OpenMP Offloading Code on Frontier: Hands on

Wael R. Elwasif, Reuben Budiardja, Swaroop Pophale

Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory
## Frontier Compilers: Summary Table

<table>
<thead>
<tr>
<th>Programming Environment</th>
<th>C</th>
<th>Compilers</th>
<th>Fortran</th>
<th>Compiler Modules</th>
<th>OpenMP Flags</th>
<th>Offloading Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrgEnv-cray</td>
<td>cc</td>
<td>(craycc     craycxx</td>
<td>ftn     crayftn)</td>
<td>cce</td>
<td>-fopenmp</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>craycc</td>
<td></td>
<td>craype-accel-amd-gfx90a rocm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrgEnv-amd</td>
<td>cc</td>
<td>(amdclang amdclang++</td>
<td>ftn     amdflang)</td>
<td>amd</td>
<td>-fopenmp</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>amdclang</td>
<td></td>
<td>rocm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrgEnv-gnu</td>
<td>cc</td>
<td>(gcc     g++     ftn</td>
<td>gfortran)</td>
<td>gcc</td>
<td>-fopenmp</td>
<td>No</td>
</tr>
</tbody>
</table>

- craycc and craycxx are based on the LLVM clang compiler suite
- crayftn front end **NOT** based on LLVM/ backend LLVM based
- craycc/craycxx/crayftn backend uses proprietary Cray/HPE code.
- amdclang and amdclang++ are based on the LLVM clang compiler
- amdflang is based on the “old” flang compiler, **not advisable for use in production**
- MI250X offloading support for GCC under development, performance ??
### GPU PARALLELISM MAPPING POLICIES: CCE

<table>
<thead>
<tr>
<th>NVIDIA</th>
<th>AMD</th>
<th>CCE Fortran OpenACC</th>
<th>Old CCE 15 Fortran OpenMP</th>
<th>Old CCE 15 C/C++ OpenMP</th>
<th>New CCE 16 OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threadblock</td>
<td>Work group</td>
<td>acc gang</td>
<td>omp teams</td>
<td>omp teams</td>
<td>omp teams</td>
</tr>
<tr>
<td>Warp</td>
<td>Wavefront</td>
<td>acc worker</td>
<td>omp parallel</td>
<td>omp parallel</td>
<td>omp parallel</td>
</tr>
<tr>
<td>Thread</td>
<td>Work item</td>
<td>acc vector</td>
<td>omp simd</td>
<td>omp simd</td>
<td>omp simd</td>
</tr>
</tbody>
</table>

- New/improved “omp parallel” support in CCE 16
- More consistent across C/C++/Fortran within CCE
- More consistent with upstream Clang and other vendor implementations
- Current best practice:
  - Use “omp teams” to express GPU threadblock/work group parallelism
  - Use “omp parallel do/for simd” to express GPU thread/work item parallelism
- Future possibilities:
  - Improve CCE performance for “omp loop”
  - Leverage multi-dimensional grids or blocks
  - Support three levels of parallelism in OpenMP

Goal: let users express parallelism with any construct they think makes sense, and CCE will map to available hardware parallelism.
Compilers & Modules on Frontier

- module reset

- CCE (15.0.0) (default)
  - module load PrgEnv-cray rocm craype-accel-amd-gfx90a
    [module load cpe/22.12]  #-- Only needed if going back from CCE/16.0.1 below

- CCE 16.0.1
  - module load PrgEnv-cray rocm craype-accel-amd-gfx90
    module load cpe/23.09  #-- this automatically loads cce/16.0.1

- ROCm
  - module load PrgEnv-amd amd

- GCC (Experimental offloading)
  - module use /sw/crusher/ums/compilers/modulefiles
    module load  module load gcc/13.2.1-dev-lates
Tutorial Code

• Repo: https://github.com/olcf/openmp-offload.git

• Simple Jacobi iterations with random initial conditions
  – C & Fortran versions of each variant

• Makefiles for the different compilers available
  – Pass COMPILER=XYZ flag to control which compiler to use
  – Make sure the right modules are loaded for each compiler
  – Example: Compile using CCE

    [elwasif@login07.frontier.openmp-offload]$ module load PrgEnv-cray cce rocm craype-accel-amd-gfx90a
    [elwasif@login07.frontier.openmp-offload]$ cd C/0-serial/
    [elwasif@login07.frontier.0-serial]$ make COMPILER=cce
    Make sure the following modules are loaded: PrgEnv-cray cce craype-accel-amd-gfx90a
    cc -Ofast -fopenmp -lm jacobi.c -o jacobi.C.cce.exe

• Command line arguments: num_cells max_iterations
  – Except for code in 5-openmp-gpu-implicit/
Jacobi iterations : Initialization

