Porting and Optimization of Search of Neighbour-particle by Using OpenACC

Takaaki Miyajima and Naoyuki Fujita
Numerical Simulation Tech. Unit
Aeronautical Tech. Directorate
Japan Aerospace Exploration Agency
JAXA is a public agency for researching and developing aerospace science and technology.

Supersonic aircraft, satellites, rockets and space science are our research target.
Apply Computational Fluid Dynamic (CFD) more broadly

- From steady-state simulation to **unsteady-state** simulation

**Simulate unsteady-state**

Understanding behavior of aircraft during takeoff, landing or turn

**Simulate steady-state**

Many research have been done on cruise condition

**Simulate all flight envelope**

Apply to actual aircrafts

Apply newly developed theory to model of actual aircrafts
An example of unsteady-state simulation

Landing in rainy weather: raindrops have negative effects on wings and tires.

- Simulation of interactions between raindrops and aircraft is needed
- Higher computing power is necessary as well :)

Raindrops decrease the lift coefficient
Cite: http://akihito114.exblog.jp/21064589/

Rain puddle makes landing run longer
Cite: http://blogs.yahoo.co.jp/qf104j/28794118.htm

We adopt MPS (Moving Particle Semi-implicit) method for simulating raindrops
1. Moving Particle Semi-implicit (MPS) Method
2. OpenACC
3. Porting and Optimization
MPS (Moving Particles Semi-Implicit) Method

【Overview】
• MPS method is attracting attention in CFD area.
• Particle-base simulation (not a stencil computation)
• Target fluids are divided to thousands of particles, each particle interacts with its neighbour-particle.

【Features computer science】
1. The # of particles becomes over ten thousands, parallel computing device is necessary
2. All the particles can be calculated independently
3. Memory-bound application
4. The “search for neighbour-particle” is the main bottleneck

MPS simulation: A collapse of water column
NSRU-MPS: in-house MPS program

We’re developing in-house MPS method program

【Features of original program】

• Physicist write the program
• +7000 lines of Fortran90
• Physical quantities are single-precision floating-point
• Structure of Array (SoA) style data structure
• Parallelized only by MPI

Simulation done by NSRU-MPS
Profiling of elapse time on Xeon CPU (IveBridge)

<table>
<thead>
<tr>
<th>Target problem</th>
<th>A collapse of water column 40[cm]x40[cm]x8[cm]</th>
</tr>
</thead>
<tbody>
<tr>
<td># of particles</td>
<td>2,247,750</td>
</tr>
<tr>
<td># of MPI processes</td>
<td>24</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2697 v2 @2.7GHz, 12 cores * 2CPUs</td>
</tr>
<tr>
<td>Memory</td>
<td>128GB of DDR3-12800</td>
</tr>
<tr>
<td>Compiler</td>
<td>PGI Fortran 16.10 with &quot;-O3 -fast&quot; option</td>
</tr>
<tr>
<td>MPI Library</td>
<td>OpenMPI 1.10.5 with &quot;-bind-to socket -npersocket 12 -n 24&quot; option</td>
</tr>
<tr>
<td>Measurement method</td>
<td>An average of first 200 steps by MPI_Wtime() function</td>
</tr>
</tbody>
</table>

**Result**
- 1 time step: 7093.75[ms]
- Search for neighbour-particle and MPI related accounted for 56% and 21% of the total processing time, respectively.
Profile elapse time by changing the # of process from 2 to 24

- Elapse time decreased along with the # of procs
- MPI communication increased in proportion to the # of procs

Decrease elapse time while keeping the # of procs small
Search for neighbour particle (w/ bucket)

【Bucket】

- Divide simulation space into squares called “bucket”
- The volume of bucket is equal to $3^3$ particles
- Effect radius (cut-off distance) is 3 buckets

【Search for neighbour particle】

1. Pickup a target particle (red)
2. Traverse adjacent $3^3$ buckets ✓ No fixed order to traverse bucket
3. Search particles in a bucket
4. Calculate distance and weight between the target particle
5. Accumulate weighted physical value to a target particle ✓ No fixed order to accumulate physical value

※ Other particle-base simulation (Molecular Dynamics or N-body sim) has similar computation
Quadraple nested-loop is used

