LBM on Multi- and Many-core acritectures

Sebastiano Fabio Schifano

University of Ferrara and INFN-Ferrara

The Lattice Boltzmann Method

- 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.
- The key idea:
  - 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.

Relevant features:

- “Easy” to implement complex physics.
- Good computational efficiency on MPAs.

The D2Q37 Lattice Boltzmann Model

- Correct treatment of:
  - Navier-Stokes equations of motion
  - heat transport equations
  - perfect gas state equation ($P = \rho T$)
- D2 model with 37 components of velocity
- Suitable to study behaviour of **compressible** gas and fluids
- optionally in presence of **combustion**[1] effects.

[1] chemical reactions turning cold-mixture of reactants into hot-mixture of burnt product.

LBM Computational Scheme

```
foreach time-step
  foreach lattice-point

    propagate();

    collide();

  endfor
endfor
```

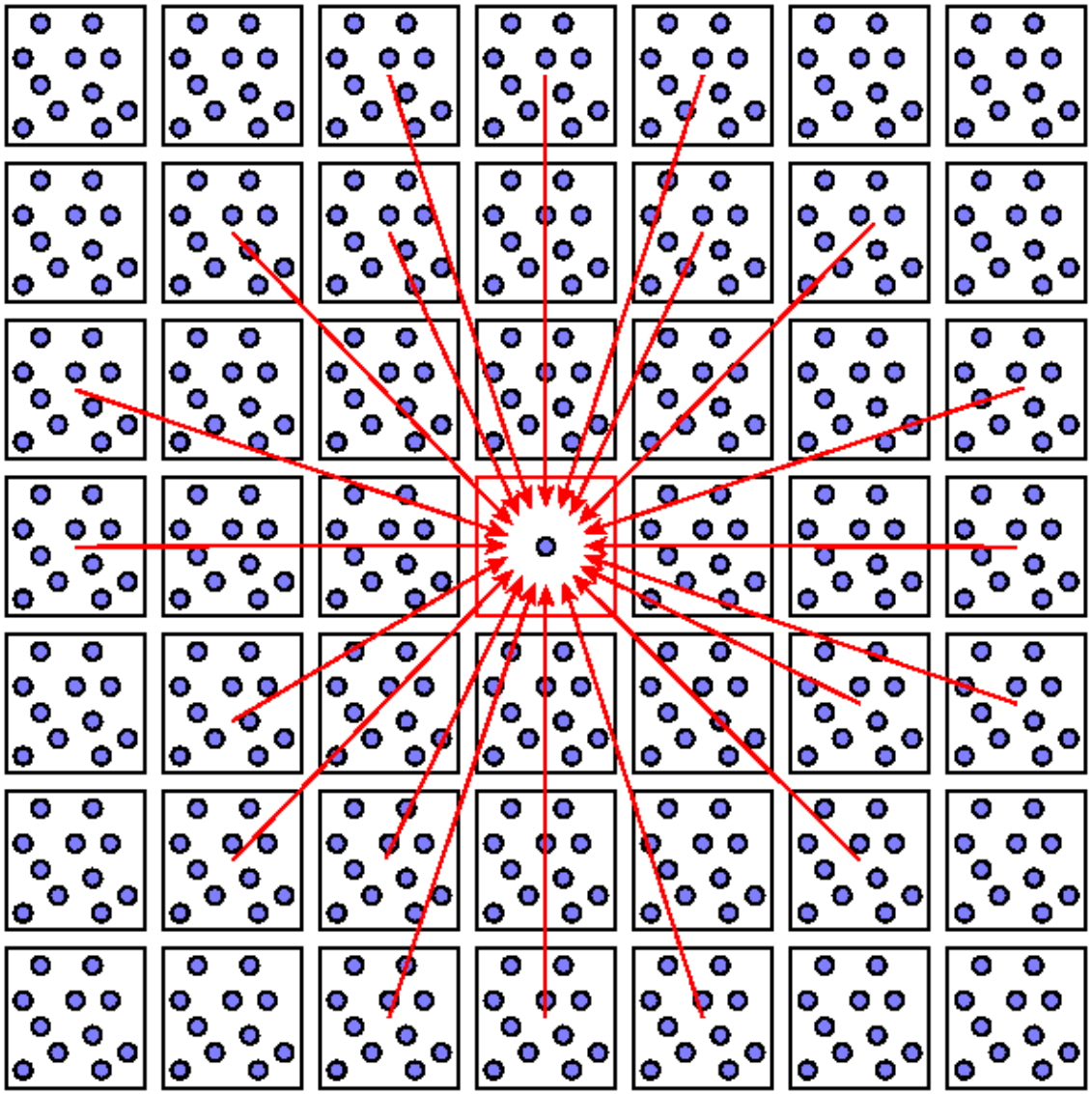
Embarrassing parallelism

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

Challenge

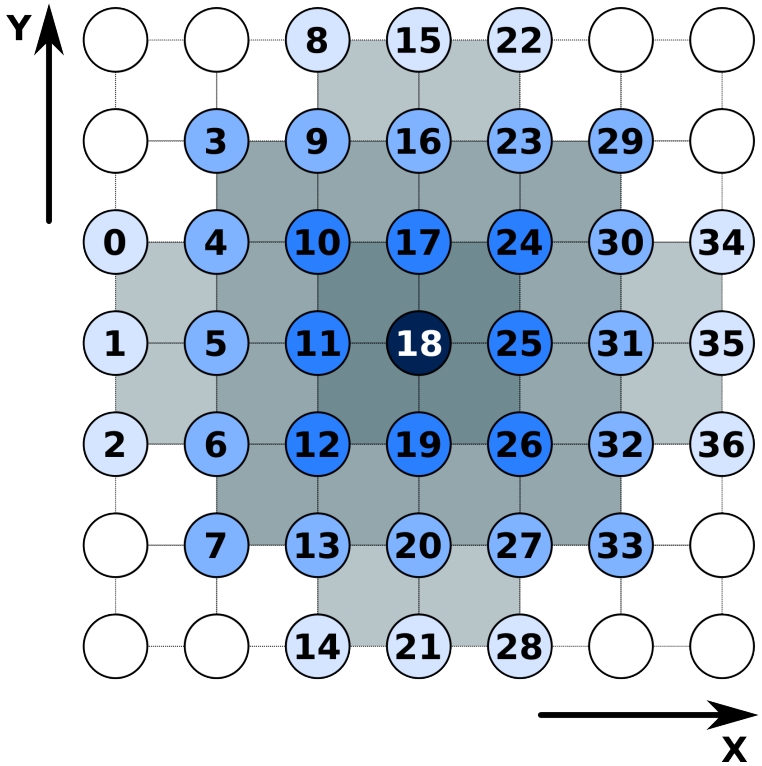
Efficient implementation on computing systems to exploit a large fraction of peak performance.

D2Q37 propagation scheme



Gather 37 populations from 37 different lattice-sites.

D2Q37 propagation



- applies to each lattice-cell,
- requires to access cells at distance 1,2, and 3,
- gathers populations at the edges of the arrows at the center point,
- performs memory accesses with **sparse** addressing patterns.

D2Q37 collision

- collision is computed to each lattice-cell
- computational intensive: for the D2Q37 model, and requires $>$ **7600** DP operations
- completely local: arithmetic operations require only the populations associate to the site

Implementation on Sandybridge CPUs

N. sockets	2
