Getting Started Guide

1. Introduction

This document describes information for GPU programming for the course. Please, read it carefully.

2. MinoTauro GPU cluster

2.1. Change your password

Use an SSH client program to login to **dt01.bsc.es** using the training account login information provided to you. **This machine is only used for password setup**, the machines you will have to connect during the hands on labs of the course are detailed below. Once logged in, follow the steps that appear in the terminal to change your default password for a new one of your choice (If no steps appear after you login, execute the command *passwd).* Note that there could be some restrictions to ensure that passwords have a minimum strength level (a minimum number of characters, use of numbers and/or punctuation symbols, etc.). Once you finish changing you password, you can logout from this machine and connect to the cluster login node.

Note: the password change may take 'a couple' of minutes to be effective.

2.2. Cluster usage

With your SSH client login to **mt1.bsc.es** using the provided username and your new password. If you just changed your password, it may take some time (up to 10 minutes) for the change to be effective, thus it is possible that you cannot login during this time.

In this section, we present a brief description of the main commands needed to work in the cluster. For further information about the MinoTauro cluster, take a look to the MinoTauro User's Guide. This guide contains detailed information about the MinoTauro Cluster, its job management system, etc.

The MinoTauro cluster uses a job management system to control access to the computing nodes. To run the labs, you will need to use the commands provided by the system to submit jobs for execution, check their state, or cancel them if necessary:

Submitting a job

To submit a job, use the command *sbatch <job_script>,* like in the following example:

```
$ sbatch ./job.sh
Submitted batch job 170798
```

After you execute the command, you shall see a message with the ID of the newly created job (170798 in the example above). The argument of *sbatch* (<job_script>) is the path of a shell script which contains a set of directives required by the job queue system, and the command to execute the program (see the example below). Some labs already provide a job.sh file, just note that to run the different programs in a lab, you will have to edit the last line of the script to execute the corresponding program. If a lab doesn't provide a job script, make a copy of a previous one and edit it accordingly for the current lab.

```
#!/bin/bash
#SBATCH --job-name=MyJob
#SBATCH --workdir=.
#SBATCH --output=%j.out
#SBATCH --error=%j.err
#SBATCH --time=00:05:00
#SBATCH --ntasks=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:1

<some command(s)>
```

Query job status

To query job status use the *squeue* command. You shall see all your submitted jobs that are waiting or running in the cluster.

A job has the PENDING state when is waiting in the queue:

```
$ squeue --long
JOBID  NAME  USER     STATE   TIME TIMELIMIT CPUS NODES NODELIST(REASON)
170798 MyJob nct00002 PENDING 0:00 10:00     6    1     (Priority)
```

A job has the RUNNING state when it is executing. When a job is in the RUNNING state, the TIME column shows the elapsed time since the job started executing, and the NODELIST (REASON) column indicates on which cluster's node/s the job is running.

```
$ squeue --long
JOBID  NAME  USER     STATE   TIME TIMELIMIT CPUS NODES NODELIST(REASON)
170798 MyJob nct00002 RUNNING 0:04 10:00     6    1     nvb118
```

When the job has finished and the job queue system has finished cleaning the job resources, the job won't appear anymore in the list:

```
$ squeue --long
JOBID NAME USER STATE TIME TIMELIMIT CPUS NODES NODELIST(REASON)
```

Canceling a job

To delete a job you can use the command *scancel <job ID>*

```
$ scancel 170798
```

When your job is finished, you can check the program output in the files "XXX.out" and "XXX.err", where "XXX" is the job ID number. Files with ".out" extension store the execution output log (stdout). Files with ".err" extension store the error or warning messages printed during the execution (stderr).

Note: Be aware that interruptions to your SSH connection may cause you to lose unsaved work. If you have reason to believe your connection to the cluster may be unstable, you may wish to use an FTP program with the SFTP protocol and your provided login information to retrieve the source files to your local machine for editing and upload them back to the cluster to compile and execute the programs.

Execution of GUI applications

In some of the labs we will use the NVIDIA Visual Profiler to profile the code and obtain a timeline of its execution. The best way to view the windows of a GUI app running in the cluster is to use the X forwarding feature of ssh. Setting up X forwarding to be able to use remote GUI apps depends on the OS installed in your laptop;

Linux

In this case you just have to add the -X flag when connecting to the cluster through ssh:

```
$ ssh -X <username>@mt1.bsc.es
```

If any error appears while trying to run a GUI app, you may try using the -Y flag instead of -X.

Windows

Windows doesn't handle X GUIs by default. There are several apps for Windows that enable this functionality, but many of them are not free. If you already have an X server emulation app installed in your laptop, you can use it for these labs. If

not, we did some tests with the free version of `MobaXterm` and seems to work well. You can download it from:

http:/mobaxterm.mobatek.net/

Mac OS

On Mac OS, the X windows system is called XQuartz. Mac OS X 10.5, 10.6 and 10.7 installed it by default, but as of 10.8 Apple has dropped support and directs users to the open source XQuartz. You can install XQuartz from the OS distribution media or download it from https://www.xquartz.org/.

When connecting to the cluster through ssh add the -Y flag (not -X) to enable X forwarding:

```
$ ssh -Y <username>@mt1.bsc.es
```

Setting up the labs

Obtain the labs source code

Use an SSH client program to login to **mt1.bsc.es** using the training account login information provided to you. Your home directory can be organized in any way you like. To unpack the labs framework including the code for all the lab assignments, execute the unpack command in the directory you would like the labs framework to be deployed:

```
$ tar -zxf ~nct00021/mv19_openacc.tgz
```

Check that a directory containing the labs has been created:

```
$ cd mv19_openacc
$ ls
```

The last command should list the different labs' directories and a few files.

Setup the environment

To be able to compile the labs, we first have to enable the PGI's OpenACC compiler. For this, execute the following command:

```
$ module load pgi
```

You will have to execute this command every time you login to the cluster.

OpenACC Course - Lab 1

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.

You can accelerate your applications using OpenACC directives with 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 4 Steps process will help you use OpenACC on your own codes more productively, and get better speed-ups in less time.

This lab is provided with C code. Some of the following labs will also provide Fortran versions.

