CUDA Implementation of a Lattice Boltzmann Method and Code Optimization

Anubhav Jain
Tata Consultancy Services
anubhav.jain1@tcs.com

INTRODUCTION

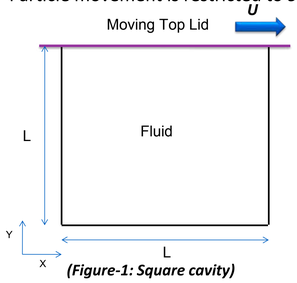
- We study fluid flow in a 2D lid driven cavity for large Reynolds numbers using multi relaxation time - Lattice Boltzmann Method(LBM).
- LBM is an alternative to conventional CFD methods that solve Navier-Stokes equations to simulate incompressible fluid dynamics [1,2,3].
- In LBM, one solves the linearized Boltzmann equation:

$$f_i(\vec{x}+\vec{c}_i\delta_t, t+\delta_t) = f_i^{eq}(\vec{x},t) + f_i^{'ne}(\vec{x},t)$$

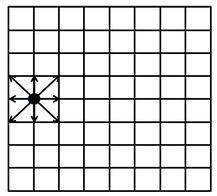
  on a discrete lattice [4,5,6] to study spatio-temporal evolution of flow field $(\vec{v}, \rho, p)$.
- The data parallel implementation of the Lattice Boltzmann Method makes the GPGPU as a platform of choice for such computation.
- Several CUDA optimizations are implemented to achieve desired performance, these are discussed below.

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.

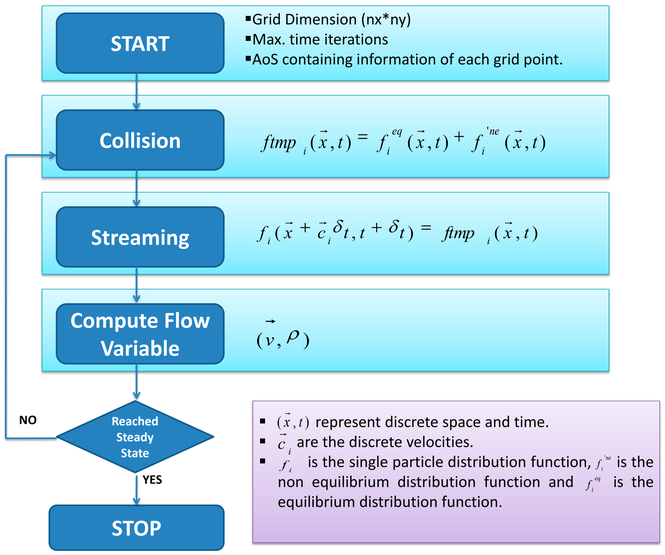


(Figure-1: Square cavity)



(Figure-2: Schematic of LBM grid)

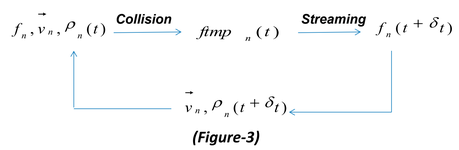
ALGORITHM



- $(\vec{x},t)$ represent discrete space and time.
- $\vec{c}_i$ are the discrete velocities.
- f_i is the single particle distribution function, f_i^{ne} is the non equilibrium distribution function and f_i^{eq} is the equilibrium distribution function.

CUDA IMPLEMENTATION

- Two kernels are implemented to solve LBM equation: **Collision** and **Streams** (Figure-3).



(Figure-3)

- **Collision** reads distribution function, velocity, density $(f, \vec{v}, \rho)$ of grid points. Each grid point is local to each thread and writes collision product to $ftmp$ (temporary variable).
- **Streams** kernel reads the collision product, $ftmp$, and writes back to f by updating collision product to appropriate neighbor grid points.

CUDA OPTIMIZATION

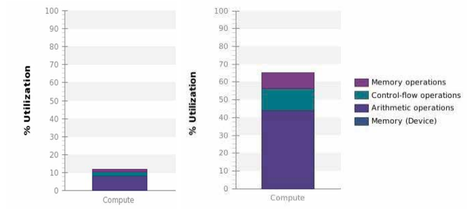
Baseline: We implemented CUDA C version of this problem by simply writing kernels iterating in a loop for a given time iterations. The data required for computations were directly copied to CUDA global memory. Each thread corresponds to one column of LBM grid (see Figure-2). With this we got **5 Million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

$$\text{MGUPS} = \frac{\text{(GridSize) * (TimeIterations)}}{\text{Time} * 10^6}$$

Problem Grid Size : (512*512)
Number of Time Iterations : 10000
CUDA Device : Tesla C2075 card (Fermi architecture, 448 cores)

We profiled 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).