CPU family	Xeon E5-2680
frequency	2.7 GHz
cores/socket	8
L3-cache/socket	20 MB
Peak Perf. DP	345.6 GFlops
Peak Memory Bw	85.3 GBytes

- *Advanced Vector Extensions* (256-bit)
- **Symmetric Multi-Processor** (SMP) system:
  - programming view: single processor with 16-24 cores
  - memory address space shared among cores
- **Non Uniform Memory Access** (NUMA) system:
  memory access time depends on relative position of thread and data allocation.

$$T_{exe} \geq \max\left(\frac{W}{F}, \frac{I}{B}\right) = \max\left(\frac{7666}{345.2}, \frac{592}{85.312}\right) \text{ ns} = \max(22.2, 6.94) \text{ ns}$$

Relevant Optimization

Applications approach peak performance if hardware features are exploited by the code:

- **core parallelism**: all cores has to work in parallel, e.g. running different functions or working on different data-sets (MIMD/multi-task or SPMD parallelism);
- **vector programming**: each core has to process data-set using vector (streaming) instructions (SIMD parallelism);
- **cache data reuse**: data loaded into cache has to be reused as long as possible to save memory access;
- **NUMA control**: time to access memory depends on the relative allocation of data and threads.

Core Parallelism

Standard POSIX Linux `pthread` library is used to manage parallelism:

```
for ( step = 0; step < MAXSTEP; step++ ) {

  if ( tid == 0 || tid == 1 ) {
    comm();       // exchange borders
    propagate(); // apply propagate to left- and right-border
  } else {
    propagate(); // apply propagate to the inner part
  }

  pthread_barrier_wait(...);

  if ( tid == 0 )
    bc(); // apply bc() to the three upper row-cells

  if ( tid == 1 )
    bc(); // apply bc() to the three lower row-cells

  pthread_barrier_wait(...);

  collide();  // compute collide()

  pthread_barrier_wait(..);
}
```

