Archiving Performance Portability for GTC-P by using OpenACC

James Lin¹, Yueming Wei¹, Stephen Wang¹, Simon See¹, William Tang², Stephane Ethier², Bei Wang²

¹Center for HPC, Shanghai Jiao Tong University
²Princeton Plasma Physics Lab, Princeton University

GTC China 2016, Beijing
Sep 13th, 2016
1. Background

2. Step 1: Implementing GTC-P with OpenACC

3. Step 2: Optimizing OpenACC GTC-P

4. Results

5. Conclusion and Future Work
1. Background: GTC-P

- The Gyrokinetic Toroidal Code developed by Princeton (aka. GTC-P) simulates plasma turbulence within Tokamak fusion devices.
- Numerical method: Gyrokinetic Particle-in-cell (PIC) method.
- Highly scalable: running on leadership-class supercomputers:
  - Tianhe-2
  - Titan
  - Sequoia
  - Mira
  - Piz Daint
  - Stampede
  - Tsubame2.5
OpenACC and Performance Portability

Now Runs On x86 Multicore CPU

One Code Path Accelerated with OpenACC

GPU Accelerator

Courtesy NVIDIA/PGI
2. Implementation: Six Major Subroutines of GTC-P

- **Charge**: particle to grid interpolation (SCATTER)
- **Smooth/Poisson/Field**: grid work (local stencil)
- **Push**: 
  - grid to particle interpolation (GATHER)
  - update position and velocity
- **Shift**: in distributed memory environment, exchange particles among processors
Two hotspots of GTC-P

- charge and push are the hotspots, which consume 86.1% overall execution time.
- Therefore, we port the two hotspots, charge and push, onto accelerator, and leave the rest four subroutines on host.
Initial OpenACC Implementation

- Use *parallel loop* directive for parallelism
- Use *data* directives to specify data transfer.
  - *copy/copyin/copyout/present*
  - Reduce data transfer between host and device
- Use *atomic* directive for correctness due to data race
Reducing Device-host Data Transfer

Reduce data transfer between host and device

- Move data clause from kernel to the beginning of the program
- Port *shift* routine to GPU by OpenACC to keep device data updated on GPU
3. Optimization: on Device (GPU)

- Opt. I: thread mapping optimization
  - `num gangs/vector_length` to adjust occupancy

```
#pragma acc parallel loop num_gangs(2048) vector_length(64)
for (m = 0; m < mi; m++) {
    ...
```

- `seq` directive to avoid auto parallelization by compiler

```
#pragma acc parallel loop
for (m=0; m<mi; m++) {
    ...
}
#pragma acc loop seq
for (larmor = 0; larmor < 4; larmor++) {
```
• Opt. II: Optimize *charge* routine in CUDA
  – Use cooperative computation to capture locality for co-scheduled threads – use global atomics, but in a coalesced way (by transposing in shared memory)

```c
#pragma acc host_data use_device
271  (density1, z0, z1, z2, z4, z5, pgyro, tgyro, jtion0, jtion1, \n    wtion0, wtion1, wzion, wpion, igrid, delt, qtinv, mtheta) 
273 { 
    call_gpu_charge_kernel(parallel_decomp->mype, density1, mzeta, nloc_over, z0, z1, z2, z4, z5, \n    pgyro, tgyro, jtion0, jtion1, wtion0, wtion1, wzion, wpion, igrid, delt, qtinv, mtheta, \n    mpsi, a0, a1, delr, delz, smu_inv, zetamin, pi2_inv, mi, igrid_in, ipsi_in, ipsi_out, rho_max);
277 }
```
In *charge* routine, array “densityi” is in data race when porting it in parallel.

```c
#pragma acc atomic update
densityi_part[ij1] += d1;
#pragma acc atomic update
densityi_part[ij1+1] += d2;
```

Allocating replication copies for array “densityi_part” and do reduction at the end can avoid conflict. **We need to obtain the OpenACC thread ID manually.**

<table>
<thead>
<tr>
<th>Problem instance: A</th>
<th>charge w atomic</th>
<th>charge w/o atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time (sec)</td>
<td>1572.83</td>
<td>8.91</td>
</tr>
</tbody>
</table>

177x speedup
Optimization on host: obtain OpenACC thread ID

Obtain OpenACC thread ID manually based on the thread mapping policy:

```c
If(mi%ACC_NUM_CORES==0)
    mi_per_core = mi/ACC_NUM_CORES;
else
    mi_per_core = mi/ACC_NUM_CORES + 1;
#elsepragma acc parallel loop
for(m=0; m<mi; ++m){
    acc_thread_id = m/mi_per_core;
    ...
}
```
4. Results: Step-by-step Optimization

- initial OpenACC implementation
  - 1.24X speedup (compared with OpenMP code.)
  - +213 Line of Code (LOC)
- + thread mapping opt.
  - 2.46x speedup. (compared with OpenACC baseline.)
  - +3 LOC
- + CUDA opt.
  - 1.38x speedup (compared with thread mapping opt.)
  - +104 LOC

Results on single node (Xeon E5-2695 V3 with 2 NVIDIA Tesla K80)
Application setup

GTC-P numerical settings

<table>
<thead>
<tr>
<th>Problem instance</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpsi</td>
<td>90</td>
<td>180</td>
<td>360</td>
</tr>
<tr>
<td>mthetamax</td>
<td>640</td>
<td>1280</td>
<td>2560</td>
</tr>
<tr>
<td>mgird</td>
<td>32449</td>
<td>128893</td>
<td>513785</td>
</tr>
<tr>
<td>particles per cell</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mstep</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>total particles</td>
<td>3235900</td>
<td>12871300</td>
<td>51342496</td>
</tr>
</tbody>
</table>

- **mps i** is the number of grid points in the radial dimension
- **mthetamax** is the maximum number grid points in the poloidal dimension
- **mgird** is the number of grid points per toroidal plane
- **mstep** is the number of calculation steps
Performance Portability

- NVIDIA K80 is a dual-GPU graphics card, we only study the portability of OpenACC code on single node without MPI, therefore, only one of the two GK210 GPUs is used.
Scalability: test bed

The evaluation is performed on Supercomputer π of SJTU. The hardware configuration of each node and machine environment is shown below.

<table>
<thead>
<tr>
<th>Component</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Xeon E5-2670 × 2 (2.60GHz)</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla Kepler K20M × 2</td>
</tr>
<tr>
<td>Interconnection</td>
<td>Infiniband: Mellanox FDR 56Gb/s</td>
</tr>
<tr>
<td>OS</td>
<td>Red Hat Enterprise Linux Server release 6.3</td>
</tr>
<tr>
<td>MPI</td>
<td>mpich-3.2</td>
</tr>
<tr>
<td>Compiler</td>
<td>PGI 16.4, CUDA Toolkit 7.5, ICC 15.02</td>
</tr>
</tbody>
</table>
Strong scaling

push routine has better scalability than charge routine
Weak scaling

Particles and grids in each process is approximately the same.
5. Conclusion and Future work

- We implement the first OpenACC GTC-P and evaluate its performance portability across the GPU, x86 CPU, and Power8.

- OpenACC shows both impressive productivity and performance. With only 213 LOC, OpenACC achieved 73% performance of CUDA code on GPUs and acceptable scalability across up to 16 nodes.

- We will extend evaluation of OpenACC GTC-P on other platforms such as ARM to further study how to archive reasonable performance portability for a production code instead of kernels or benchmarks.