

## OpenACC: 2X in 4 Steps (for C)

In this self-paced, hands-on lab, we will use [OpenACC \(http://openacc.org/\)](http://openacc.org/) directives to port a basic C program to an accelerator in four simple steps, achieving *at least* a two-fold speed-up.

Lab created by John Coombs, Mark Harris, and Mark Ebersole (Follow [@CUDAHamster \(https://twitter.com/@cudahamster\)](https://twitter.com/@cudahamster) on Twitter)

Lets begin by getting information about the GPUs on the server by running the command below.

```
In [1]: %%bash
nvidia-smi
```

```
Tue Jun 20 13:10:41 2017
```

NVIDIA-SMI 375.66					Driver Version: 375.66			
GPU	Name		Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC	
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	
0	GeForce GTX 950		Off	0000:01:00.0	On		N/A	
23%	59C	P0	27W / 99W	690MiB / 1996MiB		1%	Default	

## Introduction to OpenACC

Open-specification OpenACC directives are a straightforward way to accelerate existing Fortran and C applications. With OpenACC directives, you provide hints via compiler directives (or 'pragmas') to tell the compiler where -- and how -- it should parallelize compute-intensive code for execution on an accelerator.

If you've done parallel programming using OpenMP, OpenACC is very similar: using directives, applications can be parallelized *incrementally*, with little or no change to the Fortran, C or C++ source. Debugging and code maintenance are easier. OpenACC directives are designed for *portability* across operating systems, host CPUs, and accelerators. You can use OpenACC directives with GPU accelerated libraries, explicit parallel programming languages (e.g., CUDA), MPI, and OpenMP, *all in the same program*.

This hands-on lab walks you through a short sample of a scientific code, and demonstrates how you can employ OpenACC directives using a four-step process. You will make modifications to a simple C program, then compile and execute the newly enhanced code in each step. Along the way, hints and solution are provided, so you can check your work, or take a peek if you get lost.

## The Value of 2X in 4 Steps

You can accelerate your applications using OpenACC directives and achieve *at least* a 2X speed-up, using 4 straightforward steps:

1. Characterize your application
2. Add compute directives
3. Minimize data movement
4. Optimize kernel scheduling

The content of these steps and their order will be familiar if you have ever done parallel programming on other platforms. Parallel programmers deal with the same issues whenever they tackle a new set of code, no matter what platform they are parallelizing an application for. These issues include:

- optimizing and benchmarking the serial version of an application
- profiling to identify the compute-intensive portions of the program that can be executed concurrently
- expressing concurrency using a parallel programming notation (e.g., OpenACC directives)
- compiling and benchmarking each new/parallel version of the application
- locating problem areas and making improvements iteratively until the target level of performance is reached

The programming manual for some other parallel platform you've used may have suggested five steps, or fifteen. Whether you are an expert or new to parallel programming, we recommend that you walk through the four steps here as a good way to begin accelerating applications by at least 2X using OpenACC directives. We believe *being more knowledgeable about the four steps* will make the process of programming for an accelerator more understandable *and* more manageable. The 2X in 4 Steps process will help you use OpenACC on your own codes more productively, and get significantly better speed-ups in less time.

## Step 1 - Characterize Your Application

The most difficult part of accelerator programming begins before the first line of code is written. If your program is not highly parallel, an accelerator or coprocessor won't be much use. Understanding the code structure is crucial if you are going to *identify opportunities* and *successfully* parallelize a piece of code. The first step in OpenACC programming then is to *characterize the application*. This includes:

- benchmarking the single-thread, CPU-only version of the application
- understanding the program structure and how data is passed through the call tree
- profiling the application and identifying computationally-intense "hot spots"
  - which loop nests dominate the runtime?
  - what are the minimum/average/maximum tripcounts through these loop nests?
  - are the loop nests suitable for an accelerator?
- insuring that the algorithms you are considering for acceleration are *safely* parallel

Note: what we've just said may sound a little scary, so please note that as parallel programming methods go OpenACC is really pretty friendly: think of it as a sandbox you can play in. Because OpenACC directives are incremental, you can add one or two directives at a time and see how things work: the compiler provides a *lot* of feedback. The right software plus good tools plus educational experiences like this one should put you on the path to successfully accelerating your programs.

