Introduction to Multi GPU Programming with MPI and OpenACC

In this self-paced, hands-on lab, you will learn how to program multi GPU systems or GPU clusters using the Message Passing Interface (MPI) and OpenACC. Basic knowledge of MPI and OpenACC is a prerequisite. The topics covered by this lab are:

- Exchanging data between different GPUs using CUDA-aware MPI and OpenACC
- Handle GPU affinity in multi GPU systems
- Overlapping communication with computation to hide communication times
- Optionally how to use the NVIDIA performance analysis tools

Lab created by Jiri Kraus (based on the Lab **Accelerating C/C++ code with Multi-GPUs using CUDA** from Justin Luitjens and Mark Ebersole)

The following timer counts down to a five minute warning before the lab instance shuts down. You should get a pop up at the five minute warning reminding you to save your work!

Before we begin, let's verify WebSockets (http://en.wikipedia.org/wiki/WebSocket) are working on your system. To do this, execute the cell block below by giving it focus (clicking on it with your mouse), and hitting Ctrl-Enter, or pressing the play button in the toolbar above. If all goes well, you should see some output returned below the grey cell. If not, please consult the Self-paced Lab Troubleshooting FAQ (https://developer.nvidia.com/self-paced-labs-faq#Troubleshooting) to debug the issue.

In [1]:

```
print "The answer should be three: " + str(1+2)
```

```
The answer should be three: 3
```

Let's execute the cell below to display information about the GPUs running on the server.

In [2]:

```
!nvidia-smi
```

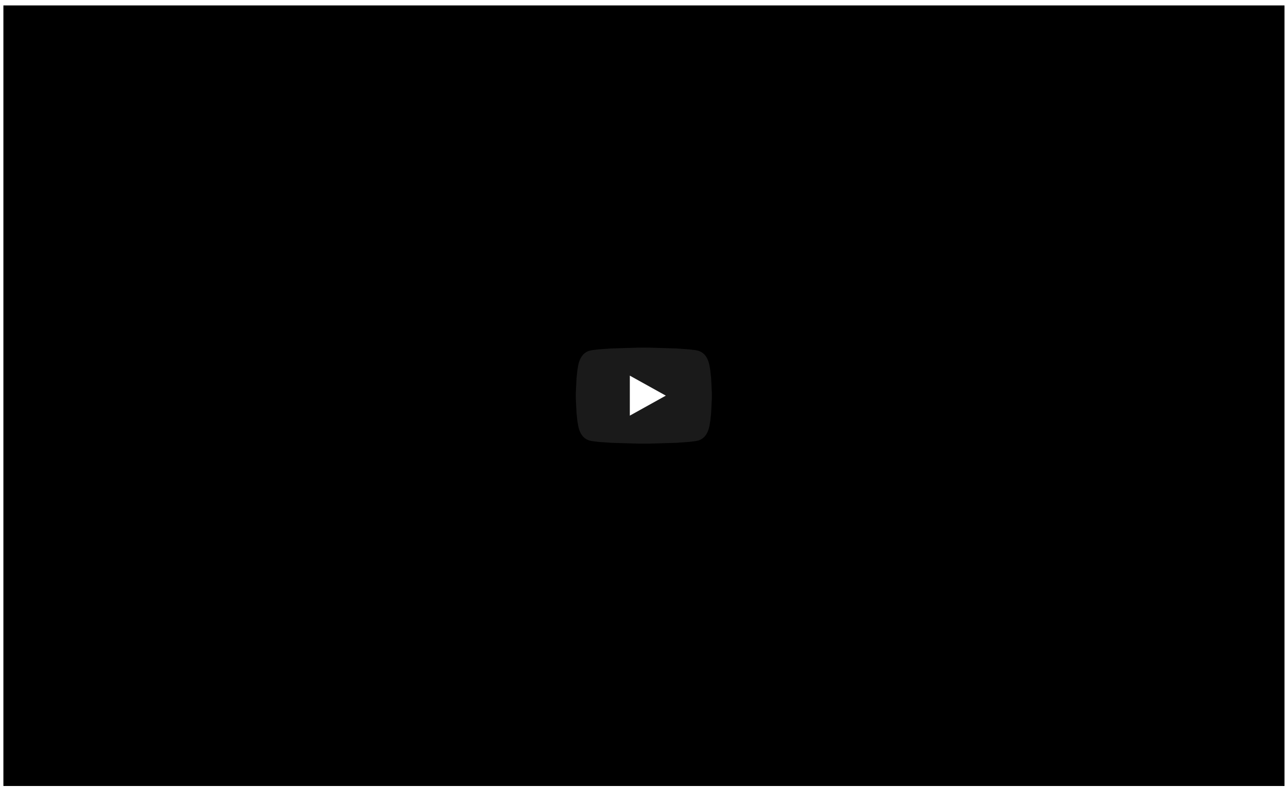
```
Tue Nov  7 20:23:57 2017
+------------------------------------------------------+

| NVIDIA-SMI 352.68     Driver Version: 352.68         |

|-------------------------------+----------------------+--------------------
--+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. EC
C |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute
 M. |
|===============================+======================+====================
==|
|   0  GRID K520           On   | 0000:00:03.0     Off |                  N/
A |
| N/A   21C    P8    17W / 125W |     11MiB /  4095MiB |      0%      Defaul
t |
+-------------------------------+----------------------+--------------------
--+
|   1  GRID K520           On   | 0000:00:04.0     Off |                  N/
A |
| N/A   20C    P8    17W / 125W |     11MiB /  4095MiB |      0%      Defaul
t |
+-------------------------------+----------------------+--------------------
--+
|   2  GRID K520           On   | 0000:00:05.0     Off |                  N/
A |
| N/A   22C    P8    17W / 125W |     11MiB /  4095MiB |      0%      Defaul
t |
+-------------------------------+----------------------+--------------------
--+
|   3  GRID K520           On   | 0000:00:06.0     Off |                  N/
A |
| N/A   19C    P8    17W / 125W |     11MiB /  4095MiB |      0%      Defaul
t |
+-------------------------------+----------------------+--------------------
--+


+-----------------------------------------------------------------------------
--+
| Processes:                                                       GPU Memor
y |
|  GPU       PID  Type  Process name                               Usage
  |
|=============================================================================
==|
|  No running processes found
  |
+-----------------------------------------------------------------------------
--+
```

