OpenACC Lab Exercise: Accelerating Laplace Code

1. Get Lab exercises from gitlab: `git clone https://git.psu.edu/sms5713/GIG.git`
2. Go to Laplace directory: `cd GIG/laplace`
3. Apply changes from .bashrc to shell environment: `source .bashrc`
   - ➔ This will load the proper modules needed for this exercise.

Preliminary Test: OpenMP scalability of Jacobi

Description: In this exercise, we will accelerate an application that solves the Laplace equation in 2D using the iterative Jacobi method. The execution speed obtained using OpenMP and OpenACC will be compared.

OpenMP execution: Set the environment variable OMP_NUM_THREADS to 1, 2, 4, 8, 16, and 24 to see how performance varies with thread count.

1. `cd lab1`
2. Verify you have proper modules loaded (pgi/15.4, gnutools/2.69, cuda/7.0) by typing `module list`
3. Depending on your preference (C/Fortran) compile the code with either Makefile or Makefile_f90

   Compile Laplace in C: `make`

   Compile Laplace in Fortran: `make -f Makefile_f90`

Note: Pay special attention to compiler output when compiling OpenMP code. The compiler will automatically vectorize some loop-nest and show details on scheduling policy (static/dynamic) as well as regions of parallelization by line number.

4. Modify your jobscript by adding the C of Fortran OpenMP executable (laplace2d_omp, laplace2d_f90_omp)
5. Set OMP_NUM_THREAD variable to 1, `export OMP_NUM_THREAD=1` in your job script
6. Submit your OpenMP job-script to scheduler: `sbatch openmp.sh`
7. Rerun your experiment with different thread counts try: 2, 4, 8, 16, 24

Part 1: Apply the OpenACC kernels directive to the Jacobi loop

Description: In this exercise, we will port the main computation loop done by CPU to GPU using the kernels directive. We expect the computation to be done entirely on GPU and give the answer back to CPU.

To DO:

1. Add kernels directive to the main computation loop for laplace calculation

2. Run your GPU code either laplace2d_acc or laplace2d_f90_acc by modifying the acc.sh script to run the appropriate executable
3. Submit your job: `sbatch acc.sh`
4. Compare the results from your OpenACC implementation versus the OpenMP version.

Q&A:

Do you see a speedup when porting the CPU code to GPU?
What part of the code is most compute intensive?
Is this program compute or memory bound in your implementation? (Hint: Profile your code to see metrics for your code. You can use **nvprof** or set **PGI_ACC_TIME=1** to see more information)

Part 2: Data Management

Description: Control data movement by using the data directive. The CPU will need to communicate with GPU by transferring the necessary data to GPU used by the kernel directive. Try to minimize data movement as much as possible, figure out where to put data directive and what variables need to be passed in.

Don't worry about scalar variables, the compiler will automatically handle those during compile time. Focus on the matrix that needs to be present on GPU. Once you finish implementation, run your code and see the results.

Pay special attention to compiler output for copy transfer counts and their total runtime. PGI reports runtime all metrics in microseconds (us) therefore, to make any sense divide by 1,000,000 to convert time in second.

To DO:

1. Apply Data Directive
2. Initialize GPU environment with acc_init() by applying the function prior to execution.

Part 3: Kernel Tuning via gang, vector values

Description: We will tune the two kernel regions by specifying our own gang vector values for the two nested loops. In previous exercise, the compiler added its own values for gang vector values which are determined by compiler based on many factors pertaining to the application.

The gang and vector translate to grid and block in PGI terms. A grid and block (or threadblock) can be 2 dimension in CUDA it's known as blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y. This means, you can apply a gang and vector clause twice in your nested loops for any compute kernel. Apply the gang vector clause in the inner and outer loop and assign some values.

Vector values should be multiple of 32 given that hardware can support 32 vector instructions per cycle also known as warp size. Vector values not multiple of 32 will waste resource and it will lead to performance loss. The Kepler GPU have a restriction of 1024 threads per thread block which means that vector size must not exceed 1024.

In the 2 nested loops, the vector values Y * Z <= 1024

```
#pragma acc kernels loop gang(X1), vector(Y)
for( int j; …)
   #pragma acc loop gang(X2), vector(Z)
   for( int i;… )
```

Experiment with Y & Z incrementally by changing the values for vector clause and see how performance changes. Initially set your gang values X1 and X2 to 1 which would be 1 grid block and tune your kernel by multiple experiments.

Tune your gang values X1 and X2 to see if you can improve your code from previous exercise. Good values for gang are 128, 256, and 512 but they vary by application to application.

RESULTS

Table 1: Laplace2d OpenMP Test - C Implementation

Threads	Test 1	Test 2	Average
1	25.722	25.813	25.767
2	12.320	13.639	12.979
4	8.063	8.318	8.191
8	5.325	6.221	5.773
16	5.671	6.001	5.836
24	5.869	5.463	5.666

Table 2: Laplace2d OpenMP Test - Fortran Implementation

Threads	Test 1	Test 2	Average
1	25.352	25.336	25.344
2	12.263	12.138	12.2005
4	6.701	6.538	6.6195
8	4.889	5.451	5.17
16	5.531	5.349	5.44
24	6.069	5.681	5.875

Table 3: Laplace2d OpenACC Implementation - Lab1

	Test 1	Test 2	Average
C	74.325	84.381	79.353
Fortran	83.462	83.967	83.7145

Table 4: Laplace2d OpenACC implementation - Lab 2

	Test 1	Test 2	Average
C	4.659	4.538	4.599
Fortran	4.690	4.755	4.723

Table 5: Laplace2d OpenACC Implementation - Lab 3

	Test 1	Test 2	Average
C	3.481	3.474	3.477
Fortran	3.471	3.423	3.447