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 coprocesor 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:

- Understanding the program structure and how data is passed through the call tree
- Profiling the CPU-only version of the application and identifying computationally-intense "hot spots"
  - Which loop nests dominate the runtime?
  - Are the loop nests suitable for an accelerator?
- Insuring that the algorithms you are considering for acceleration are *safely* parallel

What we've just said may sound a little scary, but please note that as parallel programming methods go, OpenACC is really friendly; because OpenACC directives are incremental, you can add one or two directives at a time and see how things work. Also, the compiler provides a lot of feedback that can help you during the acceleration process.

In this first lab, 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. 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);
}
```

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` 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 command to compile the code:

```
$ pgcc -fast -o jacobi_task1 task1/task1.c
```

Now submit the example job script provided with the labs (*job.sh*) using the command introduced in the Getting Started:

```
$ mnsubmit ../../job.sh
```

NOTE: In this first task the job script already contains the command that executes the program we just compiled. From now on, you will have to edit the script and modify the command it contains according to the program you want to run.

Once the job finishes, check the *.out file. It should contain the output of the program and look something like:

```
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: 4.434503 s
```

Results Correctness

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. 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 article at a later time, if you would like more details.

Profiling

Your objective in Step 2 will be to modify `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 NVIDIA's *nvprof*, PGI's *pgprof* or open-source *gprof*) to determine which functions are consuming the largest amounts of compute time. In the next labs will see how to use nvprof to do the initial profiling. For this one, the compute-intensive part of our code is the two for-loops nested inside the while loop in the function *main*.

Step 2 - Add Compute Directives

In C, an OpenACC directive is indicated in the code by '#pragma acc <*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 this step, 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
    // (note the plural, the block can contain more than one loop)
}
// 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. The kernels directive is designed to find the parallel acceleration opportunities implicit in the for-loops in the Jacobi Iteration code.

Add the directives in the file *task2/task2.c* and, once you finish, compile the code with the following command:

```
$ pgcc -acc -Minfo -o jacobi_task2 task2/task2.c
```

If you successfully added `#pragma acc kernels` in the proper spots, the compiler output messages should look similar to this:

```
main:
     36, Generating present_or_copyin(Anew[1:1022][1:1022])
         Generating copyin(A[:][:])
         Generating copyout(A[1:1022][1:1022])
         Generating NVIDIA code
     41, Loop is parallelizable
     43, Loop is parallelizable
         Accelerator kernel generated
         41, #pragma acc loop gang /* blockIdx.y */
         43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         47, Max reduction generated for error
     52, Loop is parallelizable
     54, Loop is parallelizable
         Accelerator kernel generated
         52, #pragma acc loop gang /* blockIdx.y */
         54, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         Memory copy idiom, loop replaced by call to __c_mcopy4
```

If you do not get a similar output, please check your work and try re-compiling. If you're stuck, you can compare what you have with the code in *task2/task2_solution.c*.

The output provided by the compiler is very 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 get all the output messages the compiler can provide. 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 important that you carefully inspect this information to make sure the compiler is doing what you've asked for.

Once you feel your code is correct, edit the job script to execute *jacobi_task2*, and submit it using *mnsubmit*. You'll want to review our quality check to make sure you didn't break the functionality of your application.

Step 3 - Manage Data Movement

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?!

The compiler feedback we collected earlier tells you quite a bit about data movement, and you can collect even more by setting an environment variable (`export PGI_ACC_TIME=1`) and then running the compiled code (We'll use this later in

Step 4). The reason for our slowdown in this step is *excessive data movement*: both regions spent the majority of their time *copying data*.

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 arrays *to the GPU and back to the CPU* for each accelerated section, *every time* it goes through the while loop. That is a LOT of wasted data transfers.

Ideally, we would just transfer `A` and `Anew` to the GPU at the beginning of the Jacobi Iteration, and then only transfer `A` back to the CPU at the end.

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.

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 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 1.0 or 2.0 specifications.

Edit *task3/task3.c* and add a `data` directive to minimize data transfers in the Jacobi Iteration. There's a place for the `create` clause in this exercise, too.

Hints:

- You should only have to worry about managing the transfer of data in arrays `A` and `Anew`.
- You want to put the data directive just above the outer *while* loop.
- You 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.

You can also look at *task3_solution.c* to see the answer if you get completely stuck or want to check your work.

Once you finish adding the data directives, compile the code and run it with the job system. After making these changes, our accelerator code should be 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.

Step 4 - Optimize Kernel Scheduling

The final step in our tuning process is to tune the OpenACC compute region schedule using the *gang* and *vector* clauses. These clauses let us 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 it with all your

data movement optimizations on the accelerator again, this time setting the environment variable PGI_ACC_TIME that we mentioned before.

Add the following command in the job script, before the command that was executing step 3's code:

```
export PGI_ACC_TIME=1
```

Submit the job script to run again the code from step 3, and when the job is done, examine the output files. The *.out file should contain the normal output of the program, and the *.err file should contain some timing information we haven't seen previously:

```
Accelerator Kernel Timing data
/home/gpudev1/notebook/task3/task3.c
  main  NVIDIA  devicenum=0
    time(us): 391,494
    34: data region reached 1 time
        34: data copyin reached 1 time
             device time(us): total=479 max=479 min=479 avg=479
        68: data copyout reached 1 time
             device time(us): total=519 max=519 min=519 avg=519
    37: compute region reached 1000 times
        44: kernel launched 1000 times
            grid: [8x1022]  block: [128]
             device time(us): total=248,508 max=268 min=242 avg=248
            elapsed time(us): total=258,569 max=453 min=255 avg=258
        44: reduction kernel launched 1000 times
            grid: [1]  block: [256]
             device time(us): total=27,531 max=83 min=25 avg=27
            elapsed time(us): total=37,589 max=93 min=35 avg=37
        55: kernel launched 1000 times
            grid: [8x1022]  block: [128]
             device time(us): total=114,457 max=170 min=110 avg=114
            elapsed time(us): total=124,880 max=303 min=121 avg=124
```

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 memcopy loop nest starting on line 34, which takes only a tiny fraction of the 0.39 seconds of 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: [8x1022]  block[128]
```

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 GPU *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 128 threads each. The grid the compiler has constructed is also 2D, 8 blocks wide and 1022 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.

