Portability, Performance and Scalability of LB Codes for Accelerator based Architectures

E Calore, S F Schifano, R Tripiccione

Sebastiano Fabio Schifano

University of Ferrara and INFN-Ferrara
ITALY

NVIDIA-Lab at Jülich Workshop

October 8-9, 2014

Jülich, GERMANY

Outline

1. LBM at glance, D2Q37 model
2. Programming frameworks
3. Implementation details
4. Results and conclusions

We addressed the issue of portability of code across several computing architectures preserving performances.

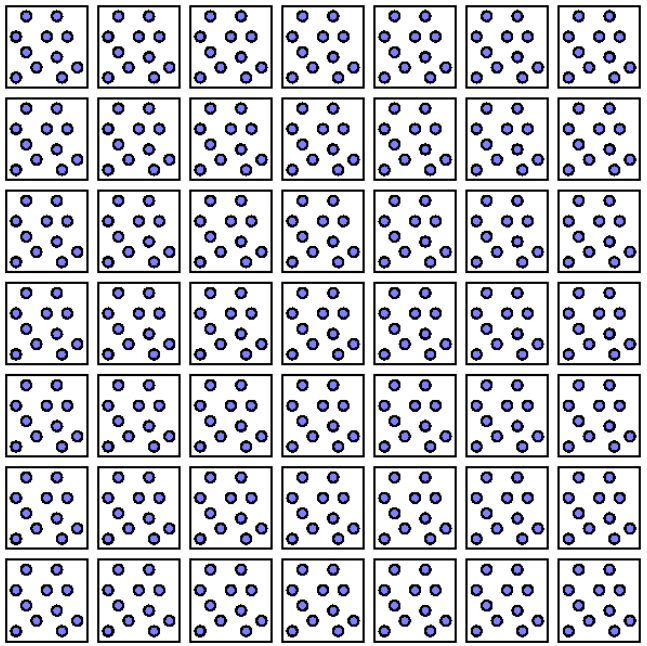
The D2Q37 Lattice Boltzmann Model

- 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** [1] 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
```



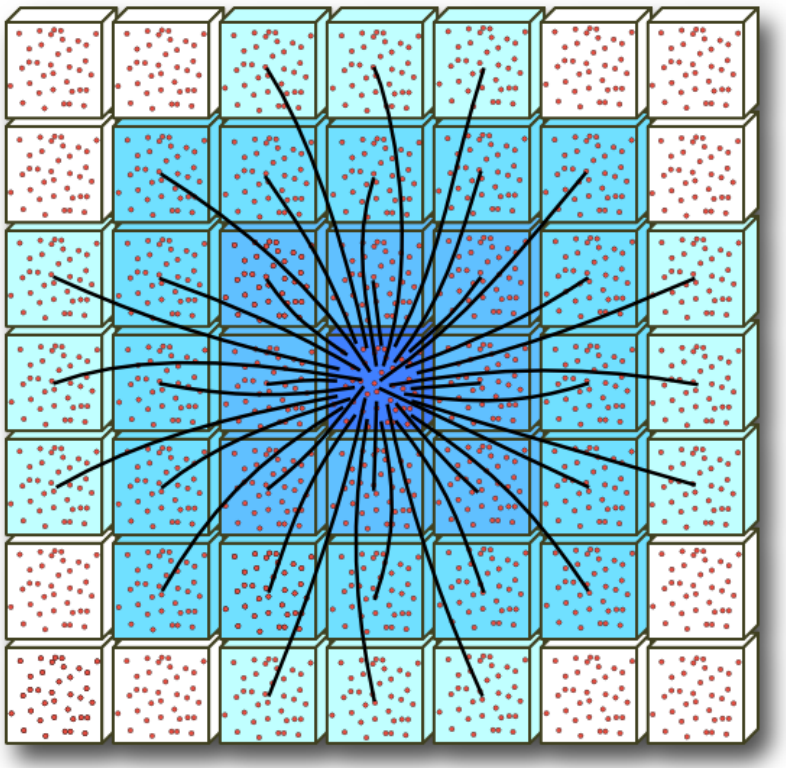
Embarassing parallelism

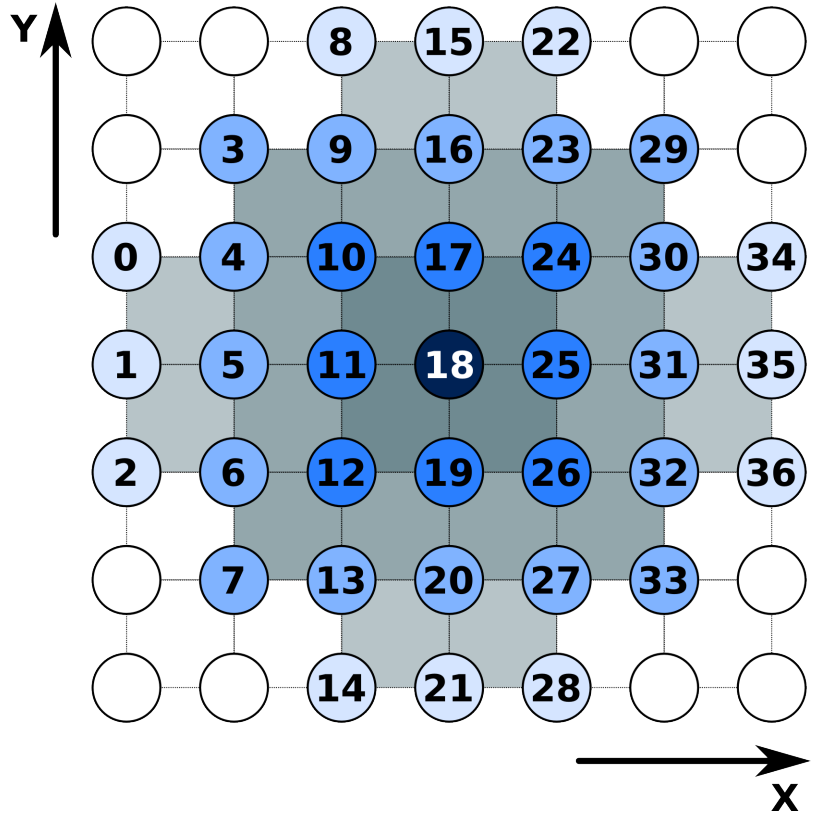
All sites can be processed in parallel applying in sequence propagate and collide.

Challenge

Design an efficient implementation able exploit a large fraction of available peak performance.

D2Q37: propagation scheme



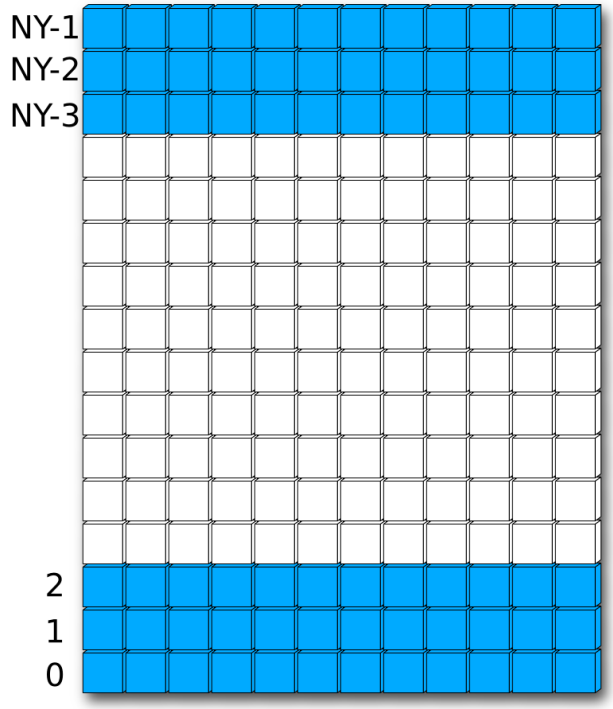


- perform accesses to neighbour-cells at distance 1,2, and 3
- generate memory-accesses with **sparse** addressing patterns

D2Q37: boundary-conditions

After propagation, boundary conditions are enforced at top and bottom edges of the lattice.

