CUDA Implementation of the Lattice Boltzmann Method

CSE 633 Parallel Algorithms

Andrew Leach

University at Buffalo

2 Dec 2010

Motivation

- The Lattice Boltzmann Method(LBM) solves the Navier-Stokes equation accurately and efficiently.

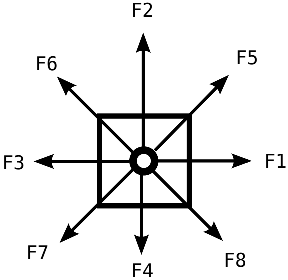
Motivation

- The Lattice Boltzmann Method(LBM) solves the Navier-Stokes equation accurately and efficiently.
- Uniformity makes it easy to parallelize.

Motivation

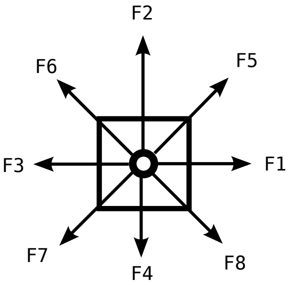
- The Lattice Boltzmann Method(LBM) solves the Navier-Stokes equation accurately and efficiently.
- Uniformity makes it easy to parallelize.
- High volume of simple calculations make it ideal for GPGPU computing.

LBM Degrees of Freedom



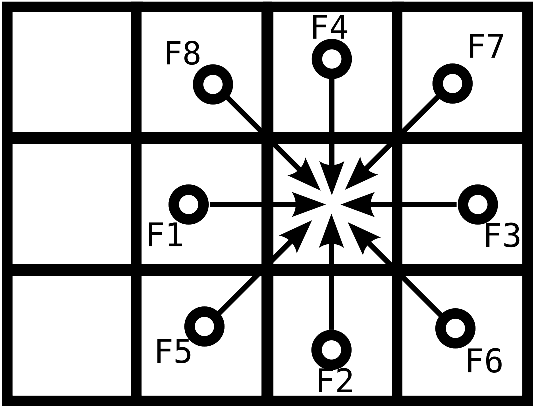
- Each lattice point has an associated mass density

LBM Degrees of Freedom



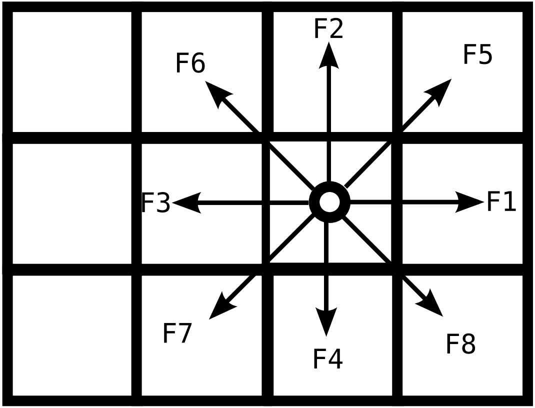
- Each lattice point has an associated mass density
- This mass density is projected in 9 directions

LBM Stream



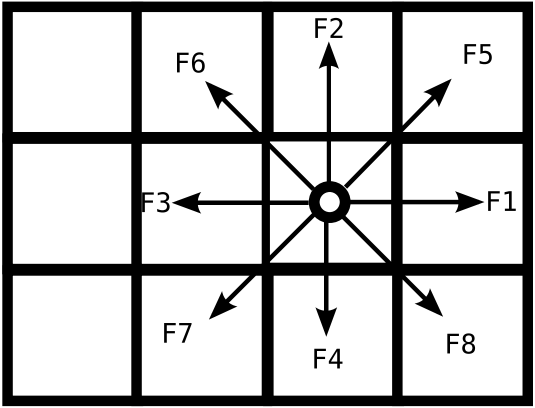
- At each time step, each neighbor passes mass density