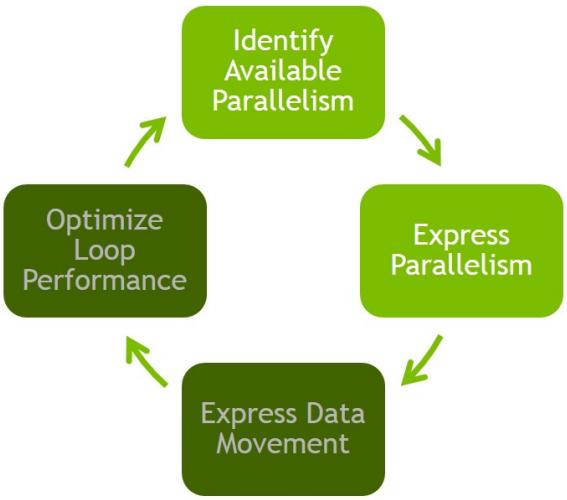
Note: You can let the compiler do the hard work of mapping loop nests, unless you are certain you can do it better (and portability is not a concern.) When you decide to intervene, think about different parallelization strategies (loop schedules): in nested loops, distributing the work of the outer loops to the GPU multiprocessors (on PGI = 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.

Try to modify the code for the main computational loop nests in the file *task4/task4.c*. 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. Look at task4_solution.c if you get stuck. When you're done, compile and execute the code to see the change in performance you obtained.

Looking at *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. You can check which value it used running the code with the env. var PGI_ACC_TIME set to 1.

OpenACC Course - Lab 2

In this lab you will profile the provided application using either NVIDIA nvprof or gprof and the PGI compiler. After profiling the application, you will use OpenACC to express the parallelism in the 3 most time-consuming routines. You will use CUDA Unified Memory and the PGI "managed" option to manage host and device memories for you. You may use either the `kernels` or `parallel loop` directives to express the parallelism in the code. Versions of the code have been provided in C99 and Fortran 90. The C99 version is available in the `c99` directory and the F90 version is available in the `f90` directory.



As discussed in the associated lecture, this lab will focus on Identifying Parallelism in the code by profiling the application and Expressing Parallelism using OpenACC. We will use CUDA Unified Memory to allow the data used on the GPU to be automatically migrated to and from the GPU as needed. Please be aware that you may see an application slowdown until you have completed each step of this lab. This is expected behavior due to the need to migrate data between the CPU and GPU memories.

Hint: You should repeat steps 2 and 3 for each function identified in step 1 in order of function importance. Gather a new GPU profile each time and observe how the profile changes after each step.

Step 0 - Building the code

Makefiles have been provided for building both the C and Fortran versions of the code. Change directory to your language of choice and run the `make` command to build the code.

C/C++

```
$ cd c99/
$ make
```

Fortran

```
$ cd f90/
$ make
```

This will build an executable named `cg` that you can run with the `./cg` command. You may change the options passed to the compiler by modifying the `CFLAGS` variable in `c99/Makefile` or `FCFLAGS` in `f90/Makefile`. You should not need to modify anything in the Makefile except these compiler flags.

Step 1 - Identify Parallelism

In this step, use the NVPROF profiler, or your preferred performance analysis tool, to identify the important routines in the application and examine the loops within these routines to determine whether they are candidates for acceleration. Use the command below in the job script to gather a CPU profile.

```
nvprof --cpu-profiling on --cpu-profiling-mode top-down ./cg
```

Once the job is done, the *.out file should have the cg program output:

```
Rows: 8120601, nnz: 218535025
Iteration: 0, Tolerance: 4.0067e+08
Iteration: 10, Tolerance: 1.8772e+07
Iteration: 20, Tolerance: 6.4359e+05
Iteration: 30, Tolerance: 2.3202e+04
Iteration: 40, Tolerance: 8.3565e+02
Iteration: 50, Tolerance: 3.0039e+01
Iteration: 60, Tolerance: 1.0764e+00
Iteration: 70, Tolerance: 3.8360e-02
Iteration: 80, Tolerance: 1.3515e-03
Iteration: 90, Tolerance: 4.6209e-05
Total Iterations: 100 Time: 28.534824s
```

The *.err file should have the profiling information, which should be something similar to this:

```
======== CPU profiling result (top down):
99.87% main
| 81.12% matvec(matrix const &, vector const &, vector const &)
| 11.53% waxpby(double, vector const &, double, vector const &, vector const &)
| 4.55% dot(vector const &, vector const &)
| 2.65% allocate_3d_poisson_matrix(matrix&, int)
| 0.03% free_matrix(matrix&)
|   0.03% munmap
0.13% __c_mset8

======== Data collected at 100Hz frequency
```

We see from the above output that the `matvec`, `waxpy`, and `dot` routines take up the majority of the runtime of this application. We will focus our effort on accelerating these functions.

NOTE: The `allocate_3d_poisson_matrix` routine is an initialization routine that can be safely ignored.

Step 2 - Express Parallelism

Within each of the routines identified above, express the available parallelism to the compiler using either the `acc kernels` or `acc parallel loop` directive. As an example, here's the OpenACC code to add to the `matvec` routine.

```
void matvec(const matrix& A, const vector& x, const vector &y) {

  unsigned int num_rows=A.num_rows;
  unsigned int *restrict row_offsets=A.row_offsets;
  unsigned int *restrict cols=A.cols;
  double *restrict Acoefs=A.coefs;
  double *restrict xcoefs=x.coefs;
  double *restrict ycoefs=y.coefs;

#pragma acc kernels
  {
    for(int i=0;i<num_rows;i++) {
      double sum=0;
      int row_start=row_offsets[i];
      int row_end=row_offsets[i+1];
      for(int j=row_start;j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
      }
      ycoefs[i]=sum;
    }
  }
}
```

Add the necessary directives to each routine one at a time in order of importance. After adding the directive, recompile the code, check that the output has remained the same, and note the performance difference from your change.