We will be accelerating a 2D-stencil called the Jacobi Iteration. Jacobi Iteration is a standard method for finding solutions to a system of linear equations. The basic concepts behind a Jacobi Iteration are described in the following video:

<http://www.youtube.com/embed/UOSYi3oLIRs> (<http://www.youtube.com/embed/UOSYi3oLIRs>)

Here is the serial C code for our Jacobi Iteration:

```

#include <math.h>
#include <string.h>
#include <openacc.h>
#include "timer.h"
#include <stdio.h>

#define NN 1024
#define NM 1024

float A[NN][NM];
float Anew[NN][NM];

int main(int argc, char** argv)
{
    int i,j;
    const int n = NN;
    const int m = NM;
    const int iter_max = 1000;
    const double tol = 1.0e-6;
    double error = 1.0;

    memset(A, 0, n * m * sizeof(float));
    memset(Anew, 0, n * m * sizeof(float));

    for (j = 0; j < n; j++)
    {
        A[j][0] = 1.0;
        Anew[j][0] = 1.0;
    }

    printf("Jacobi relaxation Calculation: %d x %d mesh\n", n, m);

    StartTimer();
    int iter = 0;

    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

        for( j = 1; j < n-1; j++)
        {
            for( i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                     + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

        for( j = 1; j < n-1; j++)
        {
            for( i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }

    double runtime = GetTimer();

    printf(" total: %f s\n", runtime / 1000);

    return 0;
}

```

In this code, the outer 'while' loop iterates until the solution has converged, by comparing the computed error to a specified error tolerance, *tol*. The first of two sets of inner nested loops applies a 2D Laplace operator at each element of a 2D grid, while the second set copies the output back to the input for the next iteration.

## Benchmarking

Before you start modifying code and adding OpenACC directives, you should benchmark the serial version of the program. To facilitate benchmarking after this and every other step in our parallel porting effort, we have built a timing routine around the main structure of our program -- a process we recommend you follow in your own efforts. Let's run the `task1.c` ([/4vwkFv7K/edit/C/task1/task1.c](#)) file without making any changes -- using the `-fast` set of compiler options on the serial version of the Jacobi Iteration program -- and see how fast the serial program executes. This will establish a baseline for future comparisons. Execute the following two commands to compile and run the program.

```
In [2]: %%bash
# To be sure we see some output from the compiler, we'll echo out "Compiled Successfully!"
#(if the compile does not return an error)
pgcc -fast -o task1_pre_out task1/task1.c && echo 'Compiled Successfully!'

Compiled Successfully!
```

```
In [3]: %%bash
# Execute our single-thread CPU-only Jacobi Iteration to get timing information.
# Make sure you compiled successfully in the
# above command first.
./task1_pre_out

Jacobi relaxation Calculation: 1024 x 1024 mesh
  0, 0.250000
 100, 0.002397
 200, 0.001204
 300, 0.000804
 400, 0.000603
 500, 0.000483
 600, 0.000403
 700, 0.000345
 800, 0.000302
 900, 0.000269
total: 2.815460 s
```

## Quality Checking/Keeping a Record

This is a good time to briefly talk about having a quality check in your code before starting to offload computation to an accelerator (or do any optimizations, for that matter). It doesn't do you any good to make an application run faster if it does not return the correct results. It is thus very important to have a quality check built into your application before you start accelerating or optimizing. This can be a simple value print out (one you can compare to a non-accelerated version of the algorithm) or something else.

In our case, on every 100th iteration of the outer `while` loop, we print the current max error. (You just saw an example when we executed `task1_pre_out`.) As we add directives to accelerate our code later in this lab, you can look back at these values to verify that we're getting the correct answer. These print-outs also help us verify that we are converging on a solution -- which means that we should see, as we proceed, that the values are approaching zero.

**Note:** NVIDIA GPUs implement IEEE-754 compliant floating point arithmetic just like most modern CPUs. However, because floating point arithmetic is not associative, the order of operations can affect the rounding error inherent with floating-point operations: you may not get exactly the same answer when you move to a different processor. Therefore, you'll want to make sure to verify your answer within an acceptable error bound. Please read [this \(https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus\)](https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus) article at a later time, if you would like more details.

After each step, we will record the results from our benchmarking and correctness tests in a table like this one:

Step	Execution	ExecutionTime (s)	Speedup vs. 1 CPU Thread	Correct?	Programming Time
1	CPU 1 thread	2.95		Yes	

Note: Problem Size: 1024 x 1024; System Information: GK520; Compiler: PGI Community Edition 17.4

(The execution times quoted will be times we got running on our GK520 -- your times throughout the lab may vary for one reason or another.)

You may also want to track how much time you spend porting your application, step by step, so a column has been included for recording time spent.

