GPU Programming with Directives Exercise
Solving the Poisson equation using Jacobi iterations

1 Introduction

The purpose of this exercise is to explore offloading a poisson solver based on the Jacobi iteration method using OpenMP. This sample code solves the poisson equation $\nabla^2\phi(x,y) = \rho(x,y)$ on a 2D domain. The field is discretized on a regular Cartesian grid. The solution is obtained using an iterative Jacobi solver. At every iteration the new value of the field is set according to the values of the old field based on the equation

$$\phi_{i,j}^{new} = \frac{1}{2}\left(\frac{\phi(i-1,j)+\phi(i+1,j)}{1+\left(\frac{dx}{dy}\right)^2}\right)+\frac{1}{2}\left(\frac{\phi(i,j-1)+\phi(i,j+1)}{1+\left(\frac{dy}{dx}\right)^2}\right)-\frac{1}{2}\frac{\rho_{i,j}}{\left(\frac{1}{dx}\right)^2+\left(\frac{1}{dy}\right)^2} \tag{1}$$

We keep iterating until we converge to an approximate solution of the Poisson equation.

2 Compiling and Running

Unpack the poisson tar file with the command `tar -zxvf poisson_exercise.tar.gz`.

In the `poisson_exercise` folder you will find the `env-archer2.sh` file. You can set-up your environment by sourcing the file.

```
cd poisson_exercise
source env-archer2.sh
```

This folder contains a CPU implementation of the Poisson solver described above. Once you have set-up your environment, you can compile by typing `make` in this folder. You can submit a job to the computing nodes using the `submit.sh` Slurm batch job script.

```
sbatch submit.sh
```

3 Offload the main loop to the device

The Jacobi iteration is implemented in the `compute jacobi` function in `jacobi.cpp`. Your task is to offload the main loop to the device using OpenMP offloading, making sure to map all required variables to the device. You can check the correctness of your solution by using the `check_output.py`. For instance, to check the correctness of the data at time step 8000 between the origin CPU version and you new offloaded version you can use

```
python3 check_output.py --compare original_phi0_80000.dat new_phi0_80000.dat
```

- This command should return 0 (within machine precision).
- Be sure to save the original output files (`phi0_*`) for comparison to your completed solution.

You can also plot the solution obtained with

```
python3 check_output.py --plot phi*.dat
```

You will need X11 forwarding in order to visualize the plots. This can be done by adding the `-XY` flag to the `ssh` command when logging in.

3.1 Offload the main computational loop with OpenMP

The main loop is containted in the `jacobi.cpp`(`jacobi.f90` for Fortran) file.

- In the `jacobi.cpp` (`jacobi.f90` for Fortran) file, offload to the device the main loop. Keep care to map all the variables accessed from the computational kernel. There are a lot of them !
- After completing the step above the data is transferred at every iteration. Try to have all the data transferred only once at the beginning of the simulation and once before every output. You might want to modify the `poisson_solver.cpp` (`poisson_solver.f90` for Fortran) file.
- There are a lot of variables to transfer between the CPU and GPU, but most of these are grouped in custom classes. Create a custom data mapper to map all of the variables in the data structure. Take care to avoid mapping the same variable twice. **(C++ only)**

3.2 Offload the main computational loop with hip (C++ only)

It is sometimes useful to mix OpenMP kernel with hip/cuda kernels or using vendor specific libraries for linear algebra, Fourier transforms and other common mathematical tasks. The `jacobi_hip.cpp` file contains an alternative implementation of the Jacobi kernel using hip. You can use this implementation by compiling with:

```
make clean
make poisson_rocm
```

This will create a new executable called `poisson_rocm`. Remember to update the name of the executable in the `submit.sh` file when running the executable. This course does not cover hip, so we do not expect you to modify the jacobi hip kernel. The hip kernel is submitted by the `launch_compute_jacobi_hip` function and expects device addresses as arguments, not host addresses.

- Use OpenMP directives to pass the correct device addresses to the `launch_compute_jacobi_hip` function in the `jacobi_hip.cpp` file.

3.3 Offload multiple fields to the device concurrently

In `poisson_solver.cpp` (`poisson_solver.f90` for Fortran) set the `nFields` variable to 10. This will solve `nFields` independent equations. As the iteration of fields are all independent of each other, they can execute concurrently on the device. However by default offloaded parallel regions are executed serially on the device. Use async constructs (such as openmp tasks and `nowait` clauses) to offload parallel regions corresponding to different fields on the device concurrently.

1. Use async constructs to offload parallel regions contained in different iterations of the loop over fields concurrently. Can you see a difference in performance ?
2. Use OpenMP parallelism to distribute different fields across different CPU threads, but still offloading the computational kernels to the device. Can you see a difference in performance ?

4 Performance optimisation

There are several performance limiters that can affect the performance of an application. Some common ones are:

1. Inefficient Memory accesses
2. Insufficient parallelism
3. Register pressure
4. Kernel launch latency

In this exercise you will explore the performance of the offloaded OpenMP region.

4.1 Profiling

In order to analyze what is limiting the performance of a kernel it is useful to use a profiler tool to obtain information on how efficiently your computation kernel is using the device. A profile contains information on how efficiently you are using the hardware: wavefront occupancy, achieved bandwidth etc...

You can can generate a profile by submitting with `sbatch submit-profiler.sh`. This will generate a `report.txt` file with a lot of information about the offloaded kernel. Below you can find a description of some useful sections:

- *Wavefront Occupancy*: Check this metric for an estimate of how much of the available parallelism on the device you are exploting.
- *Table 17.1 L2 cache. Speed-of-Light*: Look at the read and write bandwidth between the L2 cache and the Global memory. The peak achievable memory bandwidth on the Archer2 MI210 is about `1400 GB/s`. The peak theoretical bandwidth is `1600 GB/s`.
- *Table 17.3 L2 Cache Accesses*: Look at the number of read/write requests per wavefront. Do they align with your expectations ?
- *Kernel Time (Nanosec)*: Execution time on the device of the kernel.

4.2 Optimisations

1. Run the offloaded version obtained from resolving exercise 3.1 and compare with the CPU only version. How do the run times for the kernels compare ? Which of the performance limiters described above might be limiting the performance of the kernel ? Use the profiler report to help find possible area of improvements.
2. Collapse the two loops by adding the `collapse(2)` clause. Does the performance improve ? If so, which of the performance limiters above was addressed by collapsing the two loops ?
3. Exchange the loop order. Does the performance improve ? If so, why do you think exchanging the loop order improved performance ?