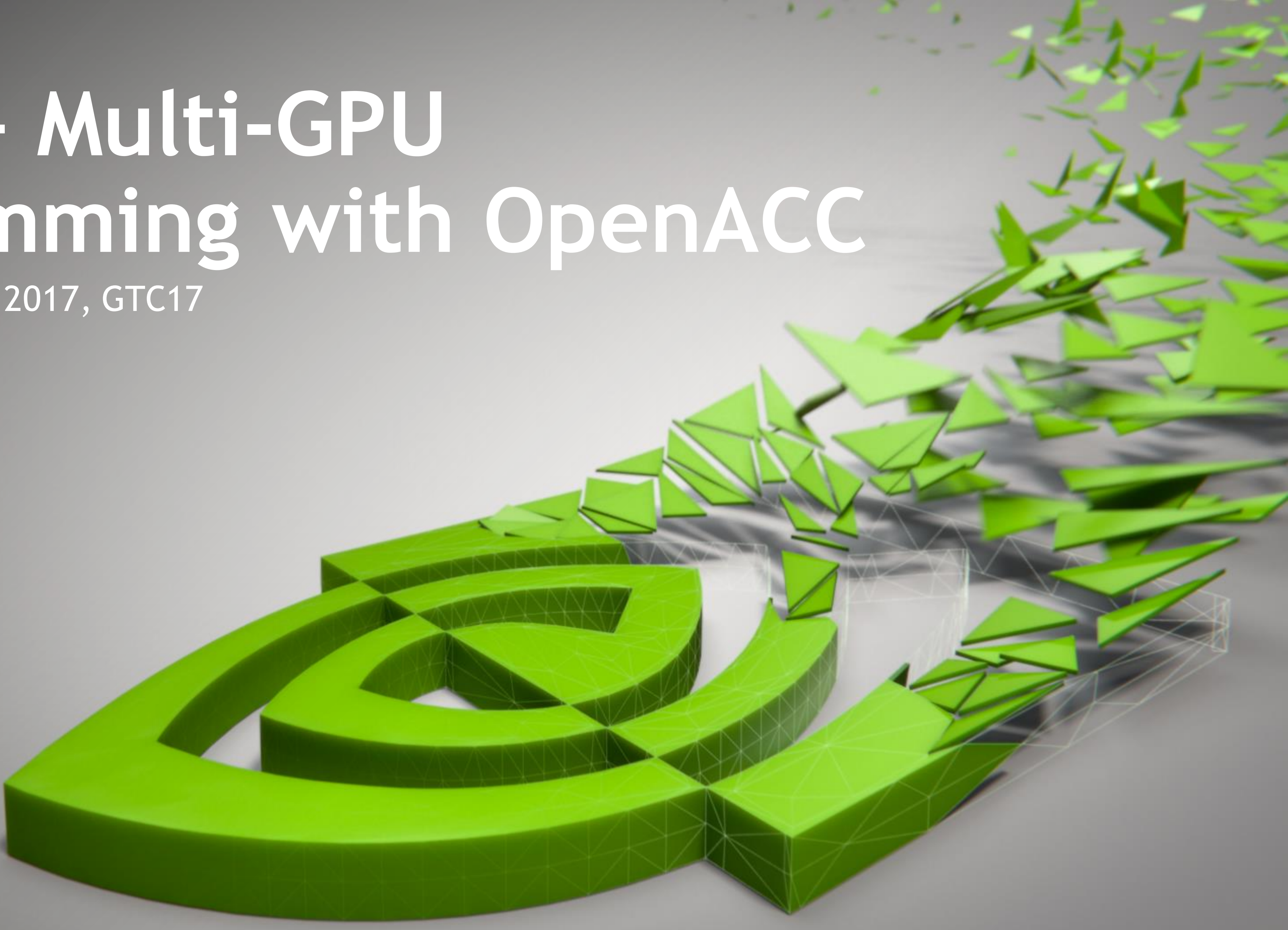
S7546 – Multi-GPU Programming with OpenACC

Jeff Larkin, May 9, 2017, GTC17



Multi-GPU Approaches

OpenACC-only

- Uses OpenACC's runtime library to select devices at runtime.
- One process manages multiple devices.

