Challenges in Programming Modern Parallel Systems

Sebastiano Fabio Schifano

University of Ferrara and INFN

Università degli Studi di Ferrara

July 23, 2018
The Eighth International Conference on Advanced Communications and Computation
INFOCOMP
Barcelona - Spain

Introduction

- several processors are available with different architectures
- application codes and perfomance are hardly portable across different processor architectures
- compilers have improved but are far from making it easy to move from one architecture to the other
- find out how to program efficiently latest processors "minimizing" changes in the code

Focus

Optimization of Lattice-based applications on latest multi- and many-core processors

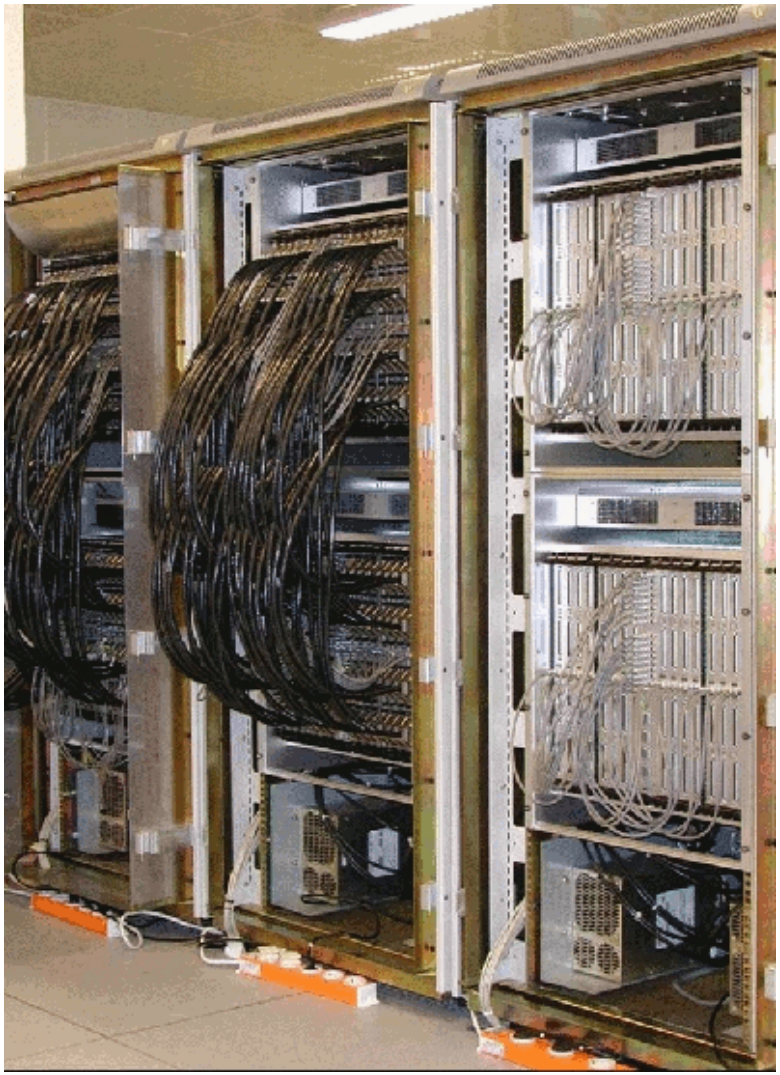
Outline

Background: Let me introduce myself

Development of computing systems optimized for computational physics:

- APEmille and apeNEXT: LQCD-machines
- AMchip: pattern matching processor, installed at CDF
- Janus I+II: FPGA-based system for spin-glass simulations
- QPACE: Cell-based machine, mainly LQCD
- AuroraScience: multi-core based machine
- EuroEXA: hybrid ARM+FPGA exascale system

APEmille e apeNEXT (2000 and 2004)



Custom HPC systems are not easy to use

Require deep knowledges of hardware structure, and lack of standard programming tools !