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>! for all the particles</td>
</tr>
<tr>
<td>2</td>
<td>do–loop1: target_ptcl = 1, all_ptcl</td>
</tr>
<tr>
<td>3</td>
<td>ib = bucket_num[m]</td>
</tr>
<tr>
<td>4</td>
<td>! traverse adjacent buckets (3–dim: 3x3x3=27)</td>
</tr>
<tr>
<td>5</td>
<td>do–loop2: x=x1,x2</td>
</tr>
<tr>
<td>6</td>
<td>do–loop3: y=y1,y2</td>
</tr>
<tr>
<td>7</td>
<td>do–loop4: z=z1,z2</td>
</tr>
<tr>
<td>8</td>
<td>ibb = get_adj_bucket_num(x,y,z)</td>
</tr>
<tr>
<td>9</td>
<td>num_of_ptcl = get_num_of_ptcl_in_bucket(ibb)</td>
</tr>
<tr>
<td>10</td>
<td>! accumulate all the neighbour–particles</td>
</tr>
<tr>
<td>11</td>
<td>do–loop5: np = 1, num_of_ptcl ! indefinite loop</td>
</tr>
<tr>
<td>12</td>
<td>if (ptcl_is_in_halo)</td>
</tr>
<tr>
<td>13</td>
<td>lcr = ptcl_halo[np] ! random access</td>
</tr>
<tr>
<td>14</td>
<td>else</td>
</tr>
<tr>
<td>15</td>
<td>lcr = ptcl[np] ! random access</td>
</tr>
<tr>
<td>16</td>
<td>end if</td>
</tr>
<tr>
<td>17</td>
<td>dist = sqrt(dot_product(m, lcr)) ! get distance</td>
</tr>
<tr>
<td>18</td>
<td>weight = get_weight(dist)</td>
</tr>
<tr>
<td>19</td>
<td>accum = accum + phys(weight) ! aggregation</td>
</tr>
<tr>
<td>20</td>
<td>m_phys[m] = m_phys[m] + accum ! in–place add</td>
</tr>
</tbody>
</table>
Analysis of Search for neighbour particle

- Not easy to vectorize and utilize cache
  - Each target particle accesses different index of bucket and particle
  - Thousands of in-flight data request to hide latency
  - No fixed order to traverse and accumulate value

- Computation naturally fits to SIMT-model
  - Indirect access; search particles in a bucket
  - Indefinite loop; # of particles in a bucket is uncertain, inefficient access pattern
1. Moving Particle Semi-implicit (MPS) Method
2. OpenACC
3. Porting and Optimization
Add directives on existing C/C++, Fortran code, and the compiler automatically generates binary for GPU. No need to write CUDA C/Fortran from scratch. Typical target of offload

- Loop
- Data transfer (CPU from/to GPU)
- User defined functions
- CUDA Library: cuBLAS, cuFFT, etc can be integrated

Not a few practical applications are ported by OpenACC

- Sunway TaihuLight added their own extensions.
- Most of applications adopts stencil computation

PGI Compiler (Community Edition) is free for personal use.
Three directives provided by OpenACC

**acc data directive**
- transfers data between the host and the device memory at an arbitrary timing.
- data transfer happens at this position.

**acc kernels directive**
- specify regions of code for offloading from CPU to GPU
- compiler automatically analyzed the loop and the necessary data