OpenACC + MPI

- Uses Message Passing Interface (MPI) for domain decomposition and GPU isolation.
- Each process (usually) only interfaces with 1 GPU, but 1 GPU may interface with many processes
- May be "free" for many apps

OpenACC multi-device API

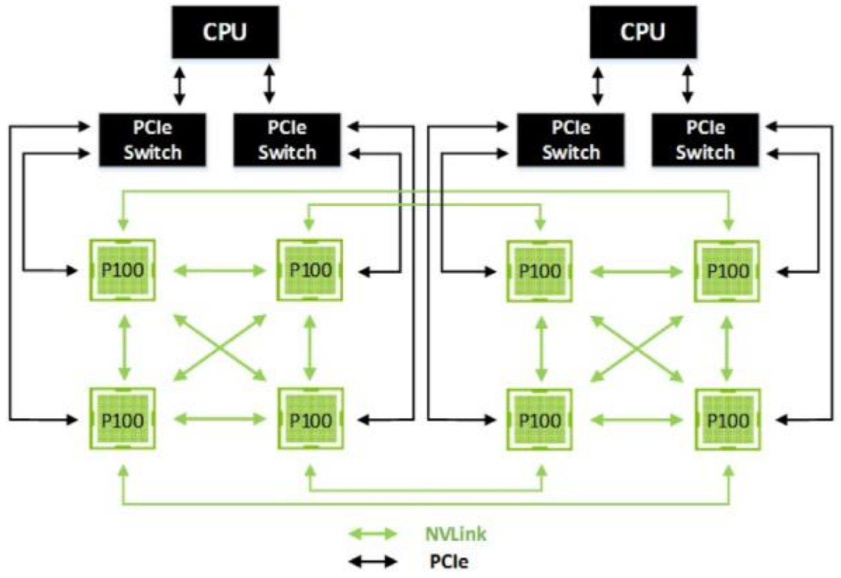
OpenACC for Multiple Devices

By default, the OpenACC runtime will choose what it believes is the most capable device.

OpenACC provides an API for enumerating the available devices, selecting device types, and selecting an individual device.

Developers can use this API to change which device the runtime will use.

Devices are uniquely identified by a number and type tuple (e.g. NVIDIA device 0).



Getting the Number of Devices

The acc_get_num_devices function returns the number of devices of a particularl type.

```
        int acc_get_num_devices(type)
```

This function should be called before attempting to use multiple devices to determine how many suitable devices are available.

Setting the Desired Device

The acc_set_device_num function selects the specific device that should be used by all upcoming OpenACC directies.

```
void acc_set_device_num(number, type)

                      or

#pragma acc set device_num(number) device_type(type)
```

Once a specific device is selected, all OpenACC directives until the next call to acc_set_device_num will go to the selected device.

This function may be called with different values from different threads.

Querying the Device Number

The acc_get_device_num function returns the device number currently being used for a given type.

```
        int acc_get_device_num(type)
```

This function is frequently confused with acc_get_num_devices. One function queries *how many devices* are available of a type, the other queries *which device* of that type will be used.

Set Device Example

```
 def= acc_get_device_num(acc_device_default);
#pragma acc parallel loop async
  for(int i=0; i<N; ++i)
    A[i] = i;

  acc_set_device_num(1,acc_device_default);
#pragma acc parallel loop async
  for(int i=0; i<N; ++i)
    B[i] = 2. * i;
#pragma acc wait

  acc_set_device_num(def,acc_device_default);
#pragma acc wait
#pragma acc parallel loop
  for(int i=0; i<N; ++i)
    C[i] = A[i] + B[i];
```

Asynchronously run loop on default device.

Asynchronously run loop on device 1, potentially concurrently with previous. Then wait for results from device 1.

