OpenACC Course

Lecture 4: Advanced OpenACC Techniques

November 12, 2015

Lecture Objective:

Demonstrate OpenACC pipelining, Interoperating with Libraries, and Use with MPI.

Course Syllabus

Oct 1: Introduction to OpenACC

Oct 6: Office Hours

Oct 15: Profiling and Parallelizing with the OpenACC Toolkit

Oct 20: Office Hours

Oct 29: Expressing Data Locality and Optimizations with OpenACC

Nov 3: Office Hours

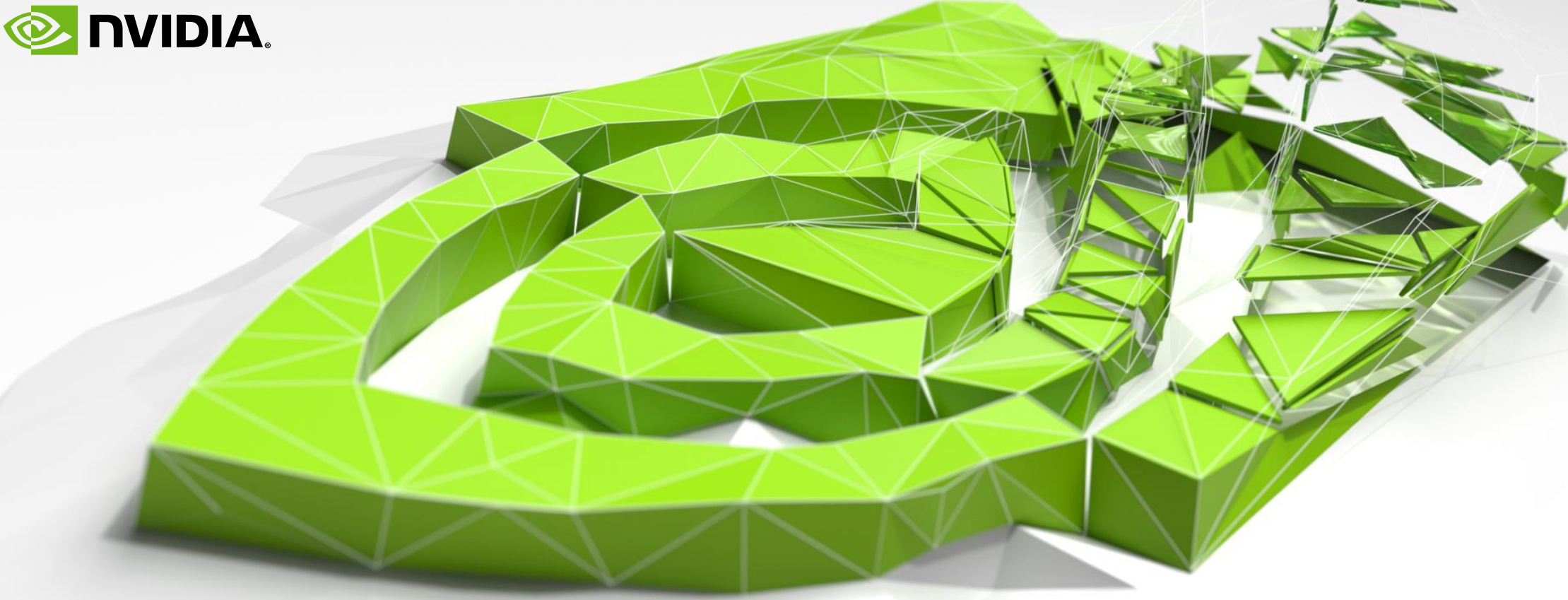
Nov 12: Advanced OpenACC Techniques

Nov 24: Office Hours

Recordings:
https://developer.nvidia.com/openacc-course

Advanced OpenACC Techniques: Pipelining, MPI and Interoperability

Jiri Kraus, NVIDIA Developer Technologies



Agenda

Part 1: Asynchronous Programming with OpenACC

Part 2: Multi GPU Programming with MPI and OpenACC

Asynchronous Programming with OpenACC

Asynchronous Programming

Programming such that two or more unrelated operations can occur independently or even at the same time without immediate synchronization.

Real World Examples:

- Cooking a Meal: Boiling potatoes while preparing other parts of the dish.
- Three students working on a project on George Washington, one researches his early life, another his military career, and the third his presidency.
- Automobile assembly line: each station adds a different part to the car until it is finally assembled.

Asynchronous Example 1

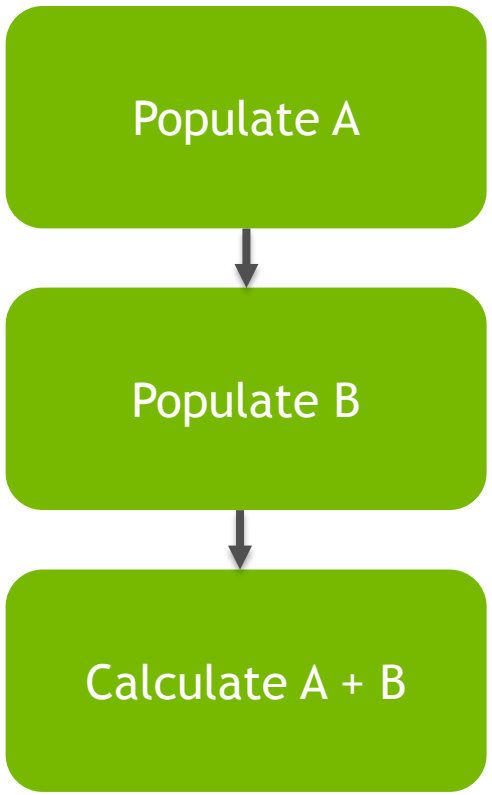
I want to populate two arrays, A and B, with data, then add them together. This requires 3 distinct operations.

1. Populate A
2. Populate B
3. Add A + B

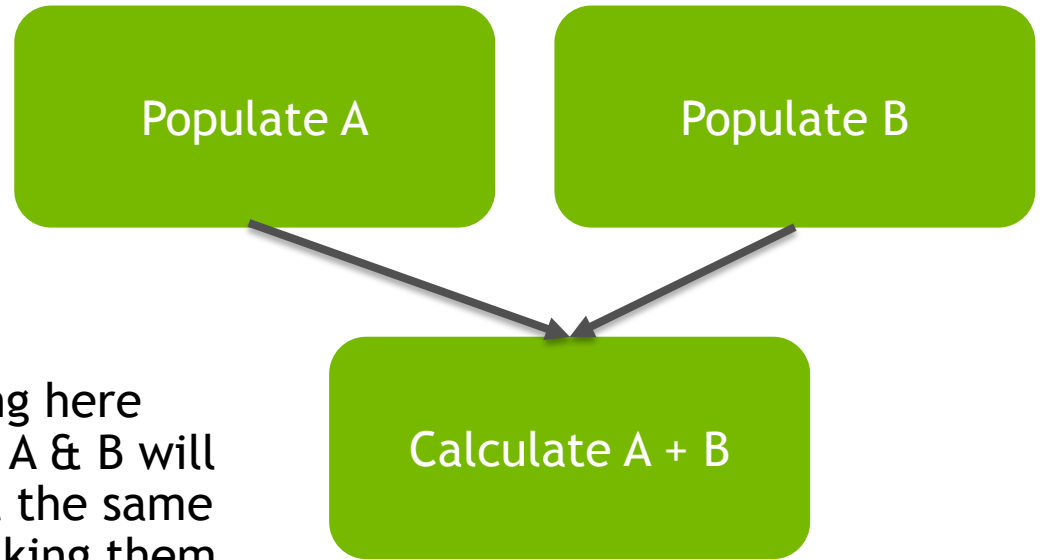
Tasks 1 and 2 are independent, but task 3 is dependent on both.

Asynchronous Example 1 cont.

Synchronous Execution



Asynchronous Execution



Note: Nothing here guarantees that A & B will be populated at the same time, but by making them asynchronous, it's now possible to run them in parallel.

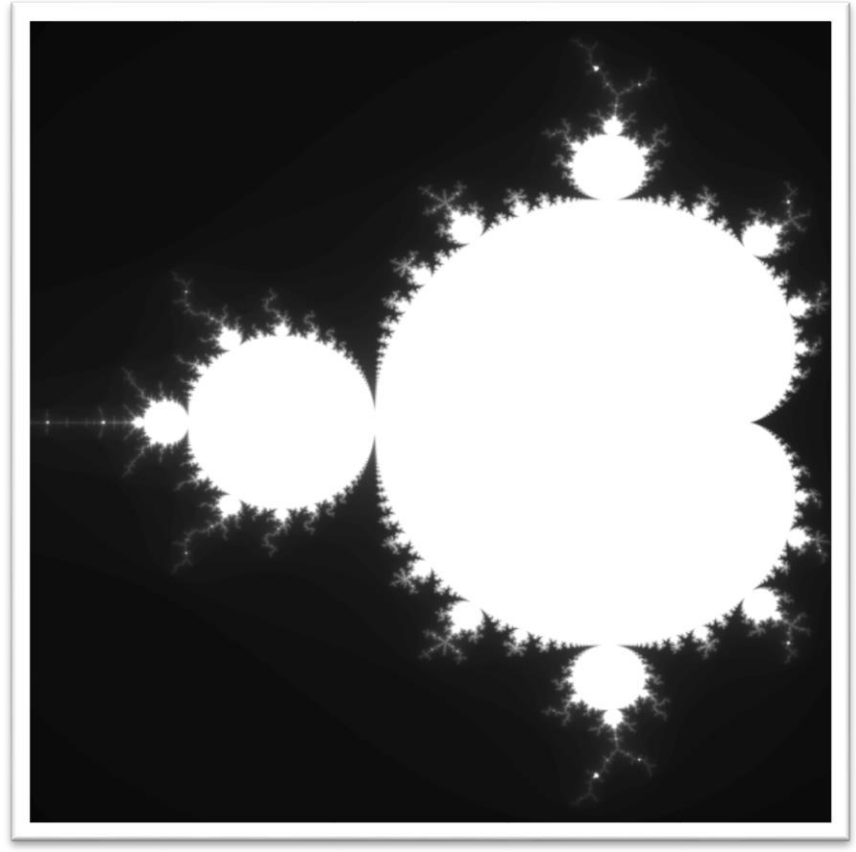
Asynchronous Pipelining

- Very large operations may frequently be broken into smaller parts that may be performed independently.
- Pipeline Stage -A single step, which is frequently limited to 1 part at a time

Photo by Roger Wollstadt, used via Creative Commons

Case Study: Mandelbrot Set

- Application generates the image to the right.
- Each pixel in the image can be independently calculated.
- Skills Used:
  - Parallel Loop or Kernels Directive
  - Data Region
  - Update Directive
  - Asynchronous Pipelining



Mandelbrot code

```
// Calculate value for a pixel
unsigned char mandelbrot(int Px, int Py) {
  double x0=xmin+Px*dx;   double y0=ymin+Py*dy;
  double x=0.0;   double y=0.0;
  for(int i=0;x*x+y*y<4.0 && i<MAX_ITERS;i++) {
    double xtemp=x*x-y*y+x0;
    y=2*x*y+y0;
    x=xtemp;
  }
  return (double)MAX_COLOR*i/MAX_ITERS;
}

// Used in main()
for(int y=0;y<HEIGHT;y++) {
  for(int x=0;x<WIDTH;x++) {
    image[y*WIDTH+x]=mandelbrot(x,y);
  }
}
```

The mandelbrot() function calculates the color for each pixel.

Within main() there is a doubly-nested loop that calculates each pixel independently.

OpenACC Routine Directive

Specifies that the compiler should generate a device copy of the function/subroutine and what type of parallelism the routine contains.

Clauses:

`gang`/`worker`/`vector`/`seq`

Specifies the level of parallelism contained in the routine.

`bind`

Specifies an optional name for the routine, also supplied at call-site

`no_host`

The routine will only be used on the device

`device_type`

Specialize this routine for a particular device type.

Routine Directive: C/C++

```
// foo.h
#pragma acc routine seq
double foo(int i);
```

```
// Used in main()
#pragma acc parallel loop
for(int i=0;i<N;i++) {
  array[i] = foo(i);
}
```

- At function source:
  - Function needs to be built for the GPU.
  - It will be called by each thread (sequentially)
- At call the compiler needs to know:
  - Function will be available on the GPU
  - It is a sequential routine

OpenACC Routine: Fortran

```
module foo_mod
 contains
 real(8) function foo(i)
    implicit none
    !$acc routine(foo) seq
    integer, intent(in), value :: i
    ...
 end function foo
end module foo_mod
```

The **routine** directive may appear in a Fortran function or subroutine definition, or in an interface block.

The save attribute is not supported.

Nested acc routines require the routine directive within each nested routine.

Step 1 code

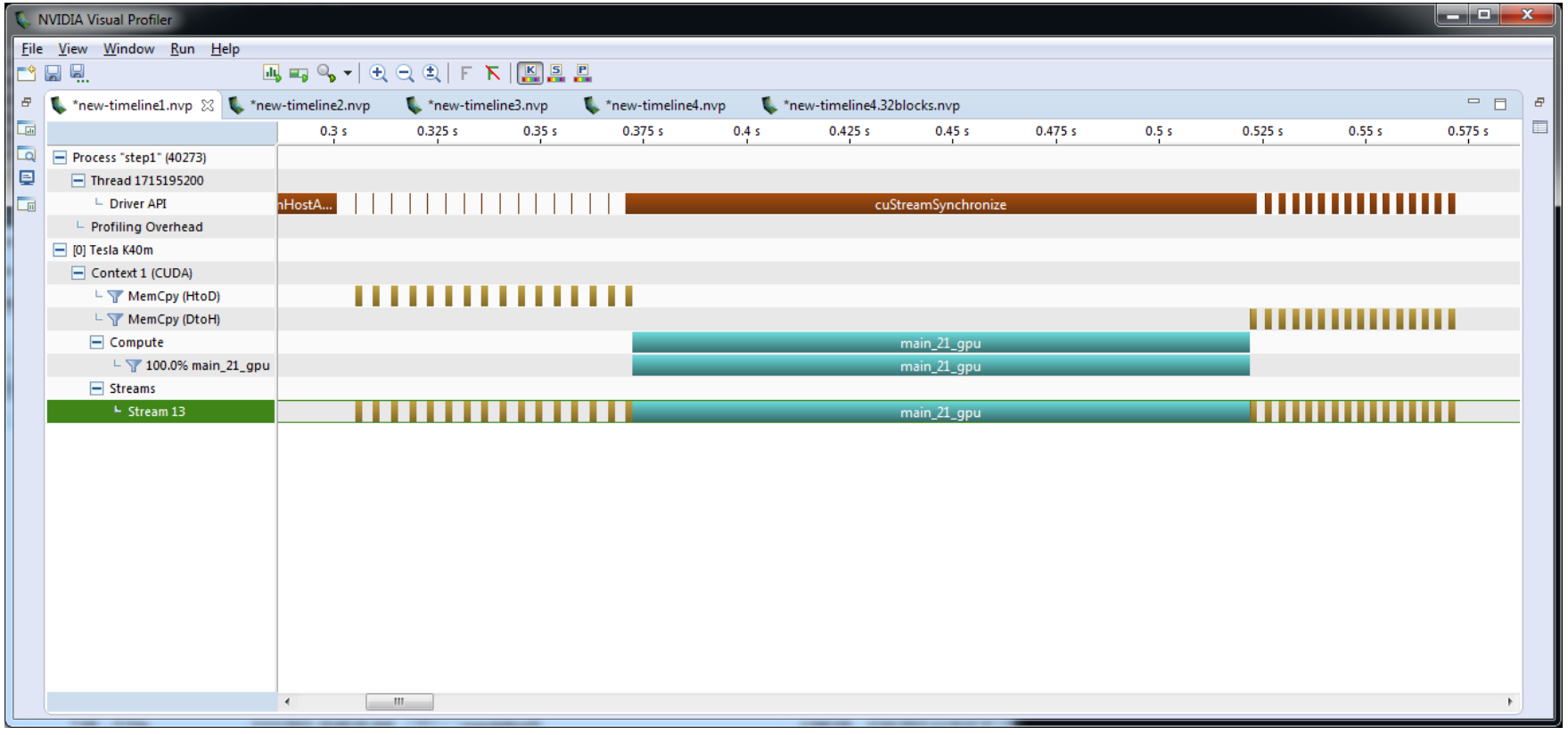
```
// In mandelbrot.h
#pragma acc routine seq
unsigned char mandelbrot(int Px, int Py);
```

The mandelbrot() function must be declared a sequential *routine*.