The following video will explain the infrastructure we are using for this self-paced lab, as well as give some tips on it's usage. If you've never taken a lab on this system before, it's highly recommended that you watch this short video first.



Motivation

Why use Multiple GPUs?

After you have accelerated your application using a single GPU, it's natural to consider extending your app to take advantage of multiple GPUs in a single node or in multiple nodes of a GPU accelerated cluster.

Multiple GPUs can:

- Compute Faster - More GPUs equals faster time to a solution
- Compute Larger - More GPUs means more memory for larger problems
- Compute Cheaper - More GPUs per node translates to less overhead in money, power and space

Using CUDA-aware MPI with OpenACC allows you to efficiently utilize all the GPUs in a single node and also to scale across the GPUs in multiple nodes of a GPU accelerated cluster. If you are accelerating an (already) existing MPI parallel CPU code with OpenACC, going multi-GPU with CUDA-aware MPI and OpenACC is straight-forward.

CUDA-aware MPI

A CUDA-aware MPI implementation allows you to exchange data directly to and from the four gpu buffers involved, avoiding host buffer staging in the user code. For this lab it is sufficient to know that you can directly pass GPU pointers to the MPI routines of a CUDA-aware MPI implementation. If you want to learn more about CUDA-aware MPI I recommend you to read my post on the Parallel Forall blog: An Introduction to CUDA-Aware MPI (http://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/)

Structure of this Lab

This lab is broken up into three tasks; instructions for each will be provided in-line below. The "solution" to each task is the starting point of the next task, so you can skip around if you'd like. In addition, reference solutions are provided for each task. You can find them by looking for files matching *.solution*. You can build and run the solution with the make target `task?.solution`, e.g. for task 1: `make -C C task1.solution`.

Instructions for downloading this IPython Notebook, as well as a .zip file of the source you worked on, are provided at the bottom of the lab in the Post Lab section.

Scalability Metrics For Success

The success of the multi GPU parallelization is measured with the following metrics. The provided tasks automatically print these metrics out at the end of each lab section/execution.

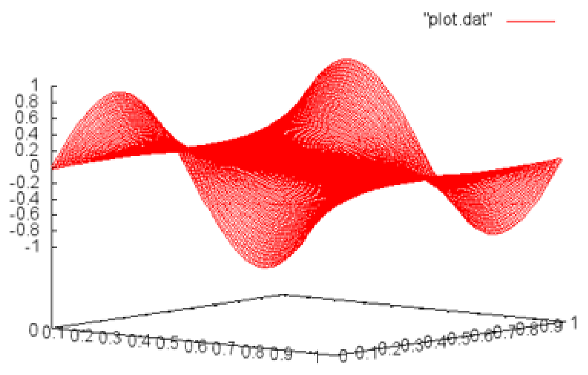
- Serial Time: T_s - how long it takes to run the problem with a single thread
- Parallel Time: T_p - how long it takes to run the problem in parallel
- Number of Processors: P - the number of processors operating in parallel
- Speedup: $S = \frac{T_s}{T_p}$ - How much faster the parallel version is versus the serial version.
  - The ideal speed up is P.
- Efficiency: $E = \frac{S}{P}$ - How efficiently the processors are being used.
  - The ideal efficiency is 1.

2D Laplace Solver

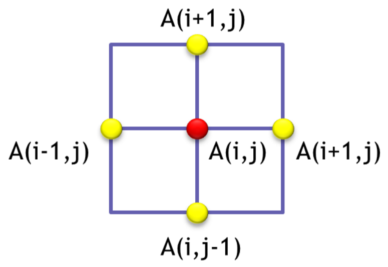
The code used in this lab is a Jacobi solver for the 2D Laplace equation on a rectangle:

$$\Delta A(x,y) = 0 \, \forall (x,y) \in \Omega \backslash \delta\Omega$$

It uses Dirichlet boundary conditions (constant values on boundaries) on the left and right boundary and periodic boundary conditions on the top and bottom boundary. With the values chosen for left and right boundaries by the provided source code the solution looks like this

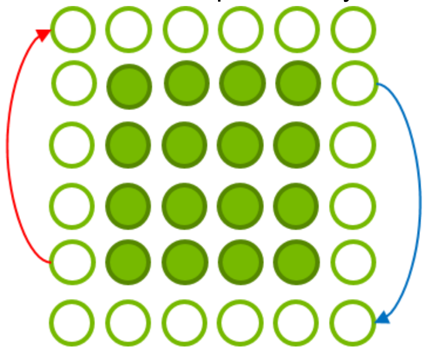


Given a 2D grid of vertexes, the solver attempts to set every vertex equal to the average of neighboring vertices. It will iterate until the system converges to a stable value. So in each iteration of the Jacobi solver for all interior vertices



$$A_{k+1}(i,j) = 0 - \frac{A_k(i-1,j)+A_k(i+1,j)+A_k(i,j-1)+A_k(i,j+1)}{4}$$

is applied and then the periodic boundary conditions are handled by copying the values of the first interior row to the bottom boundary and the last interior row to the top boundary:



Lab Tasks

This is a long lab, so please pay attention to your time. You have 120 minutes of access time from when the lab connection information was presented to you. You may want to pre-download the IPython Notebook and source in the Post Lab before continuing.