## Profiling

Back to our lab. Your objective in the step after this one (Step 2) will be to modify `task2.c` ([/4vwkFv7K/edit/C/task2/task2.c](#)) in a way that moves the most computationally intensive, independent loops to the accelerator. With a simple code, you can identify which loops are candidates for acceleration with a little bit of code inspection. On more complex codes, a great way to find these computationally intense areas is to use a profiler (such as PGI's `pgprof`, NVIDIA's `nvprof` or open-source `gprof`) to determine which functions are consuming the largest amounts of compute time. To profile a C program on your own workstation, you'd type the lines below on the command line, but in this workshop, you just need to execute the following command, and then click on the link below it to see the `pgprof` interface

```
In [4]: %%bash
pgcc -Minfo=all,ccff -fast -o task1/task1_simple_out task1/task1_simple.c && echo 'Compiled Successfully!'

Compiled Successfully!

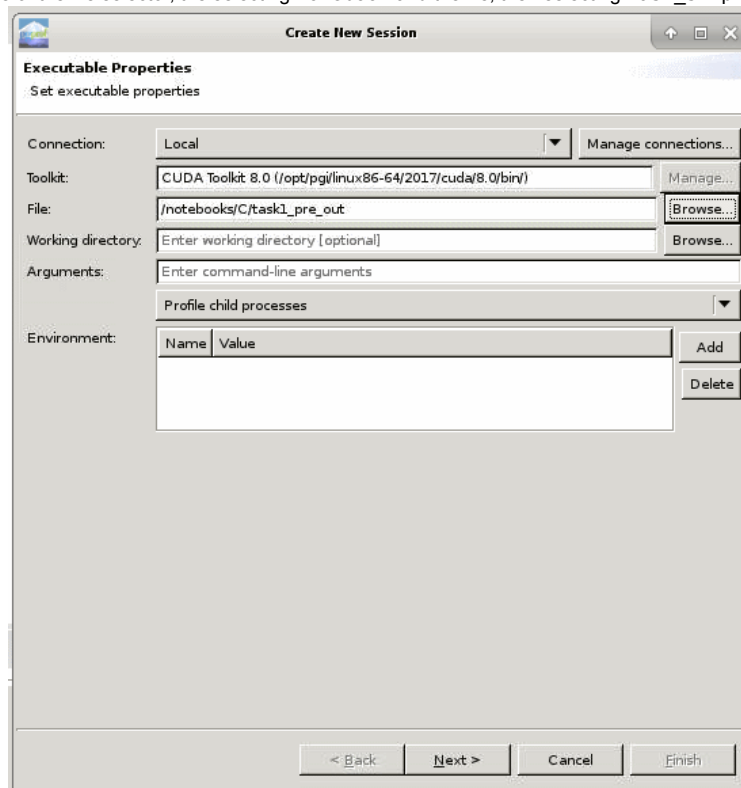
GetTimer:
    3, include "timer.h"
    62, FMA (fused multiply-add) instruction(s) generated

main:
    25, Loop not fused: function call before adjacent loop
    Loop not vectorized: may not be beneficial
    Unrolled inner loop 8 times
    Generated 7 prefetches in scalar loop
    42, Generated vector simd code for the loop containing reductions
    Generated 3 prefetch instructions for the loop
    Residual loop unrolled 2 times (completely unrolled)
    52, Memory copy idiom, loop replaced by call to __c_mcopy4
```

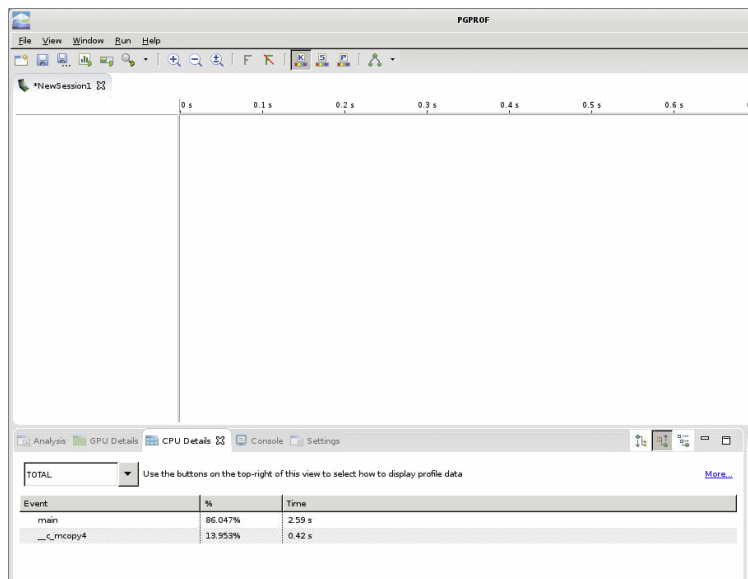
In this lab, to open the PGI profiler run the following command.

```
In [ ]: %%bash
pgprof
```