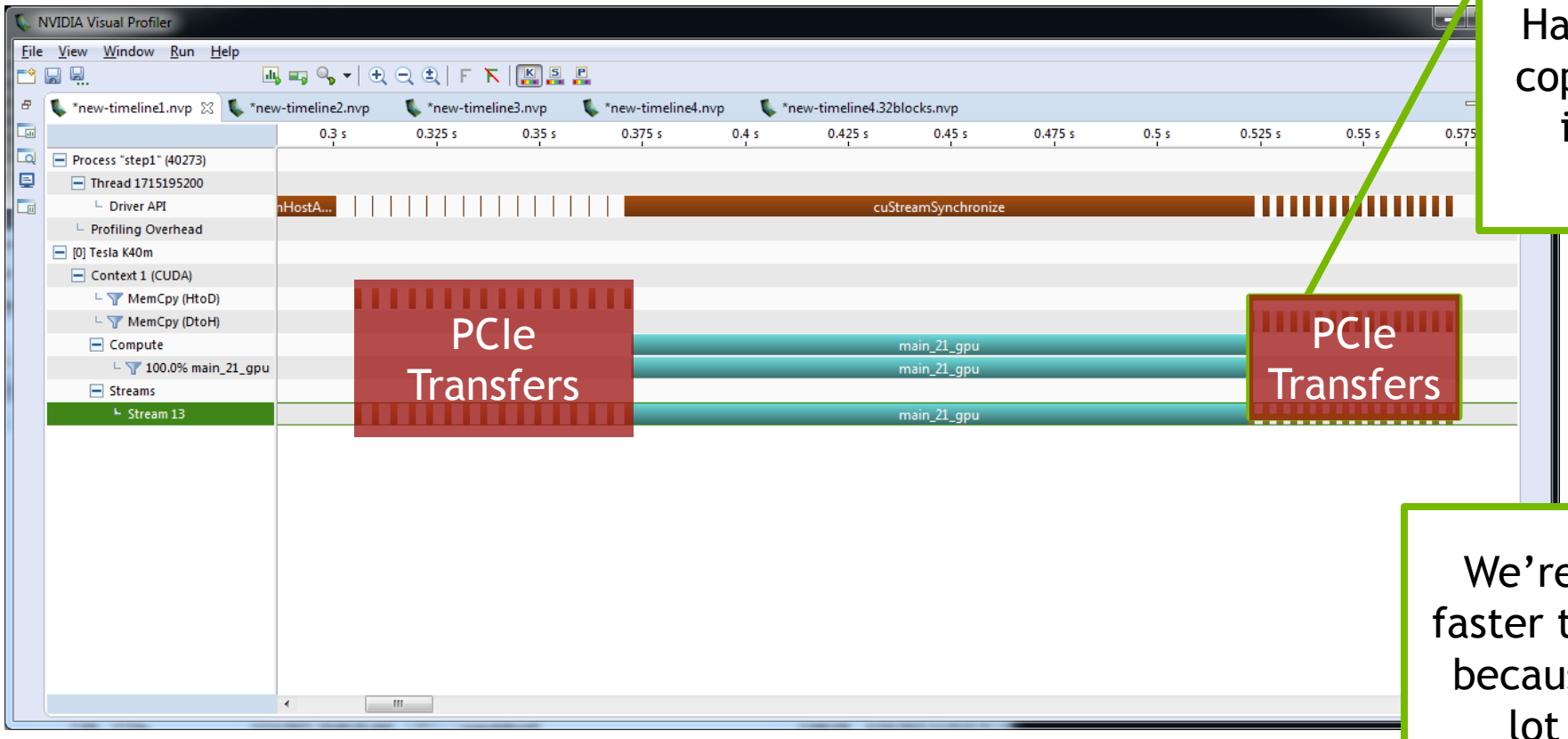
```
// Used in main()
#pragma acc parallel loop
for(int y=0;y<HEIGHT;y++) {
  for(int x=0;x<WIDTH;x++) {
    image[y*WIDTH+x]=mandelbrot(x,y);
  }
}
```

The main loops are parallelizes with *parallel loop* or *kernels*.

Step 1 Profile



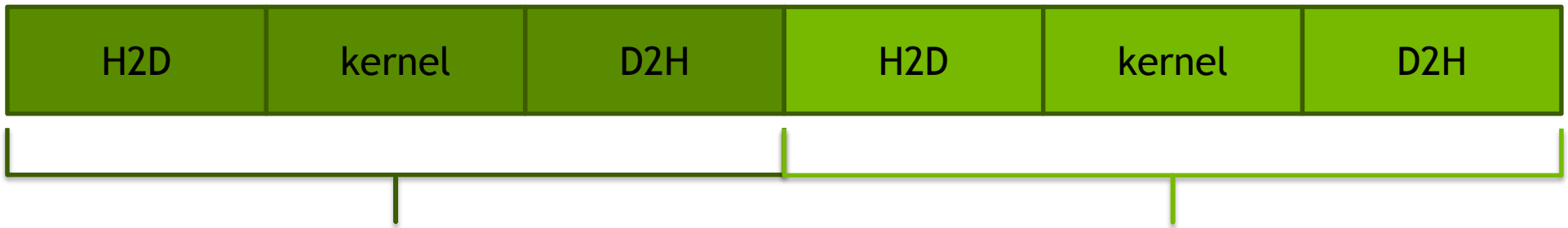
Step 1 Profile



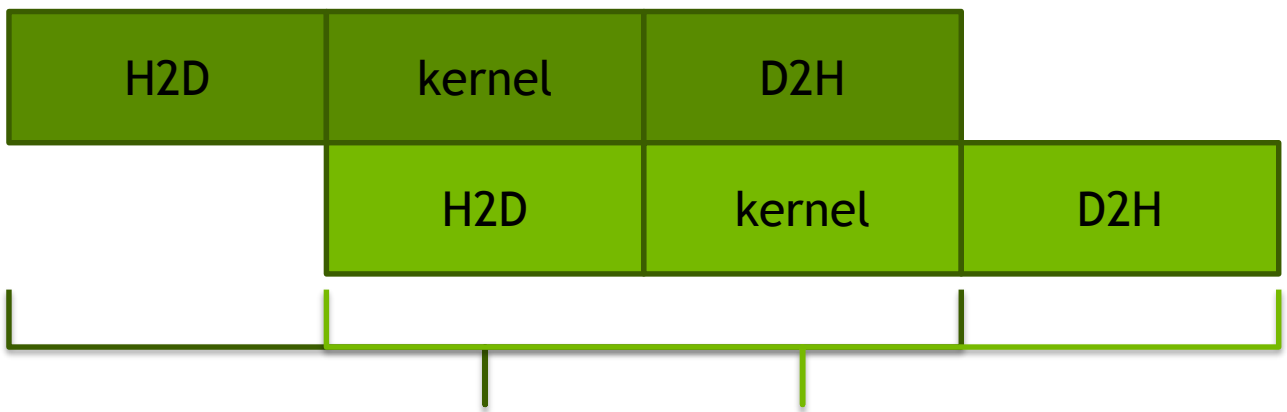
Half of our time is copying, none of it is overlapped.

We're still much faster than the CPU because there's a lot of work.

Pipelining Data Transfers



Two Independent Operations Serialized



Overlapping Copying and Computation

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

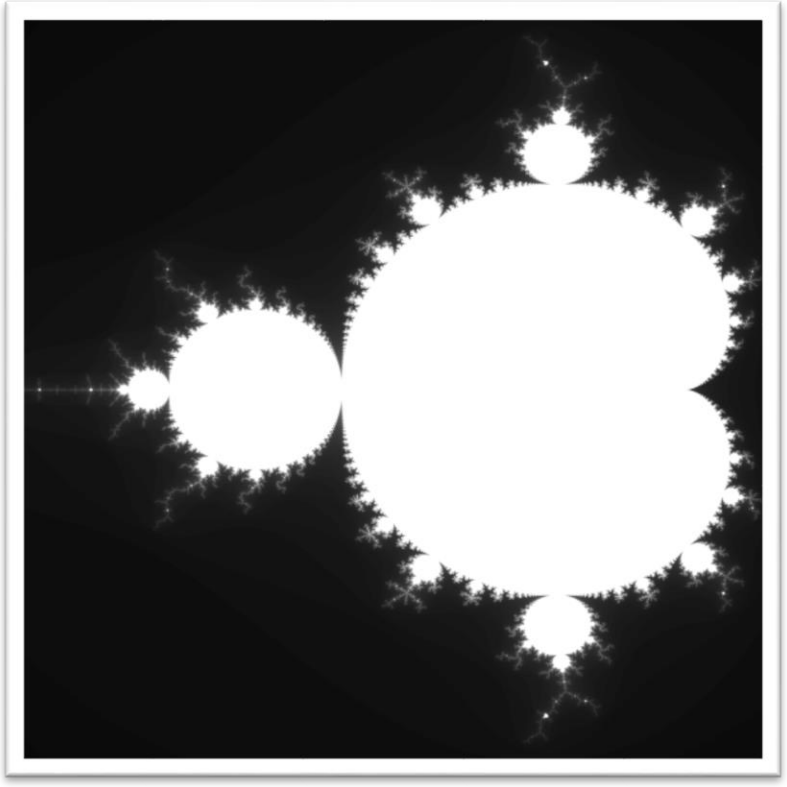
Pipelining Mandelbrot set

We only have 1 kernel, so there's nothing to overlap.

Since each pixel is independent, computation can be broken up

Steps

1. Break up computation into blocks along rows.
2. Break up copies according to blocks
3. Make both computation and copies asynchronous



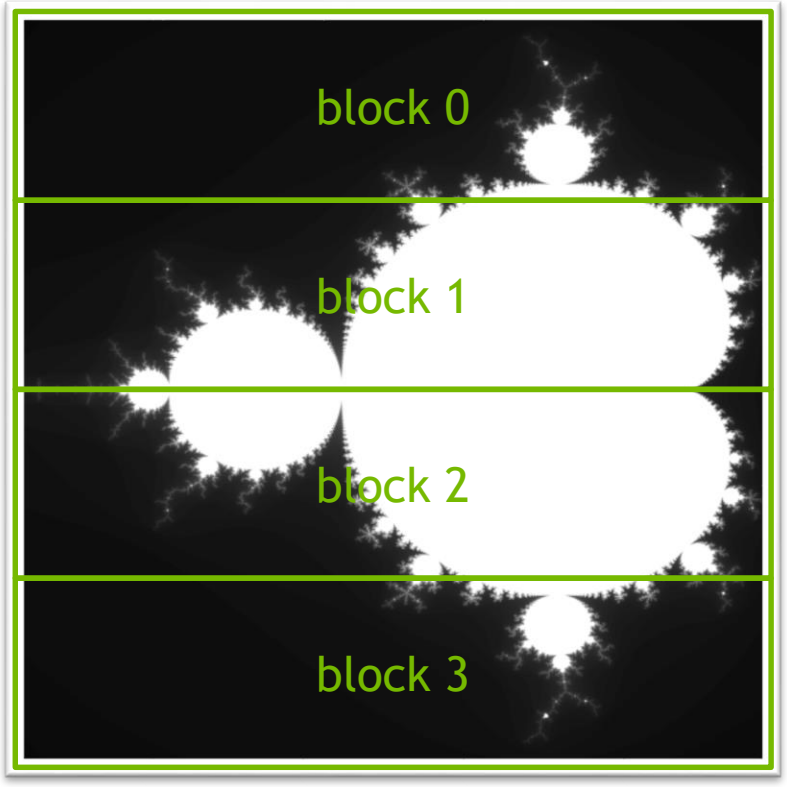
Pipelining Mandelbrot set

We only have 1 kernel, so there's nothing to overlap.

Since each pixel is independent, computation can be broken up

Steps

1. Break up computation into blocks along rows.
2. Break up copies according to blocks
3. Make both computation and copies asynchronous



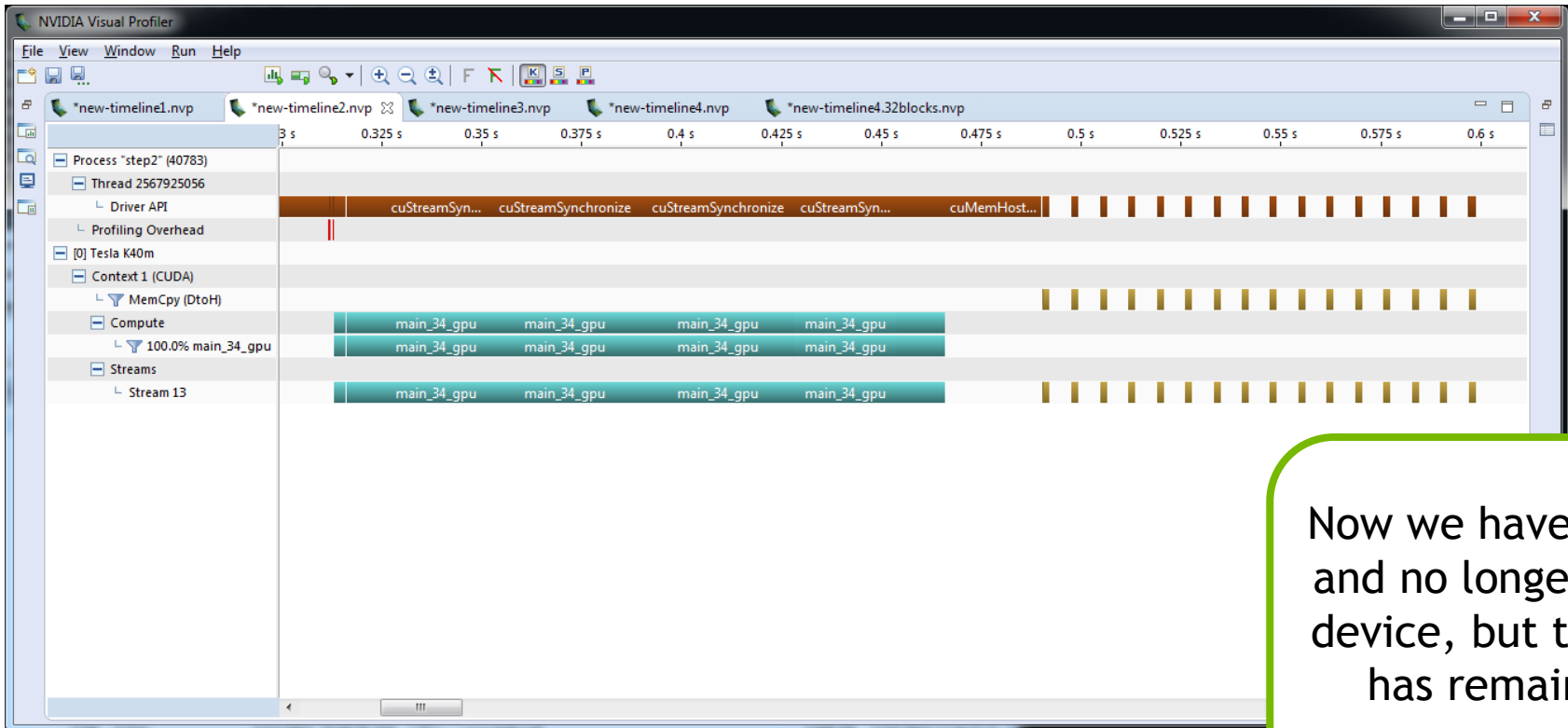
Step 2: Blocking Computation

```
24   numblocks = ( argc > 1 ) ? atoi(argv[1]) : 8;
25   blocksize = HEIGHT / numblocks;
26   printf("numblocks: %d, blocksize: %d\n",
             numblocks, blocksize);
27
28 #pragma acc data copyout(image[:bytes])
29   for(int block=0; block < numblocks; block++)
30   {
31     int ystart = block * blocksize;
32     int yend   = ystart + blocksize;
33 #pragma acc parallel loop
34     for(int y=ystart;y<yend;y++) {
35       for(int x=0;x<WIDTH;x++) {
36         image[y*WIDTH+x]=mandelbrot(x,y);
37       }
38     }
39   }
```

NOTE: We don't need to copy in the array, so make it an explicit copyout.

- Add a loop over blocks
- Modify the existing row loop to only work within blocks
- Add data region around blocking loop to leave data local to the device.
- Check for correct results.

Blocking Timeline



Now we have 8 kernel launches and no longer copy data *to* the device, but the execution time has remained roughly the same.