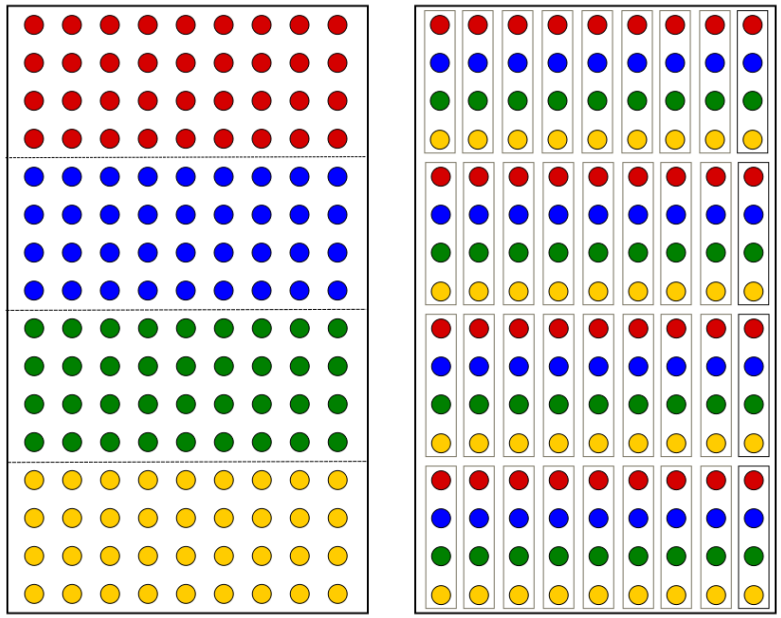
others: MPI, OpenMP, ...

Vector Programming

Components of 4 cells are combined/packed in a AVX vector of 4-doubles

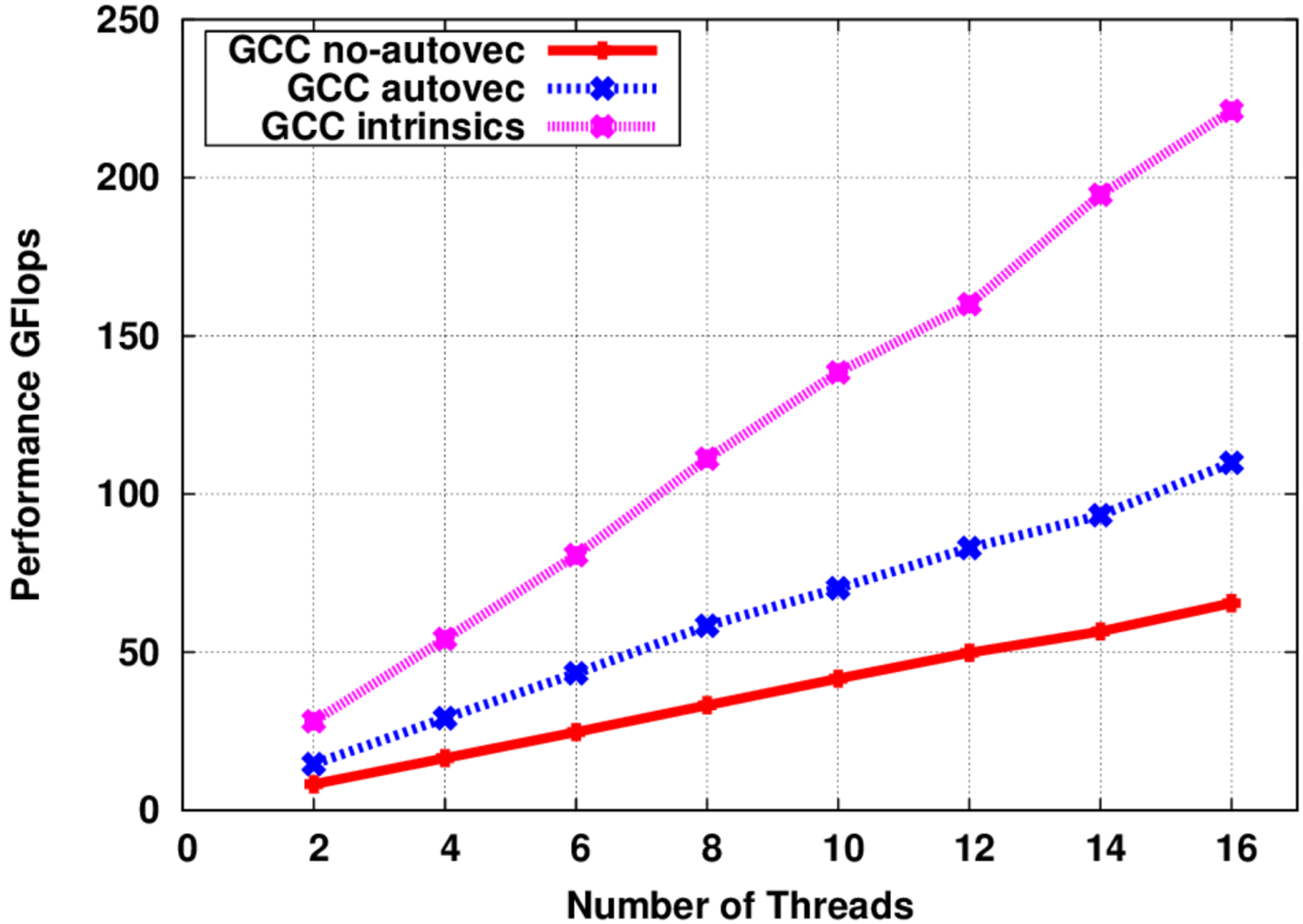
GCC and ICC vectorization by

- enabling auto-vectorization flags, e.g. `-mAVX`, `-mavx`
- using the `_mm256` vector type and intrinsics functions (`_mm256_add_pd()`,...)
- using the `vector_size` attribute (only GCC)