Before compiling, edit the Makefile and add the required flags to specify the target architecture configuration to generate GPU code using managed memory. Also add the flag that tells the compiler to output information about the parallelization process.

```
$ make
pgc++ -fast -acc -ta=tesla:managed -Minfo=accel main.cpp -o cg
"vector.h", line 16: warning: variable "vcoefs" was declared but never
          referenced
    double *vcoefs=v.coefs;
```

```
            ^

matvec(const matrix &, const vector &, const vector &):
      8, include "matrix_functions.h"
          15, Generating copyout(ycoefs[:num_rows])
              Generating
copyin(xcoefs[:],Acoefs[:],cols[:],row_offsets[:num_rows+1])
          16, Loop is parallelizable
              Accelerator kernel generated
              Generating Tesla code
              16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
*/
          20, Loop is parallelizable
```

The performance may slow down as you're working on this step. Be sure to read the compiler feedback to understand how the compiler parallelizes the code for you. If you are doing the C/C++ lab, it may be necessary to declare some pointers as `restrict` in order for the compiler to parallelize them. You will know if this is necessary if the compiler feedback lists a "complex loop carried dependency."

Step 3 - Re-Profile Application

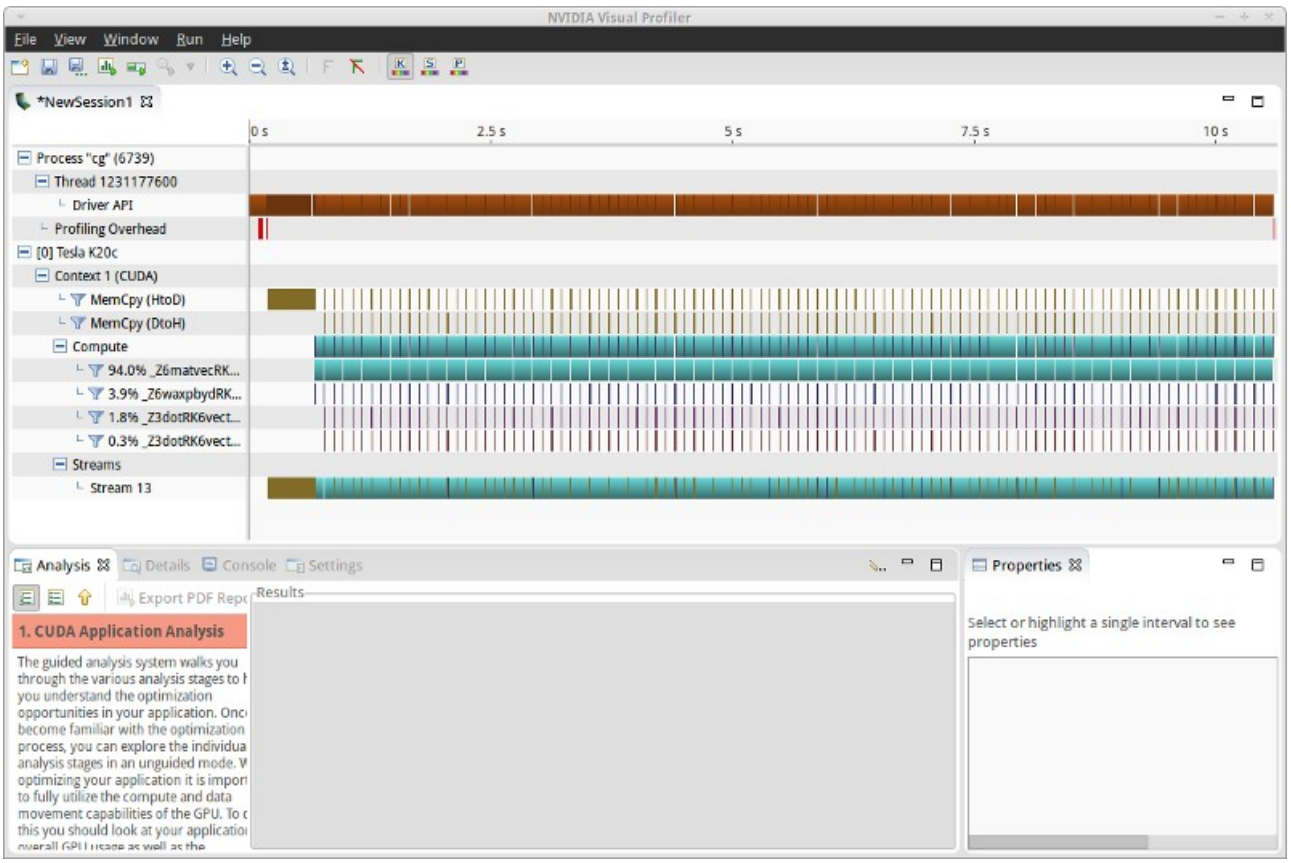
Once you have added the OpenACC directives to your code, you should obtain a new profile of the application. For this step, use the NVIDIA Visual Profiler to obtain a GPU timeline and see how the the GPU computation and data movement from CUDA Unified Memory interact. Before continuing, read the Getting Started section on how to use GUI apps in cluster, if you didn't read it yet.

To run the NVIDIA Visual Profiler in the cluster use the command:

```
$ ../../launch_nvvp
```

This will launch the profiler in one of the computating nodes of the cluster for a maximum of 20 minutes. We recommend you to close the profiler as soon as you finish to check the timeline to not occupy the computing nodes needlessly and allow other users to proceed with their jobs.

Once Visual Profiler has started, create a new session by selecting *File -> New Session*. Then select the executable that you built by pressing the *Browse* button next to *File*, browse to your working directory, select the `cg` executable, and then press *Next*. On the next screen ensure that *Enable unified memory profiling* is checked and press *Finish*. The result should look like the image below. Experiment with Visual Profiler to see what information you can learn from it.



Conclusion

After completing the above steps for each of the 3 important routines your application should show a speed-up over the unaccelerated version. You can verify this by removing the `-ta` flag from your compiler options. In the next lecture and lab we will replace CUDA Unified Memory with explicit memory management using OpenACC and then further optimize the loops using the OpenACC loop directive.

Bonus

If you used the kernels directive to express the parallelism in the code, try again with the parallel loop directive. Remember, you will need to take responsibility of identifying any reductions in the code. If you used parallel loop, try using kernels instead and observe the differences both in developer effort and performance.