OpenACC Update Directive

Programmer specifies an array (or part of an array) that should be refreshed within a data region.

```
!$acc data create(a)

do_something_on_device()
!$acc update self(a)

do_something_on_host()
!$acc update device(a)

!$acc end data
```

Copy “a” from GPU to CPU

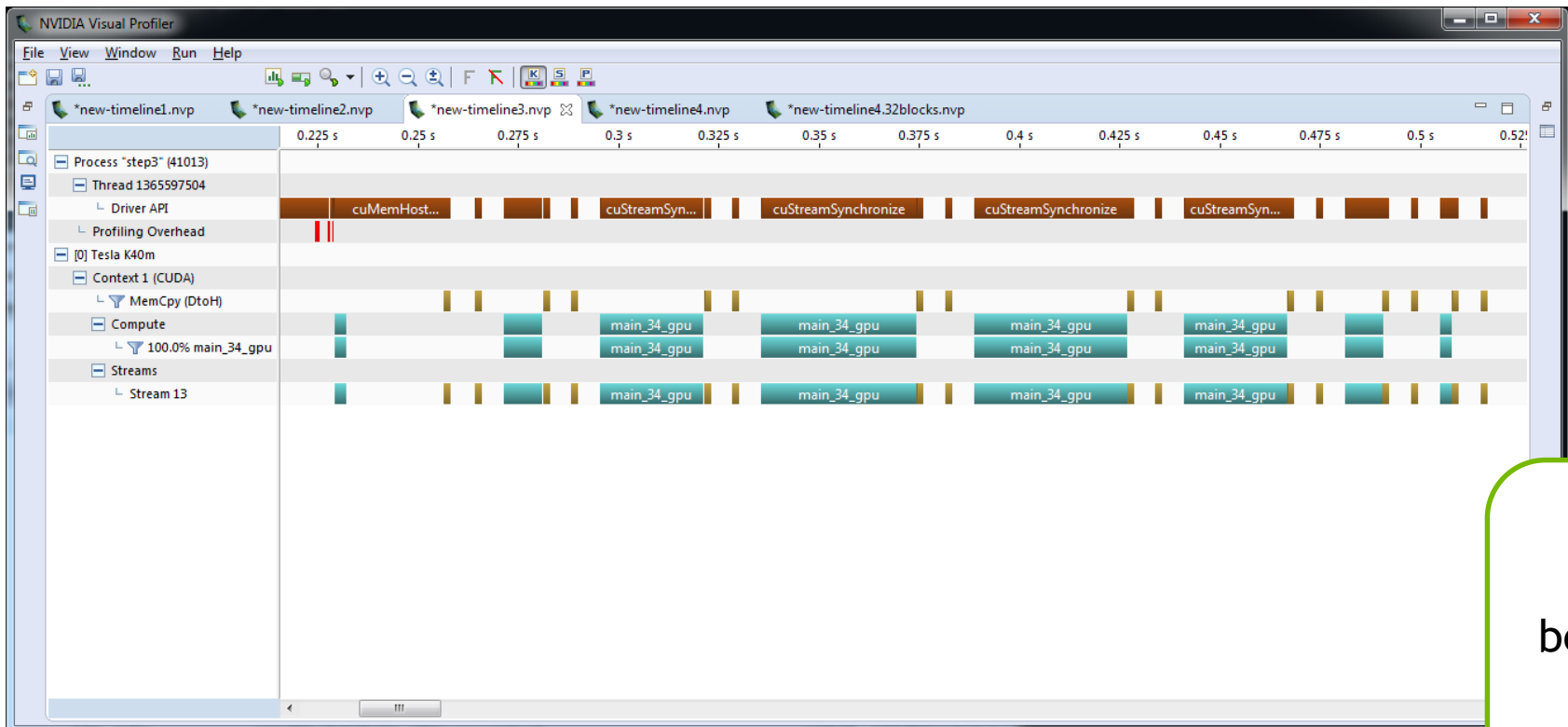
Copy “a” from CPU to GPU

Step 3: Copy By Block

```
#pragma acc data create(image[:bytes])
for(int block=0; block < numblocks;
    block++)
{
  int ystart = block * blocksize;
  int yend   = ystart + blocksize;
#pragma acc parallel loop
  for(int y=ystart;y<yend;y++) {
    for(int x=0;x<WIDTH;x++) {
      image[y*WIDTH+x]=mandelbrot(x,y);
    }
  }
#pragma acc update
 self(image[ystart*WIDTH:WIDTH*blocksize])
}
```

- Change the data region to only create the array on the GPU
- Use an update directive to copy individual blocks back to the host when complete
- Check for correct results.

Timeline: Updating by Blocks



We're now updating between blocks, but not overlapping.

OpenACC async and wait

async(n): launches work asynchronously in queue *n*

wait(n): blocks host until all operations in queue *n* have completed

Can significantly reduce launch latency, enables pipelining and concurrent operations

```
#pragma acc parallel loop async(1)
...
#pragma acc parallel loop async(1)
for(int i=0; i<N; i++)
  ...
#pragma acc wait(1)
for(int i=0; i<N; i++)
```

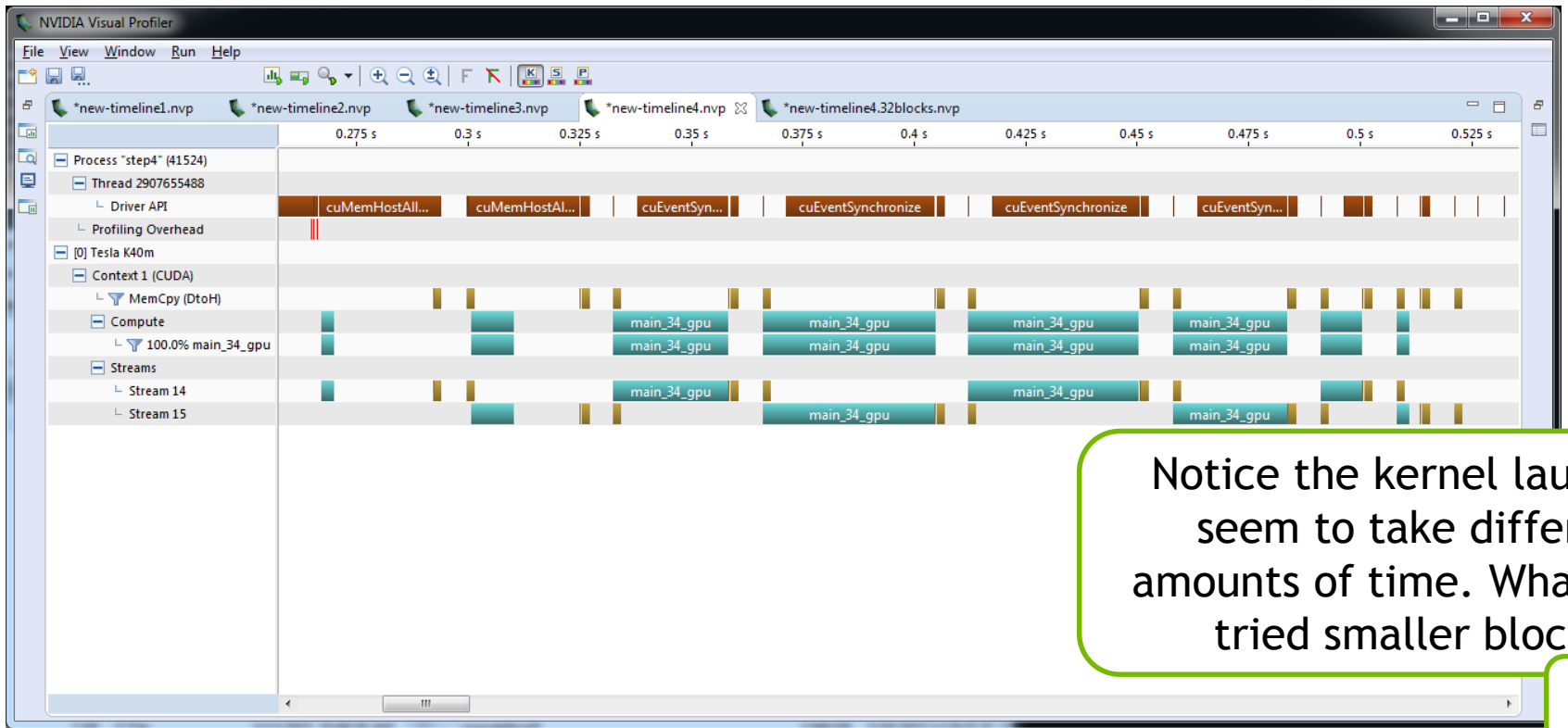
If *n* is not specified, *async* will go into a default queue and *wait* will wait all previously queued work.

Step 4: Go Asynchronous

```
#pragma acc data create(image[:bytes])
for(int block=0; block < numblocks; block++)
{
  int ystart = block * blocksize;
  int yend   = ystart + blocksize;
#pragma acc parallel loop async(block%2)
  for(int y=ystart;y<yend;y++) {
    for(int x=0;x<WIDTH;x++) {
      image[y*WIDTH+x]=mandelbrot(x,y);
    }
  }
#pragma acc update self(image[ystart*WIDTH:WIDTH*blocksize]) async(block%2)
}
#pragma acc wait
```

- Make each parallel region asynchronous by placing in different queues.
- Make each update asynchronous by placing in same stream as the parallel region on which it depends
- Synchronize for all to complete.
- Check for correct results.

Timeline: Pipelining



Notice the kernel launches seem to take differing amounts of time. What if we tried smaller blocks?

Homework

Mandelbrot

Homework

The Homework for this case study is available in the "Pipelining Work on the GPU with OpenACC" lab at https://nvidia.qwiklab.com/ and consists of 4 steps

1. Use OpenACC `routine` and `parallel loop` or `kernels` directive to make generate the image on the GPU.
2. Break the image creation into blocks by adding a blocking loop around the existing loops and changing the "y" loop to operate on blocks.
3. Change the data region to *create* the image array and use the *update* directive to copy each block back upon completion.
4. Use the block numbers to place blocks in multiple *async* queues and *wait* for all queues to complete. Experiment with the number of blocks and queues.

Multi-GPU Programming

Multi-GPU OpenACC (Single-threaded)

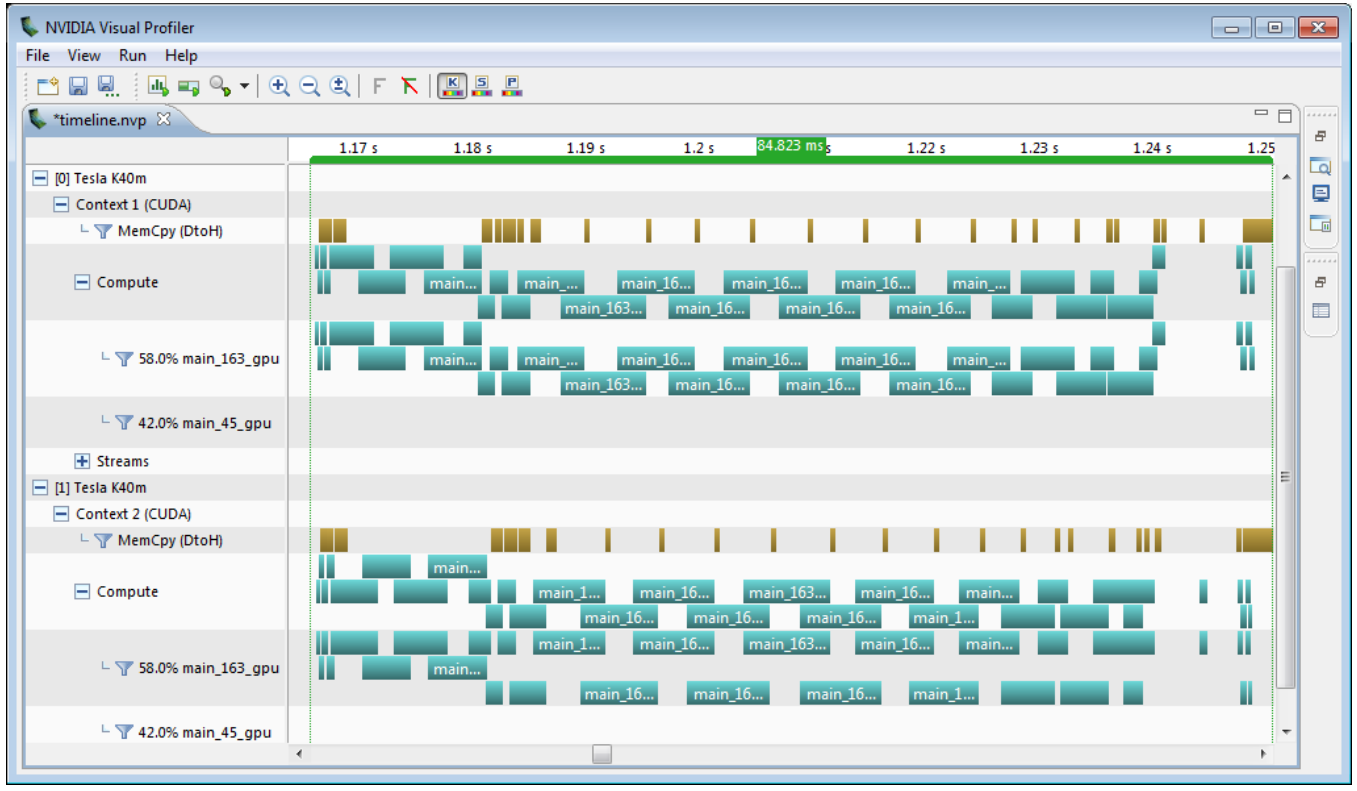
```
 for (int gpu=0; gpu < 2 ; gpu ++)
  {
    acc_set_device_num(gpu,acc_device_nvidia);
#pragma acc enter data create(image[:bytes])
  }

  for(int block=0; block < numblocks; block++)
  {
    int ystart = block * blocksize;
    int yend   = ystart + blocksize;
    acc_set_device_num(block%2,acc_device_nvidia);
#pragma acc parallel loop async(block%2)
    for(int y=ystart;y<yend;y++) {
      for(int x=0;x<WIDTH;x++) {
        image[y*WIDTH+x]=mandelbrot(x,y);
      }
    }
#pragma acc update self(image[ystart*WIDTH:WIDTH*blocksize]) async(block%2)
  }
  for (int gpu=0; gpu < 2 ; gpu ++)
  {
    acc_set_device_num(gpu,acc_device_nvidia);
#pragma acc wait
#pragma acc exit data delete(image)
  }
```

Allocate space on each device

Alternate devices per block

Clean up the devices

Multi-GPU Mandelbrot Profile



Multi-GPU OpenACC with OpenMP

```
#pragma omp parallel
{
  int my_gpu = omp_get_thread_num();
  acc_set_device_num(my_gpu,acc_device_nvidia);

  #pragma acc data create(image[0:HEIGHT*WIDTH])
  {
    int queue = 1;
    #pragma omp for schedule(static,1) firstprivate(queue)
    for(int block=0; block < numblocks; block++)
    {
      int ystart = block * blocksize;
      int yend   = ystart + blocksize;
      #pragma acc parallel loop async(queue)
      for(int y=ystart;y<yend;y++) {
        for(int x=0;x<WIDTH;x++) {
          image[y*WIDTH+x]=mandelbrot(x,y);
        }
      }
      #pragma acc update self(image[ystart*WIDTH:WIDTH*blocksize]) async(queue)
      queue = queue%2+1;
    }
    #pragma acc wait
  }
}
```