(Figure-4 : 10 % Utilization with baseline version & 65% after loop collapsing)

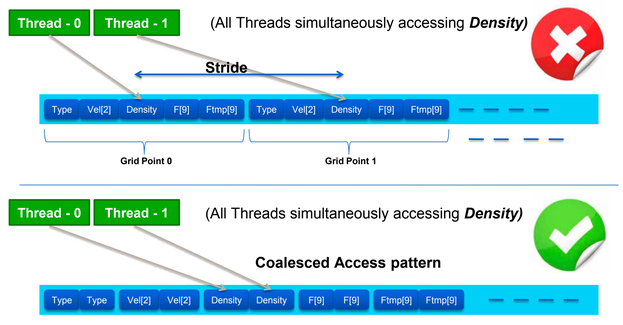
CUDA Constant Memory: Till now, we used only global memory for whole bunch of data. We Lattice data structure which in constant over the course of kernel execution and copied same in CUDA constant memory increases the performance to 23 MGUPS.

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-5: Profiler output showing memory efficiency)

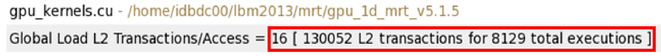
(Figure-6: Profiler output showing global memory transactions/access)

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-7: Strided and Coalesced access pattern)

(Figure-8: Memory efficiency after converting AoS to SoA)



(Figure-9: Profiler output after converting AoS to SoA)

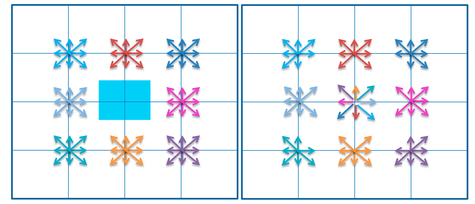
Arithmetic Optimization:

1. All the constant multiplication and division were pre computed only once and passed to kernel rather than re-computation inside kernel for each iteration.
2. Frequently required data was stored into local registers reduces global memory load/store operations.

With such optimizations we got **181 MGUPS**.

IMPLEMENTATION CHANGE

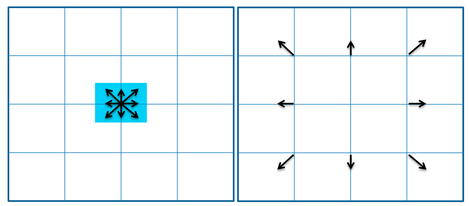
- Collision product needs to be exchange with neighboring grid points.
- For each grid point, data from neighboring grid points needs to be gathered (as shown in Figure-10) that requires synchronization between Collision and Streaming.



(Figure-10: Gathering operation from neighbors)

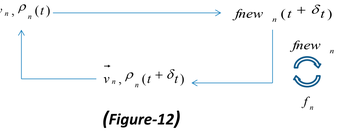
- As shown in Figure-3, maintaining two kernels needs extra Load/Store to $ftmp$ in global memory.
- Collision stores its product to $ftmp$ and Streaming loads it.
- Global memory operations are expensive.

- To avoid synchronization requirement, instead of gathering from neighbors, each grid point scatters its data to neighboring grid points (as shown in Figure-11).
- With such change in data communication approach, we save global memory operations.



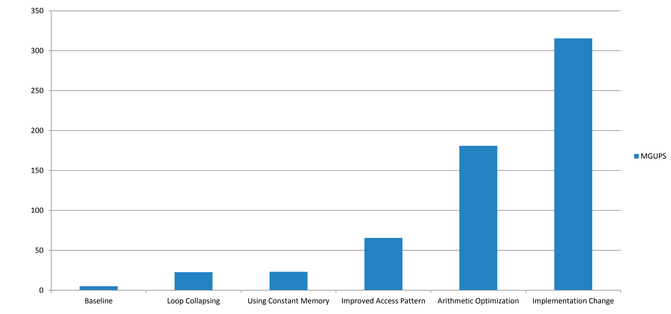
(Figure-11: Scattering operation to neighbors)

As shown in Figure-12, No need of Load/Store to $ftmp$, as both Collision and Streaming comes in one kernel itself. Store final result in $fnew$ and swap f and $fnew$ with every iteration.



(Figure-12)

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.



(Figure-13: Performance Benchmark)

FUTURE WORK

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

REFERENCES

[1] C. K. Aidun and J. R. Clausen, *Ann. Rev. Fluid Mech. 42,* 439 (2010)
[2] R. Benzi, S. Succi, and M. Vergassola, *Physics Reports 222,* 145 (1992)
[3] D. Yu, R. Mei, L. S. Luo and W. Shyy, *Prog. Aerospace Sci.* 39, 329 (2003)
[4] S. Chen and G. D. Doolen, *Ann. Rev. Fluid Mech. 30, 329* (1998)
[5] S. Succi, *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond (Oxford Univ. Press, Oxford, 2001)*
[6] D. A. Wolf-Gladrow, *Lattice Gas Cellular Automata and Lattice Boltzmann Models: An Introduction (Springer, 2004)*
[7] A. J. C. Ladd, *J. Fluid Mech. 271, 285 (1994)*