Change back to default device, wait for completion, and finally run summation loop on default device.

Multi-GPU Case Study (All-OpenACC)

Multi-Device Pipeline

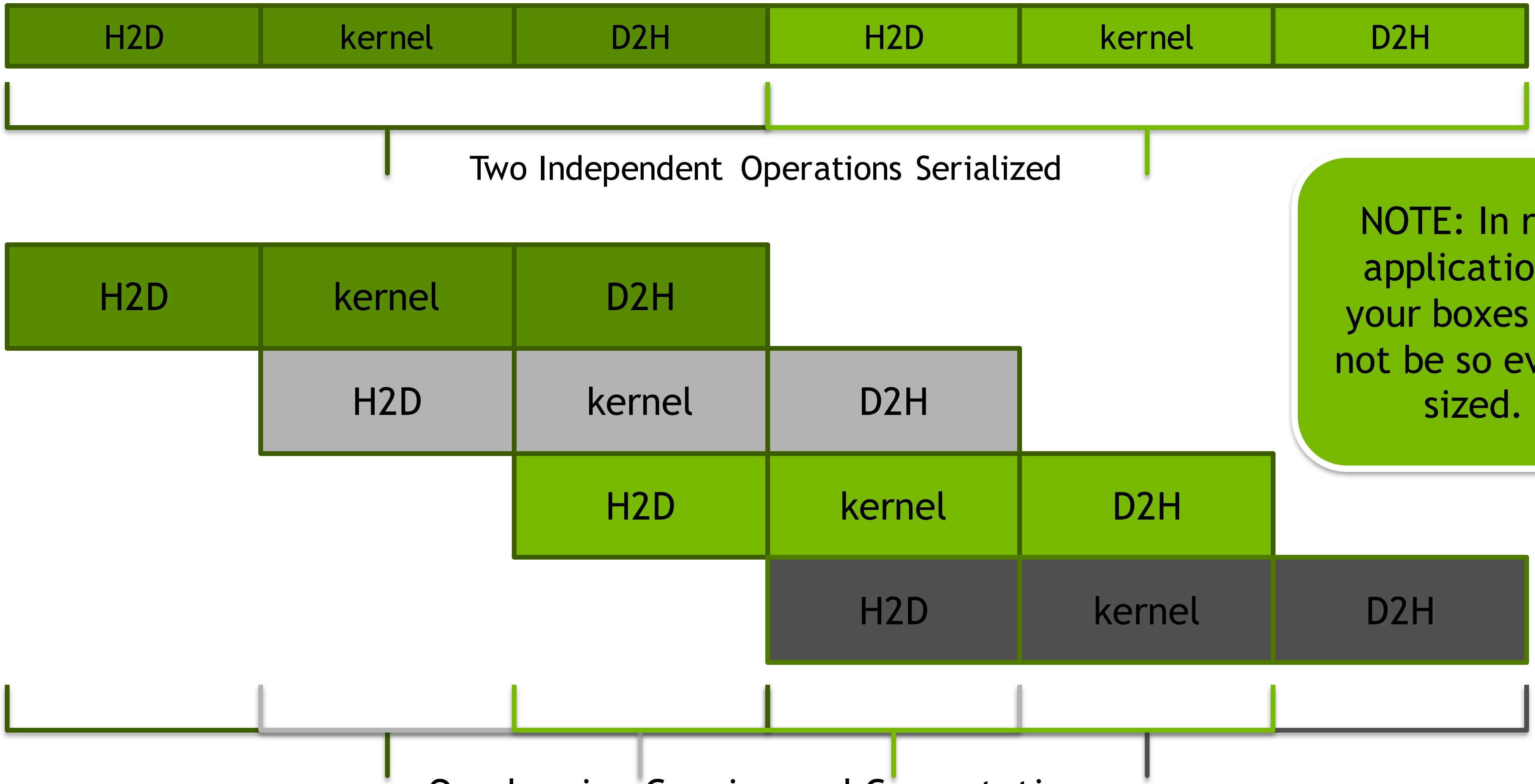
A Case Study

In NVIDIA's Fall 2016 OpenACC Course I demonstrated how to pipeline an image filter to overlap data copies and compute kernels.