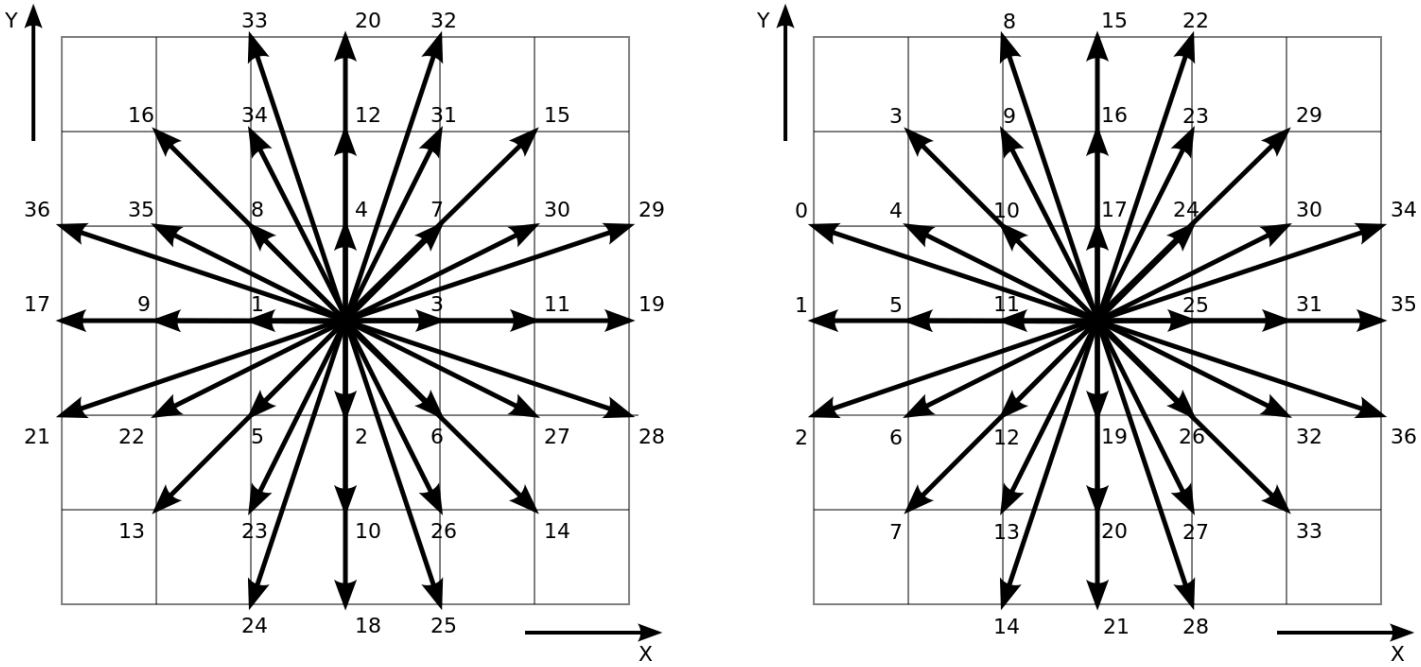
```
typedef double fourD __attribute__ ((vector_size(4*sizeof(double))));

typedef struct {
  fourD  p1;       // population 1
  fourD  p2;       // population 2
  ...
  fourD p37;       // population 37
} v_pop_type;
```