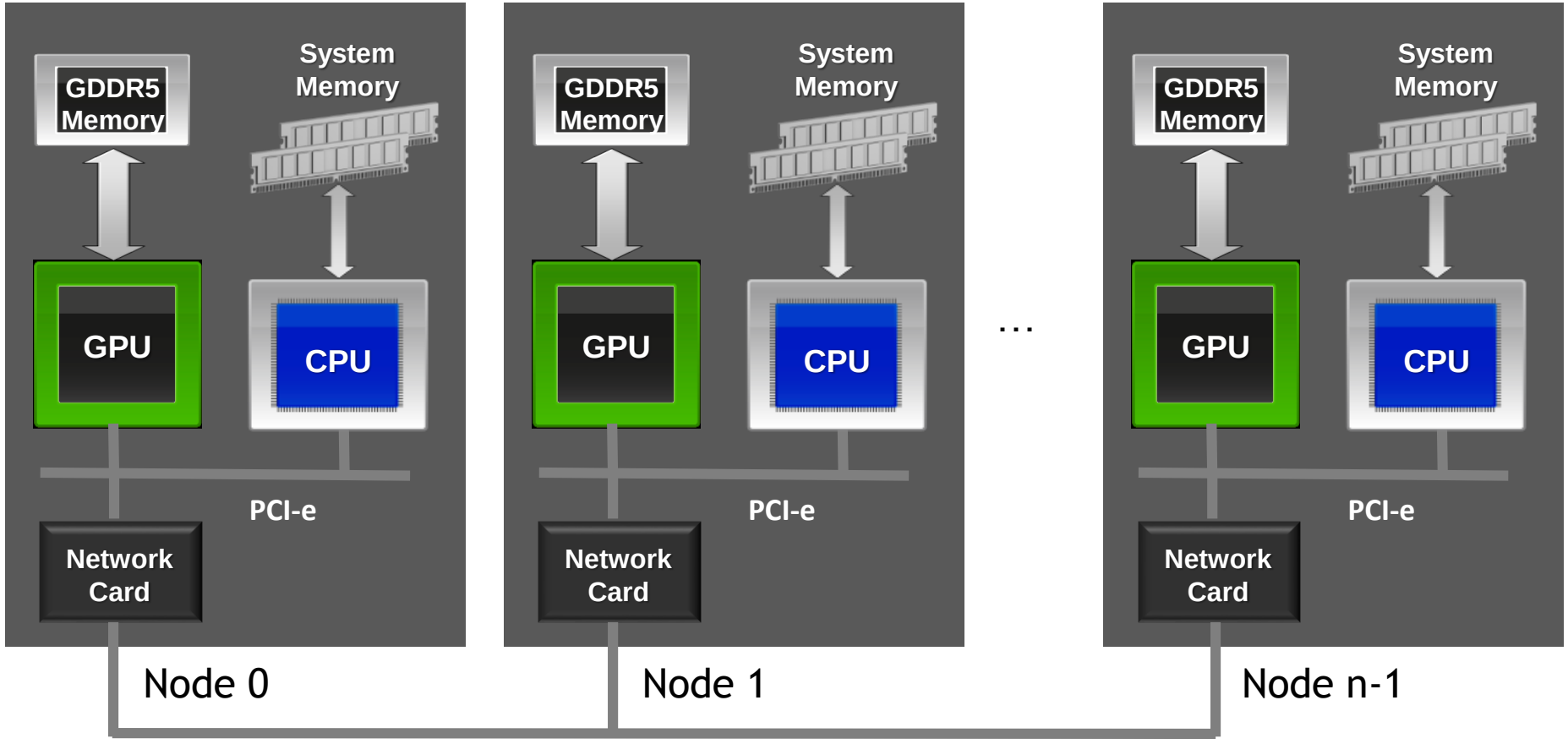
One OpenMP thread per device

Use multiple queues per device to get copy compute overlap

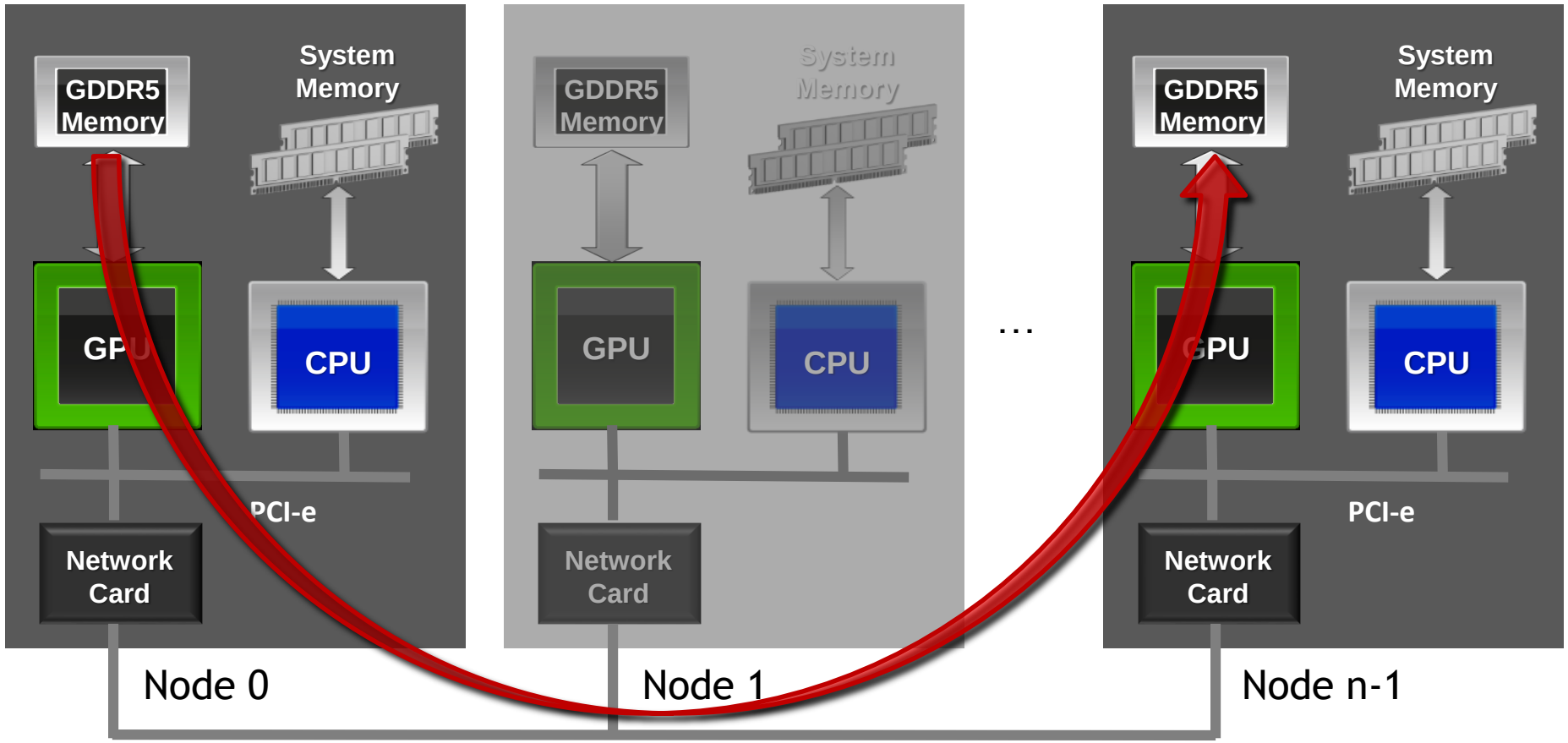
Wait for all work to complete

Multi GPU Programming with MPI and OpenACC

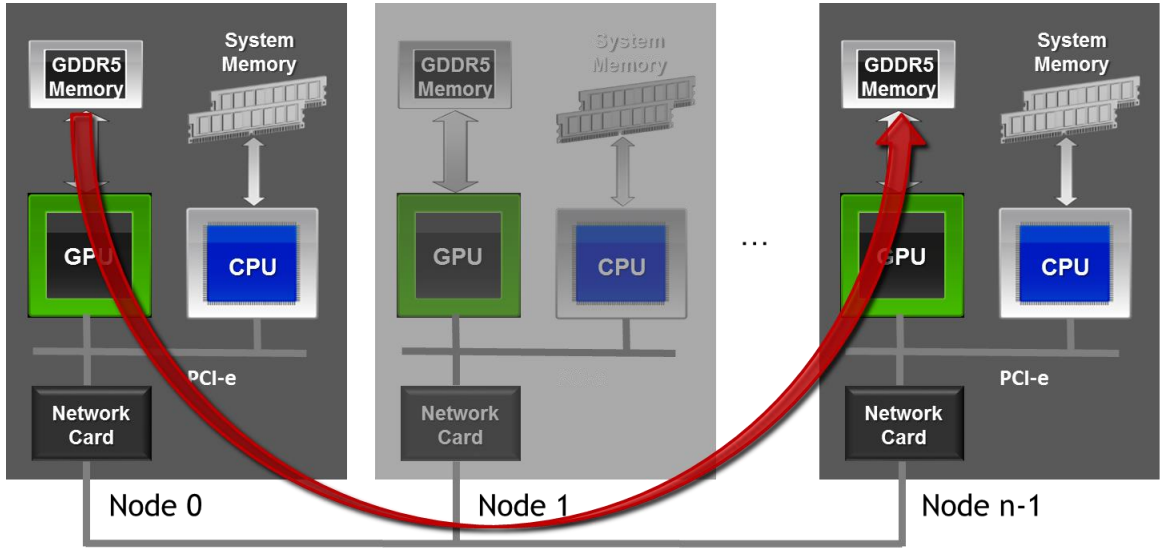
MPI+OpenACC


GDDR5 Memory
System Memory
GPU
CPU
PCI-e
Network Card
Node 0
GDDR5 Memory
System Memory
GPU
CPU
PCI-e
Network Card
Node 1
…
GDDR5 Memory
System Memory
GPU
CPU
PCI-e
Network Card
Node n-1

MPI+OpenACC


GDDR5 Memory
System Memory
GPU
CPU
PCI-e
Network Card
Node 0
GDDR5 Memory
System Memory
GPU
CPU
Network Card
Node 1
…
GDDR5 Memory
System Memory
GPU
CPU
PCI-e
Network Card
Node n-1

MPI+OpenACC




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

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

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)

- Bindings 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_size(MPI_COMM_WORLD,&size);

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    /* Call MPI routines like MPI_Send, MPI_Recv, ... */

    ...

    /* Shutdown MPI library */

    MPI_Finalize();

    return 0;

}
```

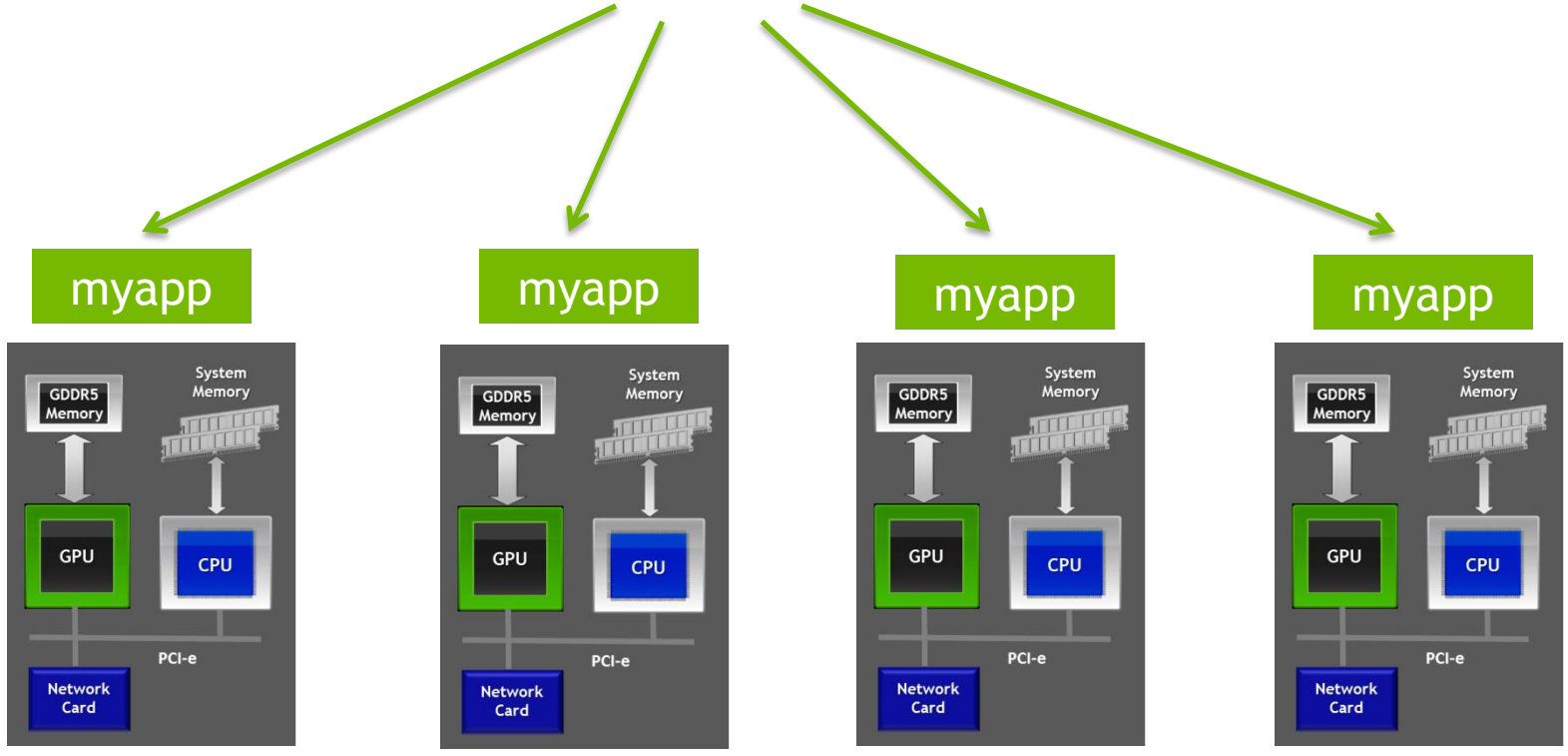
MPI – A Minimal Program

```
#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_size(MPI_COMM_WORLD,&size);

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    /* Call MPI routines like MPI_Send, MPI_Recv, ... */

    ...

    /* Shutdown MPI library */

    MPI_Finalize();

    return 0;

}
```

Remark: Almost all MPI routines return an error value which should be check. The examples and tasks leave that out for brevity.

MPI – Compiling and Launching

```
$ mpicc -o myapp myapp.c

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



Example: Jacobi Solver

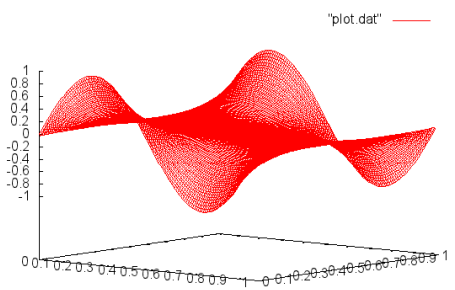
Solves the 2D-Laplace equation on a rectangle

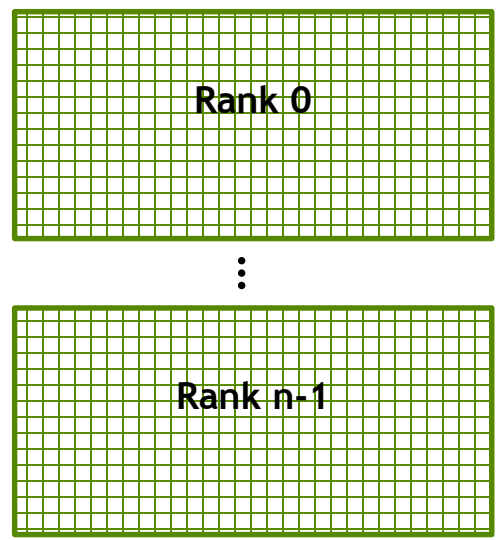
$$\Delta \boldsymbol{u}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{0} \; \forall \; (\boldsymbol{x}, \boldsymbol{y}) \in \Omega \backslash \boldsymbol{\delta}\Omega$$

Dirichlet boundary conditions (constant values on boundaries) on left and right boundary

Periodic boundary conditions Top Bottom (different from previous lectures)

1D domain decomposition with n domains





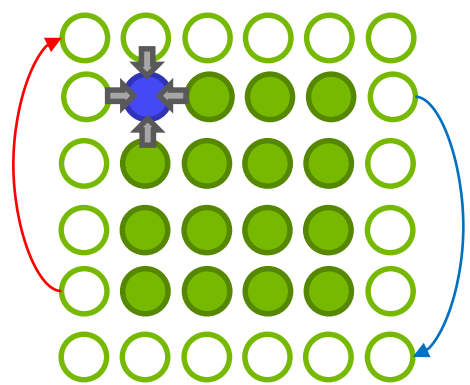
Example: Jacobi Solver – Single GPU

While not converged

- Do Jacobi step:

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

- Copy `Anew` to `A`
- Apply periodic boundary conditions (new compared to previous lectures)
- Next iteration



Handling GPU Affinity

Rely on process placement (with one rank per GPU)*

```
int rank = 0;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
int ngpus = acc_get_num_devices(acc_device_nvidia); // ngpus == ranks per node
int devicenum = rank % ngpus;
acc_set_device_num(devicenum,acc_device_nvidia);
```

*This assumes the node is homogeneous, i.e. that all the GPUs are the same. If you have different GPUs in the same node then you may need some more complex GPU selection

Domain Decomposition

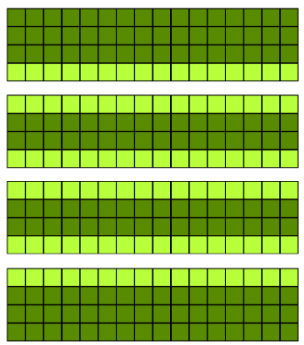
Different Ways to split the work between processes:

Minimizes number of neighbors:

- Communicate to less neighbors
- Optimal for latency bound communication

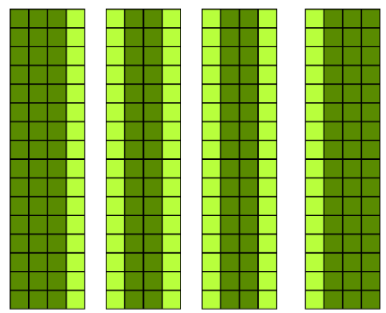
Minimizes surface area/volume ratio:

- Communicate less data
- Optimal for bandwidth bound communication



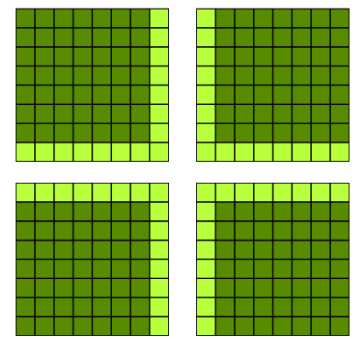
Horizontal Stripes

Contiguous if data is row-major



Vertical Stripes

Contiguous if data is column-major



Tiles

Domain Decomposition

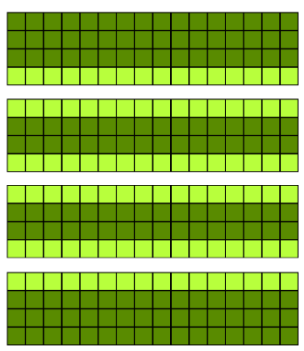
Different Ways to split the work between processes:

Minimizes number of neighbors:

- Communicate to less neighbors
- Optimal for latency bound communication

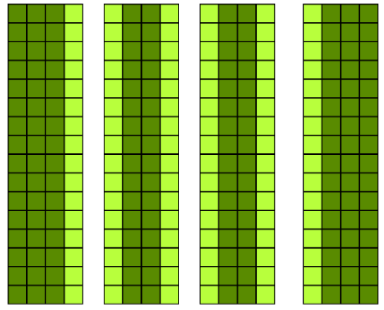
Minimizes surface area/volume ratio:

- Communicate less data
- Optimal for bandwidth bound communication



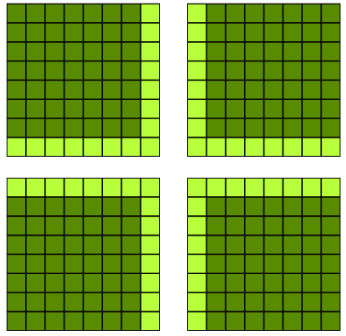
Horizontal Stripes

Contiguous if data is row-major



Vertical Stripes

Contiguous if data is column-major



Tiles

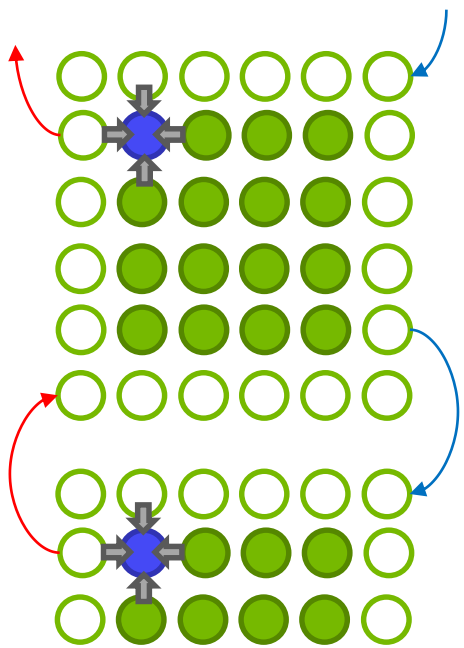
Example: Jacobi Solver – Multi GPU

While not converged

- Do Jacobi step:

```
for (int j=jstart; j < jend; j++)
   for (int i=1; i < m-1; i++)
     Anew[j][i] = 0.0f - 0.25f*(A[j-1][i] + A[j+1][i]
                                +A[j][i-1] + A[j][i+1])
```

- Copy `Anew` to `A`
- Apply periodic boundary conditions
- Exchange halo with 1 to 2 neighbors
- Next iteration



Example: Jacobi Solver – Multi GPU

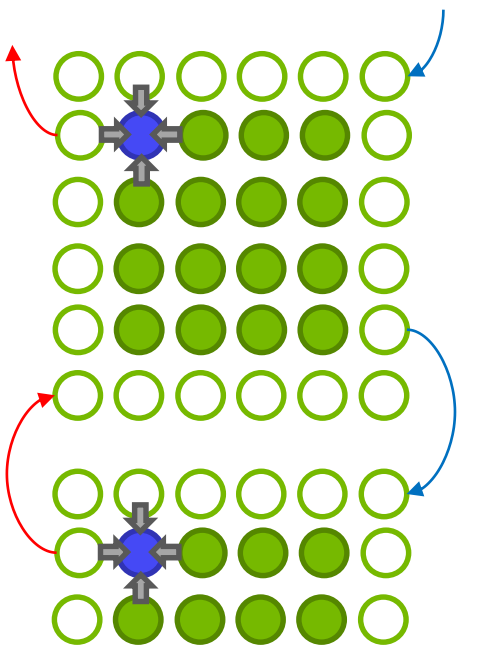
While not converged

- Do Jacobi step:

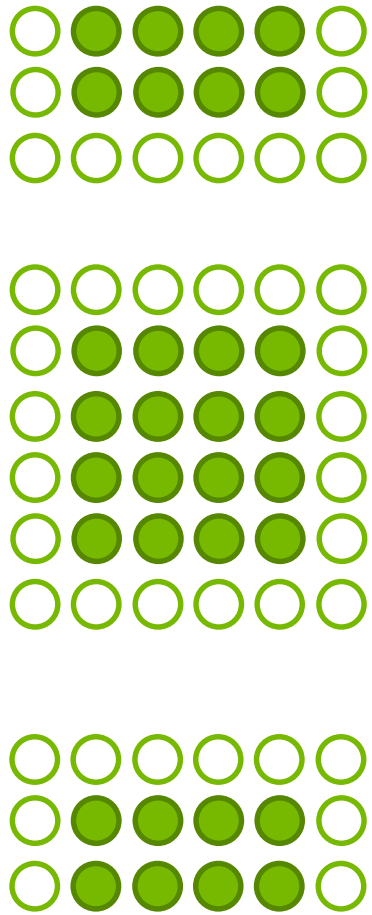
```
for (int j=jstart; j < jend; j++)
   for (int i=1; i < m-1; i++)
     Anew[j][i] = 0.0f - 0.25f*(A[j-1][i] + A[j+1][i]
                                +A[j][i-1] + A[j][i+1])
```

- Copy `Anew` to `A`
- Apply periodic boundary conditions
- Exchange halo with 1 to 2 neighbors
- Next iteration

One-step with ring exchange

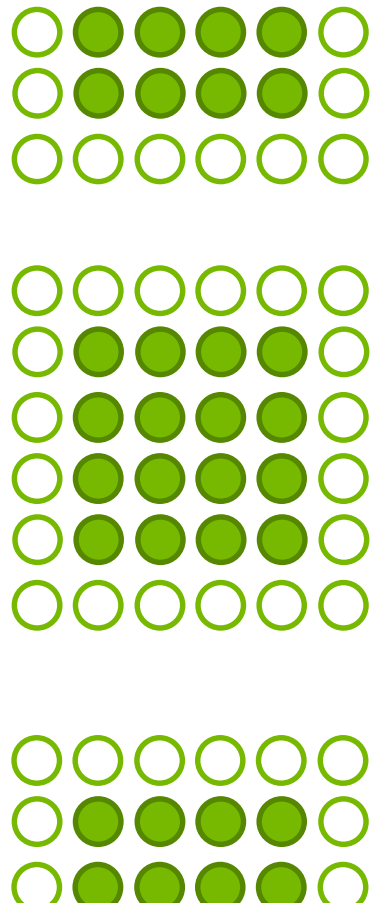


Example: Jacobi – Top/Bottom Halo



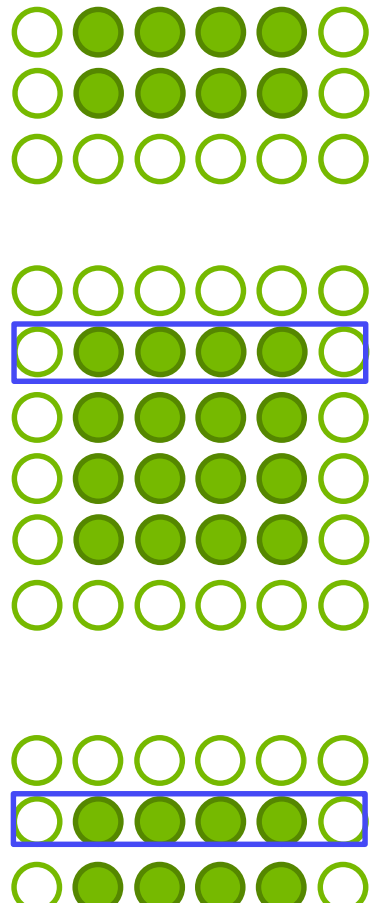
Example: Jacobi – Top/Bottom Halo

```
MPI_Sendrecv(A[jstart], M, MPI_FLOAT, top, 0,
             A[jend], M, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



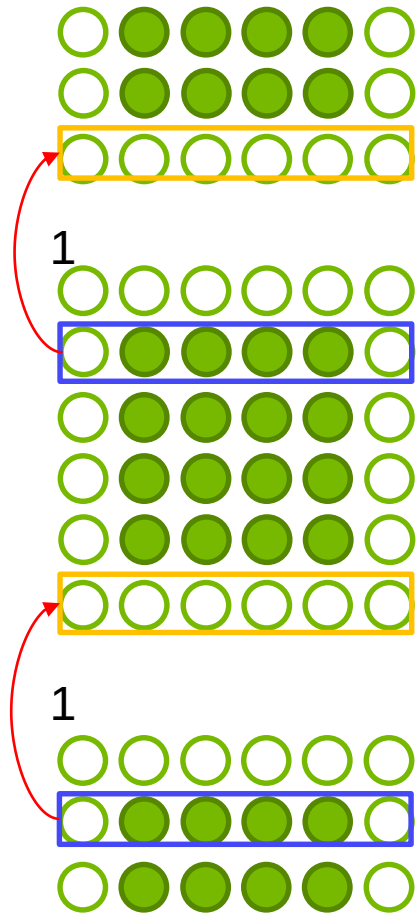
Example: Jacobi – Top/Bottom Halo

```
MPI_Sendrecv(A[jstart], M, MPI_FLOAT, top, 0,
             A[jend], M, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Example: Jacobi – Top/Bottom Halo

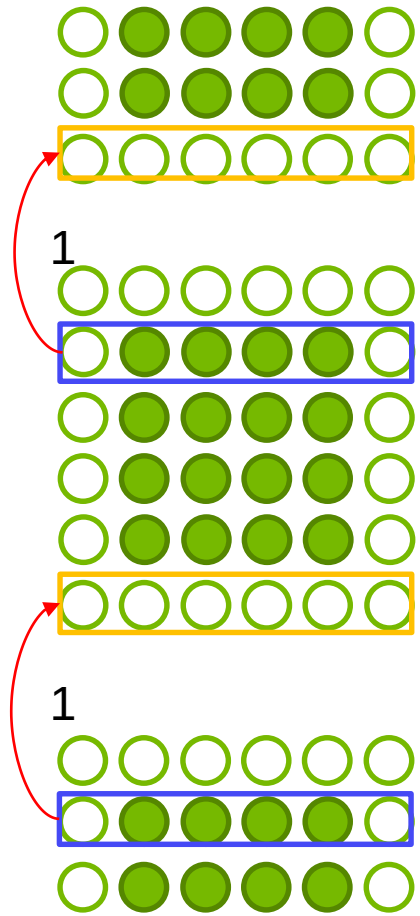
```
MPI_Sendrecv(A[jstart], M, MPI_FLOAT, top, 0,
             A[jend], M, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Example: Jacobi – Top/Bottom Halo

```
MPI_Sendrecv(A[jstart], M, MPI_FLOAT, top, 0,
             A[jend], M, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

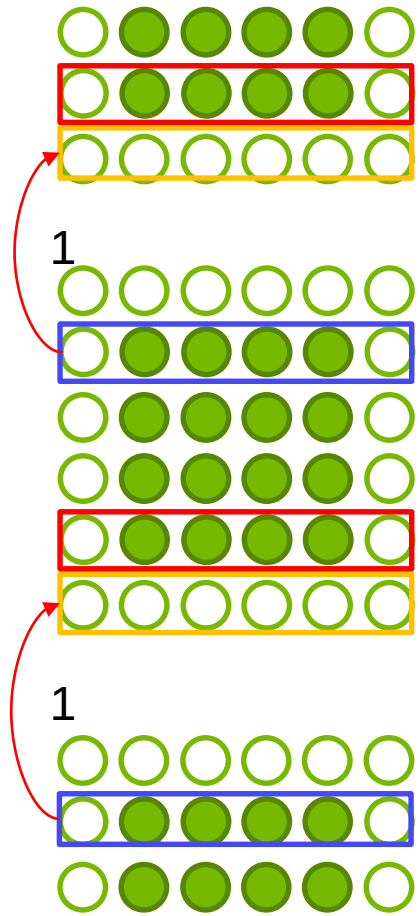
```
MPI_Sendrecv(A[(jend-1)], M, MPI_FLOAT, bottom, 0,
             A[(jstart-1)], M, MPI_FLOAT, top, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Example: Jacobi – Top/Bottom Halo

```
MPI_Sendrecv(A[jstart], M, MPI_FLOAT, top, 0,
             A[jend], M, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

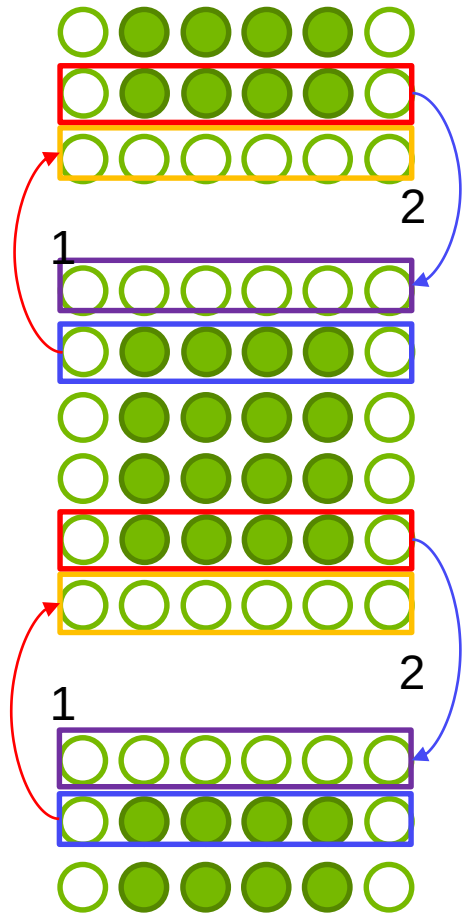
```
MPI_Sendrecv(A[(jend-1)], M, MPI_FLOAT, bottom, 0,
             A[(jstart-1)], M, MPI_FLOAT, top, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Example: Jacobi – Top/Bottom Halo

```
MPI_Sendrecv(A[jstart], M, MPI_FLOAT, top, 0,
             A[jend], M, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(A[(jend-1)], M, MPI_FLOAT, bottom, 0,
             A[(jstart-1)], M, MPI_FLOAT, top, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



OpenACC Interoperability

OpenACC Interoperability

OpenACC plays well with others.

Add CUDA or accelerated libraries to an OpenACC application

Add OpenACC to an existing accelerated application

Share data between OpenACC and CUDA

OpenACC host_data Directive

Exposes the *device* address of particular objects to the *host* code.

```
#pragma acc data copy(x,y)
{
// x and y are host pointers
#pragma acc host_data use_device(x,y)
{
 // x and y are device pointers
}
// x and y are host pointers
}
```

X and Y are device pointers here

host_data Example

OpenACC Main

```
program main
  integer, parameter :: N = 2**20
  real, dimension(N) :: X, Y
  real               :: A = 2.0

  !$acc data
  ! Initialize X and Y
  ...

  !$acc host_data use_device(x,y)
  call saxpy(n, a, x, y)
  !$acc end host_data
  !$acc end data

end program
```

CUDA C Kernel & Wrapper

```
__global__
void saxpy_kernel(int n, float a,
      float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

void saxpy(int n, float a, float *dx, float *dy)
{
  // Launch CUDA Kernel
  saxpy_kernel<<<4096,256>>>(N, 2.0, dx, dy);
}
```

- It's possible to interoperate from C/C++ or Fortran.
- OpenACC manages the data and passes device pointers to CUDA.

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.

CUBLAS Library & OpenACC

OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

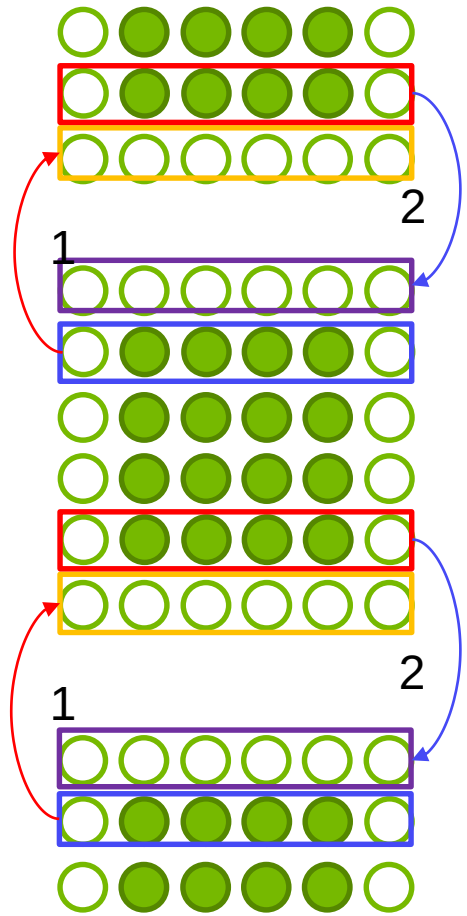
- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- Thrust
- …

OpenACC Main Calling CUBLAS

```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...

cublasInit();

#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {

    cublasSaxpy(N, 2.0, x, 1, y, 1);
  }
}

cublasShutdown();
```

Example: Jacobi – Top/Bottom Halo

```
#pragma acc host_data use_device ( A ) {
MPI_Sendrecv(A[jstart], M, MPI_FLOAT, top, 0,
             A[jend], M, MPI_FLOAT, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(A[(jend-1)], M, MPI_FLOAT, bottom, 0,
             A[(jstart-1)], M, MPI_FLOAT, top, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```



Scalability Metrics For Success

Serial Time T_s: How long it takes to run the problem with a single process

Parallel Time T_p: How long it takes to run the problem with multiple processes

Number of Processes P: The number of Processes operating on the task at hand

Speedup $S = \frac{T_s}{T_p}$: How much faster is the parallel version vs. serial. (optimal is P)

Efficiency $E = \frac{S}{P}$: How efficient are the processors used (optimal is 1)

Step 2: Results

jkraus@ivb114:~/workspace/qwiklabs/Multi-GPU-MPI/task2

```
[jkraus@ivb114 task2]$ make
mpicc -acc -ta=nvidia laplace2d.c -o laplace2d
mpirun -np 2 ./laplace2d
Jacobi relaxation Calculation: 2048 x 2048 mesh
Calculate reference solution and time serial execution.
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 2
2048x2048: 1 GPU:   0.8569 s, 2 GPUs:   0.5017 s, speedup:     1.71, efficiency:    85.39%
[jkraus@ivb114 task2]$
```

Profiling MPI+OPENACC applications

Using nvprof+NVVP:

Embed MPI Rank in output filename to be read by NVVP

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

Using nvprof only:

Only save the textual output

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

Profiling MPI+OPENACC applications

jkraus@ivb114:~/workspace/qwiklabs/Multi-GPU-MPI/task2