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

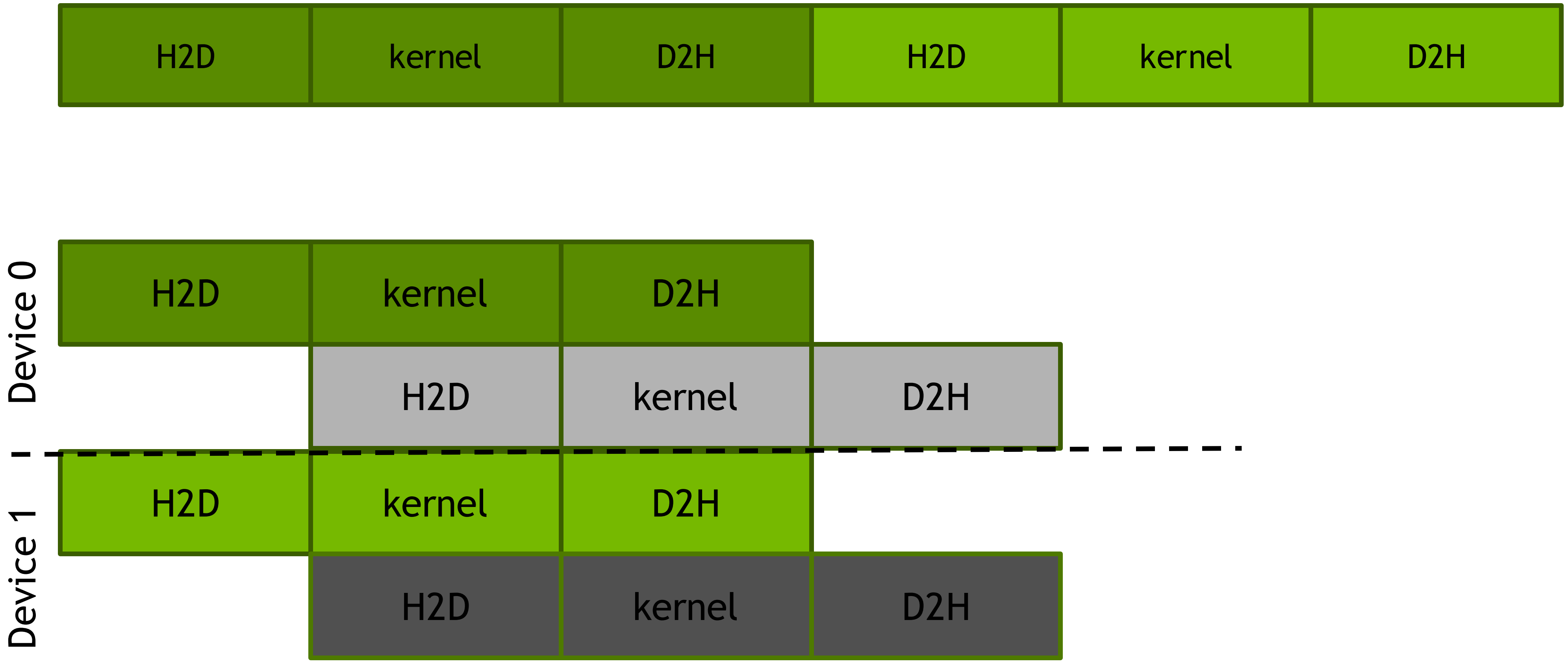
Since each part is independent, they can easily be run on different devices.