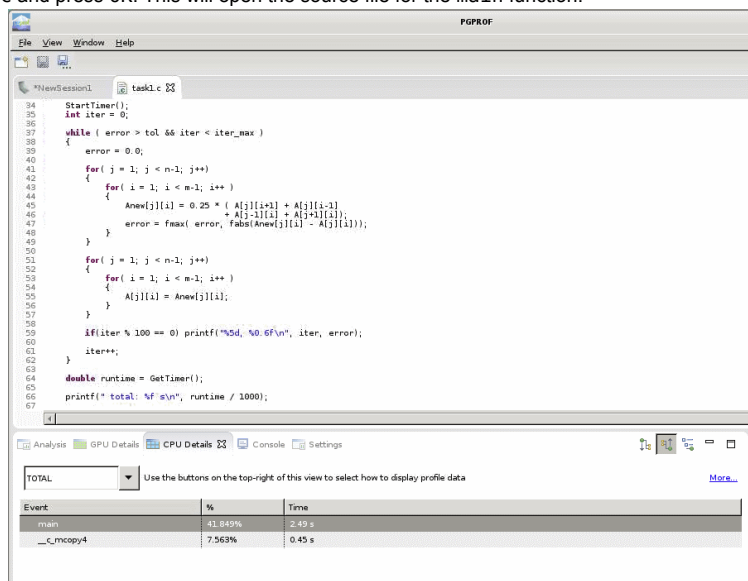
Click on File > New Session to start a new profiling session. Select the executable to profile by pressing the Browse button, clicking ubuntu from the file left side of the file selector, the selecting notebook and then C, then selecting task\_simple\_out.



Clicking Next will bring up a screen with a list profiling settings for this session. We can leave those at their default settings for now. Clicking Finish will cause pgprof to launch your executable for profiling. Since we are profiling a regular CPU application (no acceleration added yet) we should refer to the CPU Details tab along the bottom of the window for a summary of what functions in our program take the most compute time on the CPU. If you do not have a CPU Details tab, click View -> Show CPU Details View.



Double-clicking on the most time-consuming function in the table, `main` in this case, will bring up another file browser. This time click on `Recently Used` and then `C` and press `OK`. This will open the source file for the `main` function.



In our Jacobi code sample, the compute-intensive part of our code is the two for-loops nested inside the while loop in the function `main`. This is where we'll concentrate our effort in adding OpenACC to the code.

Let's see what it takes to accelerate those loops.

## Step 2 - Add Compute Directives

In C, an OpenACC directive is indicated in the code by `'#pragma acc *your directive*'`. This is very similar to OpenMP programming and gives hints to the compiler on how to handle the compilation of your source. If you are using a compiler which does not support OpenACC directives, it will simply ignore the `'#pragma acc'` directives and move on with the compilation.

In Step 2, you will add compute regions around your expensive parallel loop(s). The first OpenACC directive you're going to learn about is the *kernels* directive. The kernels directive gives the compiler a lot of freedom in how it tries to accelerate your code - it basically says, "Compiler, I believe the code in the following region is parallelizable, so I want you to try and accelerate it as best you can."

Like most OpenACC directives in C/C++, the kernels directive applies to the structured code block immediately following the `#pragma acc *directive*`. For example, each of the following code samples instructs the compiler to generate a kernel -- from suitable loops -- for execution on an accelerator:

```

#pragma acc kernels
{
    // accelerate suitable loops here
}
// but not these loops

```

or

```
#pragma acc kernels
for ( int i = 0; i < n; ++i )
{ // body of for-loop
... // The for-loop is a structured block, so this code will be accelerated
}
... // Any code here will not be accelerated since it is outside of the for-loop
```

One, two or several loops may be inside the structured block, the kernels directive will try to parallelize it, telling you what it found and generating as many kernels as it thinks it safely can. At some point, you will encounter the OpenACC *parallel* directive, which provides another method for defining compute regions in OpenACC. For now, let's drop in a simple OpenACC kernels directive in front of and embracing *both* the two for-loop codeblocks that follow the while loop using curly braces. The kernels directive is designed to find the parallel acceleration opportunities implicit in the for-loops in the Jacobi iteration code.

To get some hints about how and where to place your kernels directives, click on the links below. When you feel you are done, **make sure to save the [task2.c \(/4vwkFv7K/edit/C/task2/task2.c\)](#) file you've modified with File -> Save, and continue on.** If you get completely stuck, you can look at [task2\\_solution.c \(/4vwkFv7K/edit/C/task2/task2\\_solution.c\)](#) to see the answer.

[Hint #1](#)

[Hint #2](#)

Let's now compile our [task2.c \(/4vwkFv7K/edit/C/task2/task2.c\)](#) file by executing the command below with Ctrl-Enter (or press the play button in the toolbar above). Note that we've now added a new compiler option `-ta` to specify the type of accelerator to use. We've set it to `tesla` as we're using NVIDIA GPUs in this lab.

```
In [ ]: %%bash
# Compile the task2.c file with the pgcc compiler
# -acc tells the compiler to process the source recognizing #pragma acc directives
# -Minfo tells the compiler to share information about the compilation process
pgcc -acc -Minfo -fast -ta=tesla -o task2_out task2/task2.c && echo 'Compiled Successfully'
```