**acc parallel / loop directive**
- Optimize nested/single loop
- Loop can be mapped to block, warp, and thread

※ Each directive can have additional clauses to augment information

Sample code: Jacobi method

```c
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
  error = 0.0;
  #pragma acc kernels
  {
    #pragma acc loop
    for ( int j = 1; j < n-1; j++ ) {
      for ( int i = 1; i < m-1; i++ ) {
        Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] + A [j-1] [i] + A [j+1] [i] );
        error = fmax ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
  }
  #pragma acc loop
  for ( int j = 1; j < n-1; j++ ) {
    for (int i = 1; i < m-1; i++ ) {
      A [j] [i] = Anew [j] [i];
    }
  }
  if (iter % 100 == 0) printf ("%5d, %0.6f\n", iter, error);
  iter++;
}```
Gang, Worker, Vector are provided to model SIMT

- Map loops and functions explicitly
- Gand = Block
- Worker = Warp
- Vector = CUDA Thread

<table>
<thead>
<tr>
<th>Grid</th>
<th>Shared mem (shared clause)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block(0,0) = Gang(0,0)</td>
<td>Thread(0,0) = Vector(0,0)</td>
</tr>
<tr>
<td>Block(1,0) = Gang(1,0)</td>
<td>Thread(1,0) = Vector(1,0)</td>
</tr>
<tr>
<td>Block(2,0) = Gang(2,0)</td>
<td>Thread(2,0) = Vector(2,0)</td>
</tr>
<tr>
<td>Block(3,0) = Gang(3,0)</td>
<td>Thread(3,0) = Vector(3,0)</td>
</tr>
<tr>
<td>Block(4,0) = Gang(4,0)</td>
<td>Thread(4,0) = Vector(4,0)</td>
</tr>
<tr>
<td>Block(5,0) = Gang(5,0)</td>
<td>Thread(5,0) = Vector(5,0)</td>
</tr>
<tr>
<td>Block(6,0) = Gang(6,0)</td>
<td>Thread(6,0) = Vector(6,0)</td>
</tr>
</tbody>
</table>

num_gang = 3
num_worker = 3
vector_length = 5
Clause and its function

Used clauses in our implementation

<table>
<thead>
<tr>
<th>clause</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>gang(N)</td>
<td>map the loop to the N thread block</td>
</tr>
<tr>
<td>worker(N)</td>
<td>map the loop to the N warp</td>
</tr>
<tr>
<td>vector(N)</td>
<td>map the loop to the N thread</td>
</tr>
<tr>
<td>seq</td>
<td>run the loop sequentially</td>
</tr>
<tr>
<td>collapse(N)</td>
<td>make a N-nested loop to one large loop</td>
</tr>
<tr>
<td>independent</td>
<td>run each iteration independently</td>
</tr>
<tr>
<td>atomic</td>
<td>perform atomic operation</td>
</tr>
</tbody>
</table>
Agenda

1. Moving Particle Semi-implicit (MPS) Method
2. OpenACC
3. Porting and Optimization
Three optimization: *Naive*, *Atomic*, 3-D

1. **Naive**: $1\text{ particle} = 1\text{CUDA thread}$
   - Simplest optimization
   - Code modification is not required

2. **Atomic**: $1\text{bucket} = 1\text{CUDA thread}$
   - Use atomic operation for accumulation
   - Small code modification is required

3. **3-D Thread**: $1\text{bucket} = 1\text{CUDA thread}$
   - Consider physical background to map threads
   - Small code modification is required
Naive: \(1\) particle = \(1\) CUDA thread