LBM Collision



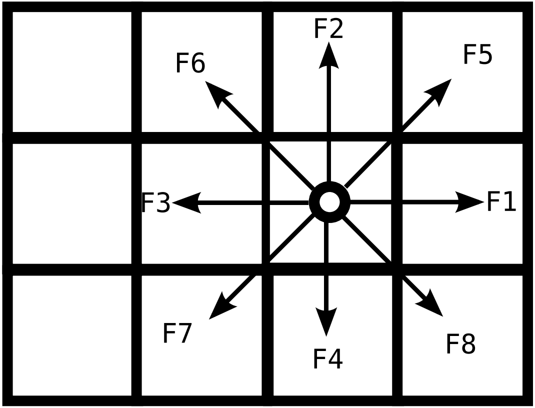
- Collision occurs with the accepted mass densities

LBM Collision



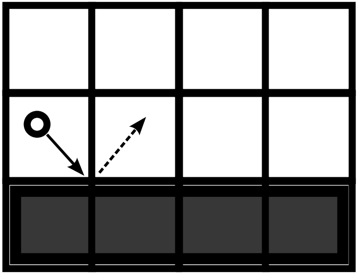
- Collision occurs with the accepted mass densities
- Equillibrium condition is solved

LBM Collision



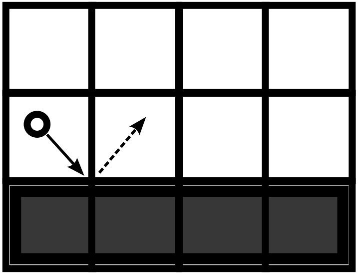
- Collision occurs with the accepted mass densities
- Equillibrium condition is solved
- New projected mass densities are assigned

LBM Boundary Conditions



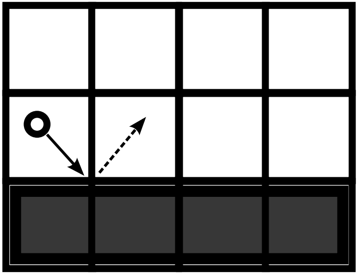
- Bounceback is implemented at solid boundaries

LBM Boundary Conditions



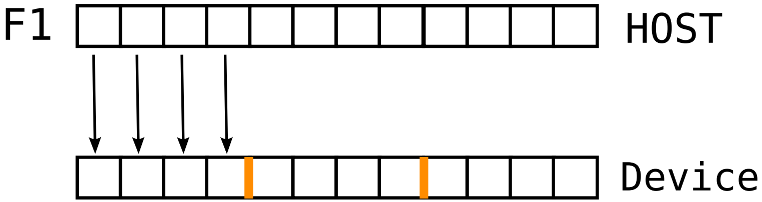
- Bounceback is implemented at solid boundaries
- The inlet has predetermined mass density

LBM Boundary Conditions



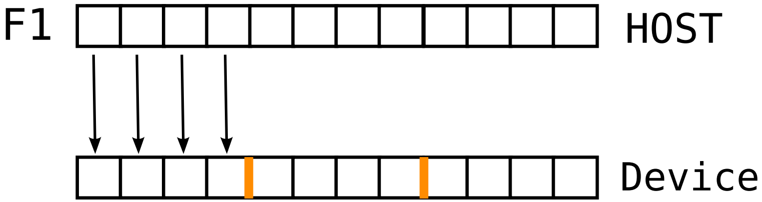
- Bounceback is implemented at solid boundaries
- The inlet has predetermined mass density
- The outlet accepts outward flow

Code: Data Structures



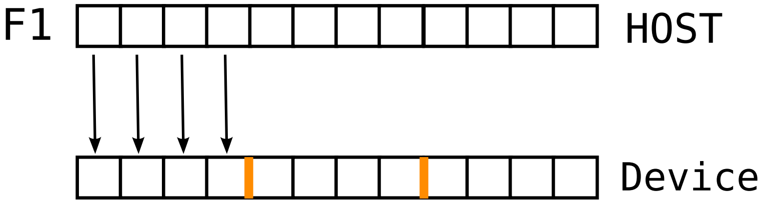
- Data initialized as an array on host

Code: Data Structures

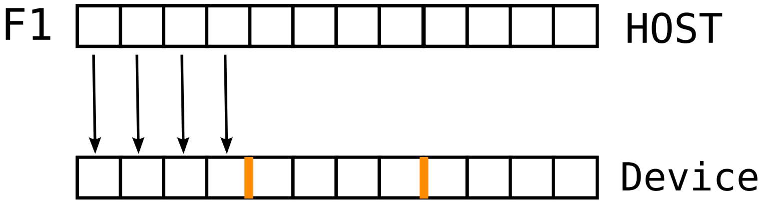


- Data initialized as an array on host
- Pitch stores the width of a row in memory, determined by CudaMallocPitch()