Janus I (2007)

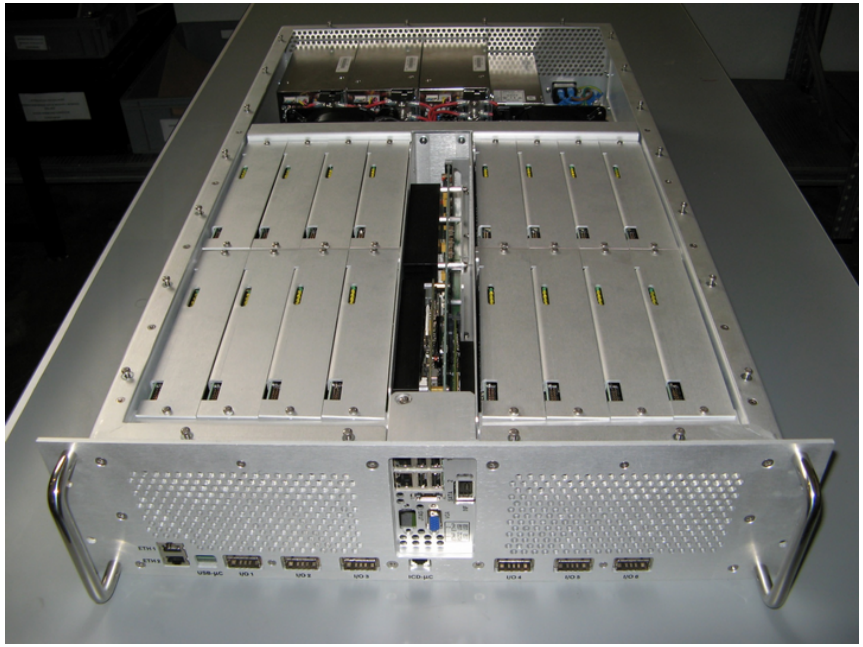
- 256 FPGAs
- 16 boards
- 8 host PC

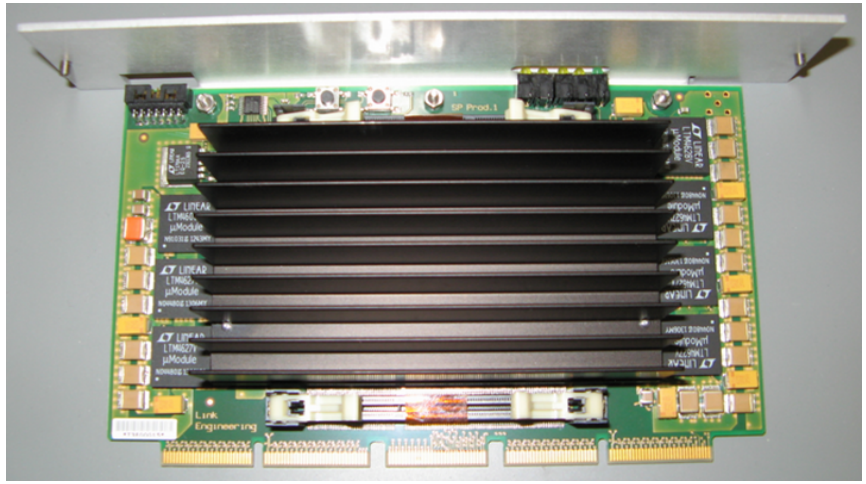
- Monte Carlo simulations of Spin Glass systems

Custom HPC systems are not easy to use

Deep knowledges are required, and lack of standard programming tools !
Code for FPGA are far to be portable across different architectures !

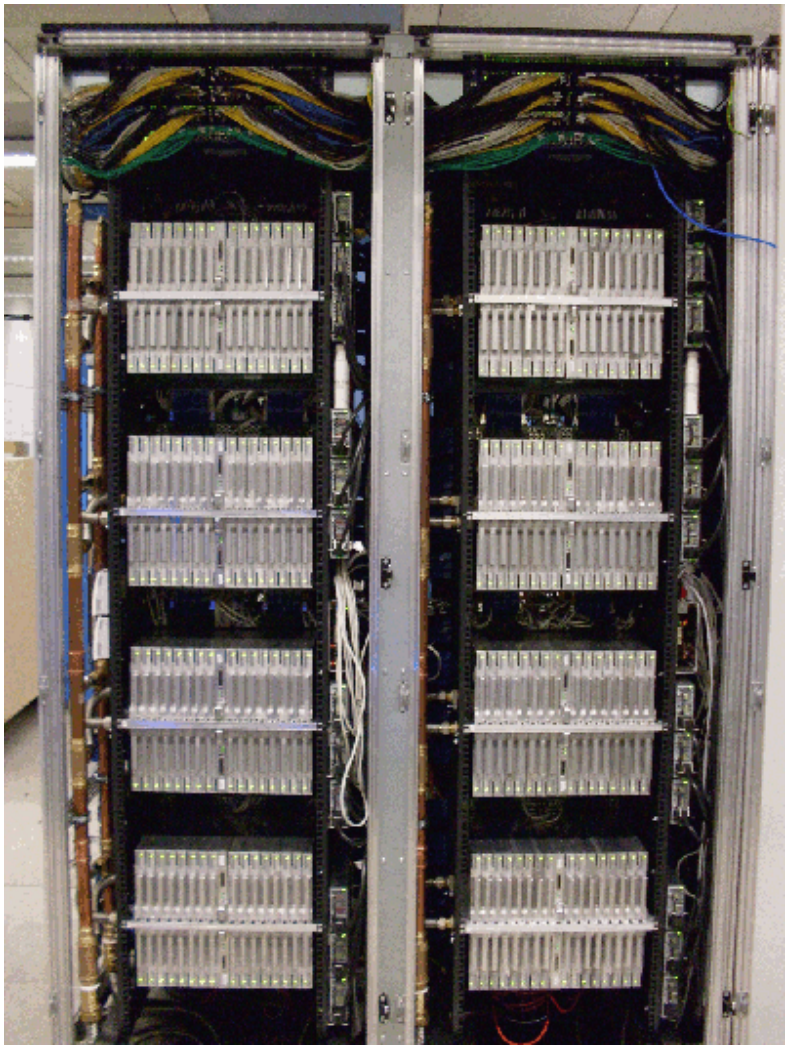
Janus II (2012)





