Early experience on running GPU-based Lattice Boltzmann simulations on POWER8 systems

Sebastiano Fabio Schifano
University of Ferrara and INFN-Ferrara

PADC Opening Workshop

October 12-13, 2015
Jülich, Germany
Outline

- lattice boltzmann at glance
- Multi-GPU implementation details: 1D vs 2D data-domain tiling
- performance results
- conclusions
The D2Q37 Lattice Boltzmann Model at Glance

- Lattice Boltzmann method (LBM) is a class of computational fluid dynamics (CFD) methods
- Simulation of synthetic dynamics described by the discrete Boltzmann equation, instead of the Navier-Stokes equations
- A set of virtual particles called populations arranged at edges of a discrete and regular grid
- Interacting by propagation and collision reproduce – after appropriate averaging – the dynamics of fluids
- D2Q37 is a D2 model with 37 components of velocity (populations)
- Suitable to study behaviour of compressible gas and fluids optionally in presence of combustion effects
- Correct treatment of Navier-Stokes, heat transport and perfect-gas ($P = \rho T$) equations

1 Chemical reactions turning cold-mixture of reactants into hot-mixture of burnt product.
Computational Scheme of LBM

foreach time-step
    
    foreach lattice-point
        propagate();
        endfor
    
    foreach lattice-point
        collide();
        endfor
    
endfor

Embarassing parallelism

All sites processed in parallel applying in sequence propagate and collide.

Challenge

Two kernels with conflicting requirements: one memory-bound and one compute-bound.
D2Q37: propagation kernel

- require to access neighbours cells at distance 1, 2, and 3
- generate memory-accesses with **sparse** addressing patterns

This kernel is strongly memory-bound.
we simulate a 2D lattice with period-boundaries along $x$-direction

at the top and the bottom boundary conditions are enforced:

- to adjust some values at sites $y = 0 \ldots 2$ and $y = N_y - 3 \ldots N_y - 1$
- e.g. set vertical velocity to zero

This step (bc) is computed before the collision step.
D2Q37: collision kernel

- collisional operator is computed for each lattice-site

- computational intensive: for the D2Q37 model, and requires \( \geq 7600 \) DP operations

- completely local: arithmetic operations require only the populations associate to the site

This kernel is strongly compute-bound.
Memory Layout: AoS vs SoA

```c
#define N (LX*LY)
typedef struct {
    double p1; // population 1
    double p2; // population 2
    ...
    double p37; // population 37
} pop_t;

pop_t lattice[N];
```

```c
#define N (LX*LY)
typedef struct {
    double p1[N]; // population 1
    double p2[N]; // population 2
    ...
    double p37[N]; // population 37
} pop_t;

pop_t lattice;
```

- data arrangement layouts: AoS (upper), SoA (lower);
- C-struct data types: AoS (left), SoA (right).
Memory Layout: AoS vs SoA

E.g.: on NVIDIA K40 GPU:
- propagate $\approx 10X$ faster
- collide $\approx 2X$ faster
Memory Data Layout: SoA

SoA data-layout exploit data-parallelism and data-coalescing

lattice allocated with halos columns
MultiGPU Tiling

- Lattice divided over a set of GPUs
- GPUs virtually arranged in 1D-ring or 2D-grid
- require an additional step to update column and row halos: GPU-to-GPU bi-directional memory copy
- we use 1 MPI-rank per GPU: simplify programming and exploit CUDA-aware MPI features

MPI solutions for GPUs:

- MPI standard libraries: OpenMPI, MVAPICH2, ...
- CUDA-aware MPI: integration of CUDA support into MPI (use of GPU-memory pointers)
- GPUDirect and RDMA: optimization of communications over IB-networks
MultiGPU: 1D-Tiling

- Lattice is divided into sub-lattices each allocated on a different GPU.
- Each sub-lattice is “divided” in three regions: bulk, left- and right-border.
- GPUs are virtually arranged on a ring, and each GPU exchanges borders with left- and right-neighbors.
MultiGPU 1D-Tiling: code and timeline execution

// Compute propagate over lattice bulk
prop_Bulk <<< dimGridB, dimBlockB, 0, stream[0] >>> ( ... );
bc_Bulk <<< dimGridB, dimBlockB, 0, stream[0] >>> ( ... );
collide_Bulk <<< dimGridB, dimBlockB, 0, stream[0] >>> ( ... );

// Update halo columns
pbc_c();