OpenACC Course - Lab 3

In this lab you will build upon the work from lab 2 to add explicit data management directives, eliminating the need to use CUDA Unified Memory, and optimize the `matvec` kernel using the OpenACC `loop` directive. If you have not already completed lab 2, please go back and complete it before starting this lab.

Step 0 - Building the code

Makefiles have been provided for building both the C and Fortran versions of the code. Change directory to your language of choice and run the `make` command to build the code.

C/C++

```
$ cd c99
$ make
```

Fortran

```
$ cd f90
$ make
```

This will build an executable named `cg` that you can run with the `./cg` command. You may change the options passed to the compiler by modifying the `CFLAGS` variable in `c99/Makefile` or `FCFLAGS` in `f90/Makefile`. You should not need to modify anything in the Makefile except these compiler flags.

Step 1 - Express Data Movement

In the previous lab we used CUDA Unified Memory, which we enabled with the `ta=tesla:managed` compiler option, to eliminate the need for data management directives. Replace this compiler flag in the Makefile with `-ta=tesla` and try to rebuild the code.

C/C++

With the managed memory option removed the C/C++ version will fail to build because the compiler will not be able to determine the sizes of some of the arrays used in compute regions. You will see an error like the one below:

```
PGCC-S-0155-Compiler failed to translate accelerator region (see -Minfo
messages): Could not find allocated-variable index for symbol (main.cpp: 15)
```

Fortran

The Fortran version of the code will build successfully and run, however the tolerance value will be incorrect with the managed memory option removed.

```
$ ./cg
 Rows:      8120601 nnz:    218535025
 Iteration:  0 Tolerance: 4.006700E+08
 Iteration: 10 Tolerance: 4.006700E+08
 Iteration: 20 Tolerance: 4.006700E+08
 Iteration: 30 Tolerance: 4.006700E+08
 Iteration: 40 Tolerance: 4.006700E+08
 Iteration: 50 Tolerance: 4.006700E+08
 Iteration: 60 Tolerance: 4.006700E+08
 Iteration: 70 Tolerance: 4.006700E+08
 Iteration: 80 Tolerance: 4.006700E+08
 Iteration: 90 Tolerance: 4.006700E+08
  Total Iterations:          100
```

We can correct both of these problems by explicitly declaring the data movement for the arrays that we need on the GPU. In the associated lecture we discussed the OpenACC structured `data` directive and the unstructured `enter data` and `exit data` directives. Either approach can be used to express the data locality in this code, but the unstructured directives are probably cleaner to use.

C/C++

In the `allocate_3d_poisson_matrix` function in matrix.h, add the following two directives to the end of the function.

```
#pragma acc enter data copyin(A)
#pragma acc enter data
copyin(A.row_offsets[:num_rows+1],A.cols[:nnz],A.coefs[:nnz])
```

The first directive copies the A structure to the GPU, which includes the `num_rows` member and the pointers for the three member arrays. The second directive then copies the three arrays to the device. Now that we've created space on the GPU for these arrays, it's necessary to clean up the space when we're done. In the `free_matrix` function, add the following directives immediately before the calls to `free`.

```
#pragma acc exit data delete(A.row_offsets,A.cols,A.coefs)
#pragma acc exit data delete(A)
  free(row_offsets);
  free(cols);
  free(coefs);
```

Notice that we are performing the operations in the reverse order. First we are deleting the 3 member arrays from the device, then we are deleting the structure containing those arrays. It's also critical that we place our pragmas *before* the arrays are freed on the host, otherwise the `exit data` directives will fail.

Now go into `vector.h` and do the same thing in `allocate_vector` and `free_vector` with the structure `v` and its member array `v.coefs`. Because we are copying the arrays to the device before they have been populated with data, use the `create` data clause, rather than `copyin`.

If you try to build again at this point, the code will still fail to build because we haven't told our compute regions that the data is already present on the device, so the compiler is still trying to determine the array sizes itself. Now go to the compute regions (`kernels` or `parallel loop`) in `matrix_functions.h` and `vector_functions.h` and use the `present` clause to inform the compiler that the arrays are already on the device. Below is an example for `matvec`.

```
#pragma acc kernels present(row_offsets,cols,Acoefs,xcoefs,ycoefs)
```

Once you have added the `present` clause to all three compute regions, the application should now build and run on the GPU, but is no longer getting correct results. This is because we've put the arrays on the device, but we've failed to copy the input data into these arrays. Add the following directive to the end of `initialize_vector` function;

```
#pragma acc update device(v.coefs[:v.n])
```

This will copy the data now in the host array to the GPU copy of the array. With this data now correctly copied to the GPU, the code should run to completion and give the same results as before.

Fortran

To make the application return correct answers again, it will be necessary to add explicit data management directives. This could be done using either the structured `data` directives or unstructured `enter data` and `exit data` directives, as discussed in the lecture. Since this program has clear routines for allocating and initializing the data structures and also deallocating, we'll use the unstructured directives to make the code easy to understand.

The `allocate_3d_poisson_matrix` in matrix.F90 handles allocating and initializing the primary array. At the end of this routine, add the following directive for copying the three arrays in the matrix type to the device;

```
!$acc enter data copyin(arow_offsets,acols,acoefs)
```

These three arrays can be copied in separate `enter data` directives as well. Notice that because Fortran arrays are self-describing, it's unnecessary to provide the array bounds, although it would be safe to do so as well. Since we've allocated these arrays on the device, they should be removed from the device when we are done with them as well. In the `free_matrix` subroutine of matrix.F90 add the following directive;

```
!$acc exit data delete(arow_offsets,acols,acoefs)
deallocate(arow_offsets)
deallocate(acols)
deallocate(acoefs)
```

Notice that the `exit data` directive appears before the `deallocate` statement. Because the OpenACC programming model assumes we always begin and end execution on the host, it's necessary to remove arrays from the device before freeing them on the host to avoid an error or crash. Now go add `enter data` and `exit data` directives to vector.F90 as well. Notice that the `allocate_vector` routine only allocates the array, but does not initialize it, so `copyin` may be replaced with `create` on the `enter data` directive.