- 2D-lattice with period-boundaries along X-direction
- top and 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



At left and and right edges we apply periodic boundary conditions.

D2Q37 collision

- collision is computed at each lattice-cell after computation of boundary conditions
- computational intensive: for the D2Q37 model requires $\approx$ 7500 DP floating-point operations
- completely local: arithmetic operations require only the populations associate to the site
- computation of propagate and collide kernels are kept separate
- after propagate but before collide we may need to perform collective operations (e.g. divergence of of the velocity field) if we include computations conbustion effects.

Implementation: Exploit Parallelism

- process all sites in parallel
- keep two copies in memory
- vectorization
- core parallelism
- node parallelism

Implementation: Memory layout for LB, AoS vs SoA

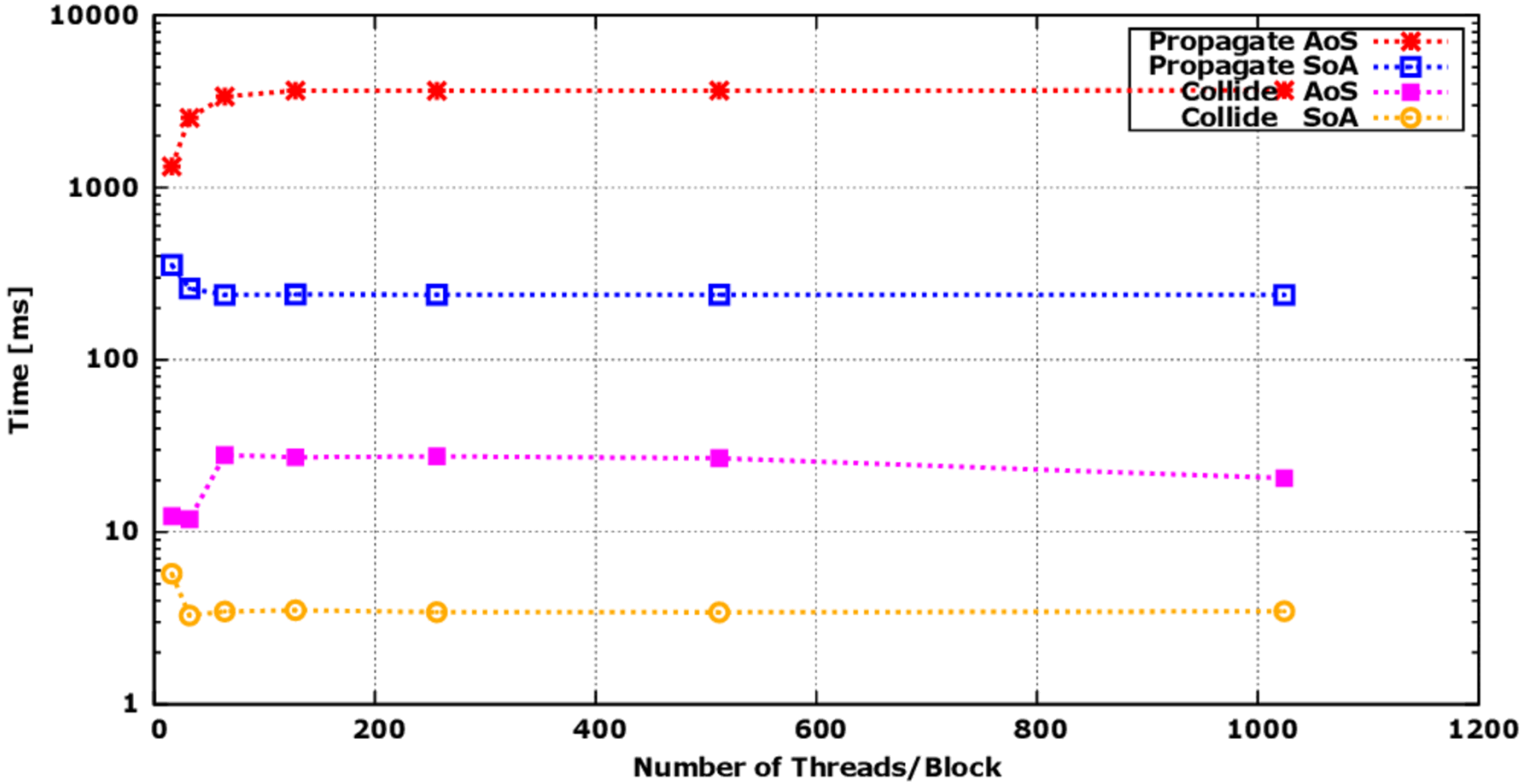
```
//lattice stored as AoS:
typedef struct {
  double  p1; // population 1
  double  p2; // population 2
  ...
  double p37; // population 37
} pop_t;

pop_t lattice2D[SIZEX*SIZEY];
```

AoS: corresponding populations of different sites are interleaved, causing strided memory-access and leading to coalescing issues.