Task #1

The purpose of this task is to show how to handle GPU affinity, add MPI boiler plate code in `C/task1/laplace2d.c` and make the necessary changes to `C/task1/Makefile`. Look out for `TODO` in these two files. These will guide you through the following steps:

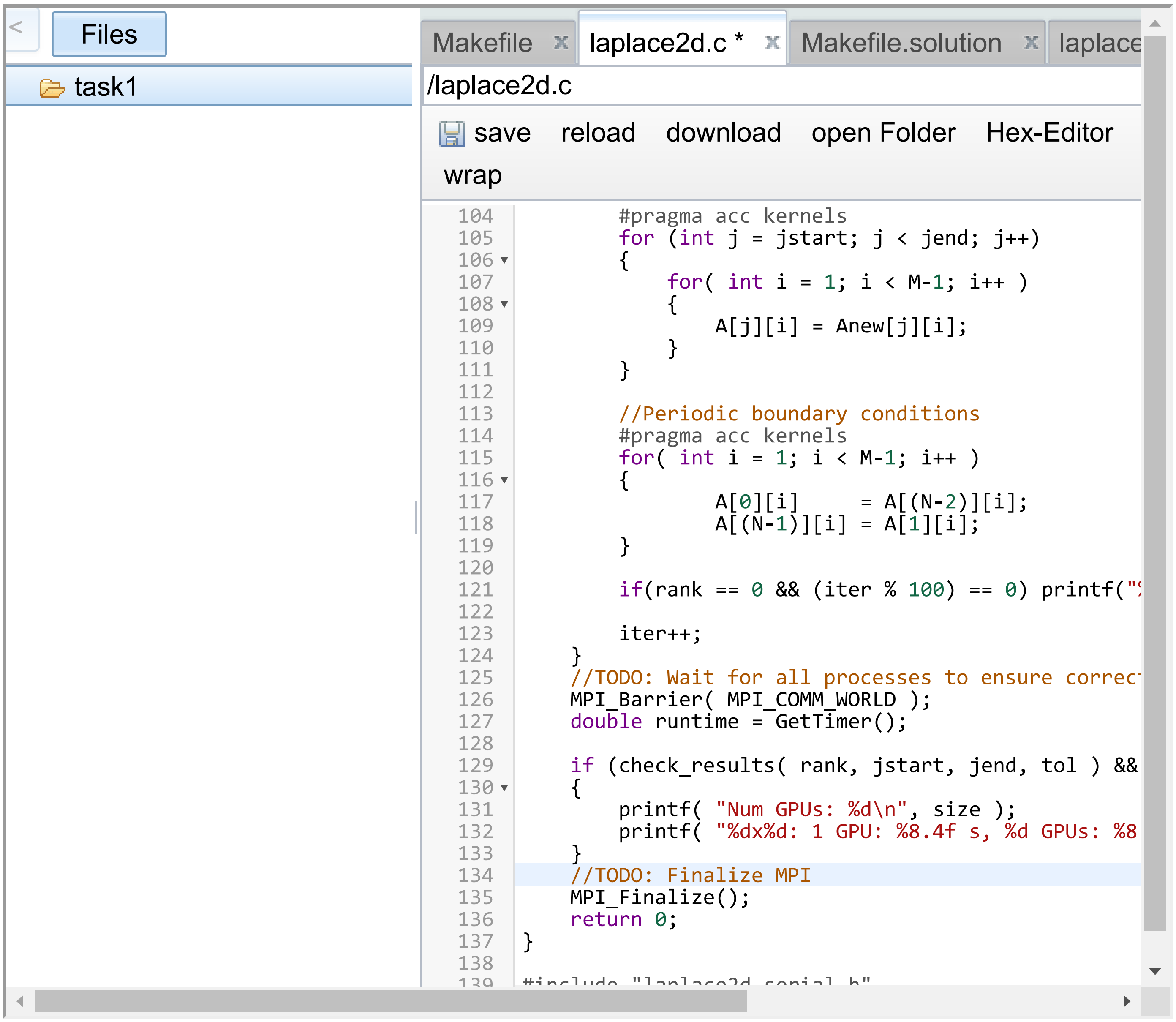
- `Makefile`: Use MPI compiler wrapper (`mpicc`)
- `Makefile`: Start with MPI launcher (`mpirun -np …`)
- `laplace2d.c`: Include MPI header (`mpi.h`)
- `laplace2d.c`: Initialize MPI (`MPI_Init, MPI_Comm_rank, MPI_Comm_size`)
- `laplace2d.c`: Handle GPU Affinity
- `laplace2d.c`: Insert barriers to ensure correct timing (`MPI_Barrier`)
- `laplace2d.c`: Finalize MPI (`MPI_Finalize`)

To compile and run simply issue `make -C C task1` as given in the cell below.