// Compute propagate on 3 leftmost columns
prop_L <<< dimGridLR, dimBlockLR, 0, stream[1] >>> ( ... );
bc_L <<< dimGridLR, dimBlockLR, 0, stream[1] >>> ( ... );
collide_L <<< dimGridB, dimBlockB, 0, stream[1] >>> ( ... );

// Compute propagate on 3 rightmost columns
prop_R <<< dimGridLR, dimBlockLR, 0, stream[2] >>> ( ... );
bc_R <<< dimGridLR, dimBlockLR, 0, stream[2] >>> ( ... );
collide_R <<< dimGridLR, dimBlockLR, 0, stream[2] >>> ( ... );

Computation of \textit{propagate} and \textit{collide} kernels kept separate: not straightforward to merge them in one single step.
MultiGPU Tiling: 1D vs 2D

1D-Tiling:
- \( T_{\text{comm}} \propto 2L \)
- \( T_{\text{comp}} \propto L \times \frac{L}{n} = \frac{L^2}{n} \)
- \( \frac{T_{\text{comm}}}{T_{\text{comp}}} \propto 2L \times \frac{n}{L^2} = \frac{2n}{L} \)

2D-Tiling:
- \( T_{\text{comm}} \propto \frac{4L}{\sqrt{n}} \)
- \( T_{\text{comp}} \propto \frac{L}{\sqrt{n}} \times \frac{L}{\sqrt{n}} = \frac{L^2}{\sqrt{n}} \)
- \( \frac{T_{\text{comm}}}{T_{\text{comp}}} \propto \frac{4L}{\sqrt{n}} \times \frac{n}{\sqrt{L^2}} = \frac{4\sqrt{n}}{L} \)
MultiGPU 2D-Tiling

- sub-lattices mapped on a 2D-grid of GPUs
- updating halos requires data from adjacent and non-adjacent (corner) nodes
- communications are scheduled in a two-step sequence:
  1. move data along X-dimension (non-contiguous elements)
  2. move data along Y-dimension (contiguous elements)
- performing Y-communications after X’s are completed, update of the four corners is included in the sequence
MultiGPU 2D-Tiling

Processing of each sub-lattice is divided in several regions:

- left- and right-borders (3 columns) are processed after update of left- and right-halos is completed
- top- bottom-borders (3 rows) are processed after update of top-, bottom-, left- and right-halos is completed
- bulk does not have dependences with any regions and can be processed in parallel

computation of *propagate* and *collide* kernels merged on bulk region.
Update Halos through CUDA-aware MPI operations

// update contiguous halo using CUDA-aware MPI
MPI_Sendrecv(
    f_d + SRC_OFF, SIZE, MPI_DOUBLE, mpi_left, 0,
    f_d + DST_OFF, SIZE, MPI_DOUBLE, mpi_right, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
);

// update non-contiguous halo using CUDA-aware MPI
for( ii=0; ii< (SIZE); ii++ )
    MPI_Sendrecv(
        f_d + (ii*STRIDE), 1, MPI_DOUBLE, mpi_left, 0,
        f_d + (ii*STRIDE), 1, MPI_DOUBLE, mpi_right, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );

- used to update contiguous halos (Y-direction): in our case requires 26 MPI operations for each halo region
- can not be used for updating non-contiguous halos (X-direction):
  - moving one cell at a time has dramatic latency overheads
  - using MPI-derived data-types is curently not efficient (both MVAPICH2 and OpenMPI)
Update Halos through Gather/Scatter Kernels

1. allocate send and receive buffers onto GPU memory
2. pack/gather non-contiguous row elements into send-buffer
3. move send-buffer through CUDA-aware MPI
4. unpack/scatter data from receive-buffer to halo-region
CUDA-aware MPI vs Gather/Scatter solution

CUDA-aware MPI solution:
- overlap with GPU computation is possible
- latencies issues: updating contiguous halos requires 26 MPI communication of small buffer as number of GPUs increases
- for buffers smaller than 4 KB ($L \approx 512$) CUDA-aware MPI is not enabled and GPU buffers are moved through memory-host (D2H + H2D).