```
//lattice stored as SoA:
typedef struct {
  double  p1[SIZEX*SIZEY]; // population 1 array
  double  p2[SIZEX*SIZEY]; // population 2 array
  ...
  double p37[SIZEX*SIZEY]; // population 37 array
} pop_t;

pop_t lattice2D;
```

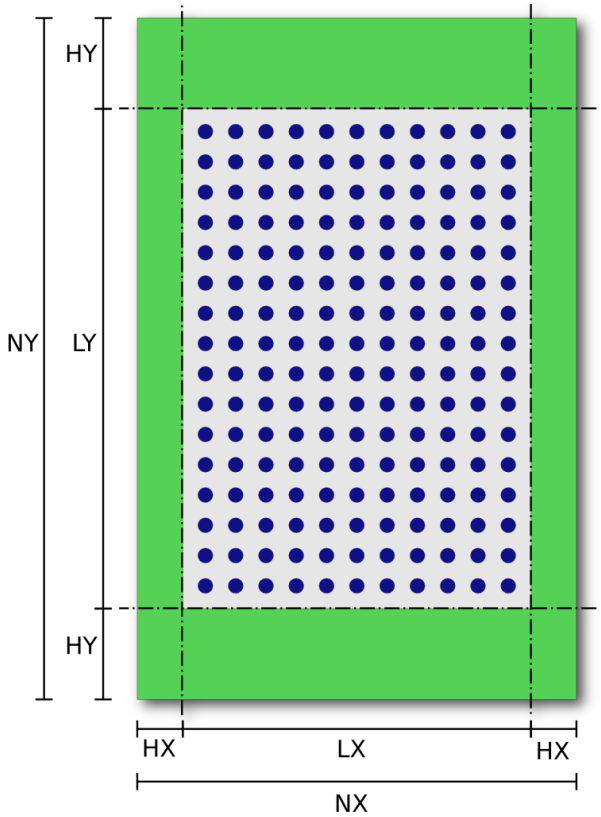
SoA: corresponding populations of different sites are allocated at contiguous memory addresses, enabling coalescing of accesses, and making use of full memory bandwidth.

AoS vs SoA in a 3D Lattice Boltzmann Application

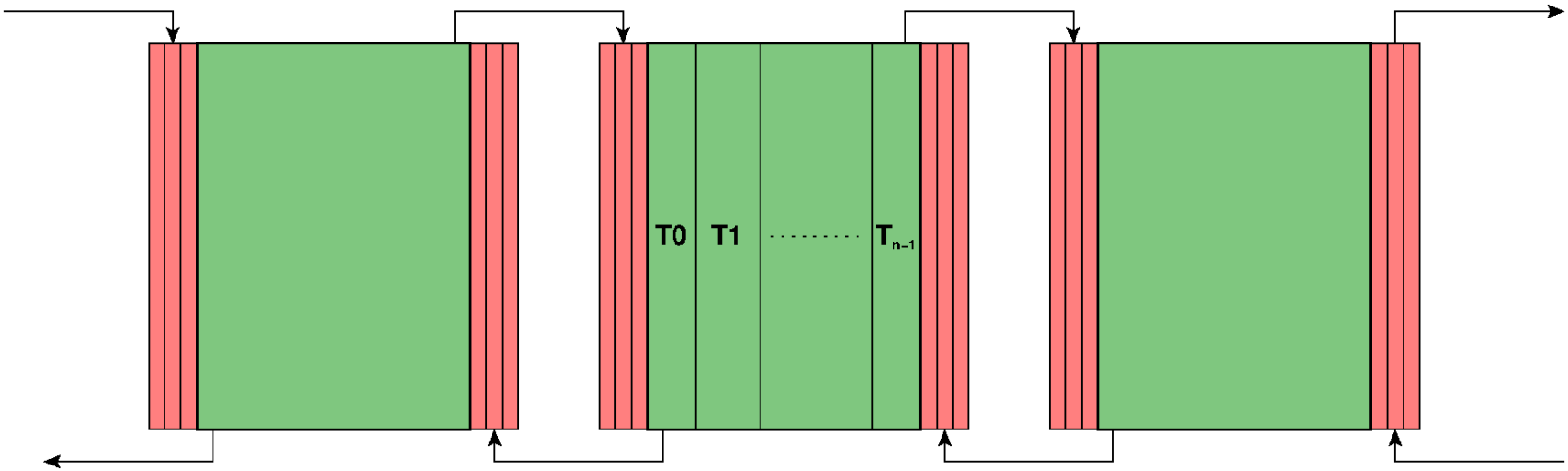


Lattice memory allocation

- lattice allocated in column-major order
- we use two copies of the lattice:
  each step reads from **prv** and write onto **nxt**
- a lattice of size $L_x \times L_y$ is stored as a grid of $(H_x + L_x + H_x) \times (H_y + L_y + H_y)$ sites:
  - make uniform computation of propagate also for sites close to borders
  - start address of lattice can be properly aligned to work-group size and cache-line.

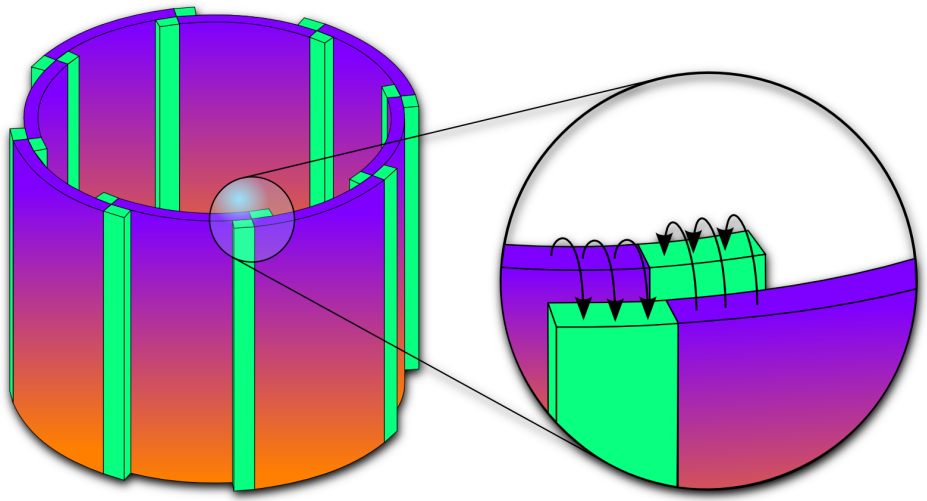


Multi-device implementation



Lattice partitioning:

- GPUs virtually arranged in a ring
- require an additional step ***PBC*** to update halo-columns at each step
- ***PBC*** is a GPU-to-GPU bi-directional (remote-)memory copy



Code Scheme

```
for ( step = 0; step < MAXSTEP; step++ ) {
  pbc ( ... );              // periodic boundary conditions

  propagate( ... );         // propagate()

  bc ( ... );               // bc()

