OpenACC Lab Exercise: Nested Parallelism with MatVec

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

Preliminary Test: In this exercise, we will see the experimental performance of the MatVec operation on the CPU.

1. `cd lab1` Go to Lab1 directory

There are a total of four source files:

nestedparallel_cpu.f90: CPU version of MatVec in Fortran
nestedparallel_cpu.c: CPU version of MatVec in C
nestedparallel_acc.f90: Your OpenACC implementation in Fortran
nestedparallel_acc.c: Your OpenACC implementation in C

Depending on your preference, choose one of the languages for this lab exercise.

2. Verify you have proper modules loaded (pgi/15.4, gnutools/2.69, cuda/7.0) by typing `module list`
3. Compile the source code by running `make`
   There will be four executable generated, pay special attention to compiler output for compiling the source code. The compiler will generate vector code for loop nest that can be parallelized with the appropriate line number. Get familiar with the compile option to see what flags were passed for generating the executable. For more info check the man pages, **man pgcc** or **man pgfortran**
4. Add the C or Fortran CPU binary (**nestedpar_cpu_C** or **nestedpar_cpu_F**) in your jobscript **cpu.sh**
5. Submit your job to scheduler: `sbatch cpu.sh`

Part 1: Port matvec function to GPU

Description: This exercise will test your ability to run the matvec function on GPU. There are several steps that need to be taken to port the function to the GPU properly. One must be aware of the data transfer when calling the matvec function to ensure GPU has the proper array for computing the Matrix Vector multiplication.

To DO:

1. Add **acc_init**(…) in your code to initialize GPU at start of application, make sure you apply **#ifdef _OPENACC … #endif** when invoking acc_init
2. Add necessary data clause to transfer array to GPU at the loop nest that calls matvec
3. Add the parallel compute construct in the loop nest that calls matvec
4. Apply the routine directive at the function declaration of matvec

5. Apply loop directive for the j loop in the matvec operation

1. Compile the code by running `make`
2. Add the binary **nestedpar_acc_C** or **nestedpar_acc_F** to the jobscript **acc.sh**
3. Submit job to job scheduler: `sbatch acc.sh`

The matvec routine is not a compute intensive task, compare the flop rate for CPU and OpenACC implementation and see which version outperforms. Try profiling your code with **nvprof** or set **PGI_ACC_TIME=1** to get timing metrics.

Part 2: Tune Parallel construct with gang vector values

Description: The Parallel compute construct can be tuned with gang vector values to further improve performance. The gang vector values associate to grid and block size for the compute construct which entails how loop nest will be translated into CUDA in terms of threadIDx.[x|y] and blockIDx.[x|y].

To DO:

1. Add **num_gangs** and **vector_length** clause to parallel construct with some values. Play around with both parameters to see if you can get better performance.
   Hint: Keep the vector values between 128-256 in multiple of 32 and try different values for gang from 32, 64, 128,…,512
2. Apply vector clause in the matvec loop
3. Apply reduction clause for sum
4. Apply vector clause on routine directive

   Try to run your code in vector only mode (set num_gangs(1)) to see how matvec operation conforms to a single grid. You can use the gang vector values from the previous exercise from compiler configuration as your starting point and then change accordingly.

Part 3: Tune Application for performance

Description: In this section, we will focus on accelerating the application even further by running the entire code on GPU and limit data movement as much as possible. We will integrate cuda runtime library for timing our code for extra precision. OpenACC is interoperable with CUDA functions which makes it easier to utilize features from CUDA that is not present in OpenACC.

In this exercise, we have made a few changes from the previous exercise. The code requires a few addition directives in order to run the code properly through proper data movement.

To DO:

1. Apply Parallel Compute Directive for the data initialization part and apply gang on outer loop and vector on inner loop. Assume data is present in the compute directive
2. Apply **enter data** directive prior to compute directive, make sure data is allocated and present on GPU using the three vectors A,V,X
3. Apply **exit data** directive at end of computation prior to cudaEventRecord(event2,0). Minimize data-transfer by copying out only the solution vector and de-allocate all vectors on GPU

RESULT

CPU-Version	Test 1	Test 2	AVERAGE
C	2666.667	2857.143	2761.905
Fortran	2735.997	2807.569	2771.783
Lab1			
C	526.316	449.438	487.877
Fortran	507.533	438.305	472.919
Lab2			
C	615.385	615.385	615.385
Fortran	618.923	618.543	618.733
Lab3			
C	7085.284	7060.220	7072.752
Fortran	4689.513	7138.313	5913.913