Gather/Scatter solution:
- execution of gather/scatter kernels may not overlap as GPU resources are busy to run application-kernels
- smaller latency: running of gather/scatter kernels is faster (compared to MPI communications)
- require to execute one MPI operation of a larger buffer

Final implementation of our code uses Gather/Scatter solution for both contiguous and non-contiguous halos.
2D-Tiling: Code Scheduling driven by CPU

// update non-contiguous halos
pack_top <<< >>;  pack_bottom <<< >>;
MPI_Sendrecv( );  MPI_Sendrecv( );
unpack_top <<< >>;  unpack_bottom <<< >>;

// update contiguous halos: pack right/left borders
pack_right <<< >>;  pack_left <<< >>

// run propagateAndCollide over Bulk (async processing)
propagateCollideBulk <<< >>

// update contiguous halos: perform MPI operations
MPI_Sendrecv( );  MPI_Sendrecv( );

// update contiguous halos: unpack right/left halos
unpack_left <<< >>;  unpack_right <<< >>

// process left/right borders
propagateCollideL <<< >>;  propagateCollideR <<< >>

// process top/bottom borders
if (uppermost−rank){
    propagateT <<< >>;  bcT <<< >>;  collideT <<< >>
} else {
    propagateCollideT <<< >>;
}

if (lowermost−rank){
    propagateB <<< >>;  bcB <<< >>;  collideB <<< >>
} else {
    propagateCollideB <<< >>;
}
2D-Tiling: Execution Timeline

Lattice 128x4096, GPU-grid 2x1:

MPI transfers to update contiguous halos fully overlap with computation.
Update of non-contiguous halos can not overlap with computation.
propagate moves $37 \times 2$ (read+write) double-precision values per site:

$$2 \times 37 \times 8 \text{ B} = 592 \text{ B/site}$$

collide performs 6472 DP floating-point operations per site:

- $\approx 10\%$ are \textit{Add}
- $\approx 20\%$ are \textit{Mul}
- $\approx 70\%$ are \textit{fused-multiply-add} (FMA)
### Performance Results: 1D tiling

**4096 × 4096, 2 GPU, 1000 iterations**

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>68.18%</td>
<td>72.680s</td>
<td>1000</td>
<td>72.680ms</td>
<td>72.288ms</td>
<td>73.083ms</td>
<td>collideB(double*, double const*)</td>
</tr>
<tr>
<td>25.32%</td>
<td>26.985s</td>
<td>1000</td>
<td>26.986ms</td>
<td>26.672ms</td>
<td>27.169ms</td>
<td>propagateB(double*, double const*)</td>
</tr>
<tr>
<td>4.31%</td>
<td>4.5905s</td>
<td>1000</td>
<td>4.5906ms</td>
<td>4.4920ms</td>
<td>4.6863ms</td>
<td>bcB(double*, double*)</td>
</tr>
<tr>
<td>0.77%</td>
<td>820.90ms</td>
<td>52020</td>
<td>15.780us</td>
<td>768ns</td>
<td>222.26ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>0.61%</td>
<td>655.38ms</td>
<td>52001</td>
<td>12.603us</td>
<td>5.3760us</td>
<td>202.47ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>0.28%</td>
<td>297.19us</td>
<td>1000</td>
<td>297.19us</td>
<td>266.65us</td>
<td>352.09us</td>
<td>collideL(double*, double const*)</td>
</tr>
<tr>
<td>0.26%</td>
<td>278.18us</td>
<td>1000</td>
<td>278.18us</td>
<td>256.00us</td>
<td>338.36us</td>
<td>collideR(double*, double const*)</td>
</tr>
<tr>
<td>0.09%</td>
<td>97.625us</td>
<td>1000</td>
<td>97.624us</td>
<td>75.039us</td>
<td>110.27us</td>
<td>propagateL(double*, double const*)</td>
</tr>
<tr>
<td>0.06%</td>
<td>67.077us</td>
<td>1000</td>
<td>67.077us</td>
<td>56.256us</td>
<td>83.263us</td>
<td>propagateR(double*, double const*)</td>
</tr>
<tr>
<td>0.06%</td>
<td>61.191us</td>
<td>1000</td>
<td>61.190us</td>
<td>54.080us</td>
<td>69.215us</td>
<td>bcR(double*, double*)</td>
</tr>
<tr>
<td>0.06%</td>
<td>60.036us</td>
<td>1000</td>
<td>60.035us</td>
<td>53.088us</td>
<td>67.871us</td>
<td>bcL(double*, double*)</td>
</tr>
</tbody>
</table>