  collide( ... );           // collide()
}
```

We have considered several hardware systems

	i7-4930K	Tesla K20X	Xeon-Phi 7120P
#physical cores	6	14	61
#logical cores	12	2688	244
Frequency (GHz)	3.4	0.735	1.238
GFLOPS (DP)	163.2	1317	1208
SIMD	AVX 64-bit	N/A	AVX2 512-bit
cache (MB)	12	1.5	30.5
Mem BW (GB/s)	59.7	250	352
Power (W)	130	235	300

- *i7-4930K*: CPU based on the Intel Ivy Bridge micro-architecture
- *Tesla K20X*: processor of the NVIDIA *Kepler* family
- *Xeon-Phi*: Intel MIC architecture

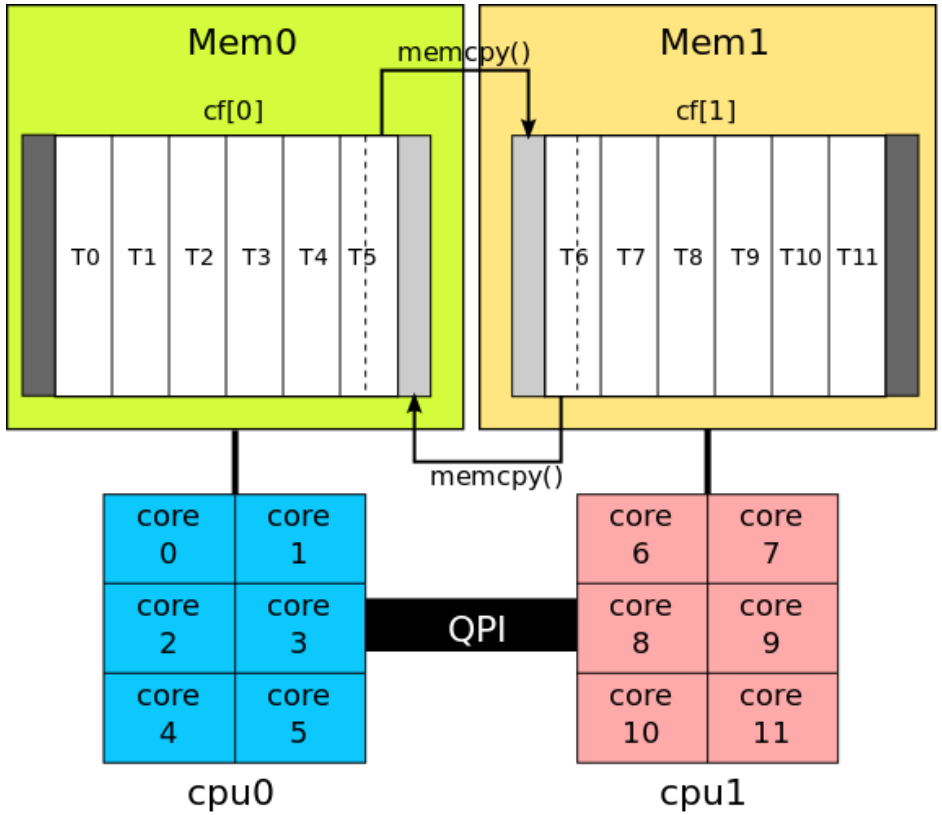
Can we run on all of them using only one code ? If YES at which price ?

We have considered several programming frameworks

- C
  - CPUs
  - Xeon-Phi
- CUDA
  - GPUs
- OpenCL
  - GPUs
  - CPUs
  - Xeon-Phi
- OpenACC
  - GPUs

C (our first) Implementation

- **core parallelism**:
  - lattice split over the cores
  - e.g. along X direction
  - borders replicated on each socket
    (make computation uniform)
  - **pthreads**/**openMP** library to manage parallelism
  - **NUMA** library to control allocations of data and threads
- **instruction parallelism**:
  - exploiting AVX vector instr.
  - processing 4 lattice-sites in parallel



C results

On a dual-Sandybridge machine running at 3.1 GHz (396.8 DP GFlops peak)

```
Lattice:  [1920x2048] 2.85 GB, NITER: 100
COLLIDE:  120740 us,  p:  252.58 GFlops, MLUP/s:  32.95 (FLOP/site: 7666)
PROPAGATE: 52800 us,  bw:  44.60 GB/s,   MLUP/s:  75.34
```

On a dual-Haswell-v3 machine running at 2.3 GHz (588.8 DP GFlops peak, preliminary)

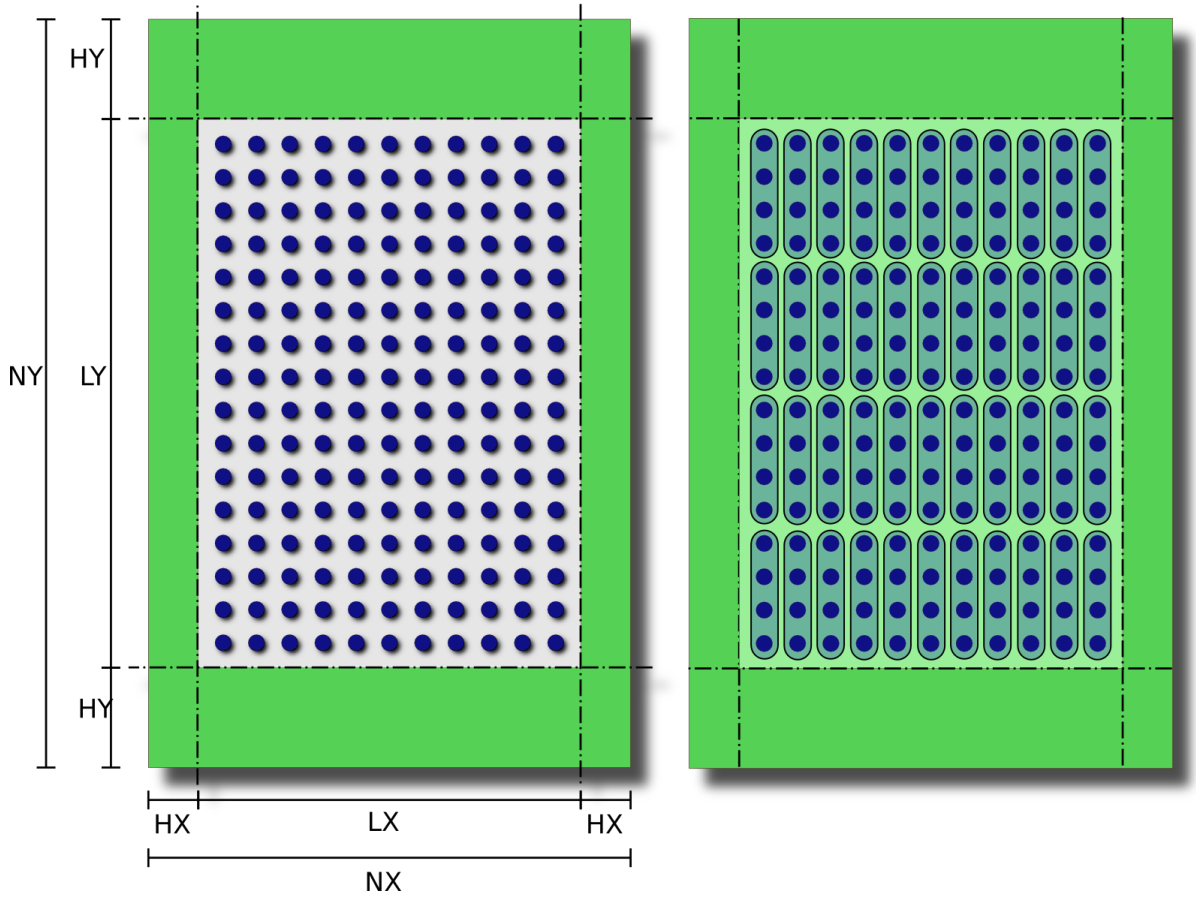
```
Lattice:  [1944x2048] 2.88 GB, NITER: 100
COLLIDE:   84320 us,   p:  366.21 GFlops, MLUP/s: 47.77 (FLOP/site: 7666)
PROPAGATE: 72480 us,   bw:  32.90 GB/s,   MLUP/s: 55.57
```

Cuda Implementation

- keep lattice data on GPU memory
- offload computation of propagate and collide kernels
- computation of pbc involves GPU and CPU
- one thread process one site

Grids Layout

Uni-dimensional array of NTHREADS, each thread processing one lattice site.



Example: physical lattice of 11 $\times$ 16 cells; the size of work-groups is 1 $\times$ 4.

$$L_y = \alpha \times N_{wi}, \;\; \alpha \in \mathbb{N}; \qquad (L_y \times L_x)/N_{wi} = N_{wg}$$

Cuda Results

On K20Xm board (1.31 TFlops DP peak), 256 threads/block, cuda 6.5:

```
Lattice:  [1024x2048] 0.578125 GB, NITER: 1000
COLLIDE:  23656.14 us,  p: 573.75 GF/s, MLUPs:  88.65 (FLOP/site: 6472)
PROPAGATE: 7945.81 us, bw: 156.25 GB/s, MLUPs: 263.93
```

On a K40 board (1.43 TFlops DP peak), 256 threads/block, cuda 6.5:

```
Lattice:  [1024x2048] 0.578125 GB, NITER: 1000
COLLIDE:  21584.96 us,  p: 628.81 GF/s, MLUPs:  97.16 (FLOP/site: 6472)
PROPAGATE: 7384.59 us, bw: 168.12 GB/s, MLUPs: 283.99
```

Open Computing Language (OpenCL)

- programming framework for **heterogenous** architectures: CPU+accelerators
- computing model:
  - host-code plus one or more **kernels** running on accelerators
  - kernels are executed by a set of **work-items** each processing an item of the data-set (data-parallelism)
  - work-items are grouped into **work-groups**, each executed by a **compute-unit** and processing K work-items in parallel using vector instructions
  - e.g.: on Xeon-Phi work-groups are mapped on (virtual-)cores processing each up to 8 double-precisions floting-point data
- memory model identifies a hierarchy of four spaces which differ for size and access-time : private, local, global and constant memory

OCL aims to guarantee portability of both code and performances across several architectures

OCL Saxpy kernel

$$C = s \cdot A \times B, \quad s \in \mathbb{R}, \quad A, B, C \in \mathbb{R}^n$$

```
__kernel void saxpy( __global double *A, __global double *B,
                     __global double *C, const double s) {

  //get global thread ID
  int id = get_global_id(0);

  C[id] = s * A[id] + B[id];
}
```

- each work-item executes the *saxpy* kernel computing just one data-item of the output array
- first it computes its unique global identifier `id`
- and then uses it to address the $\texttt{id}^{th}$ data-item of arrays `A, B` and `C`.

OCL Result Issues on GPUs

As Winter 2013: CUDA-5.5, driver-319.82

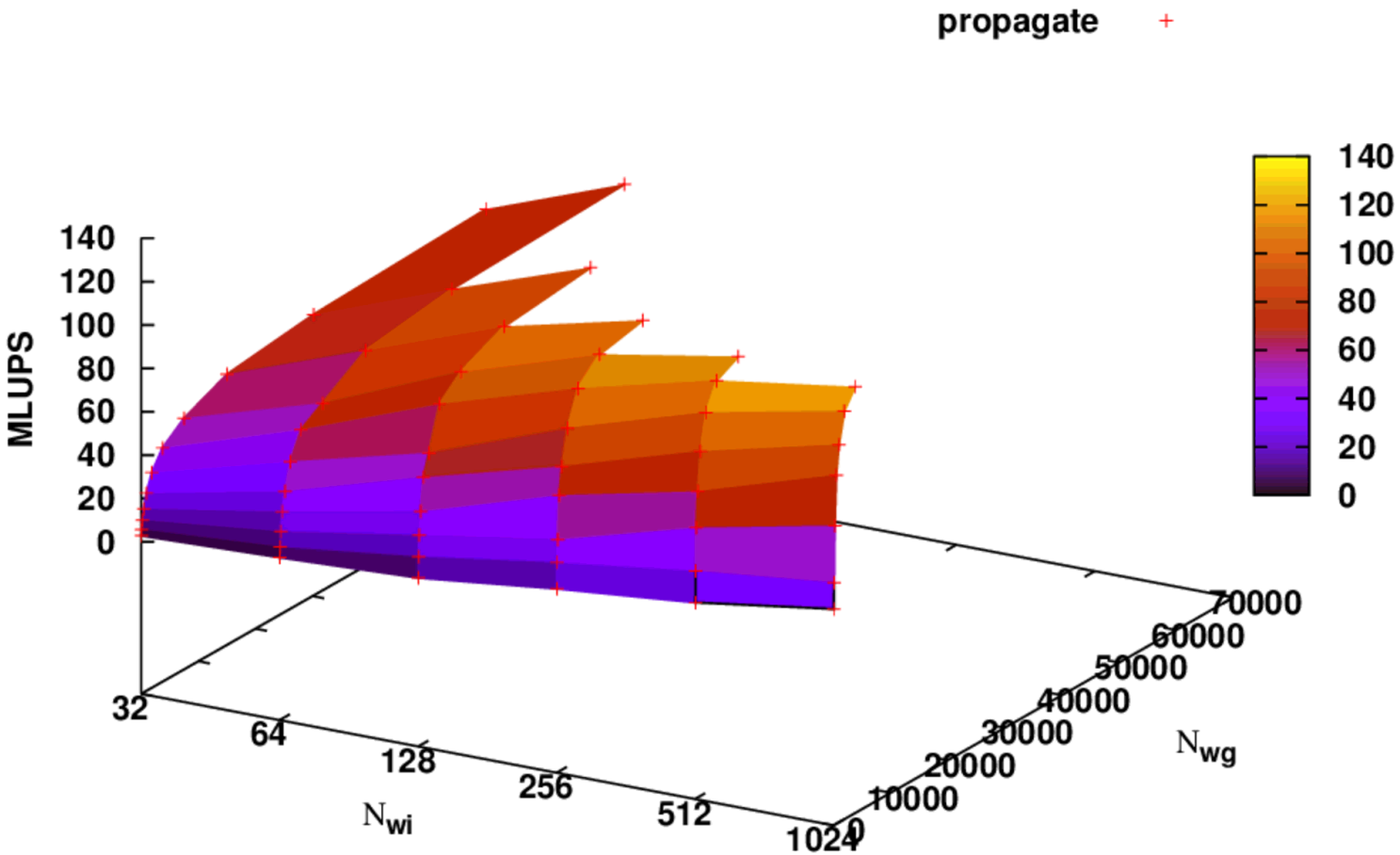
```
Pbc         time/iter:    0.06 msec
Propagate   time/iter:   17.54 msec   MLUPS: 224.167811
Bc          time/iter:    8.00 msec
Collide     time/iter:  104.78 msec   MLUPS: 37.527603
```

As Summer 2014: CUDA-5.5, driver-331.89