```c
! for all the particles
!$acc kernels
!$acc loop gang vector(128)
do-loop1: target_ptcl = 1, all_ptcl
   ib = bucket_num[m]
! traverse adjacent buckets (3-dim: 3x3x3 = 27)
!$acc loop collapse(3) seq
do-loop2: x=x1,x2
   do-loop3: y=y1,y2
      do-loop4: z=z1,z2
         ibb = get_adj_bucket_num(x,y,z)
         num_of_ptcl = get_num_of_ptcl_in_bucket(ibb)
! accumulate all the neighbour-particles
!$acc loop seq
do-loop5: np = 1, num_of_ptcl
   if (ptcl_is_in_halo)
      lcr = ptcl_halo[np] ! random access
   else
      lcr = ptcl[np] ! random access
   end if
   dist = sqrt(dot_product(m, lcr)) ! get distance
   weight = get_weight(dist)
   accum = accum + phys(weight) ! aggregation
m_phys[m] = m_phys[m] + accum ! in-place add
```
Atomic: 1 bucket = 1 CUDA thread

for all the particles
$acc$ parallel
$acc$ loop collapse(4) independent gang vector(128)
do-loop1: target $ptcl$ = 1, all $ptcl$
  ! traverse adjacent buckets (3-dim: 3x3x3 = 27)
  do-loop2: $x=x_1, x_2$
  do-loop3: $y=y_1, y_2$
  do-loop4: $z=z_1, z_2$
  ibb = get_adj_bucket_num($x, y, z$)
  num_of_ptcl = get_num_of_ptcl_in_bucket(ibb)
  ! accumulate all the neighbour-particles
  ! moved here from do-loop1
  ib = bucket_num[m]
$acc$ loop seq
do-loop5: np = 1, num_of_ptcl
  if ($ptcl$ is in halo)
    lcr = $ptcl$ halo[np] ! random access
  else
    lcr = $ptcl$[np] ! random access
  end if
  dist = $sqrt$(dot_product($m, lcr$)) ! get distance
  weight = get_weight(dist)
  accum = accum + phys(weight) ! aggregation
  ! moved here from do-loop1
$acc$ atomic update
  $m$ _phys[m] = $m$ _phys[m] + accum ! in-place add
$acc$ end atomic
Bucket traversal is mapped to threadIdx.{x,y,z}, respectively.

Physical background is considered.

Bucket traversal (3-D index)

Atomic add

27 threads / warp

atomic operation is used for accumulation
Evaluation setup (GPUs and data sets)

- Compiler: PGI Fortran 16.10 “-acc -ta = nvidia, cuda8.0, fastmath, cc60”
- Four different GPUs (One Kepler and three Pascal architecture)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K20c</td>
<td>3.5</td>
<td>706</td>
<td>2,496</td>
<td>208</td>
<td>PCIe Gen2 x16 (8GB/s)</td>
<td>Intel Xeon E5-2697 v2</td>
</tr>
<tr>
<td>GTX1080</td>
<td>8.8</td>
<td>1,733</td>
<td>2,560</td>
<td>320</td>
<td>PCIe Gen3 x16 (16GB/s)</td>
<td>Intel Xeon E5-2697 v2</td>
</tr>
<tr>
<td>P100 (PCle)</td>
<td>9.3</td>
<td>1,303</td>
<td>3,584</td>
<td>732</td>
<td>PCIe Gen3 x16 (16GB/s)</td>
<td>Intel Xeon E5-2630L v3</td>
</tr>
<tr>
<td>P100 (NVlink)</td>
<td>10.6</td>
<td>1,406</td>
<td>3,584</td>
<td>732</td>
<td>NVLink (40GB/s)</td>
<td>IBM POWER8 NVL</td>
</tr>
</tbody>
</table>

- Three data sets; Collapse of water column (40[cm] × 40[cm] × 8[cm])

<table>
<thead>
<tr>
<th>data set</th>
<th># of particles</th>
<th># of buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>25,704</td>
<td>35 × 35 × 7</td>
</tr>
<tr>
<td>Medium</td>
<td>224,910</td>
<td>70 × 70 × 14</td>
</tr>
<tr>
<td>Large</td>
<td>2,247,750</td>
<td>150 × 150 × 30</td>
</tr>
</tbody>
</table>
Evaluation: elapse time

2,247,750 particles: Naive is the fastest

224,910 particles: Naive is the fastest (Naive≈Atomic)

25,704 particles: 3-D is the fastest for P100※

※data set is too small to offload
Stall Reason of P100(PCIe)

Analysis Stall Reasons of each implementation by NVPROF

- Data Request Stall of Atomic and 3-D accounted over 80% → Too many In-flight memory requests
- Execution Dependency Stall is caused by distance calculation

※Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding.
Similar optimizations are done by OpenMP

- **Naive+reduction** is best for IvyBridge
- **Naive** is best for KNL