QPACE Machine (2008)

- 8 backplanes per rack
- 256 nodes (2048 cores)
- 16 root-cards
- 8 cold-plates
- 26 Tflops peak double-precision
- 35 KWatt maximum power consumption
- **750 MFLOPS / Watt**
- TOP-GREEN 500 in Nov.'09 and July'10



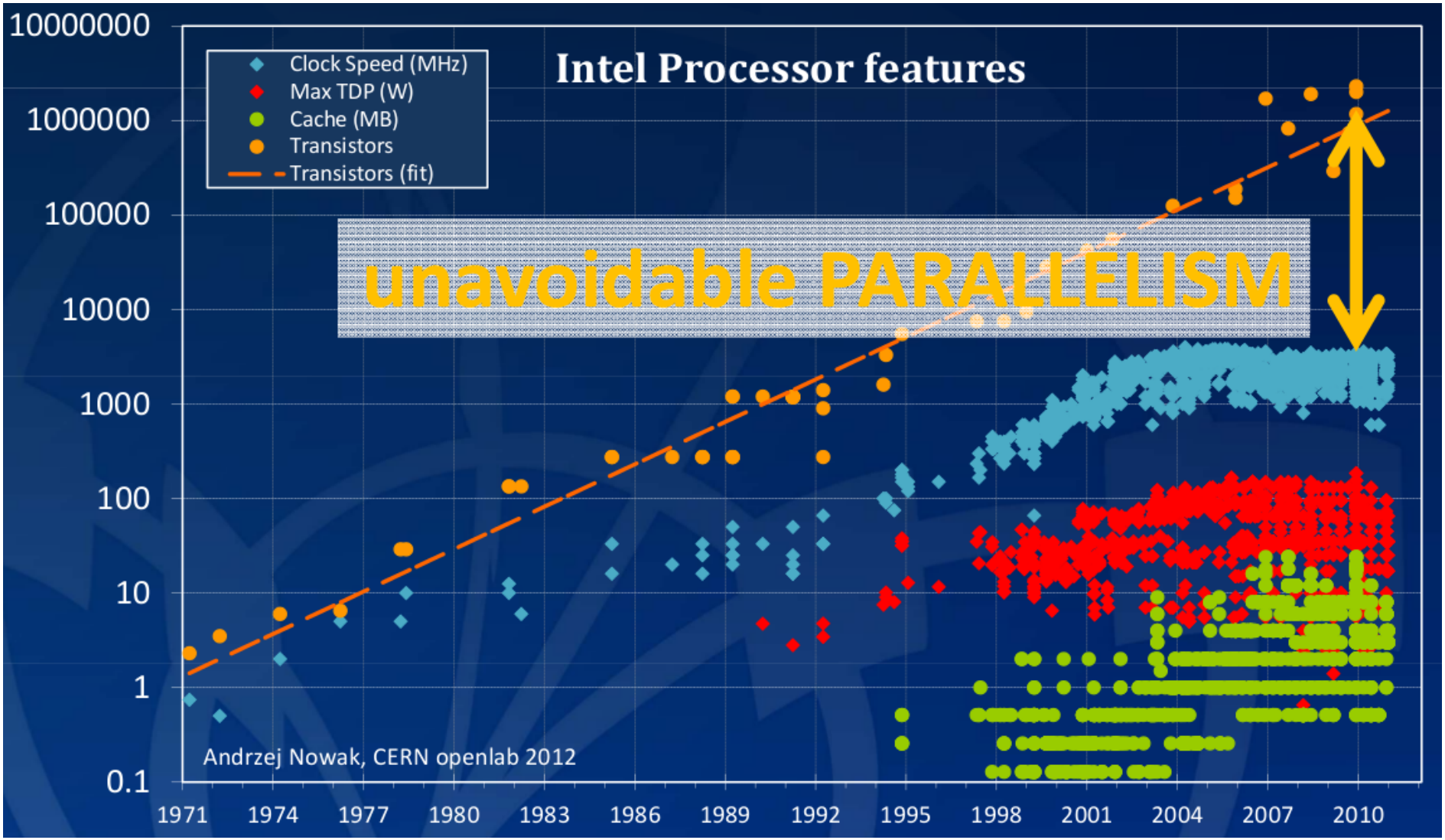
Custom HPC systems are not easy to use

Deep knowledges are required, and lack of standard programming tools !

Use of recent processors

- QPACE has been the last attempt (in our community) to use custom(-ized) HPC systems
- QPACE has been the first attempt (in our community) to use a **commodity** processor interconnected by a **custom** network