```
[jkraus@ivb114 task2]$ make profile
mpirun -np 2 nvprof -o laplace2d.%q{OMPI_COMM_WORLD_RANK}.nvvp ./laplace2d
==8873== NVPROF is profiling process 8873, command: ./laplace2d
==8872== NVPROF is profiling process 8872, command: ./laplace2d
Jacobi relaxation Calculation: 2048 x 2048 mesh
Calculate reference solution and time serial execution.
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 2
2048x2048: 1 GPU:   0.8997 s, 2 GPUs:   0.8864 s, speedup:     1.01, efficiency:    50.75%
```

Profiling MPI+OPENACC applications

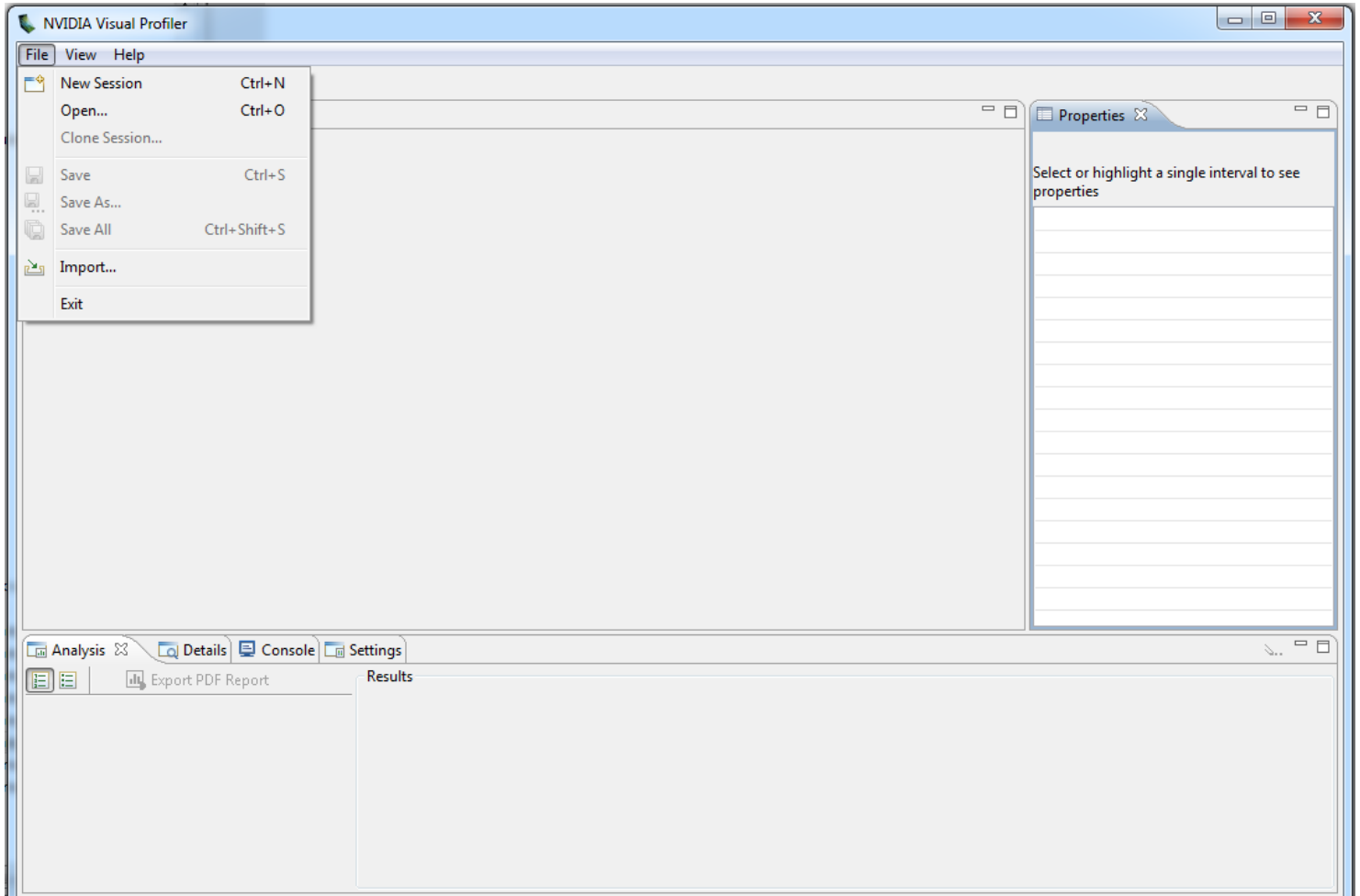
jkraus@ivb114:~/workspace/qwiklabs/Multi-GPU-MPI/task2

```
[jkraus@ivb114 task2]$ make profile
mpirun -np 2 nvprof -o laplace2d.%q{OMPI_COMM_WORLD_RANK}.nvvp ./laplace2d
==8873== NVPROF is profiling process 8873, command: ./laplace2d
==8872== NVPROF is profiling process 8872, command: ./laplace2d
Jacobi relaxation Cal
Calculate reference s
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 2
2048x2048: 1 GPU:   0
```

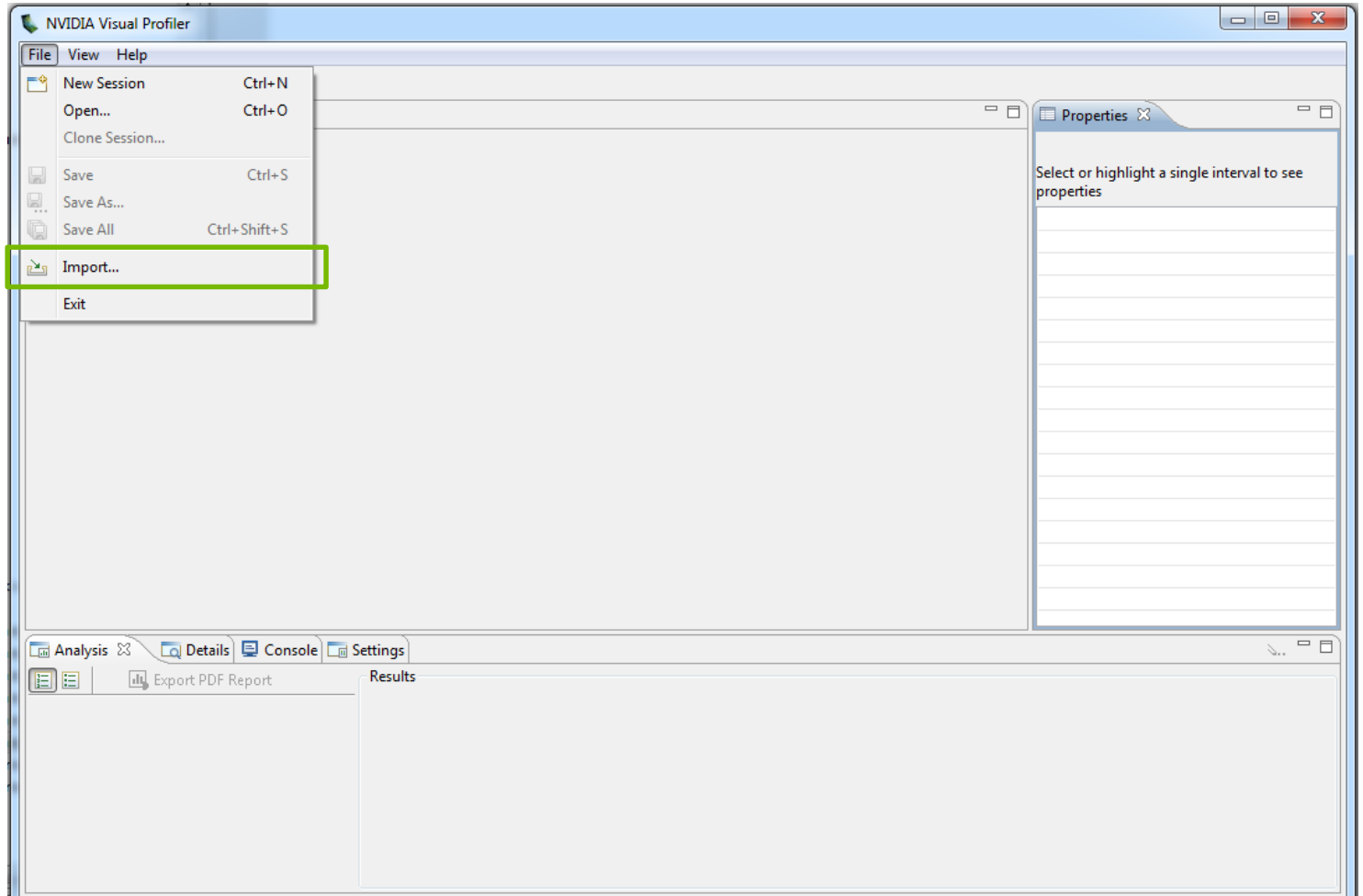
jkraus@ivb114:~/workspace/qwiklabs/Multi-GPU-MPI/task2