Collide Performance



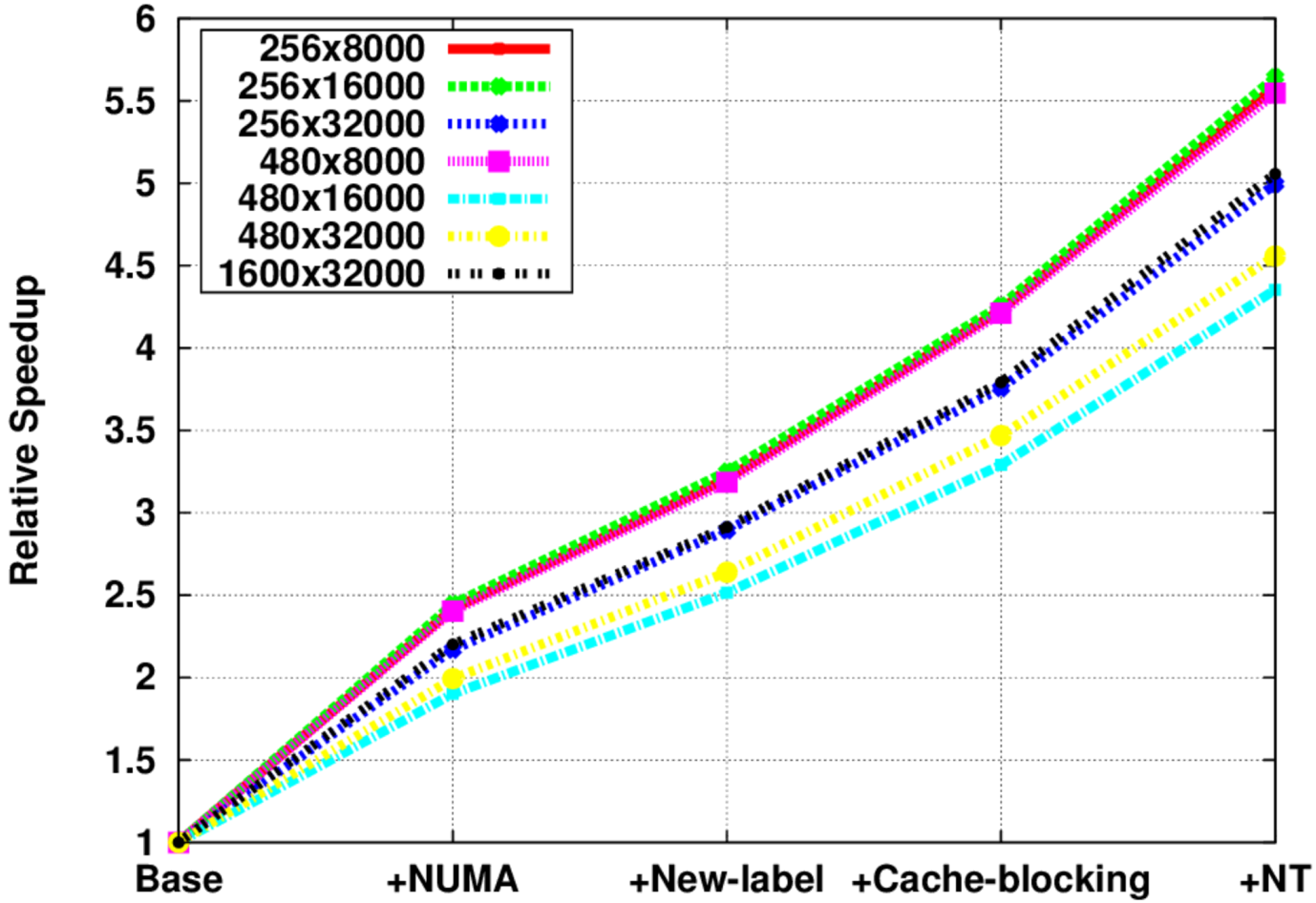
- GCC no-autovec: 18% of peak
- GCC autovec: 31% of peak
- GCC intrinsics: 62% of peak

Optimization of Propagate



- **cache data-reuse**: reordering of populations allows a better CACHE-reuse and improves performances of propagate;
- **NUMA control**: using the NUMA library to control data and thread allocation avoids overheads in accessing memory;
- **cache blocking**: load the cache with a small data-subset and work on it as long as possible;
- **non-temporal instructions**: store data directly to memory without request of *read-for-ownership*, and save time.

Optimization of Propagate



version including all optimizations performs at $\approx$ 58 MB/s, $\approx$ 67% of peak and very close to memory-copy (68.5 MB/s).

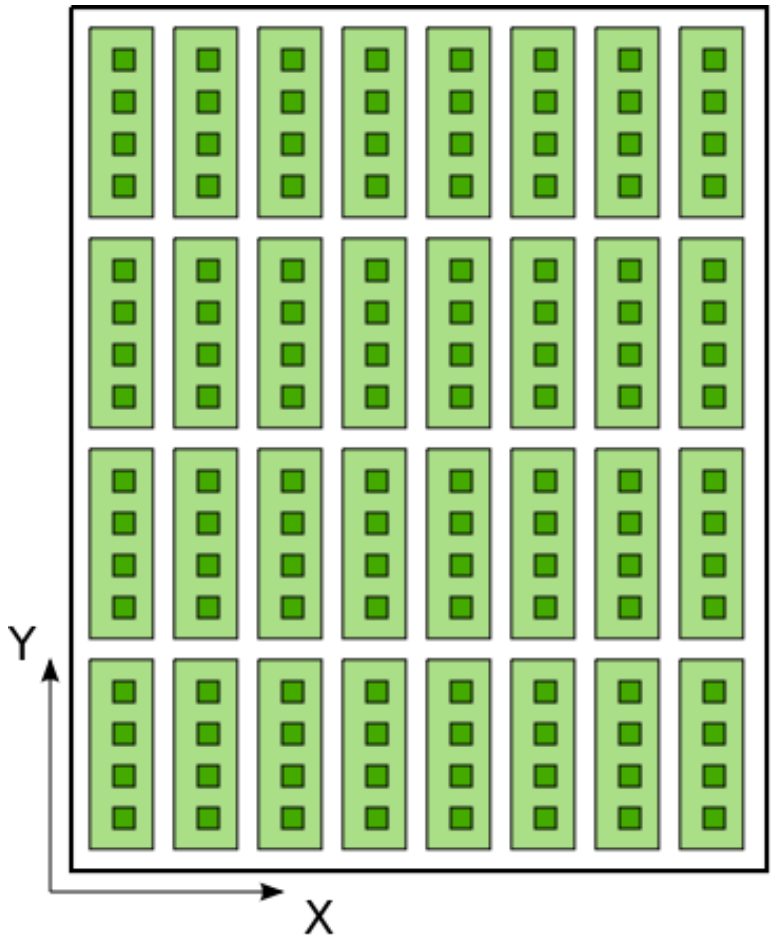
GPU Implementation: Host Program

```
typedef struct {
  double p1 [NSITES]; // population  1 array
  double p2 [NSITES]; // population  2 array
  ...
  double p37[NSITES]; // population 37 array
} pop_type;

foreach ( timestep=0; timestep < MAX_STEP; timestep++ ) {
  comm ( );                                  // exchange Y borders
  propagate <<< grid, threads >>> ( );       // run propagate
  bc        <<< grid, threads >>> ( );       // run bc
  collide   <<< grid, threads >>> ( );       // run collide
}
```