- Random seed generated and saved
- Regenerate the same problem for validation, or for runs using different configurations

```c
void init(double* T){
    static int first_time = 1;
    static int seed = 0;
    if (first_time == 1) {
        seed = time(0);
        first_time = 0;
    }
    srand(seed);

    for (unsigned i = 0; i <= n_cells + 1; i++) {
        for (unsigned j = 0; j <= n_cells + 1; j++) {
            T(i, j) = (double)rand() / (double)RAND_MAX;
        }
    }
}
```
Jacobi iterations: Serial version

```c
// simulation iterations
while (residual > MAX_RESIDUAL && iteration <= max_iterations) {

    // main computational kernel, average over neighbors in the grid
    for (unsigned i = 1; i <= n_cells; i++)
        for (unsigned j = 1; j <= n_cells; j++)
            T_new(i, j) = 0.25 * (T(i + 1, j) + T(i - 1, j) + T(i, j + 1) + T(i, j - 1));

    // reset dt
    residual = 0.0;

    // compute the largest change and copy T_new to T
    for (unsigned int i = 1; i <= n_cells; i++)
        for (unsigned int j = 1; j <= n_cells; j++)
            residual = MAX(fabs(T_new(i, j) - T(i, j)), residual);
        T(i, j) = T_new(i, j);
    iteration++;
}

printf("Serial Residual = %.9lf\n", residual);
```
Jacobi iterations : Serial version

// simulation iterations
while (residual > MAX_RESIDUAL && iteration <= max_iterations) {
    // main computational kernel, average over neighbours in the grid
    for (unsigned i = 1; i <= n_cells; i++) {
        for (unsigned j = 1; j <= n_cells; j++) {
            T_new(i, j) = 0.25 * (T(i+1, j) + T(i-1, j) + T(i, j+1) + T(i, j-1));
        }
    }
    // reset residual
    residual = 0.0;
    // compute the largest change and copy T_new to T
    for (unsigned int i = 1; i <= n_cells; i++) {
        for (unsigned int j = 1; j <= n_cells; j++) {
            residual = MAX(fabs(T_new(i, j) - T(i, j)), residual);
            T(i, j) = T_new(i, j);
        }
    }
    iteration++;
}
printf("Serial Residual = %.9lf\n", residual);
# The C/C++ & Fortran OpenMP Variants

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-serial/</td>
<td>Base serial version</td>
<td></td>
</tr>
<tr>
<td>1-openmp-cpu/</td>
<td>OpenMP CPU only</td>
<td></td>
</tr>
<tr>
<td>2-openmp-gpu-teams/</td>
<td>GPU: Teams only</td>
<td>Day 1</td>
</tr>
<tr>
<td>3-openmp-gpu-parallel/</td>
<td>GPU: Teams + Threads</td>
<td>Day 1</td>
</tr>
<tr>
<td>4-openmp-gpu-data/</td>
<td>GPU: Manage data movement</td>
<td>Day 2</td>
</tr>
<tr>
<td>5-openmp-gpu-implicit/</td>
<td>GPU: Implicit data movement</td>
<td>Day 2 – C++</td>
</tr>
<tr>
<td>6-openmp-combined/</td>
<td>All variants</td>
<td></td>
</tr>
<tr>
<td>7-loop-combined/</td>
<td>Using loop construct</td>
<td></td>
</tr>
</tbody>
</table>

**Similar Directory Structure for Fortran code**
Submitting Jobs On Frontier

• Use your own project ID
• Reservations from 11:45 – 3:30
  – --reservation=openmp
• Sample batch script for 7 CPU threads
• Template in code repo
• **Match compile and execution modules**

```bash
#!/bin/bash
#SBATCH -A ABC123
#SBATCH -J omptutorial
#SBATCH -o %x-%j.out
#SBATCH -t 00:10:00
#SBATCH -p batch
#SBATCH -N 1
#SBATCH --reservation=openmp
ulimit -s unlimited # Needed for implicit mapping example

#For CCE
module load PrgEnv-cray cpe/23.09 cce/16.0.1 rocm craype-accel-amd-gfx90a

cd /PATH/TO/TUTORIAL/openmp-offload/C/1-openmp-cpu/
export OMP_NUM_THREADS=7 # for CPU OpenMP
./jacobi.C.cce.exe <args>
```
Experiments

Please post questions here: [Google doc for OpenMP Offload Part 1](#)

- Compile and run the (GPU) code for the different compilers
  - Performance difference across compilers ??
  - Profile: [https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#profiling-applications](https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#profiling-applications)

- When is it profitable to offload to the GPU ?
  - Does it depend on the compiler ?

- Frontier GPU’s have 64 GB each: What’s the biggest problem you can solve?
  - Does the maximum problem size depend on the compiler?

- What’s the impact of changing `num_teams` and `thread_limit` on performance
  - Can you figure out the default values used by the different compilers?