- After we focus on how and how well we can use recent developed processors for our applications, LQCD LBM and others,
- and on which issues we have to face out and how we can program them at best.

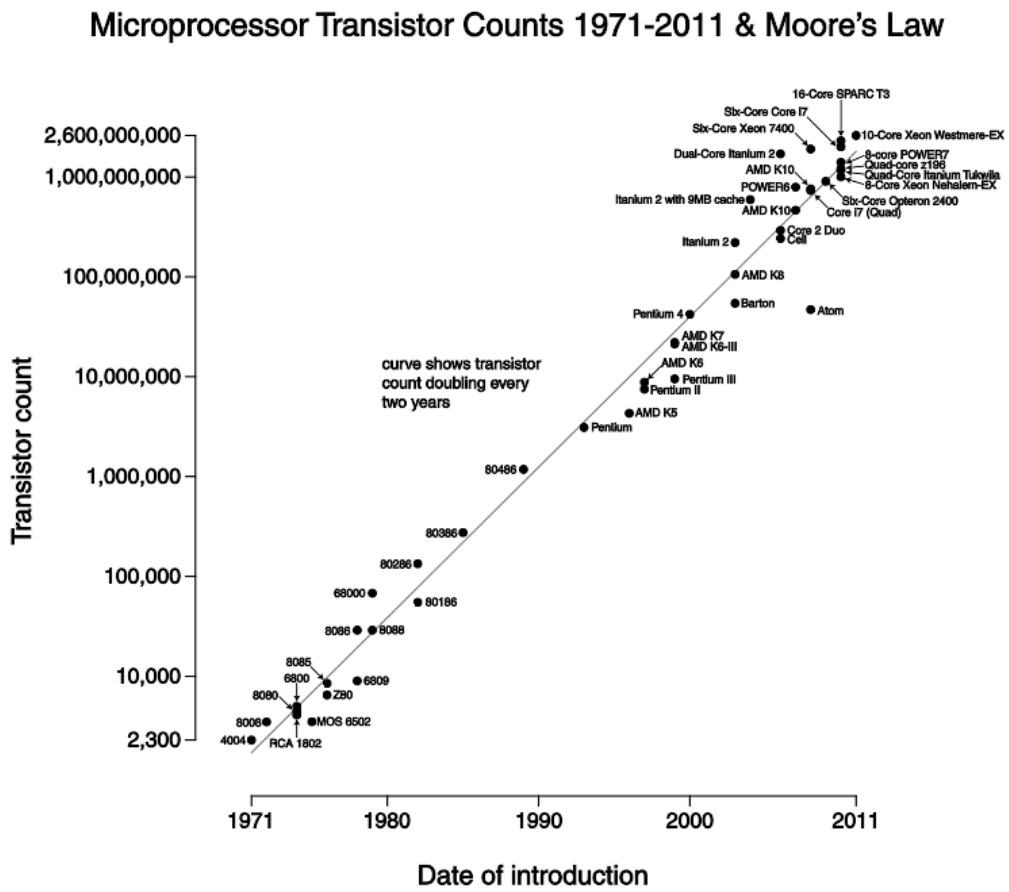
Hardware Evolution



The Multi-core processors era begins !

Multi-core architecture allows CPU performances to scale according to Moore's law.

- increase frequency beyond $\approx 3 - 4$ GHz is not possible
- assembly more CPUs in a single silicon device ✔
- CPUs capable to execute vector instructions ✔
- great impact on application performance and design ✘
- move challenge to exploit high-performance computing from HW to SW ✘



So ... what is changing in practice ?