```
Pbc         time/iter:    0.06 msec
Propagate   time/iter:   17.79 msec   MLUPS: 220.973559
Bc          time/iter:    8.70 msec
Collide     time/iter:  199.13 msec   MLUPS: 19.746729
```

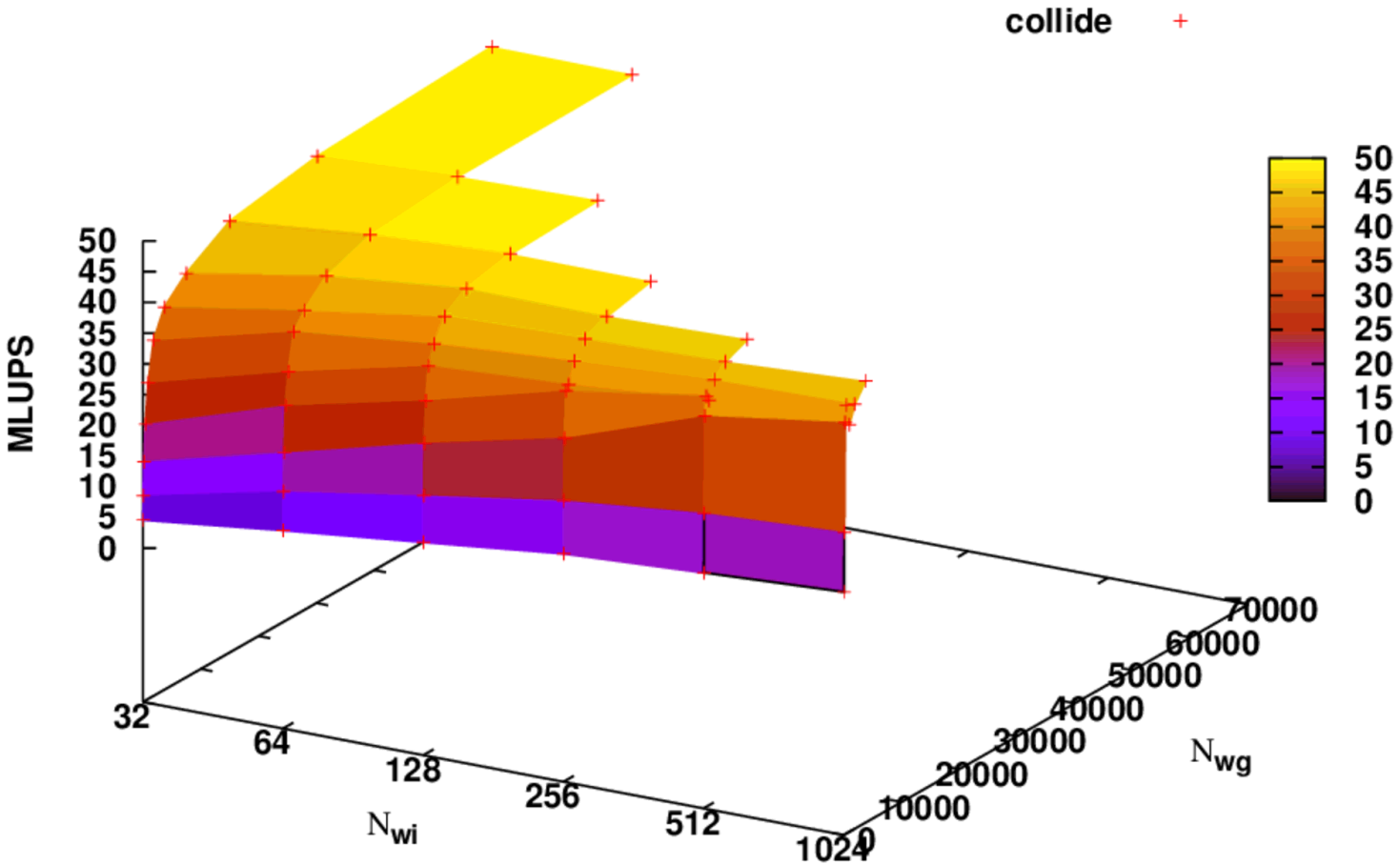
Results does not improve using CUDA-6.0 and CUDA-6.5.

OCL Benchmark of Propagate on Xeon-Phi



Performance of propagate as function of the number of work-items N_{wi} per work-group, and the number of work-groups N_{wg}.

OCL Benchmark of Collide on Xeon-Phi



Performance of collide as function of the number of work-items N_{wi} per work-group, and the number of work-groups N_{wg}.

OpenACC example: the Saxpy function

```
#pragma acc copyin(x), copy(y)
{
   my_saxpy(x, y);

   acc_async_wait(1);
}
```

```
void my_saxpy(float * x, float * y) {

  #pragma acc kernels present(x) present(y) async(1)
  #pragma acc loop gang vector(256)
  for (int i = 0; i < N; ++i)
    y[i] = a*x[i] + y[i];

}
```

`#pragma` clauses identifies regions to run on the accelerator, how to organize computation, and how to manage data transfers.

OpenACC: Propagate

```
inline void propagate (
  const data_t* restrict prv, data_t* restrict nxt )
{
  int ix, iy, site_i;

  #pragma acc kernels present(prv) present(nxt)
  #pragma acc loop gang independent
  for ( ix=HX; ix < (HX+SIZEX); ix++) {
    #pragma acc loop vector independent
    for ( iy=HY; iy<(HY+SIZEY); iy++) {
      site_i = (ix*NY) + iy;
      nxt[       site_i] = prv[       site_i-3*NY+1];
      nxt[NX*NY+site_i] = prv[NX*NY+site_i-3*NY   ];
       ....
    }
  }
}
```

OpenACC: Results

On K20Xm, 256 threads/block, cuda 5.5, PGI 14.1:

```
Lattice:   [1920x2048] 1.083984 GB, NITER: 1000
PBC+PROP: 18.83 ms, bw: 123.64 GB/s,   MLUP/s: 208.82
BC:        2.07 ms
COLLIDE: 112.66 ms,  p: 227.00 GFlops, MLUP/s: 35 (FLOPs/site: 6504)
```

On K20Xm, 256 threads/block, cuda 6.0, PGI 14.7:

```
Lattice:   [1920x2048] 1.083984 GB, NITER: 1000
PBC+PROP: 14.89 ms, bw: 156.29 GB/s,   MLUP/s: 264.01
BC:        2.37 ms
COLLIDE: 144.81 ms,  p: 176.61 GFlops, MLUP/s: 27.15 (FLOPs/site: 6504)
```

On K40, 256 threads/block, cuda 6.0, PGI 14.7:

```
Lattice:   [1920x2048] 1.083984 GB, NITER: 1000
PBC+PROP: 13.90 ms, bw: 167.44 GB/s,   MLUP/s: 282.84
BC:        2.76 ms
COLLIDE:  79.66 ms,  p: 321.07 GFlops, MLUP/s: 49.36 (FLOP/site: 6504)
```

On K40 performance improves with compiler 14.7 and some suggestions from PGI (compilation settings `loadcache:L1,maxregcount:120`).

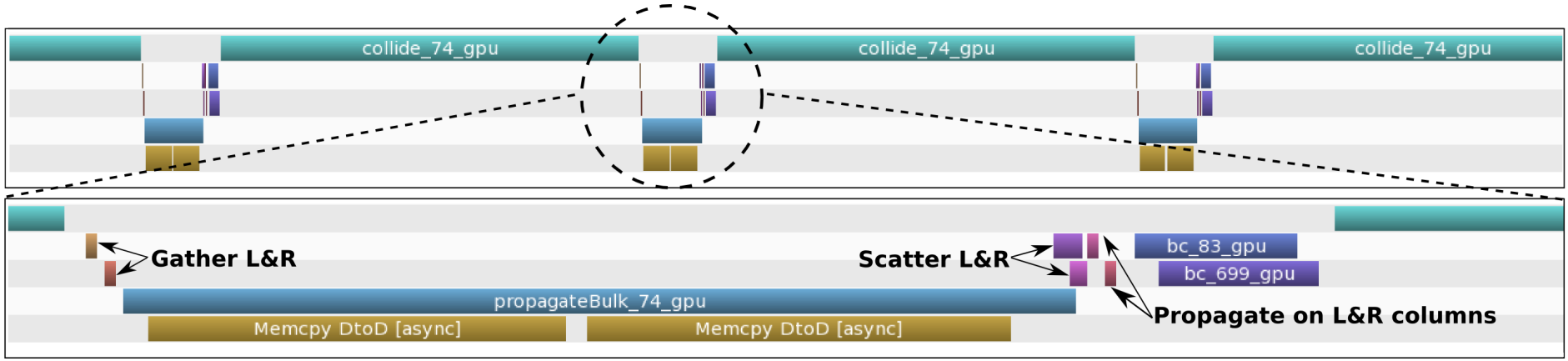
Performance of collide are signficantly slower than CUDA. We believe this could be due to the lack of proper unroll of the code.

OpenACC: Overlapping Pbc and Propagate

```
gatherL( f2, sndbufL );  // async on queue (1)
gatherR( f2, sndbufR );  // async on queue (2)

propagateBulk( f2, f1 ); // async on queue (3)

acc_async_wait(1);

MPI_Sendrecv (
 sndbufL, L, MPI_DOUBLE, mpi_rankL, tag2,
 rcvbufR, L, MPI_DOUBLE, mpi_rankR, tag2,
 MPI_COMM_WORLD, &status );

acc_async_wait(2);

MPI_Sendrecv (
 sndbufR, L, MPI_DOUBLE, mpi_rankR, tag1,
 rcvbufL, L, MPI_DOUBLE, mpi_rankL, tag1,
 MPI_COMM_WORLD, &status );

scatterL( f2, rcvbufL); // async on queue (1)
propagateL( f2, f1 );   // async on queue (1)

scatterR( f2, rcvbufR); // async on queue (2)
propagateR( f2, f1 );   // async on queue (2)