```
Calculate reference solution and time serial execution.
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002396
  200, 0.001204
  300, 0.000803
  400, 0.000603
  500, 0.000482
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 2
2048x2048: 1 GPU:   0.8997 s, 2 GPUs:   0.8864 s, speedup:     1.01, efficiency:    50.75%
==8873== Generated result file: /home-2/jkraus/workspace/qwiklabs/Multi-GPU-MPI/task2/laplace2d.
0.nvvp
==8872== Generated result file: /home-2/jkraus/workspace/qwiklabs/Multi-GPU-MPI/task2/laplace2d.
1.nvvp
[jkraus@ivb114 task2]$
```

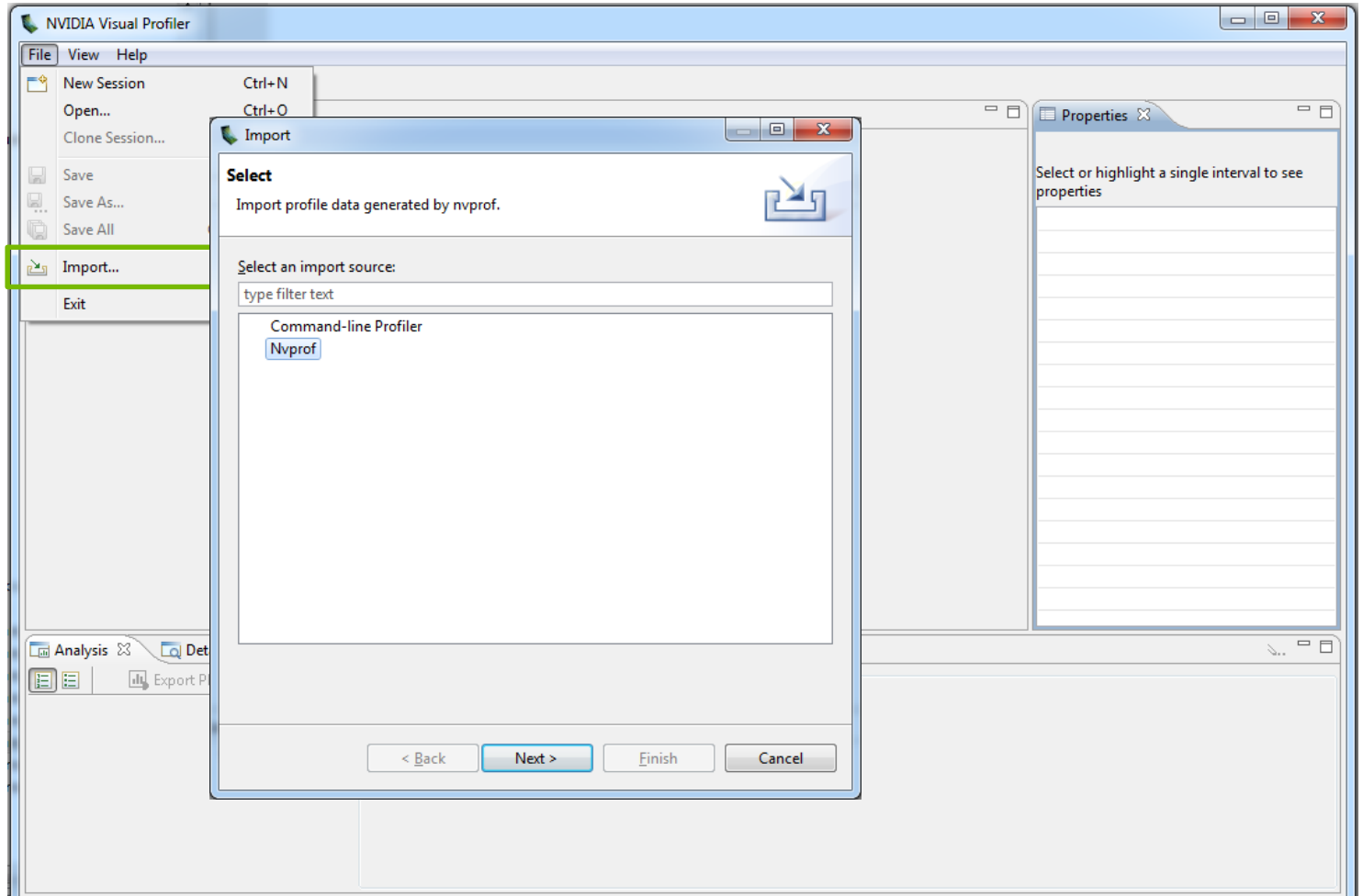
Profiling MPI+OPENACC applications



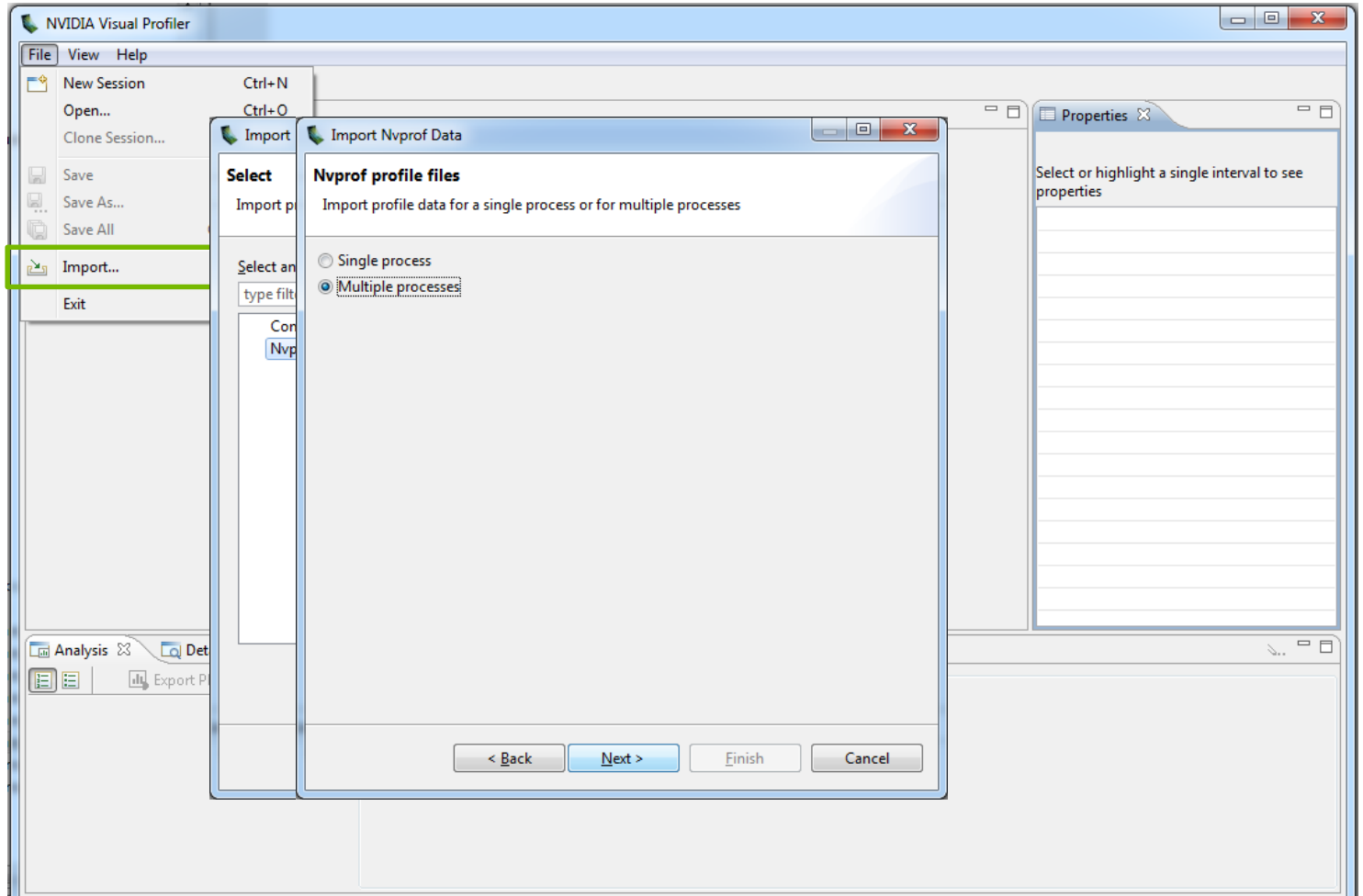
Profiling MPI+OPENACC applications



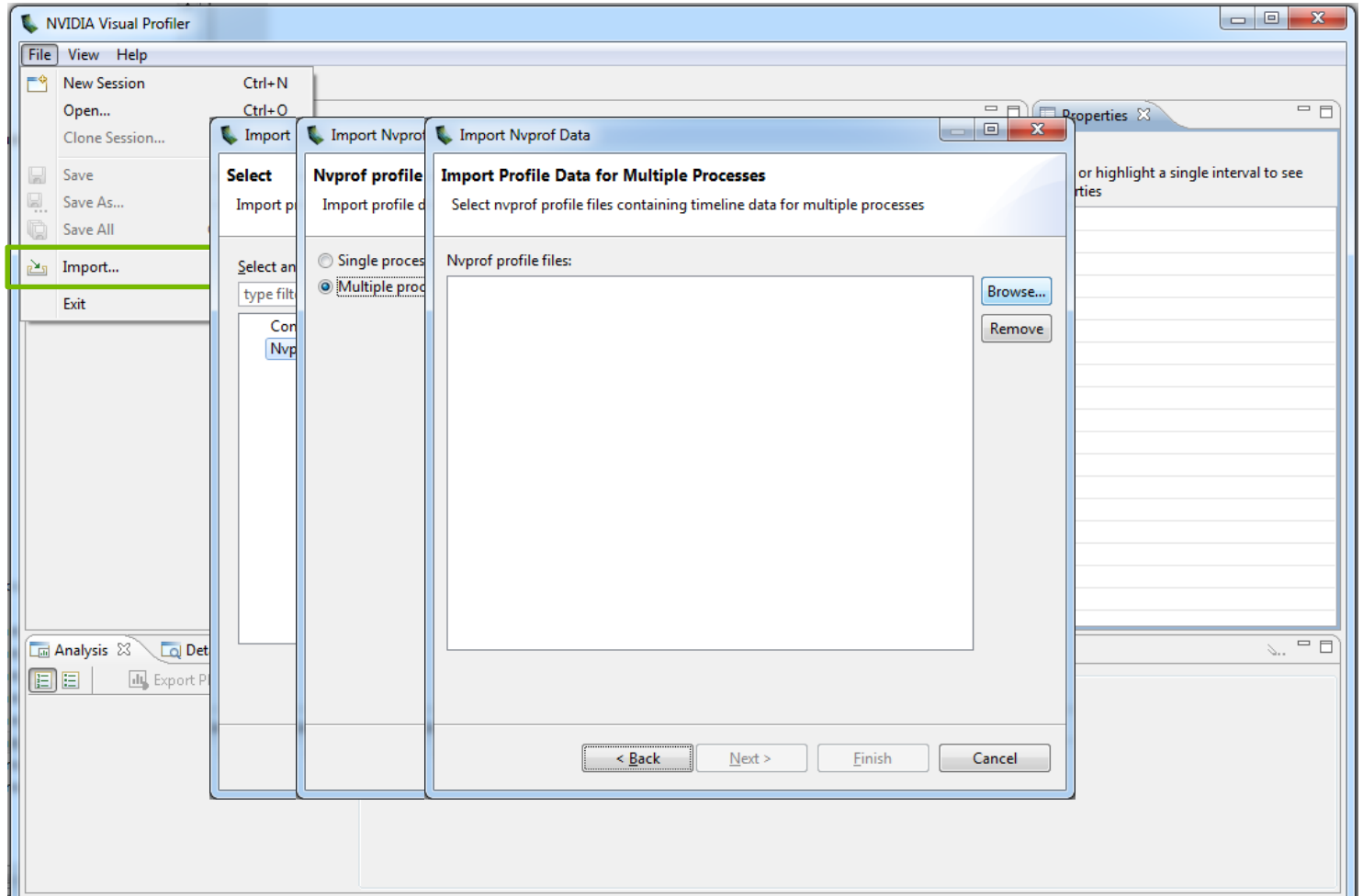
Profiling MPI+OPENACC applications



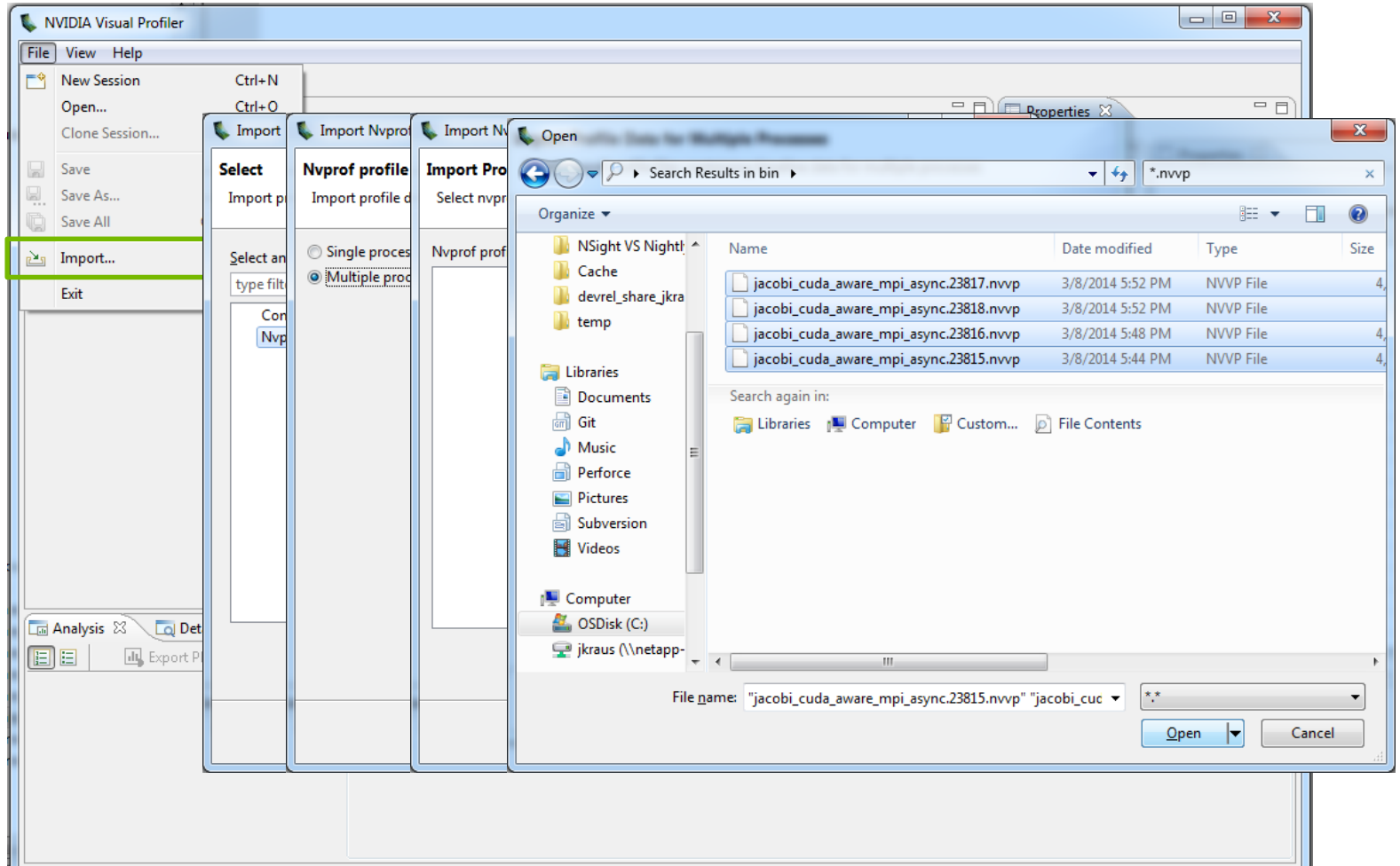
Profiling MPI+OPENACC applications



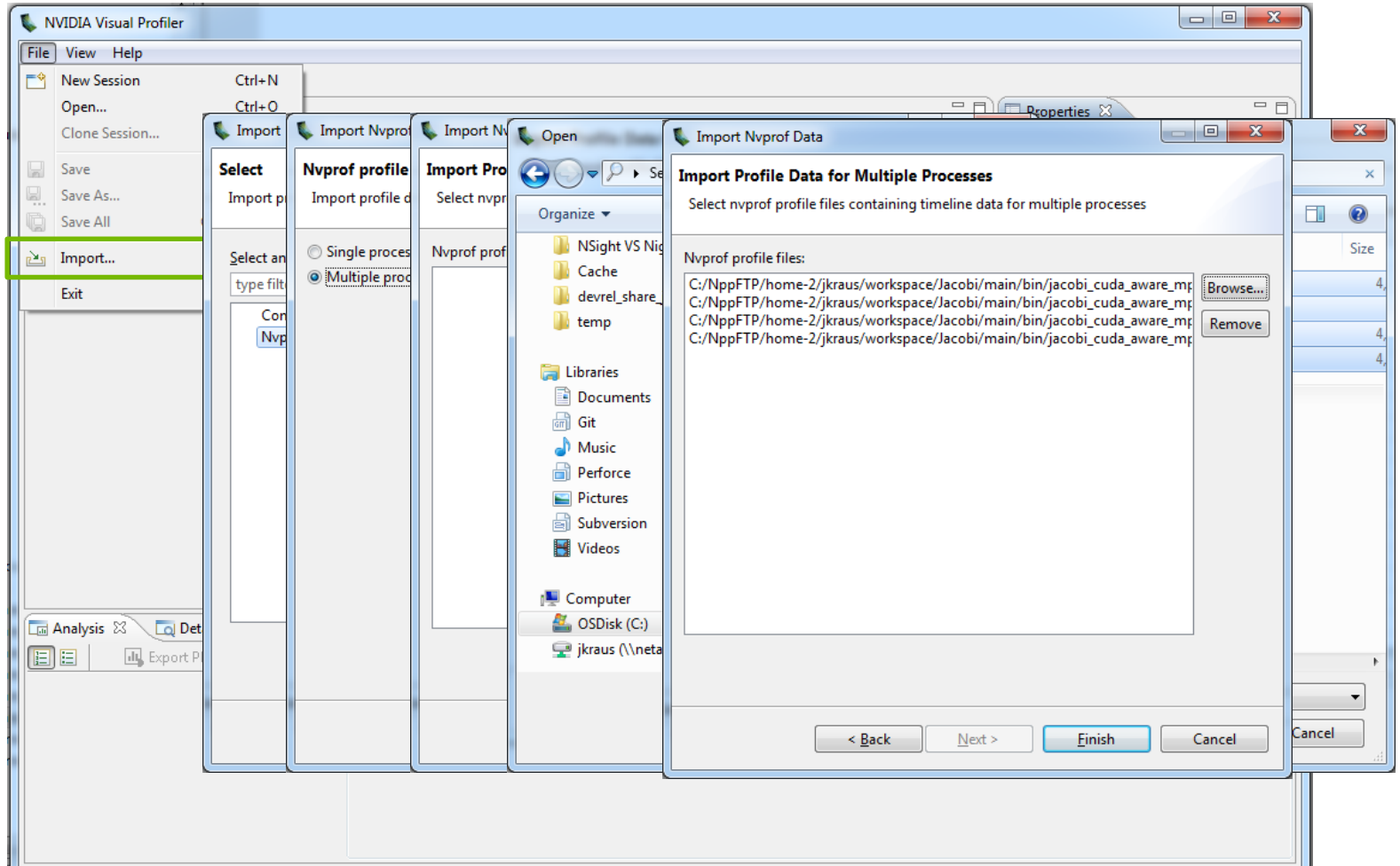
Profiling MPI+OPENACC applications



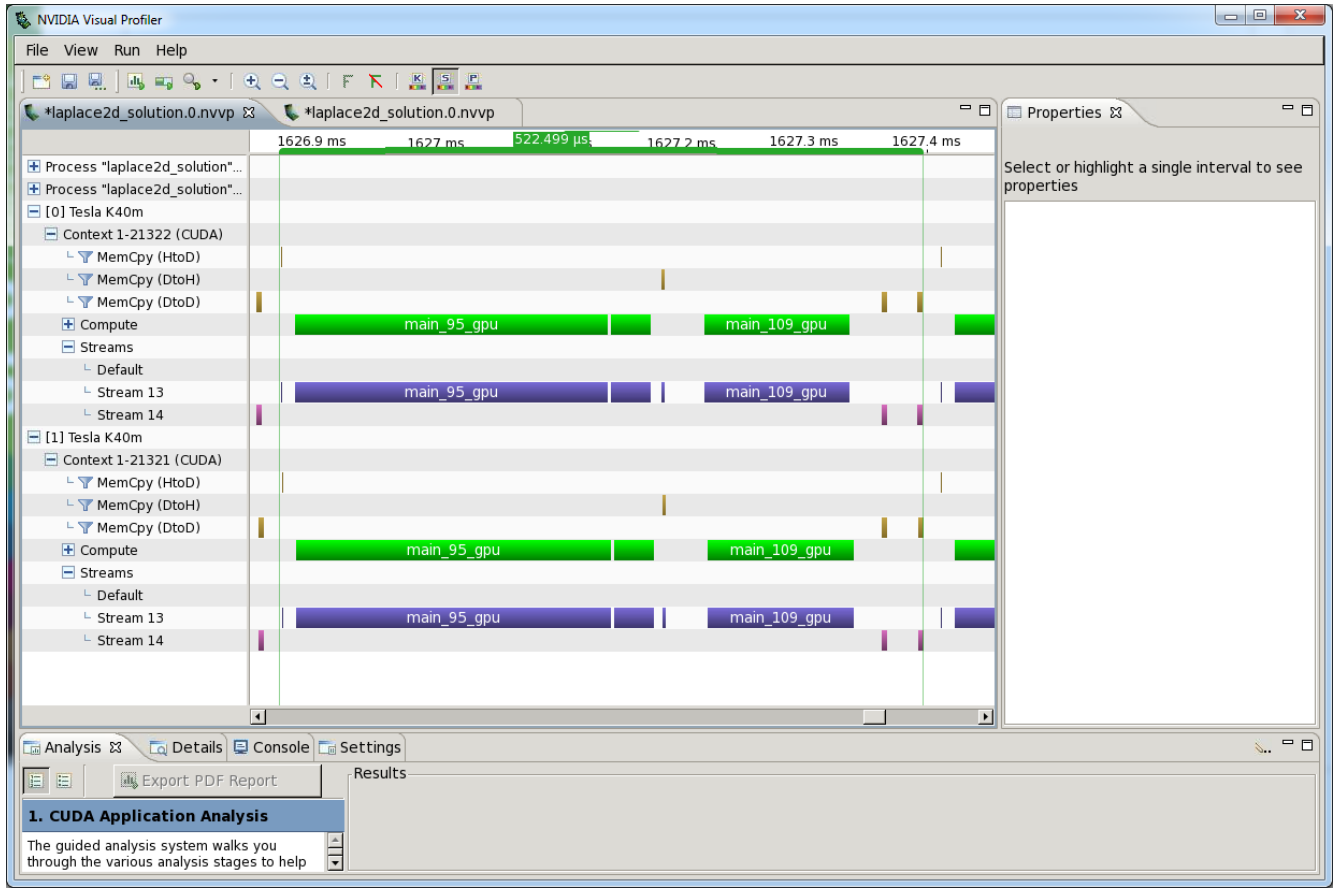
Profiling MPI+OPENACC applications



Profiling MPI+OPENACC applications

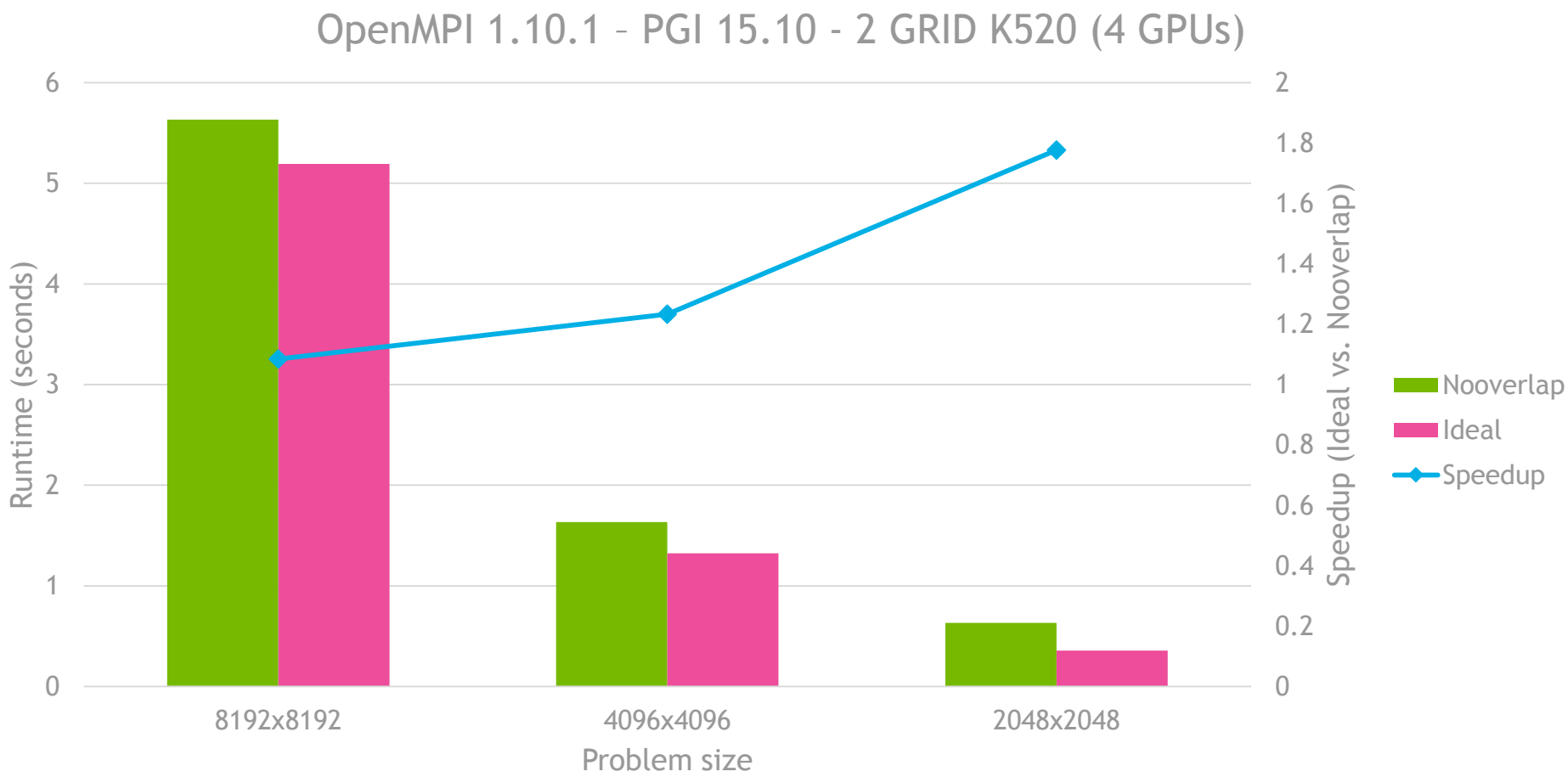


Profiling MPI+OPENACC applications

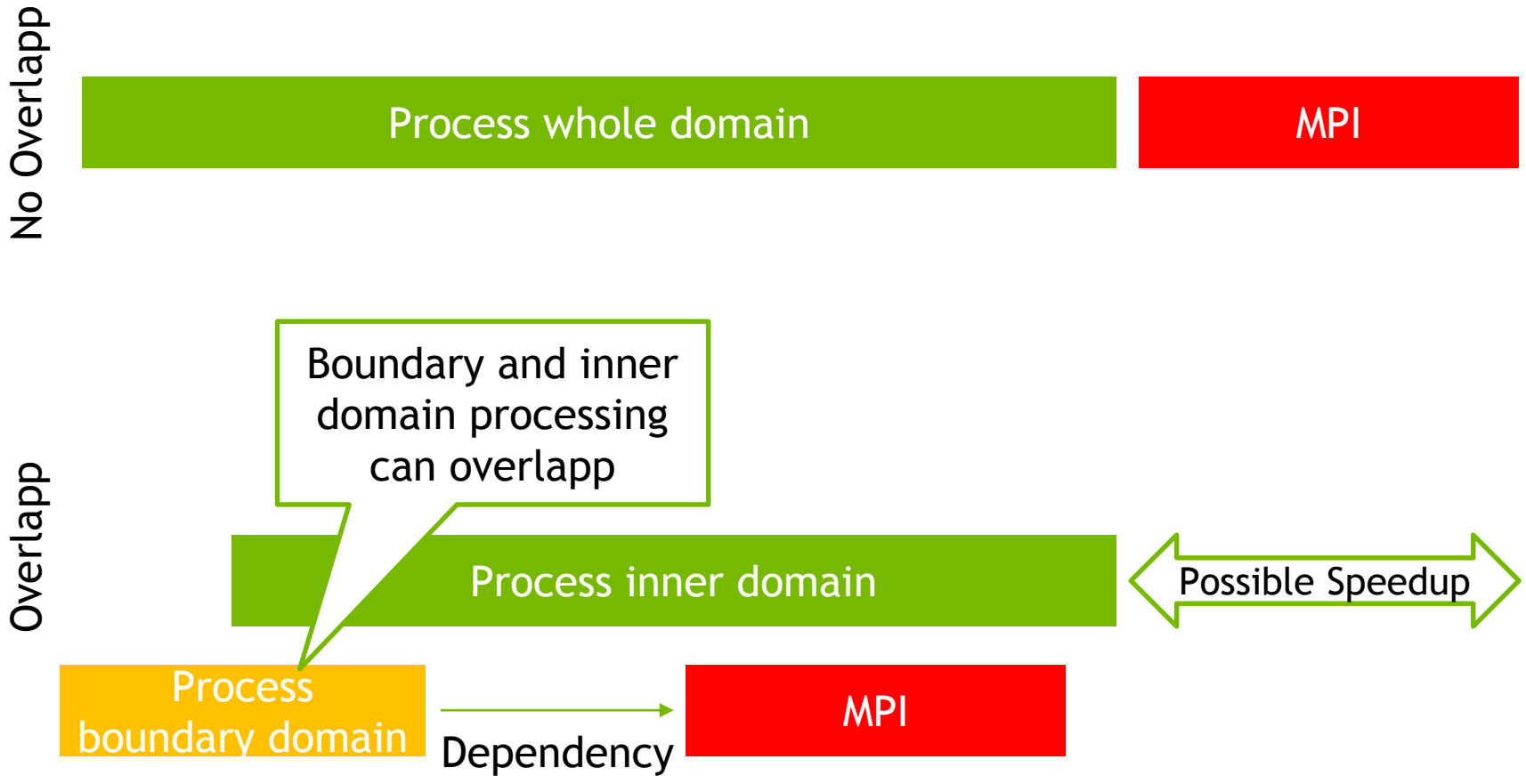


Communication + Computation Overlap

Communication + Computation Overlap



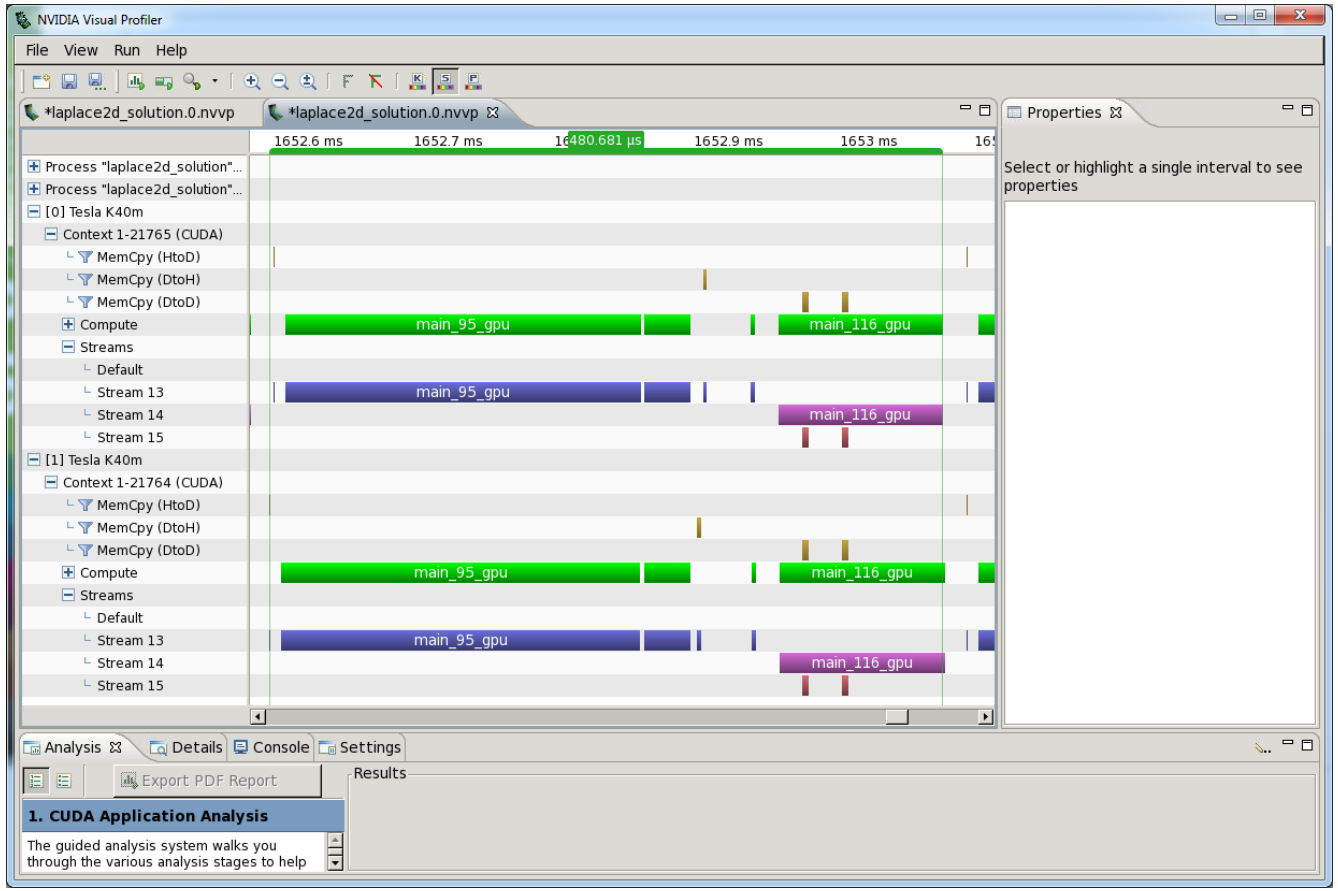
Communication + Computation Overlap



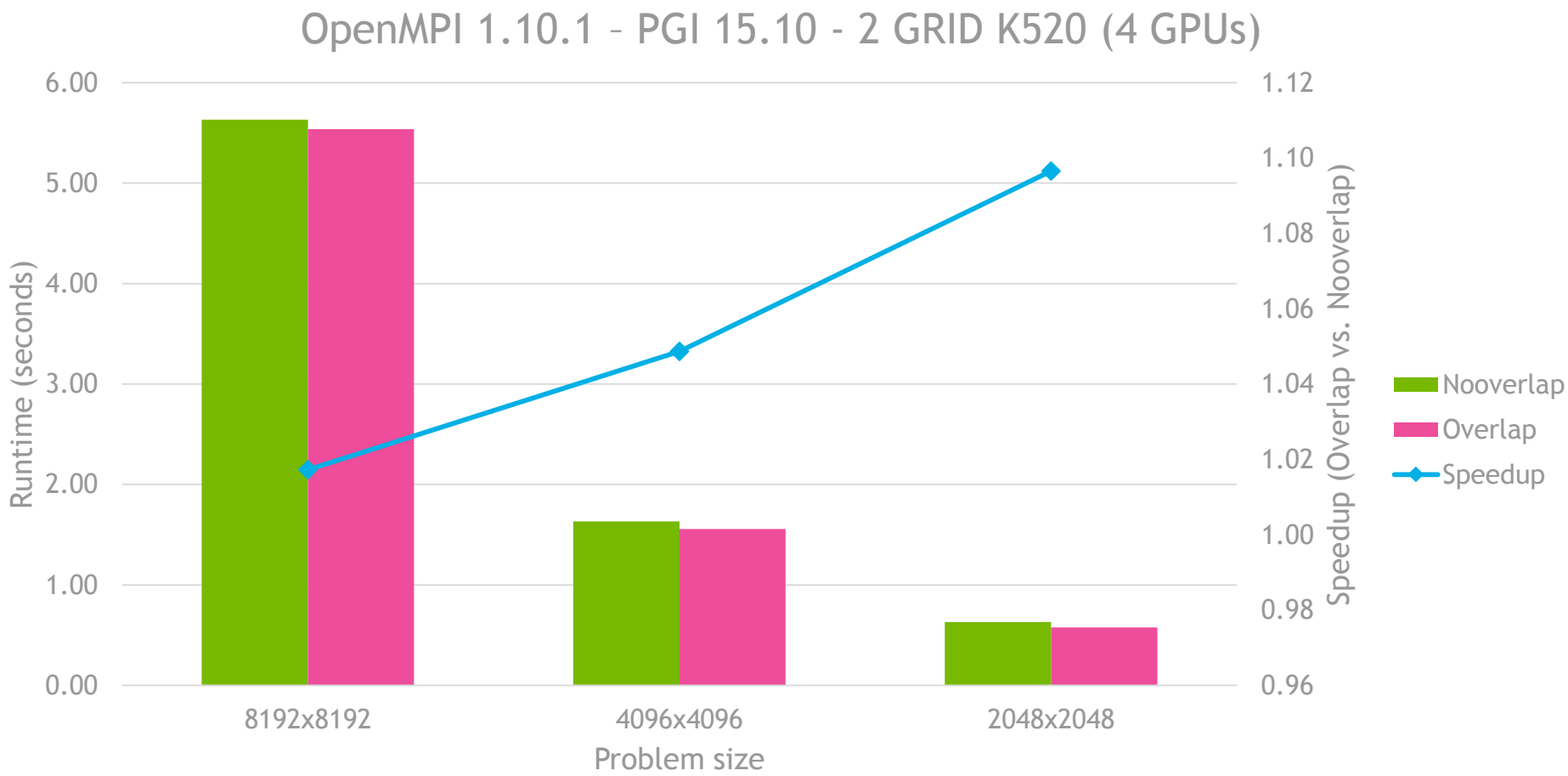
Communication + Computation Overlap