The following reference might be interesting for you:

- API documentation for MPI from the OpenMPI website https://www.open-mpi.org/doc/v1.8 (https://www.open-mpi.org/doc/v1.8).
- OpenACC 2.0 Quick Reference Guide (http://104.239.134.127/sites/default/files/213462%2010_OpenACC_API_QRG_HiRes.pdf)

Click here to see hints



```
        #pragma acc kernels
        for (int j = jstart; j < jend; j++)
        {
            for( int i = 1; i < M-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        //Periodic boundary conditions
        #pragma acc kernels
        for( int i = 1; i < M-1; i++ )
        {
                A[0][i]     = A[(N-2)][i];
                A[(N-1)][i] = A[1][i];
        }

        if(rank == 0 && (iter % 100) == 0) printf("
        
        iter++;
    }
    //TODO: Wait for all processes to ensure correc
    MPI_Barrier( MPI_COMM_WORLD );
    double runtime = GetTimer();

    if (check_results( rank, jstart, jend, tol ) &&
    {
        printf( "Num GPUs: %d\n", size );
        printf( "%dx%d: 1 GPU: %8.4f s, %d GPUs: %8
    }
    //TODO: Finalize MPI
    MPI_Finalize();
    return 0;
}

#include "laplace2d_serial.h"
```

```
In [13]:
```

```
!make -C C task1
```

```
make: Entering directory `/home/ubuntu/notebook/C'
make -C task1
make[1]: Entering directory `/home/ubuntu/notebook/C/task1'
#TODO: Use MPI compiler wrapper
mpicc -fast -acc -ta=nvidia laplace2d.c -o laplace2d
#TODO: Start with MPI run
mpirun -np 4 ./laplace2d
Jacobi relaxation Calculation: 4096 x 4096 mesh
Calculate reference solution and time serial execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 4
4096x4096: 1 GPU:   5.5852 s, 4 GPUs:   5.5574 s, speedup:     1.01, efficie
ncy:    25.13%
make[1]: Leaving directory `/home/ubuntu/notebook/C/task1'
make: Leaving directory `/home/ubuntu/notebook/C'
```

At the end of the output you will see a output similar to this:

```
Num GPUs: 4
4096x4096: 1 GPU:   5.2734 s, 4 GPUs:   5.1988 s, speedup:     1.01, efficiency:
  25.36%
```

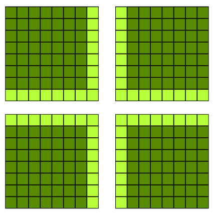
We are not getting any speed up although we are using four GPUs. The reason is that we simply quadrupled our work, that is, each GPU is solving the full problem. The alternative -- distributing the work across multiple GPUs by applying a domain decomposition -- is covered in task 2.

Task #2

The purpose of this task is to apply a domain decomposition using horizontal stripes. To let you focus on the logical domain decomposition and GPU to GPU communication the data here is still fully replicated on each GPU. (This is something one would normally not do, as it has a significant storage overhead, but it avoids some extensive boilerplate code that should not be part of this lab.)

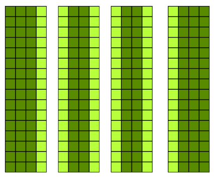
Domain Decomposition

Here are three options for breaking up our 2D grid of vertexes, or domain, to parallelize the work across the multiple GPUs. The halo region shown in light green in the images is the data that needs to be shared among the GPUs working on the problem.


Tiles

Minimizes surface area/volume ratio:

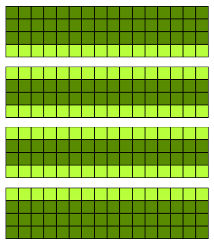
- Communicate less data
- Optimal for bandwidth bound communication


Vertical Stripes

Minimizes number of neighbors:

- Communicate to fewer neighbors
- Optimal for latency bound communication

Contiguous if data is column-major (https://en.wikipedia.org/wiki/Column-major_order)


Horizontal Stripes

Minimizes number of neighbors:

- Communicate to fewer neighbors
- Optimal for latency bound communication

Contiguous if data is row-major (https://en.wikipedia.org/wiki/Row-major_order)

In our case, we'll be using the Horizontal Stripes decomposition as row-major order is used in C/C++. To actually do the domain decomposition we divide the number of rows of our data array by the number of MPI ranks participating in a run and assign a chunk of that size to each rank. E.g. with 500 rows and 2 MPI ranks: rank 0 would process from 1 to 250 and rank 1 from 251 to 498. In case the number of rows is not divisible by the number of MPI ranks you need to take care of rounding errors.

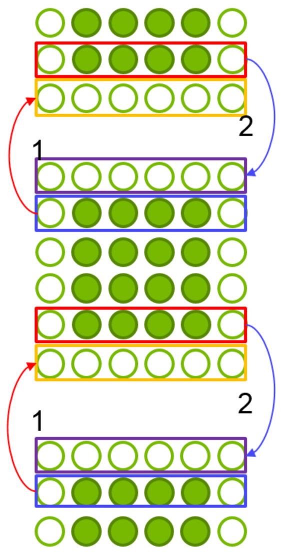
Halo updates

If the calculation is decomposed and distributed across the GPUs/MPI ranks each GPU/MPI rank works on its private copy of the data. To propagate information across the whole computation domain, we need to update the borders of each domain (so called halos) with the current values of the neighbouring GPUs/MPI ranks in each iteration. For example, rank `i` needs to sent the first row it has modified to the last row (the bottom domain boundary (halo)) of rank `i-1` (top neighbour), and the last row it has modified to the first row (the top domain boundary (halo)) of rank rank `i+1` (bottom neighbour):

```
#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);
}
```