acc_async_wait_all();
```

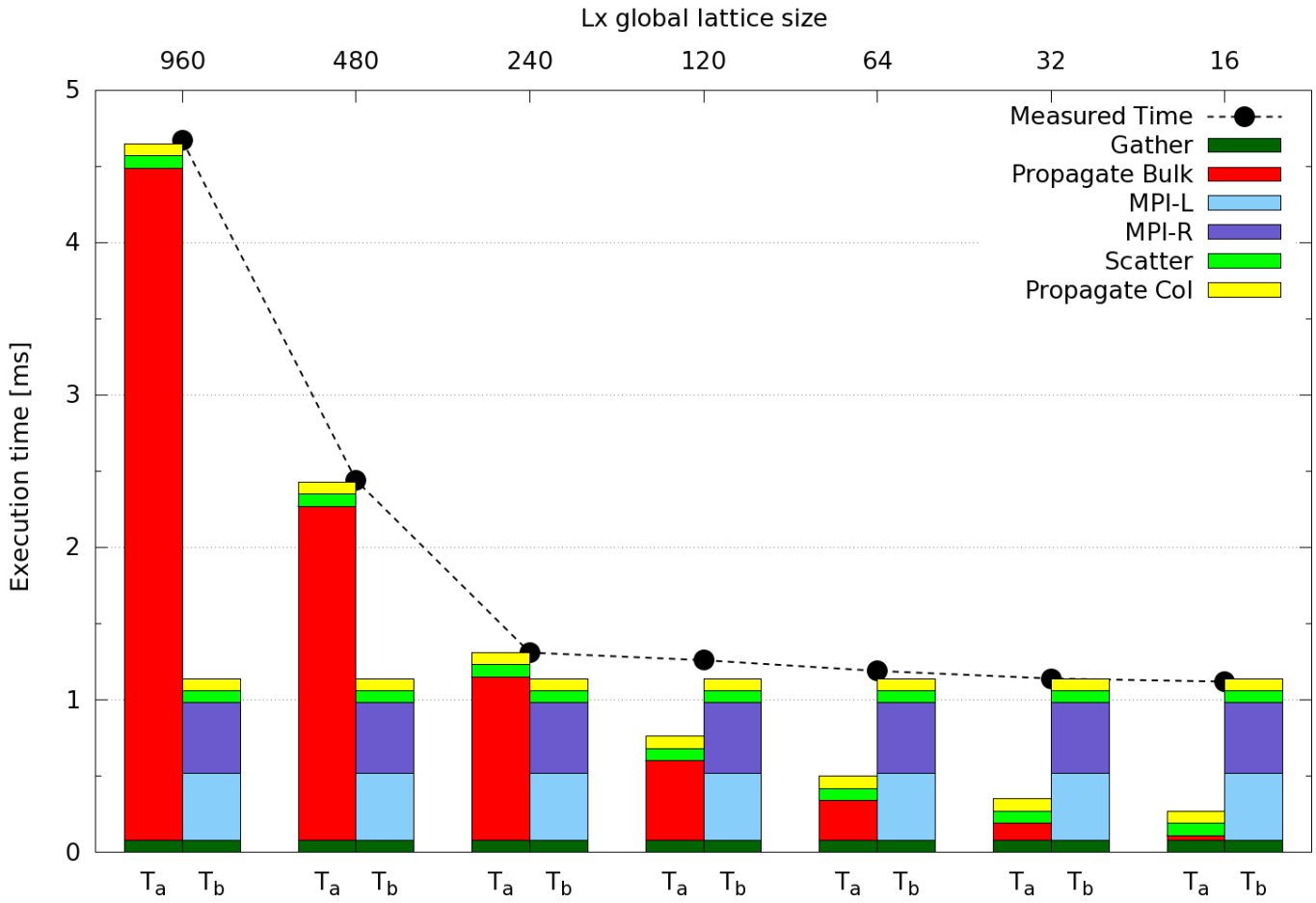
Critical optimization for scalability

OpenACC: Overlapping Pbc and Propagate



OpenACC: Overlapping Pbc and Propagate

$$T_{\mathrm{PBC}} = \max \begin{cases} T_a &= T_G + T_P + T_S + T_{P'} \\ T_b &= T_G + T_{\mathrm{MPI(L)}} + T_{\mathrm{MPI(R)}} + T_S + T_{P'} \end{cases}$$

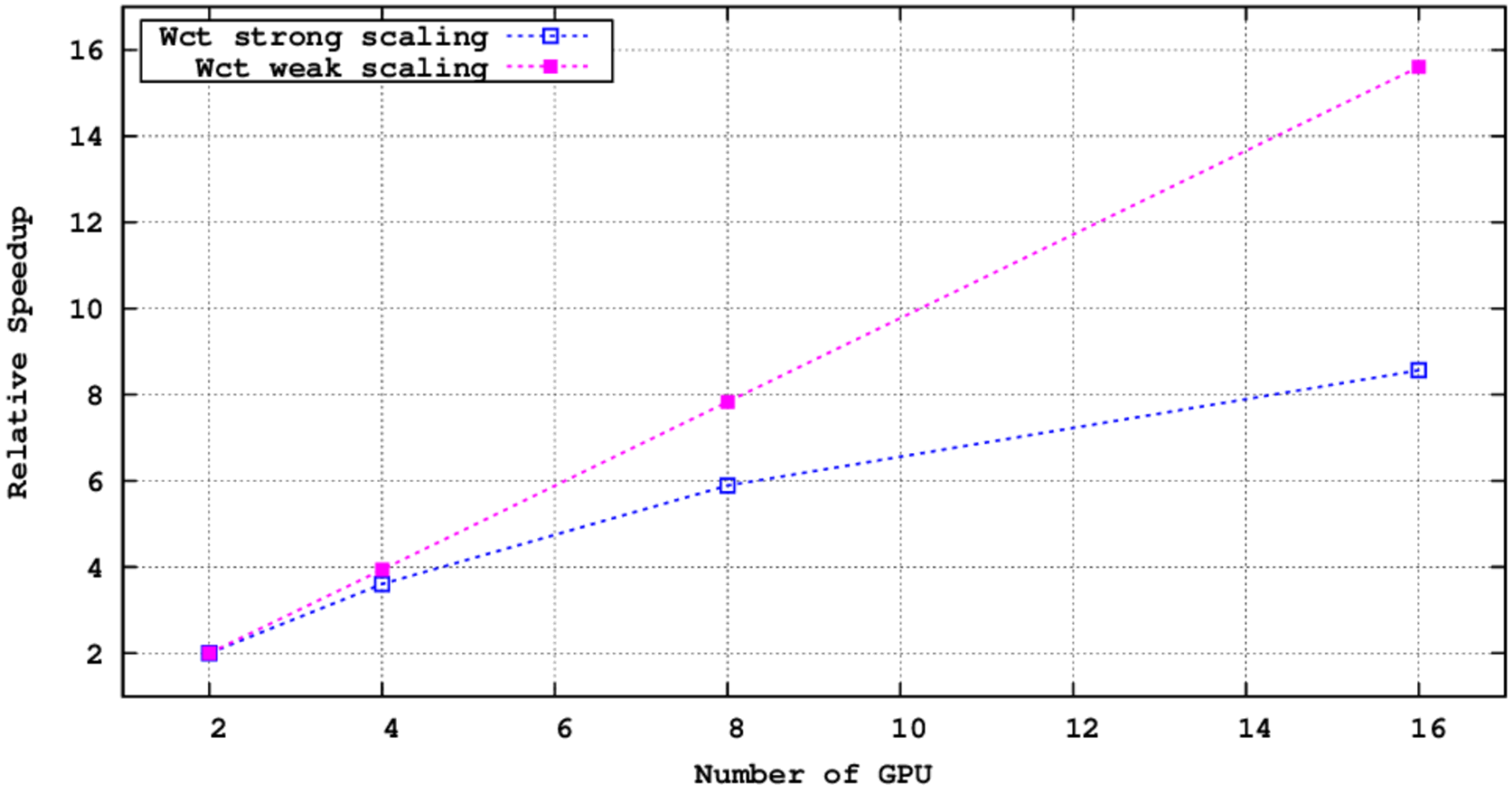


Comparison Results

	i7-4930K	Tesla K20Xm			Xeon-Phi 7120
Code Version	C	CUDA	OCL	OACC	OCL
$T_{\text{Pbc+Prop}}$ [msec]	162.00	15.38	14.89	18.83	30.46
GB/s	14.54	151.36	156.33	123.64	76.42
$\mathcal{E}_p$	24%	60%	62%	49%	22%
T_{Bc} [msec]	4.87	5.70	7.08	2.07	3.20
T_{Collide} [msec]	307.42	43.96	93.27	112.66	72.79
MLUPS	13	89	42	35	54
$\mathcal{E}_c$	59%	52%	24%	20%	34%
μJ / site	10.04	2.63	5.57	6.73	5.55
T_{WC}/iter [msec]	489.98	65.03	115.24	135.37	106.45
MLUPS	8	60	34	29	37

Lattice size: 1920×2048

OACC Scalability



- strong regime lattice size: 1024×8192
- weak regime lattice size 256×8192 / GPU

Conclusions

Today scenario faced by programmers:

- CUDA gives the best performance but
  - rewriting of the code is necessary
  - lack of portability
- OpenCL is portable with good performance
  - implementation for GPUs seems not supported
  - coding is lengthy and low-level
- OpenACC is promising
  - today performance are lower if compared with CUDA
  - not supported by all accelerator

Take-away conclusions:

- several programming frameworks are available
- solutions for portability of code and performance still away

Acknowledgments

- Luca Biferale, Mauro Sbragaglia, Patrizio Ripesi
  University of Tor Vergata and INFN Roma, Italy
- Andrea Scagliarini
  University of Barcelona, Spain
- Filippo Mantovani
  BSC institute, Spain
- Enrico Calore, Sebastiano Fabio Schifano, Raffaele Tripiccione,
  University and INFN of Ferrara, Italy
- Federico Toschi
  Eindhoven University of Technology The Netherlands, and CNR-IAC, Roma Italy

This work has been performed in the framework of the INFN COKA and SUMA projects.

We would like to thank CINECA (ITALY) and JSC (GERMANY) institutes for access to their systems.