So ... what is changing in practice ?

. . . from one big-plow to **many** small-plows model !

- one big processor: low-latency and good throughput;
- many small processors (core): high throughput and reasonable latency;
- how to manage many processor cores ?

Better if you can use both !!! May be hard to program and get good efficiency !

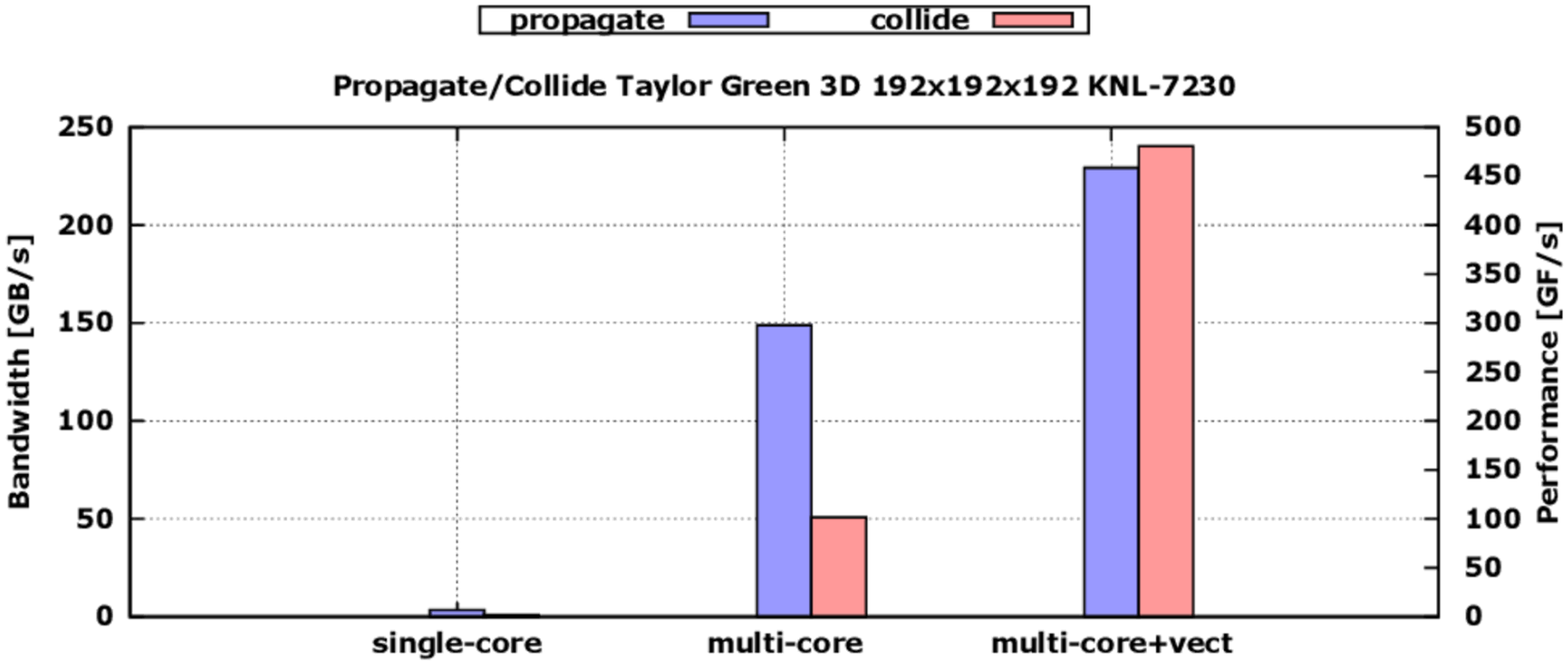
... and ... what is changing in practice ?

... and ... what is changing in practice ?

. . . and from one-hand one-beer to one-hand **many**-beers model !

- vector instructions improve computing throughput . . .
- . . . provided there are many data to process !
- How to exploit vector instructions ?

... just to give an idea of what changes in practice !