If you successfully added `#pragma acc kernels` in the proper spots, you should see the following in the output of the compiler:

main:

```
23, Loop not fused: function call before adjacent loop
    Loop not vectorized: may not be beneficial
    Unrolled inner loop 8 times
    Generated 7 prefetches in scalar loop
34, Loop not vectorized/parallelized: potential early exits
36, Generating copyout(Anew[1:1022][1:1022])
    Generating copyin(A[:][:])
    Generating copyout(A[1:1022][1:1022])
41, Loop is parallelizable
43, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
    41, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
    43, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    47, Max reduction generated for error
52, Loop is parallelizable
54, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
    52, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
    54, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

If you do not get similar output, please check your work and try re-compiling. If you're stuck, you can compare what you have to [task2\\_solution.c](#) in the editor above.

*The output provided by the compiler is extremely useful, and should not be ignored when accelerating your own code with OpenACC.* Let's break it down a bit and see what it's telling us.

1. First since we used the `-Minfo` command-line option, we will see all output from the compiler. If we were to use `-Minfo=accel` we would only see the output corresponding to the accelerator, in this case an NVIDIA GPU.
2. The first line of the output, *main*, tells us which function the following information is in reference to.
3. The line starting with 41, *Loop is parallelizable* of the output tells us that on line 41 in our source, an accelerated kernel was generated. This is the the loop just after where we put our `#pragma acc kernels`.
4. The following lines provide more details on the accelerator kernel on line 42. It shows we created a parallel OpenACC loop. This loop is made up of gangs (a grid of blocks in CUDA language) and vector parallelism (threads in CUDA language) with the vector size being 128 per gang.
5. At line 54, the compiler tells us it found another loop to accelerate.
6. The rest of the information concerns data movement which we'll get into later in this lab.

So as you can see, lots of useful information is provided by the compiler, and it's very important that you carefully inspect this information to make sure the compiler is doing what you've asked of it.

Finally, let's execute this program to verify we are getting the correct answer (execute the command below).

Once you feel your code is correct, try running it by executing the command below. You'll want to review our quality check from the beginning of task2 to make sure you didn't break the functionality of your application.

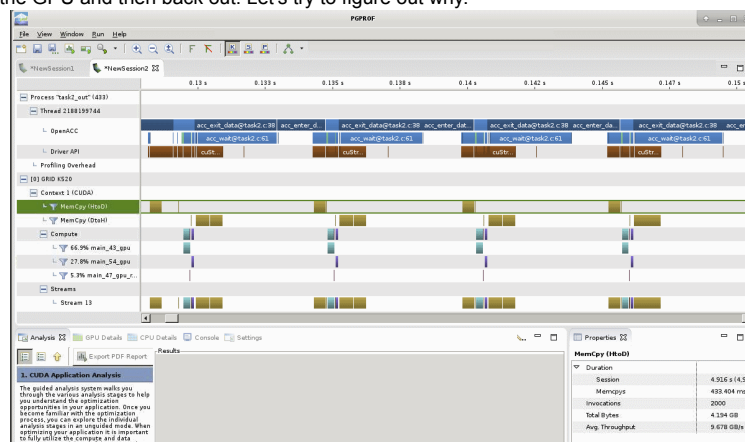
```
In [ ]: %%bash
        ./task2_out
```

Let's record our results in the table:

Step	Execution	Time(s)	Speedup vs. 1 CPU Thread	Correct?	Programming Time
1	CPU 1 thread	2.95			
2	Add kernels directive	4.93	0.60X	Yes	

*Note: Problem Size: 1024x1024; System Information: GK520; Compiler: PGI Community Edition 17.4*

Now, if your solution is similar to the one in task2\_solution.c, you have probably noticed that we're executing **slower** than the non-accelerated, CPU-only version we started with. What gives?! Let's see what `pgprof` can tell us about the performance of the code. Return to your PGPROF window from earlier, start another new session, but this time loading `task2_out` as your executable (it's in the same directory as before). This time we'll find a colorful graph of what our program is doing, this is the GPU timeline. We can't tell much from the default view, but we can zoom in by using the + magnifying glass at the top of the window. If you zoom in far enough, you'll begin to see a pattern like the one in the screenshot below. The teal and purple boxes are the compute kernels that go with the two loops in our kernels region. Each of these groupings of kernels is surrounded by tan colored boxes representing data movement. What this graph is showing us is that for every step of our while loop, we're copying data to the GPU and then back out. Let's try to figure out why.



The compiler feedback we collected earlier tells you quite a bit about data movement. If we look again at the compiler feedback from above, we see the following.

