CUDA Implementation of a Lattice Boltzmann Method and Code Optimization

INTRODUCTION

- We study fluid flow in a 2D lid driven cavity for large Reynolds numbers using multi relaxation time - Lattice Boltzmann Method (LBM).
- LBM is an alternative to conventional CFD methods that solve Navier-Stokes equations to simulate incompressible fluid dynamics [1,2,3].
- In LBM, one solves the linearized Boltzmann equation:
  \[ f(x, \xi; t) \approx f(x, \xi; t) + f(x, \xi;\xi') \]
  on a discrete lattice \([4,5,6]\) to study spatio-temporal evolution of flow field \((\xi')\).
- The data parallel implementation of the Lattice Boltzmann Method makes the GPGPU as a platform of choice for such computation.
- Several CUDA optimizations are implemented to achieve desired performance, these are obtained which enabled us to do performance optimization at different levels.

SIMULATION PROBLEM:

- We consider a square cavity of size L (Figure-1).
- Flow is generated by continuously moving the top wall at a constant velocity (u).
- Other walls of the cavity are stationary.
- LBM assumes that fluid particles lives on grid points (Figure-2).
- Particle movement is restricted to 9 discrete directions in 2D space.
- The data parallel implementation of the Lattice Boltzmann Method makes the GPGPU as a platform of choice for such computation.

ALGORITHM

1. Grid Dimension \((nx*ny)\)
2. Max. time iterations
3. AoS containing information of each grid point.
4. Collision reads distribution function, velocity, density \((f, \xi, \rho)\) of grid points. Each grid point is local to each thread and writes collision product to \(\text{tvf}^i\) (temporal variable).
5. Streams kernel reads the collision product, \(\text{tvf}^i\) and writes back to \(\text{tvf}^i\) by updating collision product to appropriate neighbor grid points.

Cuda Implementation

- Two kernels are implemented to solve LBM equation: Collision and Streams (Figure-3).
- Collision stores its product to \(\text{ttv}^i\) and Streaming loads it.
- Global memory operations are expensive.
- As both Collision and Streaming comes in one kernel itself. Store final result in \(\text{ttf}^i\) and swap \(\text{ttv}^i\) and \(\text{ttf}^i\) with \(\text{tvf}^i\).
- Converting AoS to SoA (Figure-7) helps us to achieve coalesced access pattern. Figure-8,9 shows improvement in device compute utilization as compare to baseline (Figure-4).

CUDA OPTIMIZATION

Baseline: We implemented CUDA C version of this problem by simply writing kernels iterating in a loop for a given time iterations. The data required for computations were directly copied to CUDA global memory. Each thread corresponds to one column of LBM grid (see Figure-2). With this we got 5 Million Grid Updates Per Second (MGUPS).

<table>
<thead>
<tr>
<th>MGUPS</th>
<th>GridSize</th>
<th>Time * 10^5</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.6</td>
<td>2975</td>
<td>315.96</td>
</tr>
<tr>
<td>65.5</td>
<td>2975</td>
<td>181.06</td>
</tr>
<tr>
<td>315</td>
<td>512*512</td>
<td>22.6 MGUPS</td>
</tr>
<tr>
<td>181</td>
<td>512*512</td>
<td>22.6 MGUPS</td>
</tr>
</tbody>
</table>

We profilled this application using Nvidia Visual Profiler and many interesting facts and figures were obtained which enabled us to do performance optimization at different levels.

Loop Collapsing: To exhibit more parallelism, we collapsed loops so that each thread corresponds to one grid point (see Figure-2). With this we got 22.6 MGUPS. Profiling also shows improvements in device compute utilization as compare to baseline (Figure-4).

Cuda Constant Memory: Till now, we used only global memory for whole bunch of data. We Lattice data structure which in constant over the course of kernel execution and copied same in CUDA constant memory increases the performance to 23 MGUPS.

Achieving Coalesced Access Pattern: Profiler output shows that global memory load/store access pattern is not optimal (Figure-6). Memory Load/Store efficiency was also very low (Figure-5).

| (Figure-8: Strided and Coalesced access pattern) |
| (Figure-9: Proﬁler output showing global memory transactions/Access) |

We figured out the strides in access pattern in our data structure that causes inefficient access. Converting AoS to SoA (Figure-7) helped us to achieve coalesced access pattern. Figure-8,9 shows improvement in proﬁler outputs. With this we got 65.5 MGUPS.

| (Figure-10: Gathering operation from neighbors) |
| (Figure-11: Scattering operation to neighbors) |

This algorithmic change saves a lot on memory operations and gave around 315 MGUPS of performance. Figure-13 shows performance benchmark with every optimization level.

FUTURE WORK

- Further optimization of the problem on kepler architecture.
- Speed up the application on multiple gpus for high resolution grid sizes.

REFERENCES