Many Different Cores

- several processors are available with different architectures
- different programming models
- different level of parallelism
- different vector instructions

Lattice Boltzmann Methods

Lattice Boltzmann Methods (LBM)

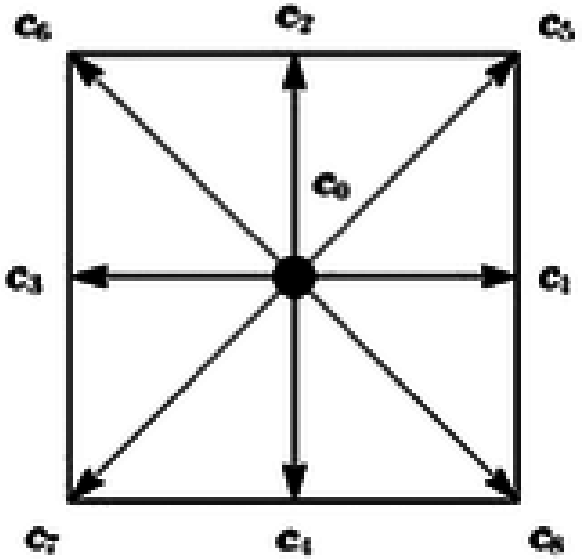
- a class of computational fluid dynamics (CFD) methods
- discrete **Boltzmann** equation instead of **Navier-Stokes** equations
- sets of **virtual particles**, called **populations**, arranged at edges of a D-dimensional ($D = 2, 3$) lattice
- each population $f_i(x, t)$ has a given fixed lattice velocity $\boldsymbol{c}_i$, drifting – at each time step – towards a nearby lattice-site;
- populations evolve in discrete time according to the following equation:

$$f_i(\boldsymbol{x} + \boldsymbol{c}_i \Delta t, t + \Delta t) = f_i(\boldsymbol{x}, t) - \frac{\Delta t}{\tau} \left(f_i(\boldsymbol{x}, t) - f_i^{(eq)} \right)$$

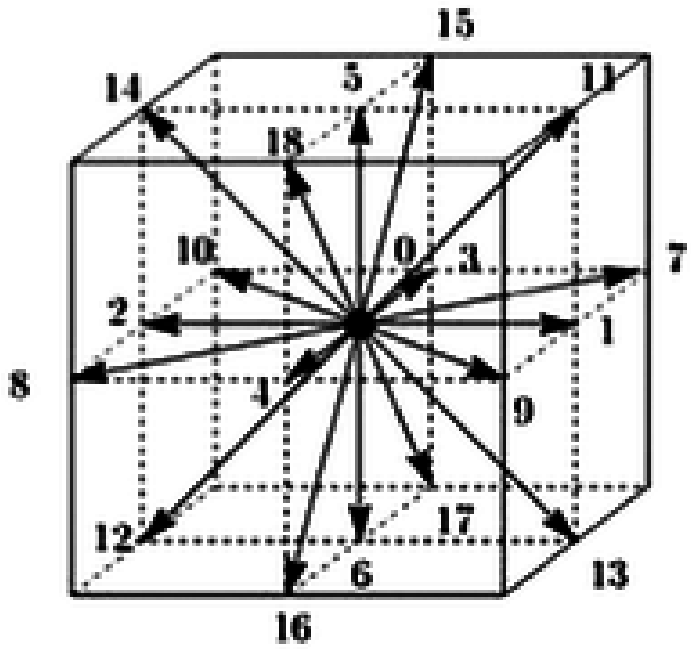
- macroscopic observables, like density ρ, velocity $\boldsymbol{u}$ and temperature T, are defined in terms of populations $f_i(x, t)$:

$$\rho = \sum_i f_i \quad \rho \boldsymbol{u} = \sum_i \boldsymbol{c}_i f_i \quad D \rho T = \sum_i |\boldsymbol{c}_i - \boldsymbol{u}|^2 f_i$$

LBM models



D2Q9



D3Q19

DnQk:

- *n* is the spatial dimension,
- *k* is the number of populations per lattice site

LBM Computational Scheme

Rewriting evolution equation as

$$f_i(\boldsymbol{y}, t + \Delta t) = f_i(\boldsymbol{y} - \boldsymbol{c}_i \Delta t, t) - \frac{\Delta t}{\tau}\left(f_i(\boldsymbol{y} - \boldsymbol{c}_i \Delta t, t) - f_i^{(eq)}\right)$$

being $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{c}_i \Delta t$, we can handle it by a two-step algorithm:

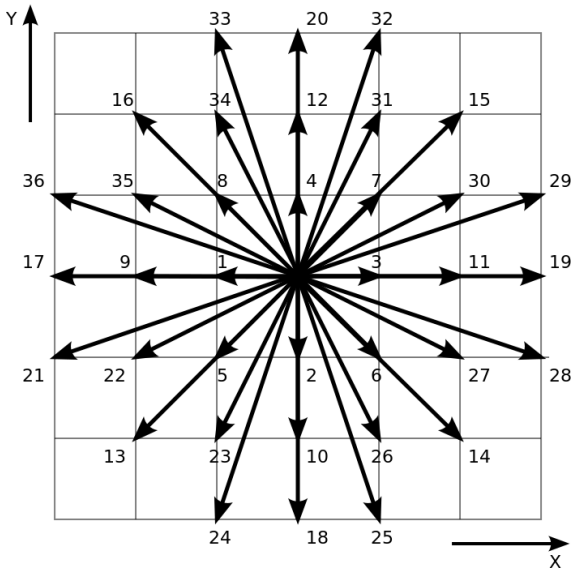
1. **propagate**:
$$f_i(\boldsymbol{y} - \boldsymbol{c}_i \Delta t, t)$$
gathering from neighboring sites the values of the fields f_i corresponding to populations drifting towards $\boldsymbol{y}$ with velocity $\boldsymbol{c}_i$;
2. **collide**:
$$-\frac{\Delta t}{\tau}\left(f_i(\boldsymbol{y} - \boldsymbol{c}_i \Delta t, t) - f_i^{(eq)}\right)$$
compute the bulk properties density ρ and velocity $\boldsymbol{u}$, use these to compute the equilibrium distribution $f_i^{(eq)}$, and then relax the fluid distribution functions to the equilibrium state (τ relaxation time).