- **propagate BW ≈ 184 GB/s (ε ≈ 64%)**
- **collide P ≈ 747 GFLOP/s (ε ≈ 45%)**
- **Wct: 104.72 s (104.72 ms/iter), Tswap: 2.48 s (2.48 ms/iter) P: 1036.85 GFLOPs, MLUP/s: 160.20**

S. F. Schifano (Uni and INFN of Ferrara)
Performance Results: 2D tiling

4096 × 4096, 2 × 1 GPU-grid, 1000 iterations

--52900-- Profiling result:

<table>
<thead>
<tr>
<th>Name</th>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>propagateCollideBulk(double*, double const *, int)</td>
<td>95.33%</td>
<td>89.454s</td>
<td>1000</td>
<td>89.454ms</td>
<td>89.178ms</td>
<td>89.733ms</td>
</tr>
<tr>
<td>[CUDA memcpy HtoD]</td>
<td>0.72%</td>
<td>678.02ms</td>
<td>14020</td>
<td>48.361us</td>
<td>896ns</td>
<td>237.78ms</td>
</tr>
<tr>
<td>propagateT(double*, double const *)</td>
<td>0.63%</td>
<td>593.60ms</td>
<td>1000</td>
<td>504.64us</td>
<td>425.15us</td>
<td>648.09us</td>
</tr>
<tr>
<td>propagateB(double*, double const *)</td>
<td>0.54%</td>
<td>448.20ms</td>
<td>14001</td>
<td>32.011us</td>
<td>9.3110us</td>
<td>237.79ms</td>
</tr>
<tr>
<td>[CUDA memcpyDtoH]</td>
<td>0.48%</td>
<td>396.42ms</td>
<td>1000</td>
<td>379.72us</td>
<td>341.09us</td>
<td>433.28us</td>
</tr>
<tr>
<td>collideT(double*, double const *, int)</td>
<td>0.40%</td>
<td>368.87ms</td>
<td>1000</td>
<td>368.87us</td>
<td>332.89us</td>
<td>433.85us</td>
</tr>
<tr>
<td>bcB(double*, double*)</td>
<td>0.39%</td>
<td>340.82ms</td>
<td>1000</td>
<td>340.82us</td>
<td>309.18us</td>
<td>398.69us</td>
</tr>
<tr>
<td>collideB(double*, double const *, int)</td>
<td>0.36%</td>
<td>307.26ms</td>
<td>1000</td>
<td>307.26us</td>
<td>172.00us</td>
<td>489.76us</td>
</tr>
<tr>
<td>propagateCollideR(double*, double const *)</td>
<td>0.22%</td>
<td>207.26ms</td>
<td>1000</td>
<td>207.26us</td>
<td>172.00us</td>
<td>489.76us</td>
</tr>
<tr>
<td>propagateCollideL(double*, double const *)</td>
<td>0.04%</td>
<td>14.823ms</td>
<td>1000</td>
<td>14.822us</td>
<td>13.984us</td>
<td>18.144us</td>
</tr>
<tr>
<td>unpack_left(double*, double*)</td>
<td>0.03%</td>
<td>14.823ms</td>
<td>1000</td>
<td>14.822us</td>
<td>13.984us</td>
<td>18.144us</td>
</tr>
<tr>
<td>unpack_right(double*, double*)</td>
<td>0.02%</td>
<td>14.823ms</td>
<td>1000</td>
<td>14.822us</td>
<td>13.984us</td>
<td>18.144us</td>
</tr>
<tr>
<td>pack_left(double*, double*)</td>
<td>0.02%</td>
<td>14.823ms</td>
<td>1000</td>
<td>14.822us</td>
<td>13.984us</td>
<td>18.144us</td>
</tr>
<tr>
<td>pack_right(double*, double*)</td>
<td>0.02%</td>
<td>14.823ms</td>
<td>1000</td>
<td>14.822us</td>
<td>13.984us</td>
<td>18.144us</td>
</tr>
</tbody>
</table>

- **propagateCollide BW** ≈ 55 GB/s
- **propagateCollide P** ≈ 607 GFLOP/s
- **Wct**: 91.43 s (91.43 ms/iter), **Tswapnc**: 0.00 s (0.03 us/iter), **P**: 1187.54 GFLOPs, **MLUP/s**: 183.49
Performance Results
Simulation of the Rayleigh-Taylor (RT) Instability

Instability at the interface of two fluids with different density and temperature

A cold-dense fluid over a less dense and warmer fluid triggers an instability that mixes the two fluid-regions (till equilibrium is reached).