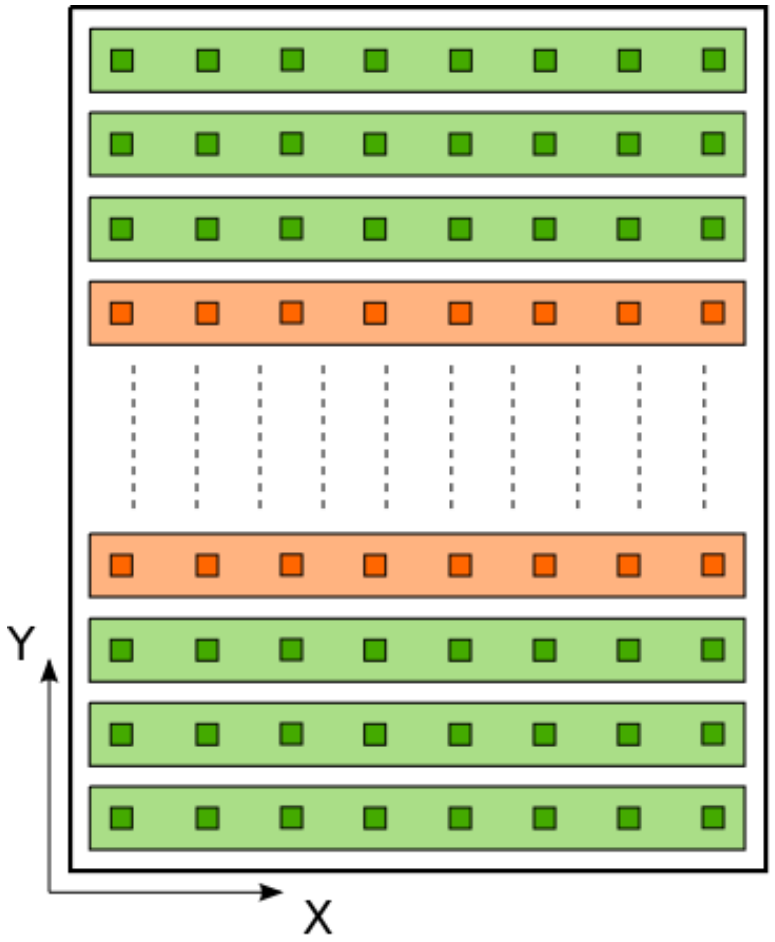
The lattice is stored as a Structure of Arrays (SOA) to exploit data-coalescing.

GPU Implementation: CUDA Grids Layouts

Physical lattice of 8 $\times$ 16 sites: each CUDA-thread process a lattice-point.