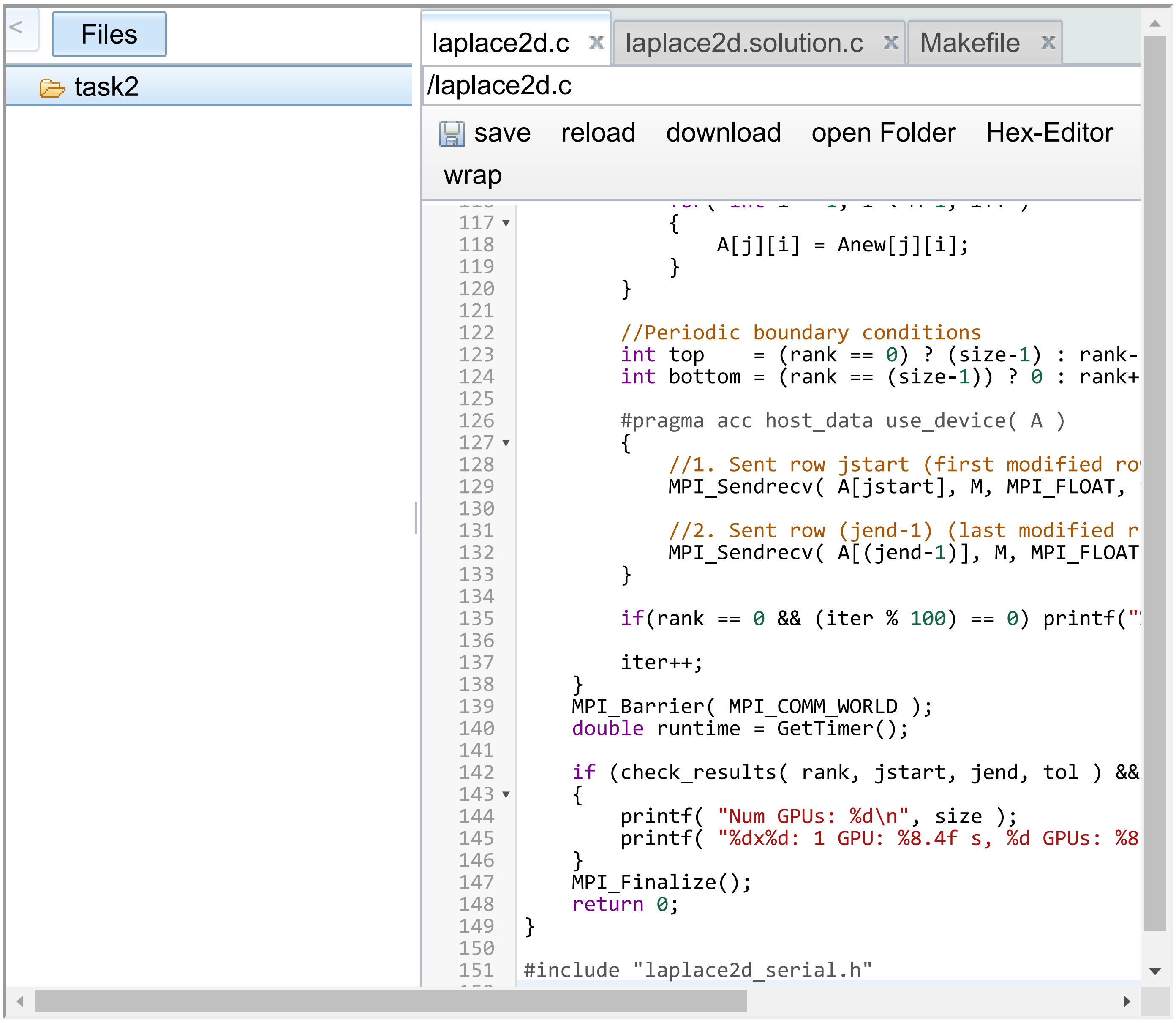
If we treat rank `0` as the bottom neighbour of rank `n-1` and rank `n-1` as the top neighbour of rank `0` this step will also handle the periodic boundary conditions and therefore the loop handling for those can be removed.

`#pragma acc host_data use_device( A )`

Since the iterations of the Jacobi solver are carried out on the GPUs the necessary halo updates also need to be done from and to the buffers in GPU memory. This is ensured with the directive `host_data use_device`, which was already inserted into the source code for you. `host_data use_device(A)` tells the OpenACC compiler to use the device representation of `A` in the following code block. Because we are using a CUDA-aware MPI the MPI implementation can handle these and do the halo updates directly to and from GPU memory. Without `host_data use_device(A)` the host representations of `A` would be passed to MPI and thus stale data would be exchanged between buffers in host memory which are not being used during the iterations of the Jacobi solver. If you want, you can try to remove the `host_data use_device(A)` directive and see what happens.

Like in Task #1 you should look out for `TODO` in `C/task2/laplace2d.c`. These will guide you through the following steps:

- Decompose the calculation across the GPUs/MPI ranks by adjusting the first and last row to be processed by each rank.
- Use `MPI_Allreduce` to calculate the global error across all GPUs/MPI ranks.
- Handle the periodic boundary conditions and the halo exchange with MPI as described above.



```
            {
                A[j][i] = Anew[j][i];
            }
        }

        //Periodic boundary conditions
        int top    = (rank == 0) ? (size-1) : rank-
        int bottom = (rank == (size-1)) ? 0 : rank+

        #pragma acc host_data use_device( A )
        {
            //1. Sent row jstart (first modified ro
            MPI_Sendrecv( A[jstart], M, MPI_FLOAT,

            //2. Sent row (jend-1) (last modified r
            MPI_Sendrecv( A[(jend-1)], M, MPI_FLOAT
        }

        if(rank == 0 && (iter % 100) == 0) printf("

        iter++;
    }
    MPI_Barrier( MPI_COMM_WORLD );
    double runtime = GetTimer();

    if (check_results( rank, jstart, jend, tol ) &&
    {
        printf( "Num GPUs: %d\n", size );
        printf( "%dx%d: 1 GPU: %8.4f s, %d GPUs: %8
    }
    MPI_Finalize();
    return 0;
}

#include "laplace2d_serial.h"
```