If we build and run the application at this point we should see our tolerance changing once again, but the answers will still be incorrect. Next let go to each compute directive (`kernels` or `parallel loop`) in matrix.F90 and vector.F90 and inform the compiler that the arrays used in those regions are already present on the device. Below is an example from matrix.F90.

```
!$acc kernels present(arow_offsets,acols,acoefs,x,y)
```

At this point the compiler knows that it does not need to be concerned with data movement in our compute regions, but we're still getting the wrong answer. The last change we need to make is to make sure that we're copying the input data to the device before execution. In vector.F90 add the following directive to the end of `initialize_vector`.

```
vector(:) = value
!$acc update device(vector)
```

Now that we have the correct input data on the device the code should run correctly once again.

NOTE for C/C++ and Fortran: One could also parallelize the loop in `initialize_vector` on the GPU, but we choose to use the `update` directive here to illustrate how this directive is used.

Step 2 - Optimize Loops - Vector Length

Now that we're running on the GPU and getting correct answers , let's apply our knowledge of the code to help the compiler make better decisions about how to parallelize our loops. We know from the `allocate_3d_poisson_matrix` routine that the most non-zero elements we'll have per row is 27. By examining the compiler output, as shown below, we know that the compiler chose a vector length of 128 for the `matvec` loops. This means that with the compiler-selected vector length of 128, 101 vector lanes (threads) will go unused. Let's tell the compiler to choose a better vector length for these loops.

```
matvec(const matrix &, const vector &, const vector &):
      8, include "matrix_functions.h"
          15, Generating
present(row_offsets[:],cols[:],Acoefs[:],xcoefs[:],ycoefs[:])
          16, Loop is parallelizable
              Accelerator kernel generated
              Generating Tesla code
              16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
*/
          20, Loop is parallelizable
```

On an NVIDIA GPU the vector length must be a multiple of the *warp size* of the GPU, which on all NVIDIA GPUs to-date is 32. This means that the closest vector length we can choose is 32. Depending on whether the code uses `kernels` or `parallel loop`, we can specify the vector length one of two ways.

Kernels

When using the `kernels` directive, the vector length is given by adding `vector(32)` to the loop we want to use as the `vector` loop. So for our `matvec` loops, we'd apply the vector length as shown below.

C/C++

```
#pragma acc kernels present(row_offsets,cols,Acoefs,xcoefs,ycoefs)
  {
    for(int i=0;i<num_rows;i++) {
      double sum=0;
```

```
      int row_start=row_offsets[i];
      int row_end=row_offsets[i+1];
      #pragma acc loop device_type(nvidia) vector(32)
      for(int j=row_start;j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
      }
      ycoefs[i]=sum;
    }
  }
```

Fortran

```
!$acc kernels present(arow_offsets,acols,acoefs,x,y)
do i=1,a%num_rows
  tmpsum = 0.0d0
  row_start = arow_offsets(i)
  row_end   = arow_offsets(i+1)-1
  !$acc loop device_type(nvidia) vector(32)
  do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum + acoef*xcoef
  enddo
  y(i) = tmpsum
enddo
!$acc end kernels
```

Parallel Loop

When using `parallel loop` the vector length is given at the top of the region, as shown below.

C/C++

```
#pragma acc parallel loop present(row_offsets,cols,Acoefs,xcoefs,ycoefs) \
        device_type(nvidia) vector_length(32)
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum) device_type(nvidia) vector
    for(int j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```

Fortran

```
!$acc parallel loop private(tmpsum,row_start,row_end) &
!$acc& present(arow_offsets,acols,acoefs,x,y)        &
!$acc& device_type(nvidia) vector_length(32)
```

```
do i=1,a%num_rows
  tmpsum = 0.0d0
  row_start = arow_offsets(i)
  row_end   = arow_offsets(i+1)-1
  !$acc loop reduction(+:tmpsum) device_type(nvidia) vector
  do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum + acoef*xcoef
  enddo
  y(i) = tmpsum
enddo
```

Notice that the above code adds the `device_type(nvidia)` clause to the affected loops. Because we only want this optimization to be applied to NVIDIA GPUs, we've protected that optimization with a `device_type` clause and allowed the compiler to determine the best value on other platforms. Now that we've adjusted the vector length to fit the problem, let's profile the code again to see how well it's performing. Using Visual Profiler, let's see if we can find a way to further improve performance.

The folders `intermediate.kernels` and `intermediate.parallel` contain the correct code for the end of this step. If you have any trouble, use the code in one of these folders to help yourself along.

Step 3 - Optimize Loops - Profile The Application

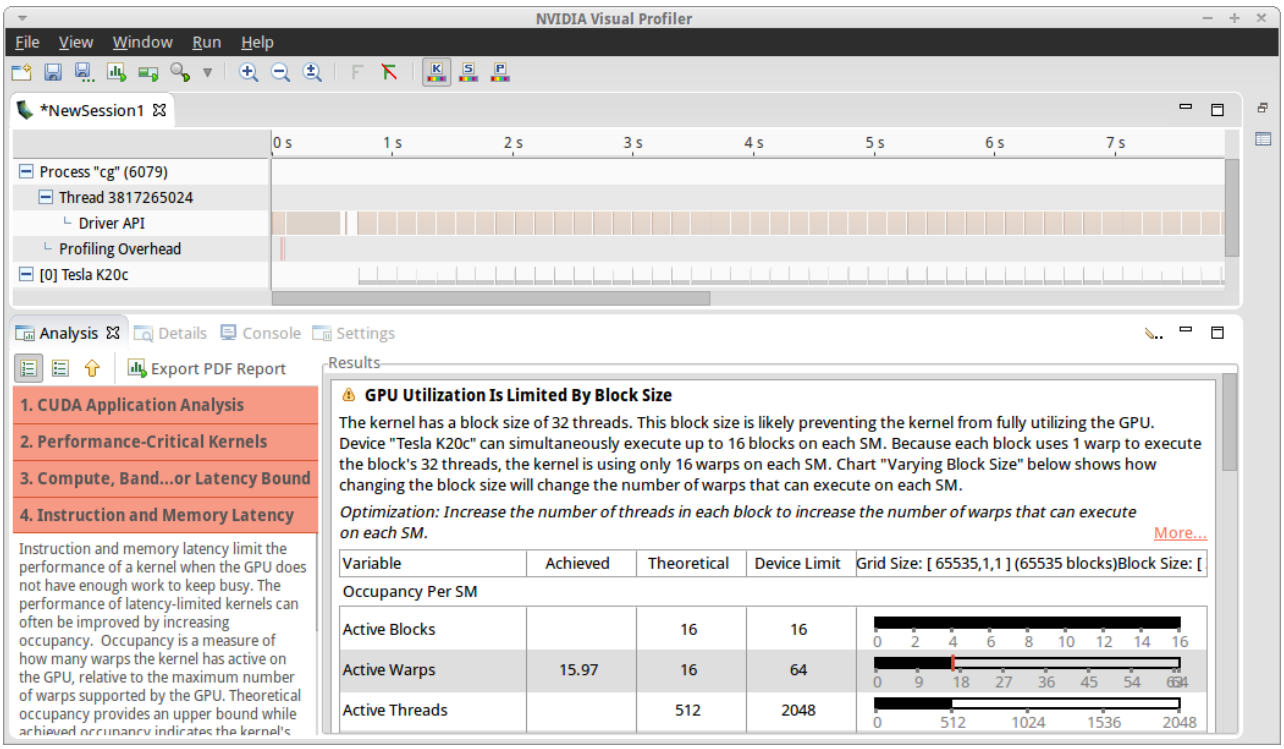
Just as in the last lab, we'll use the NVIDIA Visual Profiler to profile our application. Start it with the command:

```
$ ../../launch_nvvp
```

Once Visual Profiler has started, create a new session by selecting *File -> New Session*. Then select the executable that you built by pressing the *Browse* button next to *File*, browse to your working directory, select the `cg` executable, and then press *Next*. On the next screen press *Finish*. Visual Profiler will run for several seconds to collect a GPU timeline and begin its *guided analysis*.

In the lower left, press the "Examine GPU Usage" button. You may need to enlarge the bottom panel of the screen by grabbing just below the horizontal scroll bar at the middle of the window and dragging it up until the button is visible. After this runs, click on "Examine Individual Kernels" and select the top kernel in the table. After selecting the top kernel, press the "Perform Kernel Analysis" button to gather further performance information about this kernel and wait while Visual Profiler collects additional data ***(this make take several minutes)***. When this

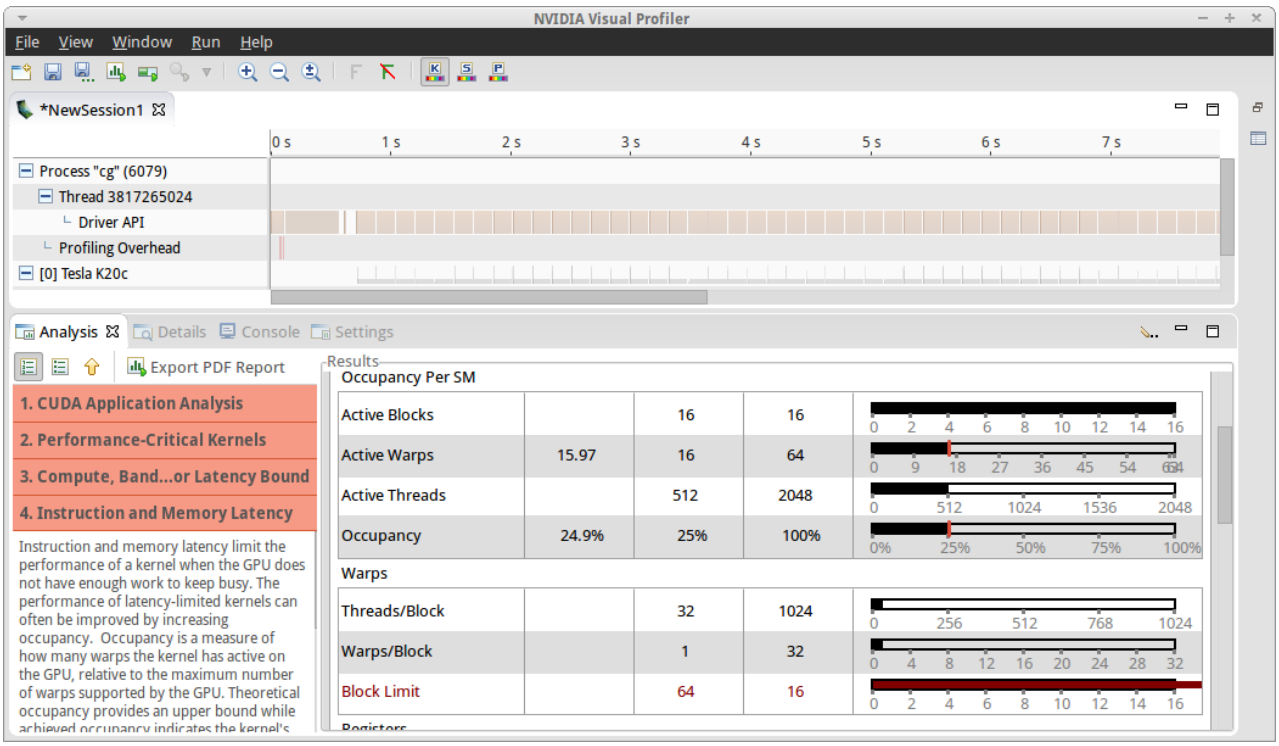
completes, press "Perform Latency Analysis". The screenshot below shows Visual Profiler at this step.



Visual Profiler is telling us that the performance of the matvec kernel is limited by the amount of parallelism in each gang (referred to as *"block size"* in CUDA). Scrolling down in the *Results* section I see that the *Occupancy* is 25%. Occupancy is a measure of how much parallelism is running on the GPU versus how much theoretically could be running. 25% occupancy indicates that resources are sitting idle due to the size of the blocks (OpenACC gangs).

NOTE: 100% occupancy is not necessary for high performance, but occupancy below 50% is frequently an indicator that optimization is possible.

Scrolling further down in the *Results* section we reach the *Block Limit* metric, which will be highlighted in red. This is shown in the screenshot below.



This table is showing us that the GPU *streaming multiprocessor (SM)* can theoretically run 64 *warps* (groups of 32 threads), but only has 16 to run. Looking at the *Warps/Block* and *Threads/Block* rows of the table, we see that each block contains 1 warp, or 32 threads, although it could run many more. This is because we've told the compiler to use a vector length of 32. As a reminder, in OpenACC many *gangs* run independently of each other, each gang has 1 or more *workers*, each of which operates on a *vector*. With a vector length of 32, we'll need to add workers in order to increase the work per gang. Now we need to inform the compiler to give each gang more work by using *worker* parallelism.

Step 4 - Optimize Loops - Increase Parallelism

To increase the parallelism in each OpenACC gang, we'll use the worker level of parallelism to operate on multiple vectors within each gang. On an NVIDIA GPU the *vector length X number of workers* must be a multiple of 32 and no larger than 1024, so let's experiment with increasing the number of workers. From just 1 worker up to 32. We want the outermost loop to be divided among gangs and workers, so we'll specify that it is an gang *and* worker loop. By only specifying the number of workers, we allow the compiler to generate enough gangs to use up the rest of the loop iterations applying worker parallelism.

Kernels

When using the `kernels` directive, use the `loop` directive to specify that the outer loop should be a *gang* and *worker* loop with 32 workers as shown below. Experiment with the number of workers to find the best value.

C/C++