Code: Data Structures



- Data initialized as an array on host
- Pitch stores the width of a row in memory, determined by CudaMallocPitch()
- Memory is allocated on the device linear memory with CudaMallocArray()



- Data initialized as an array on host
- Pitch stores the width of a row in memory, determined by CudaMallocPitch()
- Memory is allocated on the device linear memory with CudaMallocArray()
- Array copied from host to device with CudaMemcpy2D()

Code: Textures

- The stream step requires a lot of data retrieval

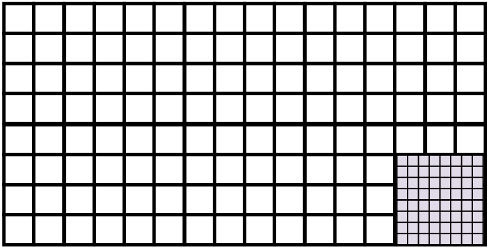
Code: Textures

- The stream step requires a lot of data retrieval
- Texture memory has fast retrieval but limited space

Code: Textures

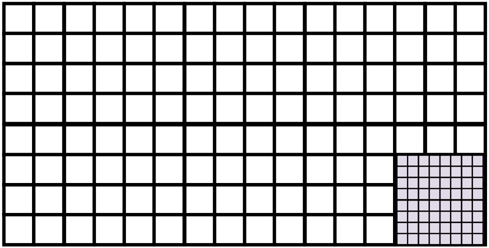
- The stream step requires a lot of data retrieval
- Texture memory has fast retrieval but limited space
- Use cudaBindTextureToArray() to copy data as a texture

Code: Kernels



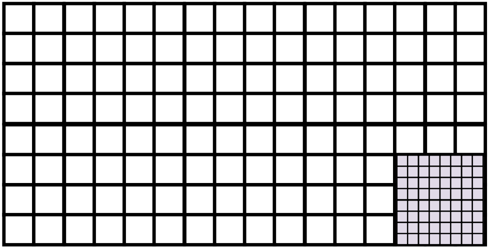
- A kernel is launched on a grid of blocks

Code: Kernels



- A kernel is launched on a grid of blocks
- Each block consists of threads which will independently run the kernel(SIMD)

Code: Kernels



- A kernel is launched on a grid of blocks
- Each block consists of threads which will independently run the kernel(SIMD)
- What follows is the Kernel for the stream() method. This example utilizes a lock-step texture look up.

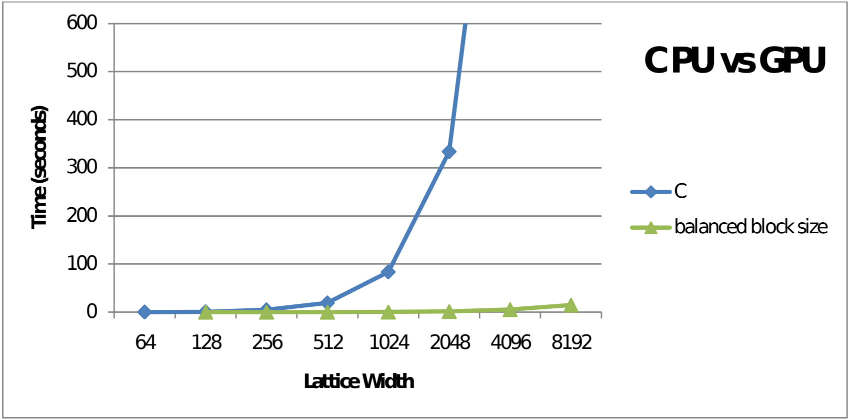
Code: Stream

```
//Define stream kernel//
__global__ void stream_kernel(int pitch, float *f1, float *f2,
            float *f3, float *f4, float *f5, float *f6, float *f7, float *f8){

  int i, j, id;
  i = blockDim.x * blockIdx.x + threadIdx.x;
  j = blockDim.y * blockIdx.y + threadIdx.y;
  id = i + j * (pitch/sizeof(float));

  //Gather adjacent f's for the stream step//
  f1[id] = tex2D(f1_tex, (float) i-1, (float) j);
  f2[id] = tex2D(f2_tex, (float) i, (float) j-1);
  f3[id] = tex2D(f3_tex, (float) i+1, (float) j);
  f4[id] = tex2D(f4_tex, (float) i, (float) j+1);
  f5[id] = tex2D(f5_tex, (float) i-1, (float) j-1);
  f6[id] = tex2D(f6_tex, (float) i+1, (float) j-1);
  f7[id] = tex2D(f7_tex, (float) i+1, (float) j+1);
  f8[id] = tex2D(f8_tex, (float) i-1, (float) j+1);

}
```