Pipelining in a Nutshell


H2D
kernel
D2H
H2D
kernel
D2H
Two Independent Operations Serialized
H2D
kernel
D2H
H2D
kernel
D2H
H2D
kernel
D2H
H2D
kernel
D2H
Overlapping Copying and Computation

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

Multi-device Pipelining in a Nutshell



Multi-GPU Pipelined Code

```
#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++ ) {
        float blue = 0.0, green = 0.0, red = 0.0;
        <filter code removed for space>
        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(queue)
    queue = (queue%3)+1;
  }
#pragma acc wait
  }
  }
```

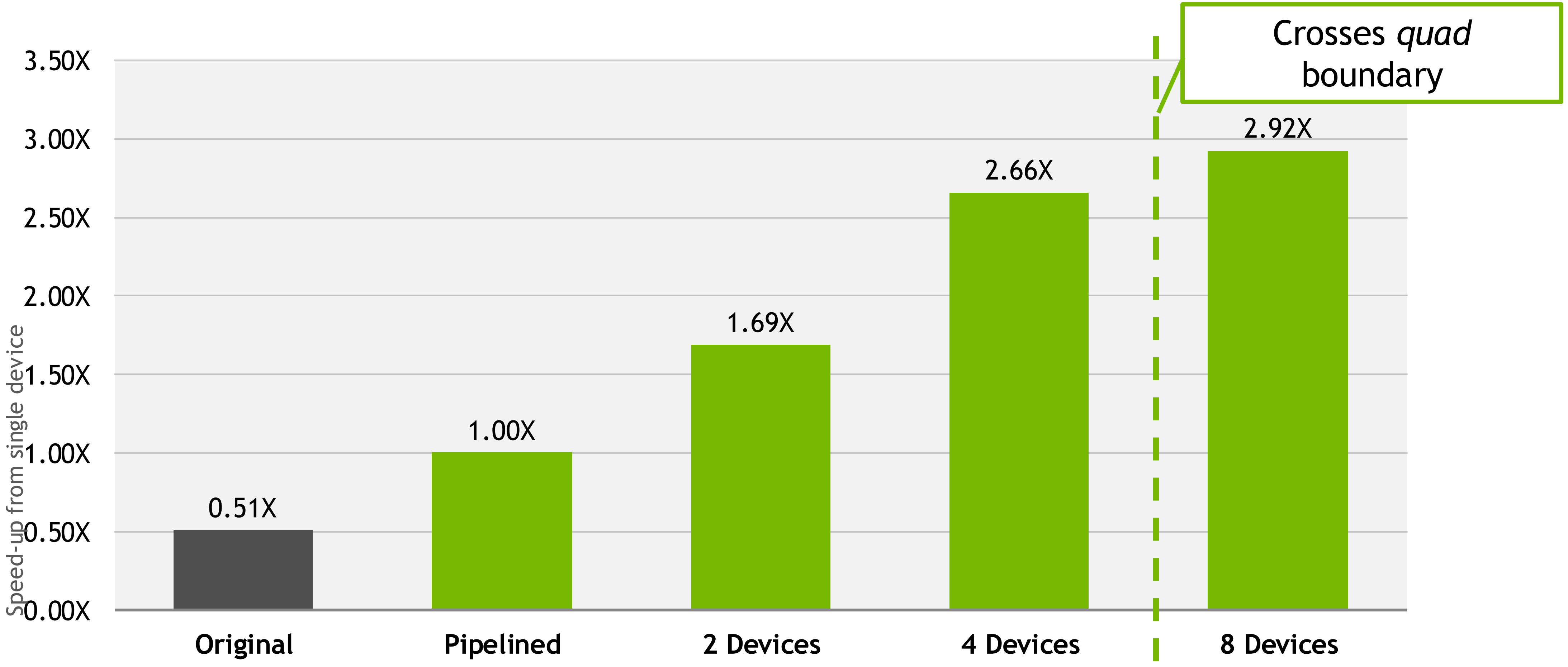
Spawn 1 thread per device.

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)

Multi-GPU Case Study (MPI+OpenACC)

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.

There is no portable way to get a local rank or to map a rank to the GPU(s) with good affinity to the rank.

The MPI launcher (mpirun, mpiexec, aprun, etc.) can generally help you place ranks on particular CPUs to improve affinity mapping.