```
#pragma acc kernels present(row_offsets,cols,Acoefs,xcoefs,ycoefs)
{
#pragma acc loop device_type(nvidia) gang worker(32)
    for(int i=0;i<num_rows;i++) {
      double sum=0;
      int row_start=row_offsets[i];
      int row_end=row_offsets[i+1];
      #pragma acc loop device_type(nvidia) vector(32)
      for(int j=row_start;j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
      }
      ycoefs[i]=sum;
    }
}
```

Fortran

```
!$acc kernels present(arow_offsets,acols,acoefs,x,y)
!$acc loop device_type(nvidia) gang worker(32)
do i=1,a%num_rows
  tmpsum = 0.0d0
  row_start = arow_offsets(i)
  row_end   = arow_offsets(i+1)-1
  !$acc loop device_type(nvidia) vector(32)
  do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum + acoef*xcoef
  enddo
  y(i) = tmpsum
enddo
!$acc end kernels
```

Parallel Loop

When using the `parallel loop` directive, use `gang` and `worker` to specify that the outer loop should be a *gang* and *worker* loop and then add `num_workers(32)` to specify 32 workers, as shown below. Experiment with the number of workers to find the best value.

C/C++

```
#pragma acc parallel loop present(row_offsets,cols,Acoefs,xcoefs,ycoefs) \
        device_type(nvidia) gang worker vector_length(32) num_workers(32)
```

```
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum) device_type(nvidia) vector
    for(int j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```

Fortran

```
!$acc parallel loop private(tmpsum,row_start,row_end) &
!$acc& present(arow_offsets,acols,acoefs,x,y)         &
!$acc& device_type(nvidia) gang worker num_workers(32) vector_length(32)
do i=1,a%num_rows
  tmpsum = 0.0d0
  row_start = arow_offsets(i)
  row_end   = arow_offsets(i+1)-1
  !$acc loop reduction(+:tmpsum) device_type(nvidia) vector
  do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum + acoef*xcoef
  enddo
  y(i) = tmpsum
enddo
```

After experimenting with the number of workers, performance should be similar to the table below.

Workers	K40
1	
2	61.03544
4	31.36616
8	16.71916
16	8.81069
32	6.488389

Conclusion

In this lab we started with a code that relied on CUDA Unified Memory to handle data movement and added explicit OpenACC data locality directives. This makes the code portable to any OpenACC compiler and accelerators that may not

have Unified Memory. We used both the unstructured data directives and the `update` directive to achieve this.

Next we profiled the code to determine how it could run more efficiently on the GPU we're using. We used our knowledge of both the application and the hardware to find a loop mapping that ran well on the GPU, achieving a 2-4X speed-up over our starting code.

The table below shows runtime for each step of this lab on an NVIDIA Tesla K40 GPU.

Step	K40
Unified Memory	8.458172
Explicit Memory	8.459754
Vector Length 32	11.656281
Final Code	4.802727

OpenACC Course - Lab 4

During this lab you will implement the advanced OpenACC techniques explained in the last lecture;

- A pipelined GPU version of a Mandelbrot set generation app as an example of asynchronous programming
- A multi-GPU Jacobi solver combining OpenACC and MPI

Pipelined Mandelbrot set

Since this code is the one presented during the lecture, you should already be familiar with the process to implement the pipelined GPU version. To recap, here are the steps needed to incrementally improve the Mandelbrot code to reach our goal:

1. Implement the GPU version using the pragmas we've been using in all the labs. You may need the `routine` pragma, too.
2. Break the image creation into several kernels, where each generates a part of the final image.
3. Improve the handling of data by removing unecessary HtoD copies and telling OpenACC to start DtoH copies as soon as each kernel finishes (check the `update` directive).
4. Use several asynchronous queues to enable the parallel execution of unrelated copies and kernel executions. Remember to wait for the async work to finish. If the queue creation time is noticeable, try reusing a few queues instead of creating a new one for each block.

You can experiment with the number of blocks you divide the image to see which one gives the best time.

Multi-GPU Jacobi with OpenACC + MPI

Before starting, we need extra steps to setup the environment for this lab to work in the Minotauro cluster. Run the following commands:

```
$ module unload pgi
$ module unload bullxmpi
$ module load pgi/16.9
$ module load openmpi/1.10.2_cuda_pgi
```

For the C version, also run:

```
$ export OMPI_CC=pgcc
```

You will have to run these commands every time you login to the cluster to work in this lab.

Notice that this lab also includes a new job script (*job_mpi.sh*) that must be used to run this lab's code in the cluster.

The lab is divided in three tasks:

1. Add MPI boiler plate code
2. Distribute work across the GPUs
3. Overlap communication and computation to improve multi-GPU scalability

Complete the missing parts of the code and execute the code to see the improvement you get after each task. If you want to make sure your code is correct or you get stuck, check the solution provided with each task.