In [19]:

```
!make -C C task2
```

```
make: Entering directory `/home/ubuntu/notebook/C'
make -C task2
make[1]: Entering directory `/home/ubuntu/notebook/C/task2'
mpicc -fast -acc -ta=nvidia laplace2d.c -o laplace2d
mpirun -np 4 ./laplace2d
Jacobi relaxation Calculation: 4096 x 4096 mesh
Calculate reference solution and time serial execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 4
4096x4096: 1 GPU:   5.5903 s, 4 GPUs:   1.7159 s, speedup:     3.26, efficie
ncy:    81.45%
make[1]: Leaving directory `/home/ubuntu/notebook/C/task2'
make: Leaving directory `/home/ubuntu/notebook/C'
```

At the end of the output you will see a output similar to this:

```
Num GPUs: 4
4096x4096: 1 GPU:   5.2692 s, 4 GPUs:   1.6237 s, speedup:     3.25, efficiency:
  81.13%
```

Now we are getting a speed up when using four GPUs. The efficiency is already quite decent but we can do better by hiding communication times. This will be covered in the next task.

Optional: Using the NVIDIA Visual Profiler (NVVP)

As described in CUDA Pro Tip: Profiling MPI Applications (http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-profiling-mpi-applications/) `nvprof` can be used to generate profiles of MPI+OpenACC applications. Execute the `make task2.profile` in the cell below to generate a profile for each MPI rank executing your solution of task 2.

To view the generated profiles we'll be using the NVIDIA Visual Profiler (NVVP) tool which comes standard with the CUDA Toolkit software. To launch the tool please click here (/vnc) which will open a new browser window. **Note that it may take a few seconds for NVVP to start.**

After NVVP has started, import the generated profiles by clicking on "File" and then "Import...". In the dialog select "nvprof" and "Multiple Processes". Browse to `ubuntu/notebook/C/task2` and select `laplace2d.[0-3].nvvp`.

If you've never used NVVP before or if you want to read more about you can [click here (https://developer.nvidia.com/nvidia-visual-profiler)](https://developer.nvidia.com/nvidia-visual-profiler) for more information.

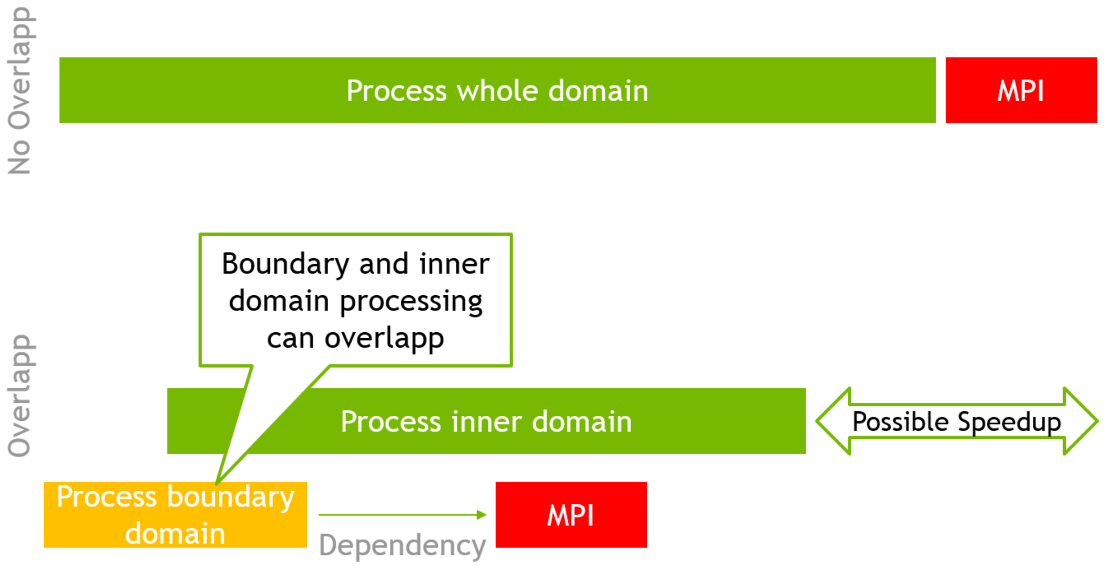
In [20]:

```
!make -C C task2.profile
```

```
make: Entering directory `/home/ubuntu/notebook/C'
make -C task2 profile
make[1]: Entering directory `/home/ubuntu/notebook/C/task2'
mpirun -np 4 nvprof -o laplace2d.%q{OMPI_COMM_WORLD_RANK}.nvvp ./laplace2d
==26365== NVPROF is profiling process 26365, command: ./laplace2d
==26366== NVPROF is profiling process 26366, command: ./laplace2d
==26368== NVPROF is profiling process 26368, command: ./laplace2d
==26367== NVPROF is profiling process 26367, command: ./laplace2d
Jacobi relaxation Calculation: 4096 x 4096 mesh
Calculate reference solution and time serial execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 4
4096x4096: 1 GPU:   5.8463 s, 4 GPUs:   1.8471 s, speedup:     3.17, efficie
ncy:    79.13%
==26366== Generated result file: /home/ubuntu/notebook/C/task2/laplace2d.0.n
vvp
==26368== Generated result file: /home/ubuntu/notebook/C/task2/laplace2d.3.n
vvp
==26367== Generated result file: /home/ubuntu/notebook/C/task2/laplace2d.1.n
vvp
==26365== Generated result file: /home/ubuntu/notebook/C/task2/laplace2d.2.n
vvp
make[1]: Leaving directory `/home/ubuntu/notebook/C/task2'
make: Leaving directory `/home/ubuntu/notebook/C'
```