```
#pragma acc kernels
for ( ... )
      //Process boundary
#pragma acc kernels async
for ( ... )
      //Process inner domain

#pragma acc host_data use_device ( A )
{
  //Exchange halo with top and bottom neighbor
  MPI_Sendrecv( A…);
  //…
}
//wait for iteration to finish
#pragma acc wait
```

Profiling MPI+OPENACC Applications



Communication + Computation Overlap



Multi GPU Jacobi Solver

Homework

The Homework for this case study is available in the “Introduction to Multi GPU Programming with MPI and OpenACC“ lab at https://nvidia.qwiklab.com/ and consists of 3 tasks

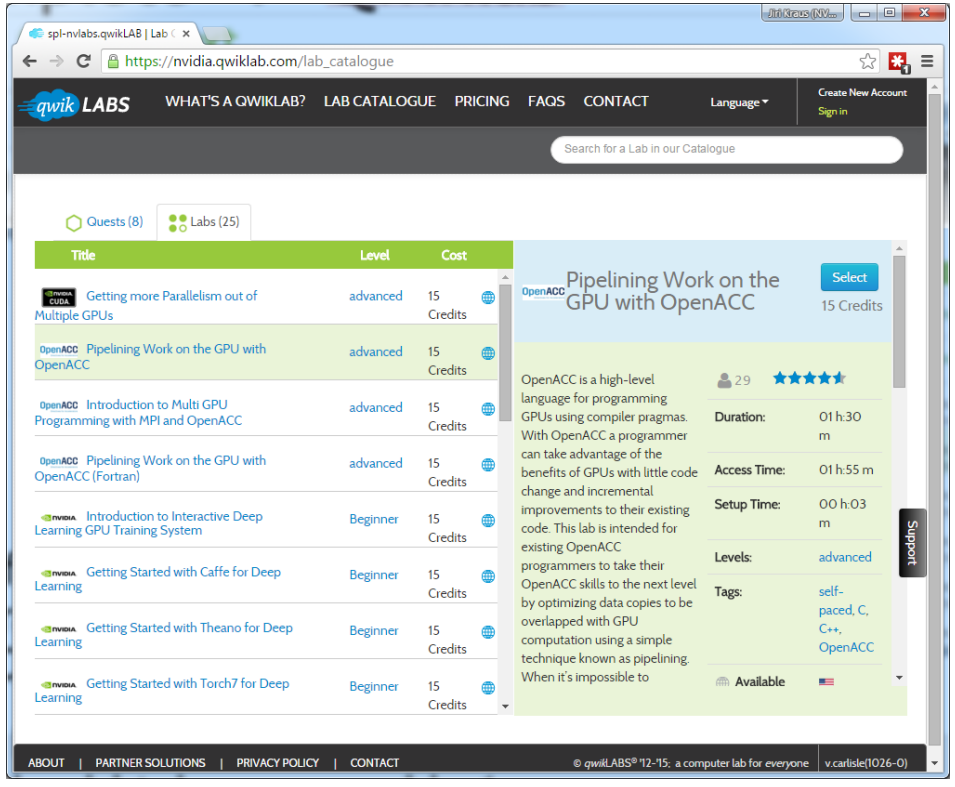
1. Add MPI boiler plate code: Use MPI compiler wrapper, Intialize MPI, …
2. Distribute work across GPUs
3. Overlap communication and computation to improve multi GPU scalability.

Homework

Complete Pipelining and MPI Qwiklab

From the NVIDIA Qwiklab website, select the Home Work

- Pipelining Work on the GPU with OpenACC (~1.5 hours)
  - bit.ly/nvoacclab4
- Introduction to Multi GPU Programming with MPI and OpenACC (~1.5 hours)
  - bit.ly/nvoacclab4b



Office Hours In Two Week

Last session in two weeks will be an office hours session.

Bring your questions from this week's lecture and homework to that session.

If you can't wait until then, post a question on StackOverflow tagged with openacc.

Course Syllabus

Oct 1: Introduction to OpenACC

Oct 6: Office Hours

Oct 15: Profiling and Parallelizing with the OpenACC Toolkit

Oct 20: Office Hours

Oct 29: Expressing Data Locality and Optimizations with OpenACC

Nov 3: Office Hours

Nov 12: Advanced OpenACC Techniques

Nov 24: Office Hours

Recordings:
https://developer.nvidia.com/openacc-course