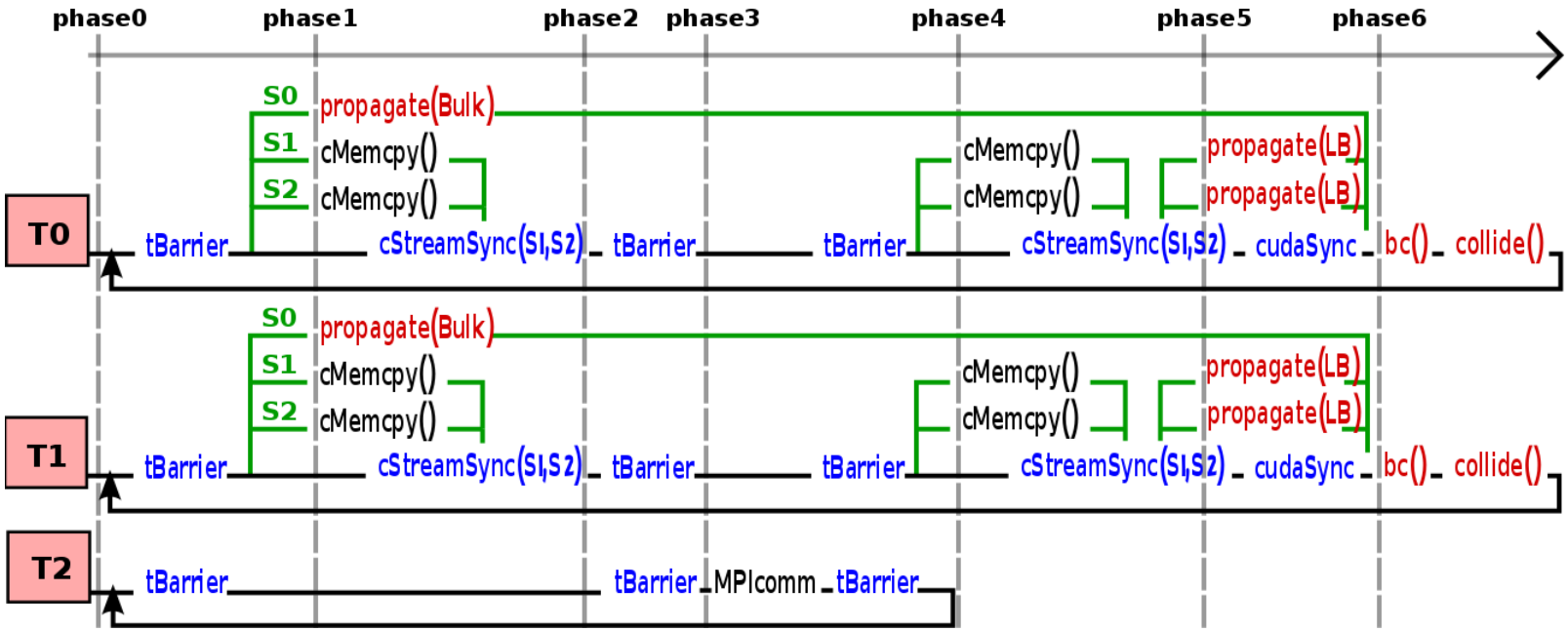


`propagate()` and `collide()`



`bc()`.

GPU Implementation: Flow Diagram



Host-CPU runs three threads:

- T0 and T1 manage runs on GPUs
- T2 executes communication with neighbour nodes

Each GPU runs three streams:

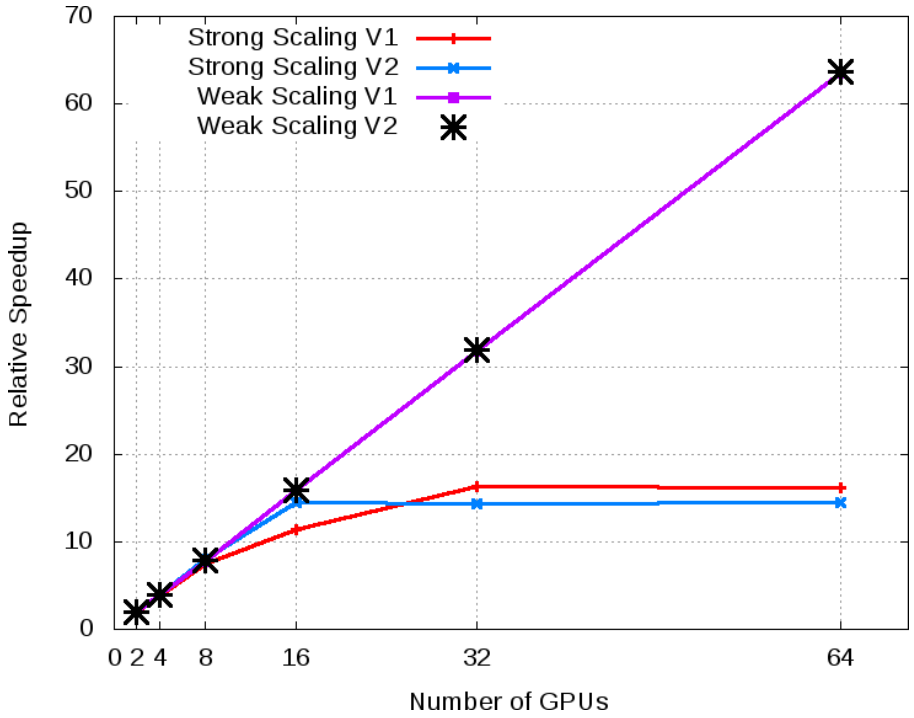
- S0 applies propagate to the bulk
- S1 and S2 copies borders to and from memory buffers