Task #3

By applying a domain decomposition and distributing the work across multiple GPUs in task #2 we could get a speed-up but do not attain optimal efficiency. This is because of the time that is needed (spent) to carry out the halo updates using MPI. This wasted time is called "parallel overhead" because it is a step not necessary for execution with a single GPU. We can lower the parallel overhead by doing computations in parallel with the MPI communication and therefore hide the communication time. In the case of our Jacobi solver, this is best done by splitting each domain into a boundary part (which updates all values that we need to communicate) and an inner part. By doing this split, we can start the MPI communication after the boundary part has finished, and let it run in parallel with the inner part:



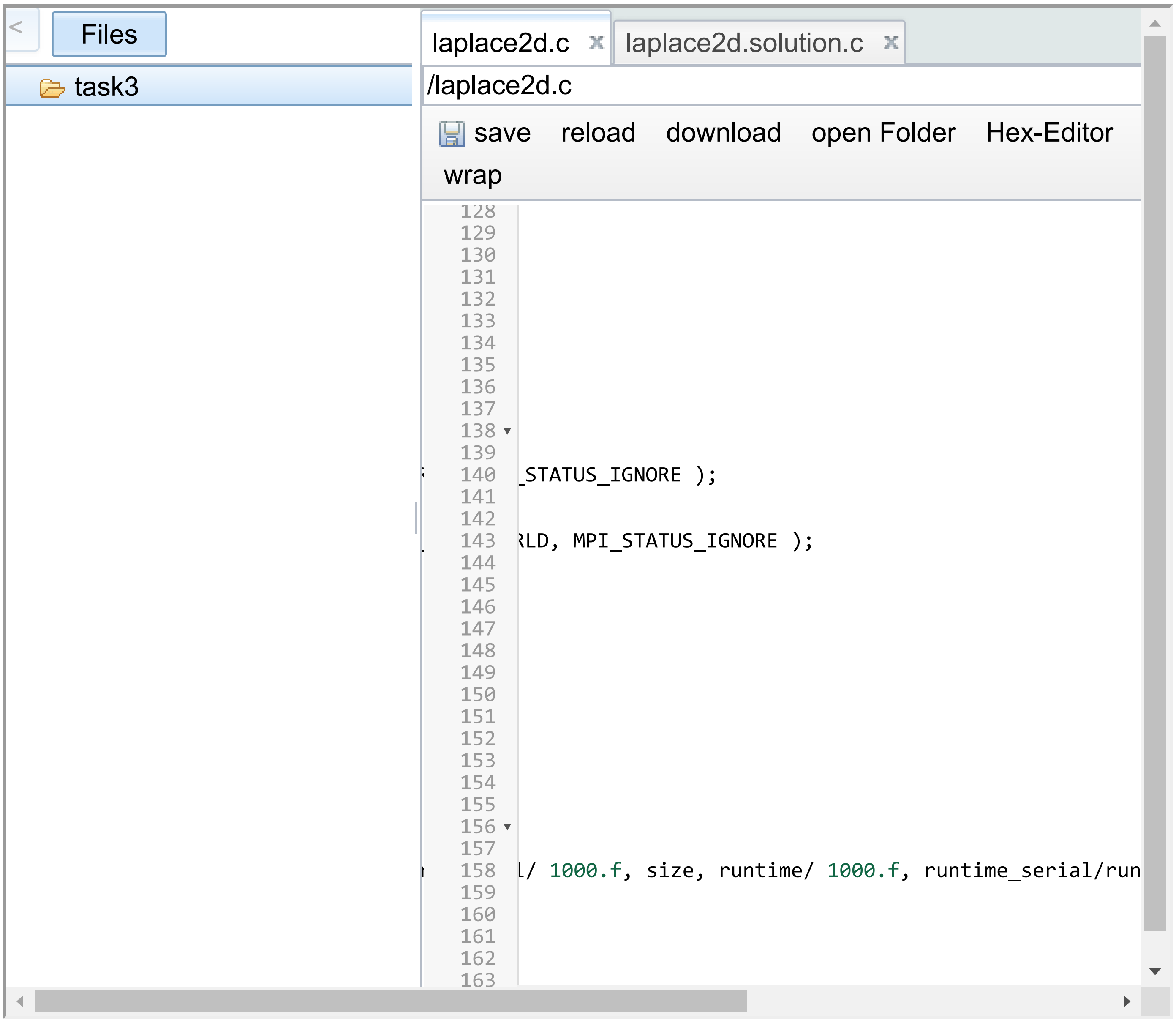
In OpenACC this can be done by using the async clause on a kernels region as outlined below.

```
#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
```

In this task you should apply this approach to the copy loop of the Jacobi solver. As in in the earlier tasks in this lab, you should look for `TODO` in `C/task3/laplace2d.c`. These will guide you through the following steps:

- Split the copy loop into its constituent halo and bulk parts.
- Start the computation of the bulk part asynchronously.
- Wait for the bulk part to complete at the end of the iteration.



```
_STATUS_IGNORE );


RLD, MPI_STATUS_IGNORE );
```

```
l/ 1000.f, size, runtime/ 1000.f, runtime_serial/run
```