For more details about best practices using MPI and GPUs, including GPU affinity see S7133 - MULTI-GPU PROGRAMMING WITH MPI

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();

$ mpirun --bind-to core --npersocket 4 ...
```

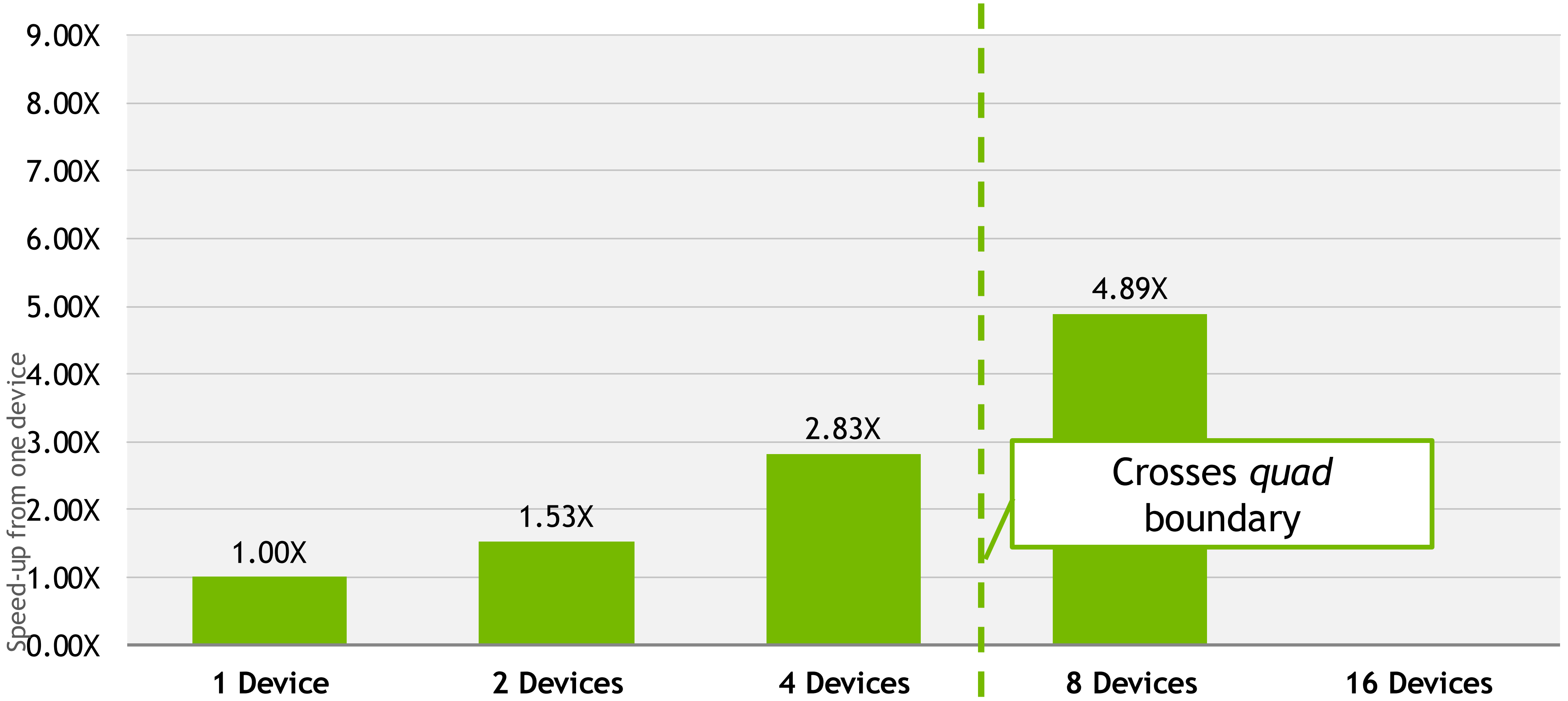
Decompose image across processes (ranks)

Receive final parts from all ranks.

Launch with good GPU/process affinity

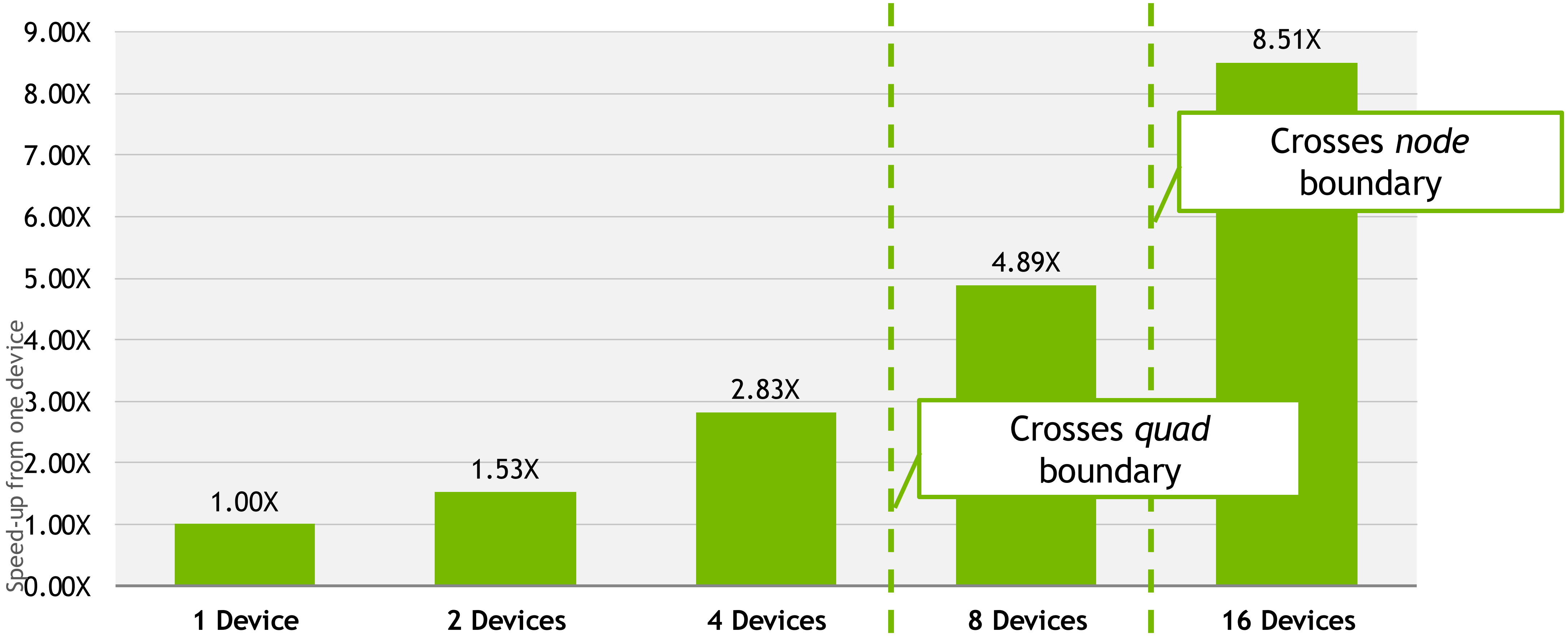
There's a variety of ways to do MPI decomposition, this is what I used for this particular example.

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

Conclusions

Conclusions

OpenACC provides an API for managing multiple devices on a single node.

One approach to multiple devices is to manage them all with a single process.

- Developer only needs to worry about a single process; all GPUs read from/write to the same host memory; no additional API dependency required.
- Must be cautious of shared memory race conditions and improper use of asynchronous directives; does not handle GPU affinity very well when significant

Another approach is to use MPI with OpenACC

- Many applications already use MPI (little/no changes required); simple way to handle affinity; opens possibility of running on multiple nodes
- Adds dependency on MPI; domain decomposition can be tricky, may increase memory footprint