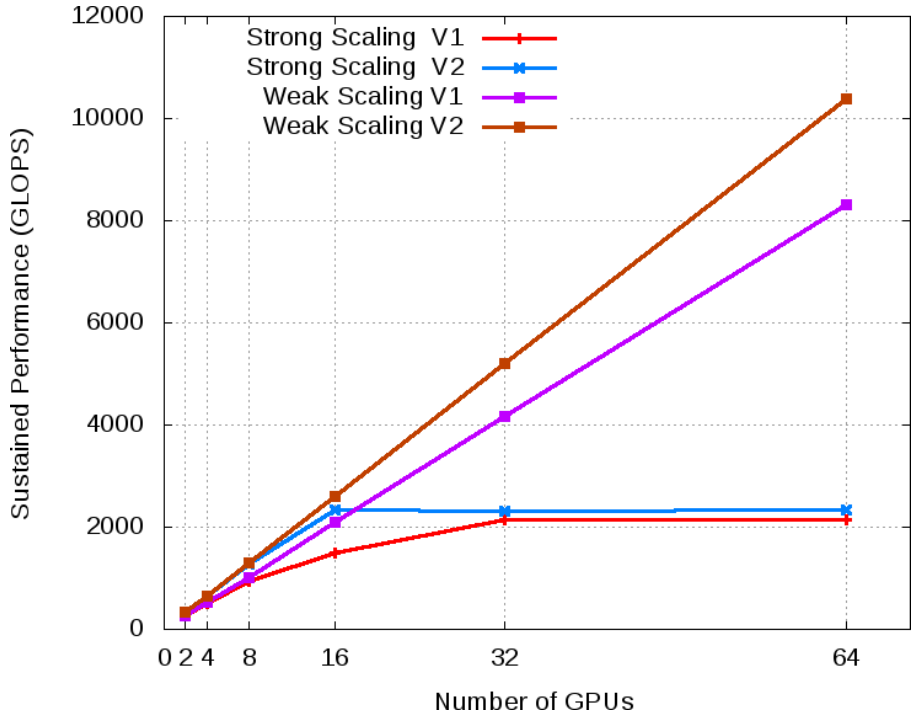
Benchmark Results

	C2050	2-WS	2-SB	xxx	yyy
propagate GB/s	84	17.5	60	120	52
$\mathcal{E}$	58%	29%	70%	58%	16%
collide GF/s	205.4	88	220	350	274
$\mathcal{E}$	41%	55%	63%	30%	27%
ξ (collide)	–	1.19	1.27	–	0.52

$$\xi = \frac{P}{N_c \times v \times f}$$

Results: Code production @PLX

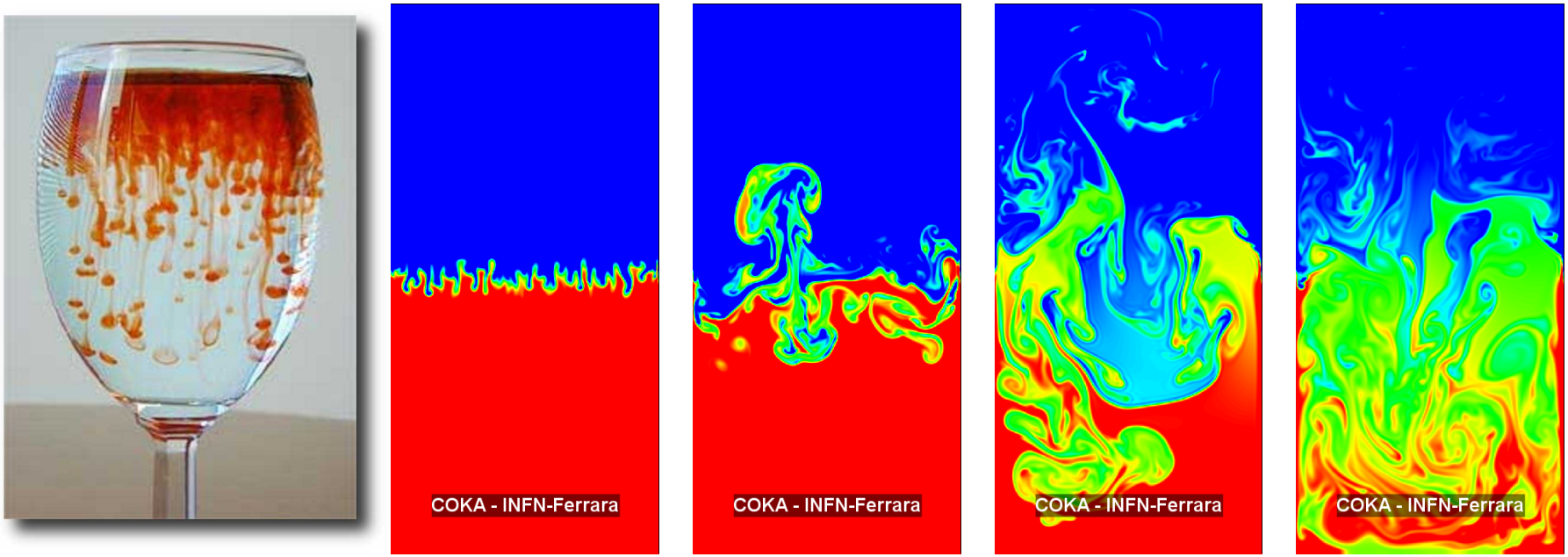




- strong regime: code runs on a lattice size $L_x \times L_y = 1024 \times 7168$
- weak-regime: sub-lattice size on each node is $L_x \times L_y = 254 \times 14464$

Simulation of the Rayleigh-Taylor (RT) Instability

Instability at the interface of two fluids of different densities triggered by gravity.



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