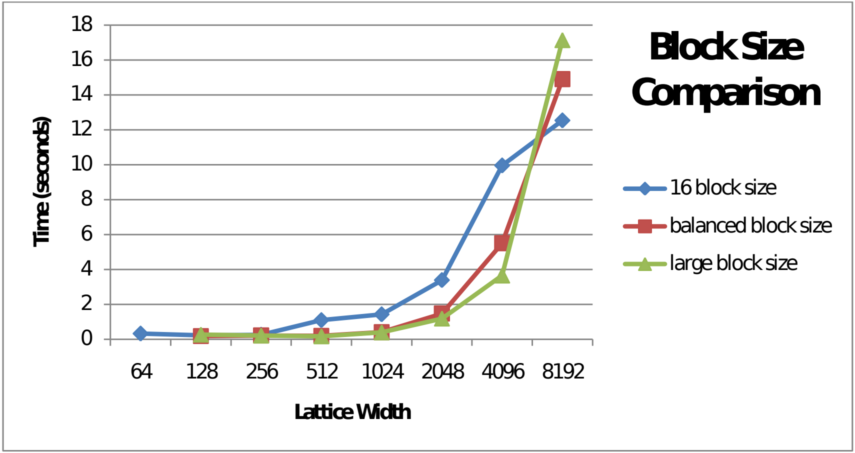
Runtime Analysis

The following slides contain graphs comparing run times for the LBM on a laptop with 1.3 GHZ processor running sequential C code and a single Tesla GPU running parallel code in CUDA. The change in performance based on block size is also explored.

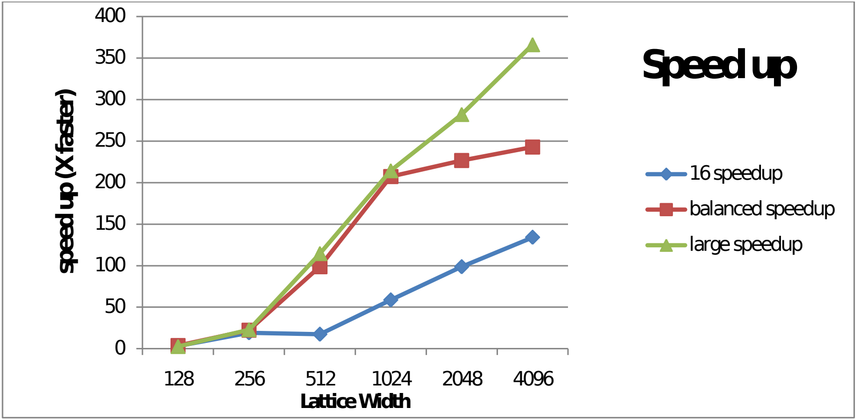
Sequential vs Parallel



Sequential vs Parallel



Sequential vs Parallel



Thank You

Thanks to Dr.Graham Pullan from Cambridge University for letting me use and modify his code.

Bibliography

Alexander Wagner, A Practical Introduction to the Lattice Boltzmann Method. North Dakota State University, March 2008.

Graham Pullan, A 2D Lattice Boltzmann Flow Solver Demo. http://www.many-core.group.cam.ac.uk/projects/LBdemo.shtml, University of Cambridge.