```
36, Generating copyout(Anew[1:1022][1:1022])
    Generating copyin(A[1:1022][1:1022])
    Generating copyout(A[1:1022][1:1022])
```

This is telling us that the compiler has inserted data movement around our kernel's region at line 36 which copies the A array *in* and *out* of GPU memory and also copies Anew out. This problem of copying back and forth on every iteration of a loop is sometimes called "data sloshing".

The OpenACC compiler can only work with the information we have given it. It knows we need the A and Anew arrays on the GPU for each of our two accelerated sections, but we didn't tell it anything about what happens to the data outside of those sections. Without this knowledge, it has to copy the full arrays *to the GPU and back to the CPU* for each accelerated section, *every time* it went through the while loop. That is a LOT of wasted data transfers.

Ideally, we would just transfer A to the GPU at the beginning of the Jacobi Iteration, and then only transfer A back to the CPU at the end. As for Anew, it's only used within this region, so we don't need to copy any data back and forth, we only need to create space on the device for this array.

Because overall accelerator performance is determined largely by how well memory transfers are optimized, the OpenACC specification defines the data directive and several modifying clauses to manage all the various forms of data movement.

### Step 3 - Manage Data Movement

We need to give the compiler more information about how to reduce unnecessary data movement for the Jacobi Iteration. We are going to do this with the OpenACC data directive and some modifying clauses defined in the OpenACC specification.

In C, the data directive applies to the next structured code block. The compiler will manage data according to the provided clauses. It does this at the beginning of the data directive code block, and then again at the end. Some of the clauses available for use with the data directive are:

- `copy( list )` - Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- `copyin( list )` - Allocates memory on GPU and copies data from host to GPU when entering region.



- `copyout( list )` - Allocates memory on GPU and copies data to the host when exiting region.
- `create( list )` - Allocates memory on GPU but does not copy.
- `present( list )` - Data is already present on GPU from another containing data region.

As an example, the following directive copies array A to the GPU at the beginning of the code block, and back to the CPU at the end. It also copies arrays B and C *to the CPU* at the *end* of the code block, but does **not** copy them both to the GPU at the beginning:

```
#pragma acc data copy( A ), copyout( B, C )
{
    ....
}
```

For detailed information on the data directive clauses, you can refer to the [OpenACC 2.5](http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf) ([http://www.openacc.org/sites/default/files/OpenACC\\_2pt5.pdf](http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf)) specification.

In the `task3.c` ([/4vwkFv7K/edit/C/task3/task3.c](#)) file, see if you can add in a data directive to minimize data transfers in the Jacobi iteration. There's a place for the `create` clause in this exercise too. As usual, there are some hints provided, and you can look at `task3_solution.c` ([/4vwkFv7K/edit/C/task3/task3\\_solution.c](#)) to see the answer if you get stuck or want to check your work. **Don't forget to save with File -> Save in the editor below before moving on.**

[Hint #1](#)

[Hint #2](#)

[Hint #3](#)

Once you think you have `task3.c` ([/4vwkFv7K/edit/C/task3/task3.c](#)) saved with a directive to manage data transfer, compile it with the below command and note the changes in the compiler output in the areas discussing data movement (lines starting with `Generating ...`). Then modify Anew using the `create` clause, if you haven't yet, and compile again.

```
In [ ]: %%bash
pgcc -fast -acc -Minfo=accel -ta=tesla -o task3_out task3/task3.c && echo 'Compiled Successfully'
```

How are we doing on our timings? Let's execute our step 3 program and see if we have indeed accelerated the application versus the execution time we recorded after Step #2.

```
In [ ]: %%bash
./task3_out
```

After making these changes, our accelerator code is much faster -- with just a few lines of OpenACC directives we have made our code more than twice as fast by running it on an accelerator, as shown in this table.

Step	Execution	Time (s)	Speedup vs. 1 CPU thread	Correct?	Programming Time
1	CPU 1 thread	2.95			
2	Add kernels directive	4.93	0.60X	Yes	
3	Manage data movement	0.45	6.56X	Yes	

*Note: Problem Size: 1024x1024; System Information: GK520; Compiler: PGI Community Edition 17.4*

We are making good progress, but we can still improve performance.

## Step 4 - Optimize Kernel Scheduling

The final step in our tuning process is to tune the OpenACC compute region schedules using the *gang* and *vector* clauses. These clauses let us use OpenACC directives to take more explicit control over how the compiler parallelizes our code for the accelerator we will be using.

Kernel scheduling optimizations *may* give you significantly higher speedup, but be aware that these particular optimizations can significantly reduce performance portability. The vast majority of the time, the default kernel schedules chosen by the OpenACC compilers are quite good, but other times the compiler doesn't do as well. Let's spend a little time examining how we could do better, if we were in a situation where we felt we needed to.

First, we need to get some additional insight into how our Jacobi iteration code with the data optimizations is running on the accelerator. Let's run the C code with all your data movement optimizations on the accelerator again. We could use `pgprof` again, but this time setting just the environment variable `PGI_ACC_TIME`, which will print some high level timers for us without leaving our command shell.