```c
! for all the particles
!$ omp parallel do reduction private(i, j, k, ib, ...) do-loop1: target_ptcl = 1,all_ptcl
    ib = bucket_num[m]
    ! traverse adjacent buckets (3-dim: 3x3x3=27)
do-loop2: x=x1,x2
    do-loop3: y=y1,y2
    do-loop4: z=z1,z2
    ibb = get_adj_bucket_num(x,y,z)
    num_of_ptcl = get_num_of_ptcl_in_bucket(ibb)
    ! accumulate all the neighbour-particle
do-loop5: np = 1,num_of_ptcl ! indefinite loop
    if (ptcl_is_in_halo)
        lcr = ptcl_halo[np] ! random access
    else
        lcr = ptcl[np] ! random access
    end if
    dist = sqrt(dot_product(m, lcr)) ! get distance
    weight = get_weight(dist)
    accum = accum + phys(weight) ! aggregation
m_phys[m] = m_phys[m] + accum ! in-place add
```
Comparison: Elapse time

- **KNL-7210 @ 1.3GHz 64 core (Quadrant+Flat), MCDRAM**
- **IveBridge E5-2697 v2 @ 2.70GHz (12 core*2CPU), DDR3-1600MHz**

<table>
<thead>
<tr>
<th>GPU</th>
<th>Elapse time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20c</td>
<td>263.7</td>
</tr>
<tr>
<td>GTX1080</td>
<td>80.3</td>
</tr>
<tr>
<td>P100</td>
<td>45.2</td>
</tr>
<tr>
<td>8p6t</td>
<td>373.8</td>
</tr>
<tr>
<td>12p4t</td>
<td>395.4</td>
</tr>
<tr>
<td>24p2t</td>
<td>330.4</td>
</tr>
<tr>
<td>2p128t</td>
<td>541.2</td>
</tr>
<tr>
<td>4p64t</td>
<td>570.3</td>
</tr>
<tr>
<td>8p32t</td>
<td>554.7</td>
</tr>
</tbody>
</table>

- **x7.3 faster**
- **x12.0 faster**
Conclusion

Search for neighbour particle (w/ Bucket) is ported and optimized by OpenACC

- *Naive* optimization: Simplest optimization. Fastest for large data set
- *Atomic* optimization: Use atomic operation for accumulation
- *3-D thread* optimization: Physical background is considered
- Four GPUs and three data sets are used for evaluation
- Preliminary evaluation of Xeon and Xeon Phi is shown

Evaluation

<table>
<thead>
<tr>
<th></th>
<th>E5-2697 v2 (MPI only)</th>
<th>E5-2697 v2 (MPI+OpenMP)</th>
<th>KNL-7210 (MPI+OpenMP)</th>
<th>P100 (MPI+OpenACC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization</td>
<td>N/A</td>
<td>Naive+reduction</td>
<td>Naive</td>
<td>Naive</td>
</tr>
<tr>
<td>Elapse time [ms]</td>
<td>1927.4</td>
<td>330.4</td>
<td>541.2</td>
<td>45.2</td>
</tr>
<tr>
<td>Speed up</td>
<td>1.0</td>
<td>5.8</td>
<td>3.6</td>
<td>42.6</td>
</tr>
</tbody>
</table>

x12.0 faster  x7.3 faster
Thank you :)
Comparison of each implementation

Size of Grid, Block and occupancy (Np: # of particles)

<table>
<thead>
<tr>
<th>実装</th>
<th>全スレッド数</th>
<th>Gridサイズ</th>
<th>Blockサイズ</th>
<th>occupancy</th>
<th>used regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>Np</td>
<td>Np/128</td>
<td>128</td>
<td>100%</td>
<td>70</td>
</tr>
<tr>
<td>Atomic</td>
<td>Np*(3^3) = Np*27</td>
<td>Np*(3^3)/128 = Np*0.21</td>
<td>128</td>
<td>100%</td>
<td>70</td>
</tr>
<tr>
<td>3-D thread</td>
<td>Np*(3^3) = Np*27</td>
<td>Np*(3^3)/(3^3) = Np</td>
<td>3^3 = 27</td>
<td>21%</td>
<td>40</td>
</tr>
</tbody>
</table>

Particle

Bucket traversal (3-D index)

Bucket traversal

Atomic add

3-D thread
CPUからGPUへのデータ転送帯域

実測値で2.72倍

<table>
<thead>
<tr>
<th>データセット</th>
<th>Minsky</th>
<th>Intel マシン</th>
</tr>
</thead>
<tbody>
<tr>
<td>小(1.14MB)</td>
<td>44.6[us]</td>
<td>158.4[us]</td>
</tr>
<tr>
<td>大(1.71MB)</td>
<td>81.9[us]</td>
<td>236.9[us]</td>
</tr>
</tbody>
</table>

今回のアプリでも実測で1.8倍ほど速くデータ転送が完了している
CPUからGPUへのデータ転送帯域

device 2 host

<table>
<thead>
<tr>
<th>データセット</th>
<th>Minsky</th>
<th>Intel マシン</th>
</tr>
</thead>
<tbody>
<tr>
<td>小(1.14MB)</td>
<td>44.6[us]</td>
<td>158.4[us]</td>
</tr>
<tr>
<td>大(1.71MB)</td>
<td>81.9[us]</td>
<td>236.9[us]</td>
</tr>
</tbody>
</table>

今回のアプリでも実測で1.8倍ほど速くデータ転送が完了している

実測値で2.68倍
データセット中小での処理時間

図示は、異なる処理方式とブケットサイズにおける処理時間の比較を示しています。x軸には処理方式（Naive、Atomic、3-D Thread）が、y軸には時間（ms）が示されています。各グループに含まれるデータは、ブケットサイズ35×35×7と70×70×14の2種類の設定があります。図中の棒グラフは、各処理方式とブケットサイズにおける処理時間の極値を示しています。また、図の右側には、ブケットサイズとスレッド数ごとの具体的な処理時間が記載されています。