In [21]:

```
!make -C C task3
```

```
make: Entering directory `/home/ubuntu/notebook/C'
make -C task3
make[1]: Entering directory `/home/ubuntu/notebook/C/task3'
mpicc -fast -acc -ta=nvidia laplace2d.c -o laplace2d
mpirun -np 4 ./laplace2d
Jacobi relaxation Calculation: 4096 x 4096 mesh
Calculate reference solution and time serial execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Parallel execution.
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000402
  700, 0.000345
  800, 0.000302
  900, 0.000268
Num GPUs: 4
4096x4096: 1 GPU:   5.5845 s, 4 GPUs:   1.6482 s, speedup:     3.39, efficie
ncy:    84.71%
make[1]: Leaving directory `/home/ubuntu/notebook/C/task3'
make: Leaving directory `/home/ubuntu/notebook/C'
```

Summary

In this lab you have learned that using CUDA-aware MPI in combination with OpenACC is a effective way to exploit the power of multiple GPUs in a node or a cluster.

Post-Lab

Finally, don't forget to save your work from this lab before time runs out and the instance shuts down!!

1. Save this IPython Notebook by going to `File -> Download as -> IPython (.ipynb)` at the top of this window
2. You can execute the following cell block to create a zip-file of the files you've been working on, and download it with the link below.

In [3]:

```
%%bash
rm -f multi_gpu_mpi_openacc_files.zip
zip -r multi_gpu_mpi_openacc_files.zip C FORTRAN
```

```
  adding: C/ (stored 0%)
  adding: C/Makefile (deflated 70%)
  adding: C/task1/ (stored 0%)
  adding: C/task1/Makefile.solution (deflated 63%)
  adding: C/task1/laplace2d_serial.h (deflated 59%)
  adding: C/task1/Makefile (deflated 47%)
  adding: C/task1/laplace2d.c (deflated 61%)
  adding: C/task1/laplace2d.solution.c (deflated 60%)
  adding: C/task1/common.h (deflated 53%)
  adding: C/task3/ (stored 0%)
  adding: C/task3/laplace2d_serial.h (deflated 59%)
  adding: C/task3/Makefile (deflated 62%)
  adding: C/task3/laplace2d.c (deflated 60%)
  adding: C/task3/laplace2d.solution.c (deflated 61%)
  adding: C/task3/common.h (deflated 53%)
  adding: C/task2/ (stored 0%)
  adding: C/task2/laplace2d_serial.h (deflated 59%)
  adding: C/task2/Makefile (deflated 63%)
  adding: C/task2/laplace2d.c (deflated 63%)
  adding: C/task2/laplace2d.solution.c (deflated 60%)
  adding: C/task2/common.h (deflated 53%)
  adding: FORTRAN/ (stored 0%)
  adding: FORTRAN/Makefile (deflated 70%)
  adding: FORTRAN/task1/ (stored 0%)
  adding: FORTRAN/task1/laplace2d.F03 (deflated 62%)
  adding: FORTRAN/task1/Makefile.solution (deflated 64%)
  adding: FORTRAN/task1/Makefile (deflated 42%)
  adding: FORTRAN/task1/laplace2d_serial.F03 (deflated 61%)
  adding: FORTRAN/task1/laplace2d.solution.F03 (deflated 62%)
  adding: FORTRAN/task3/ (stored 0%)
  adding: FORTRAN/task3/laplace2d.F03 (deflated 63%)
  adding: FORTRAN/task3/Makefile (deflated 64%)
  adding: FORTRAN/task3/laplace2d_serial.F03 (deflated 61%)
  adding: FORTRAN/task3/laplace2d.solution.F03 (deflated 64%)
  adding: FORTRAN/task2/ (stored 0%)
  adding: FORTRAN/task2/laplace2d.F03 (deflated 64%)
  adding: FORTRAN/task2/Makefile (deflated 64%)
  adding: FORTRAN/task2/laplace2d_serial.F03 (deflated 61%)
  adding: FORTRAN/task2/laplace2d.solution.F03 (deflated 63%)
```

After executing the above cell, you should be able to download the zip file here (files/multi_gpu_mpi_openacc_files.zip)

References/Further Reading

- Learn more at the CUDA Developer Zone (https://developer.nvidia.com/category/zone/cuda-zone).
- If you have an NVIDIA GPU in your system, you can download and install the CUDA tookit (https://developer.nvidia.com/cuda-toolkit).
- Take the fantastic online and **free** Udacity Intro to Parallel Programming (https://www.udacity.com/course/cs344) course which uses CUDA C.

- Search or ask questions on Stackoverflow (http://stackoverflow.com/questions/tagged/cuda) using the cuda tag
- Read the GPU Computing developer blog Parallel Forall (http://devblogs.nvidia.com/parallelforall/)

Lab FAQ

Q: I'm encountering issues executing the cells, or other technical problems?
A: Please see this (https://developer.nvidia.com/self-paced-labs-faq#Troubleshooting) infrastructure FAQ.

Hints

Task #1 - Hints

Hint #1

To handle GPU affinity we assign each MPI rank to one GPU so you need to map the rank to the id of the device to use.

Hint #2

You can assume that MPI ranks are started consecutively on each node. I.e., with 8 processes on 2 nodes rank 0,1,2, and 3 are started on node 0 and rank 4,5,6,7 are started on node 1.

Hint #3

```
You can use code like the following to handle the GPU affinity.
#if _OPENACC
int ngpus=acc_get_num_devices(acc_device_nvidia);
int devicenum=rank%ngpus;
acc_set_device_num(devicenum,acc_device_nvidia);
#endif
```

Return to Task #1