```
In [ ]: %%bash
export PGI_ACC_TIME=1
pgcc -acc -fast -ta=tesla -Minfo=accel -o accel_timing_out task3/task3.c
./accel_timing_out
```

This generates some information we haven't seen previously from the PGI compiler:

```
Accelerator Kernel Timing data
/notebooks/C/task3/task3.c
```

```

main NVIDIA devicenum=0
time(us): 379,107
34: data region reached 2 times
    34: data copyin transfers: 1
        device time(us): total=474 max=474 min=474 avg=474
    68: data copyout transfers: 1
        device time(us): total=473 max=473 min=473 avg=473
37: compute region reached 1000 times
    37: data copyin transfers: 1000
        device time(us): total=8,775 max=24 min=2 avg=8
    44: kernel launched 1000 times
        grid: [32x256] block: [32x4]
        device time(us): total=234,338 max=245 min=233 avg=234
        elapsed time(us): total=255,302 max=542 min=252 avg=255
    44: reduction kernel launched 1000 times
        grid: [1] block: [256]
        device time(us): total=20,969 max=28 min=20 avg=20
        elapsed time(us): total=41,157 max=61 min=38 avg=41
    44: data copyout transfers: 1000
        device time(us): total=18,007 max=29 min=13 avg=18
    55: kernel launched 1000 times
        grid: [32x256] block: [32x4]
        device time(us): total=96,071 max=105 min=95 avg=96
        elapsed time(us): total=117,348 max=191 min=115 avg=117

```

There is a lot of information here about how the compiler mapped the computational kernels in our program to our particular accelerator (in this case, an NVIDIA GPU). We can see three regions. The first one is the memcpy loop nest starting on line 34, which takes only a tiny fraction of the total system time. The second region is the nested computation loop starting on line 44, which takes about 0.25 seconds. The copyback (*copyout*) loop then executes beginning with line 68. We can see that region takes very little time -- which tells us there is no other part of the program that takes significant time. If we look at the main loop nests, we can see these lines:

```
grid: [32x256] block[32x4]
```

The terms *grid* and *block* come from the CUDA programming model. A GPU executes groups of threads called *thread blocks*. To execute a kernel, the application launches a *grid* of these thread blocks. Each block runs on one of the GPUs *multiprocessors* and is assigned a certain range of IDs that it uses to address a unique data range. In this case our thread blocks have 32x4, 128 threads each. The grid the compiler has constructed is also 2D, 32 blocks wide and 256 blocks tall. This is just enough to cover our 1024x1024 grid. But we don't really need that many blocks -- if we tell the compiler to launch fewer, it will automatically generate a sequential loop over data blocks within the kernel code run by each thread.

*Note: You can let the compiler do the hard work of mapping loop nests, unless you are certain you can do it better (and portability is not a concern.) When you decide to intervene, think about different parallelization strategies (loop schedules): in nested loops, distributing the work of the outer loops to the GPU multiprocessors (on OpenACC = gangs) in 1D grids. Similarly, think about mapping the work of the inner loops to the cores of the multiprocessors (CUDA threads, vectors) in 1D blocks. The grids (gangs) and block (vector) sizes can be viewed by setting the environment variable ACC\_NOTIFY. To get you started, here are some experiments we conducted for these computational kernels and this accelerator:*

Accelerator	Grid	Outer Loop Gang	Outer Loop Vector	Inner Loop Gang	Inner Loop Vector	Seconds
GK520	1024x1024		8		32	0.508
			4		64	0.510
				8	32	0.379
				16	32	0.410
				4	64	0.379

Try to modify the `task4.c (/4vwkFv7K/edit/C/task4/task4.c)` code for the main computational loop nests in the window below. You'll be using the openacc loop constructs `gang()` and `vector()`. Look at `task4_solution.c` if you get stuck:

#### Hint #1

After you've made some changes, save your work, then compile and run in the boxes below:

```
In [ ]: %bash
pgcc -acc -Minfo=accel -fast -ta=tesla -o task4_out task4/task4.c && echo 'Compiled Successfully'
```

```
In [ ]: %bash
./task4_out
```

Looking at `task4_solution.c (/4vwkFv7K/edit/C/task4/task4_solution.c)`, the `gang(8)` clause on the inner loop tells it to launch 8 blocks in the X(column) direction. The `vector(32)` clause on the inner loop tells the compiler to use blocks that are 32 threads (one warp) wide. The absence of clause on the outer loop lets the compiler decide how many rows of threads and how many blocks to use in the Y(row) direction. We can see

what it says, again, with:

```
In [ ]: %%bash
        export PGI_ACC_TIME=1
        ./task4_out
```

*\*Note: we usually want the inner loop to be vectorized, because it allows coalesced loading of data from global memory. This is almost guaranteed to give a big performance increase. Other optimizations are often trial and error. When selecting grid sizes, the most obvious mapping is to have\**

the number of gangs \* the number of workers \* the number of vectors = the total problem size.

*\*We may choose to manipulate this number, as we are doing here, so that each thread does multiple pieces of work -- this helps amortize the cost of setup for simple kernels.\**

*\*Note: Low-level languages like CUDA C/C++ offer more direct control of the hardware. You can consider optimizing your most critical loops in CUDA C/C++ if you need to extract every last bit of performance from your application, while recognizing that doing so may impact the portability of your code. OpenACC and CUDA C/C++ are fully interoperable.\**

A similar change to the copy loop nest benefits performance by a small amount. After you've made all your changes (look at task4\_solution.c to be sure) compile your code below:

```
In [ ]: %%bash
        pgcc -acc -fast -ta=tesla -Minfo=accel -o task4_out task4/task4.c && echo 'Compiled Successfully'
```

Then run it and record the run time of the optimized code in the table:

```
In [ ]: %%bash
        ./task4_out
```

Here is the performance after these final optimizations:

Step	Execution	Time (s)	Speedup vs. 1 CPU thread	Correct?	Programming Time
1	CPU 1 thread	2.95			
2	Add kernels directive	4.93	0.60X		
3	Manage data movement	0.45	6.56X	Yes	
4	Optimize kernel scheduling	0.33	8.9X	Yes	

*Note: Problem Size: 1024x1024; System Information: GK520; Compiler: PGI Community Edition 17.4*

At this point, some of you may be wondering what kind of speed-up we get against the OpenMP version of this code. If you look at [task1\\_omp.c](#) ([/4yvwkFv7K/edit/C/task4/task1\\_omp.c](#)) in the text editor above, you can see a simple OpenMP version of the Jacobi Iteration code. Running this using 8-OpenMP threads on an Intel Xeon E5-2670, our Kepler GK520 about 2X faster. If we scale the matrix up to an even larger 4096x4096, our Kepler GK520 GPU becomes significantly faster than the 8-OpenMP thread version. If you have some time remaining in this lab, feel free to compile & run the OpenMP and OpenACC versions below with the larger matrices.

First, compile the OpenMP version:

```
In [ ]: %%bash
        pgcc -fast -mp -Minfo -o task4_4096_omp task4/task4_4096_omp.c
```

Now run the OpenMP code you just created, and record your results in the new table for the larger matrix.

*Note: because our dataset has now grown by 16-fold your CPU may not seem as responsive. We're using -Minfo in the compile so you can see that something is indeed happening, but you may need to be patient.*

```
In [ ]: %%bash
        OMP_NUM_THREADS=8 ./task4_4096_omp
```

Now, compile and run the OpenACC solution for the larger 4096x4096 matrix using the next two boxes:

```
In [ ]: %%bash
        pgcc -acc -fast -ta=tesla -Minfo=accel -o task4_4096_out task4/task4_4096_solution.c && echo 'Compiled Suc
```

```
In [ ]: %%bash
        ./task4_4096_out
```

Here's our comparison with the larger matrix size:

Execution	matrix size	Time (s)	Speedup vs. 8 CPU threads	Correct?	Programming Time
CPU 8 threads	4096x4096	15.03		YES	
GPU optimized kernel	4096x4096	3.44	4.37X	Yes	

Note: System Information: GK520; Compiler: PGI Community Edition 17.4

## Learn More

If you are interested in learning more about OpenACC, you can use the following resources:

- [openacc.org \(http://openacc.org/\)](http://openacc.org/)
- [OpenACC on CUDA Zone \(https://developer.nvidia.com/openacc\)](https://developer.nvidia.com/openacc)
- Search or ask questions on [Stackoverflow \(http://stackoverflow.com/questions/tagged/openacc\)](http://stackoverflow.com/questions/tagged/openacc) using the openacc tag
- Get the free [PGI Community Edition \(https://www.pgroup.com/products/community.htm\)](https://www.pgroup.com/products/community.htm) compiler.
- Attend an in-depth workshop offered by XSEDE (<https://portal.xsede.org/overview> (<https://portal.xsede.org/overview>)) or a commercial provider (see the 'education' page at OpenACC.org)

## Hints

### Step #2 - Hint #1

Remember that in C, an OpenACC directive applies to the next structured code block. So for example, the following applies the `ernels` directive to the outer `for` loop and everything inside of it:

```
#pragma acc kernels
for ( int i = 0; i < n-1; i++ )
{
    for ( int j = 0; j < n-1; j++)
        ...
}
```

### Step #2 - Hint #2

If you choose to use only one `#pragma acc kernels` region -- which we recommend, because it demonstrates the power of the `ernels` directive -- you will need to add some additional `{ }` brackets so it applies to the correct region of code.

[Return to Step #2](#)

### Step #3 - Hint #1

You should only have to worry about managing the transfer of data in arrays `A` and `Anew`.

### Step #3 - Hint #2

You want to put the `data` directive just above the outer `while` loop.

### Step #3 - Hint #3

You'll want to copy ( `A` ) so it is transferred to the GPU and back again after the final iterations through the `data` region. But you only need to create ( `Anew` ) as it is just used for temporary storage on the GPU, so there is no need to ever transfer it back and forth.

[Return to step #3](#)

### Step #4 - Hint #1

You'll want a `gang()` and `vector()` clause on the inner loops, but you may want to let the compiler decide the dimensions of the outer loops. In that case, you can use a loop directive without any modifying clauses.

[Return to step #4](#)