LBM Computational Scheme

```
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
- two relevant kernels:
  - *propagate* memory-intensive,
  - *collide* compute-intensive;
- *propagate* and *collide* can be fused in a single step;
- good tool to stress, test and benchmark computing systems.

D2Q37 LBM Application

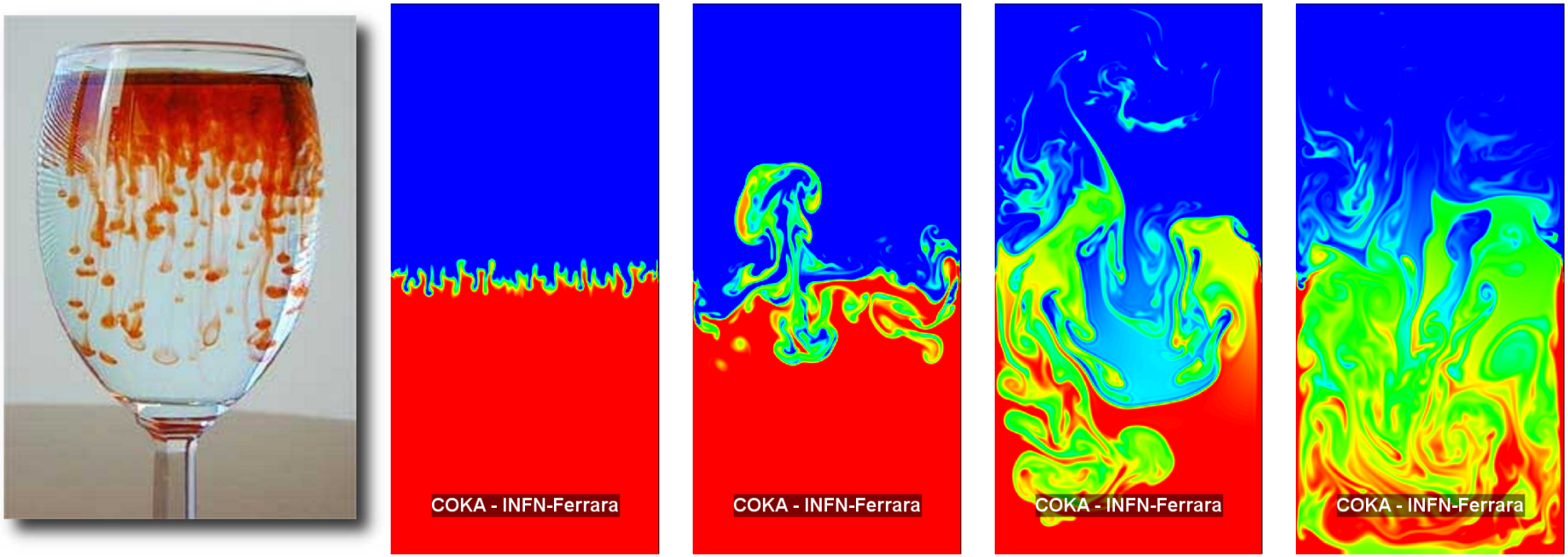


- D2Q37 is a 2D LBM model with 37 velocity components (populations);
- suitable to study behaviour of **compressible** gas and fluids optionally in presence of **combustion**[1] effects;
- include correct treatment of *Navier-Stokes*, heat transport and perfect-gas ($P = \rho T$) equations;
- used to study Rayleight-Taylor effects of stacked fluids at different temperature and density with periodic boundary conditions along one dimension;
- *propagate*: memory-intensive, access neighbours cells at distance 1,2, and 3, generate memory-accesses with **sparse** addressing patterns;
- *collide* compute-intensive, requires $\approx$ 6500 DP floating-point operations, is local.

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

Rayleigh-Taylor Instability Simulation with D2Q37

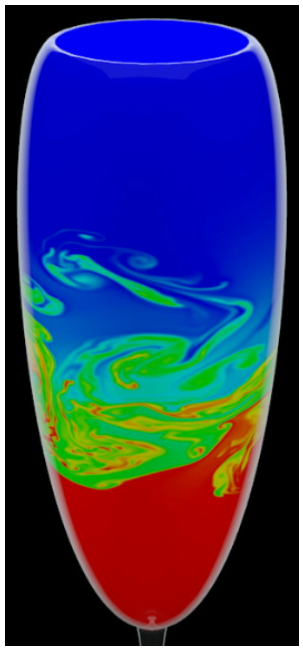
Instability at the contact-surface of two fluids of different densities and temperature triggered by gravity.

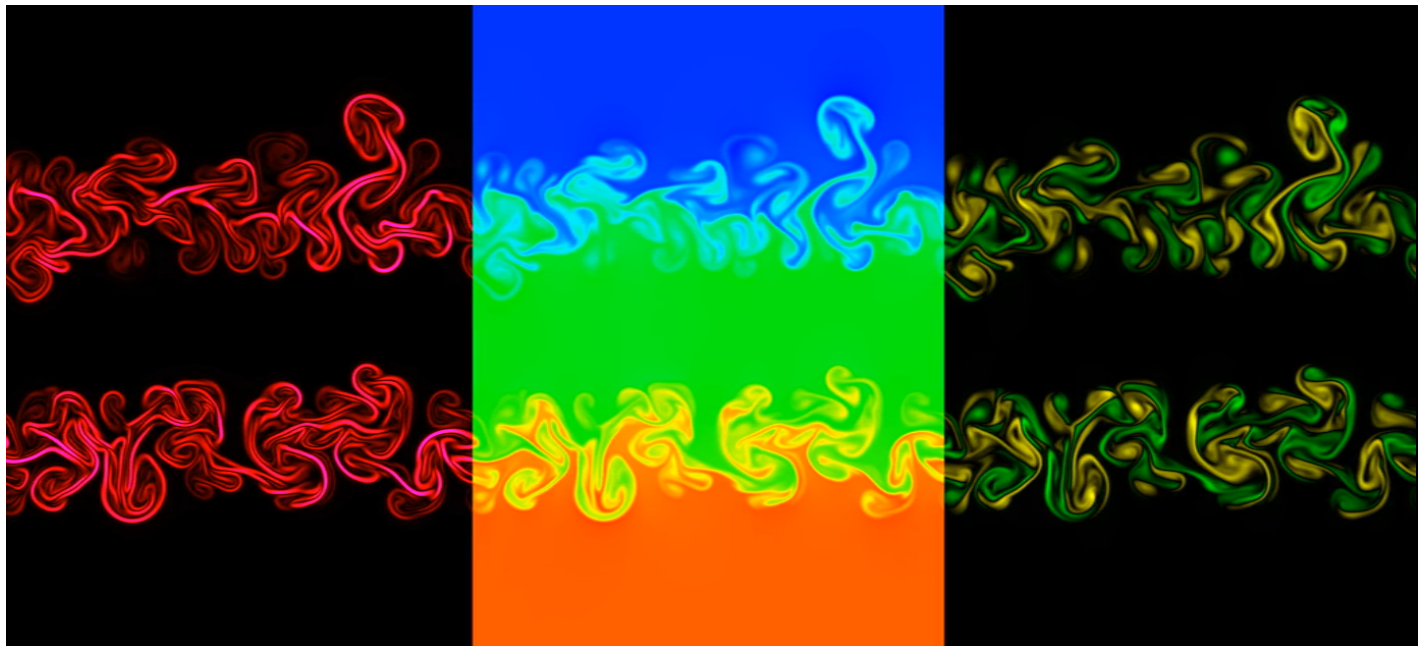


A cold-dense fluid over a less dense and warmer fluid triggers an instability that mixes the two fluid-regions (till equilibrium is reached).

D2Q37

- Rayleight-Taylor effects on two stacked fluids
- Rayleight-Taylor effects on three stacked fluids

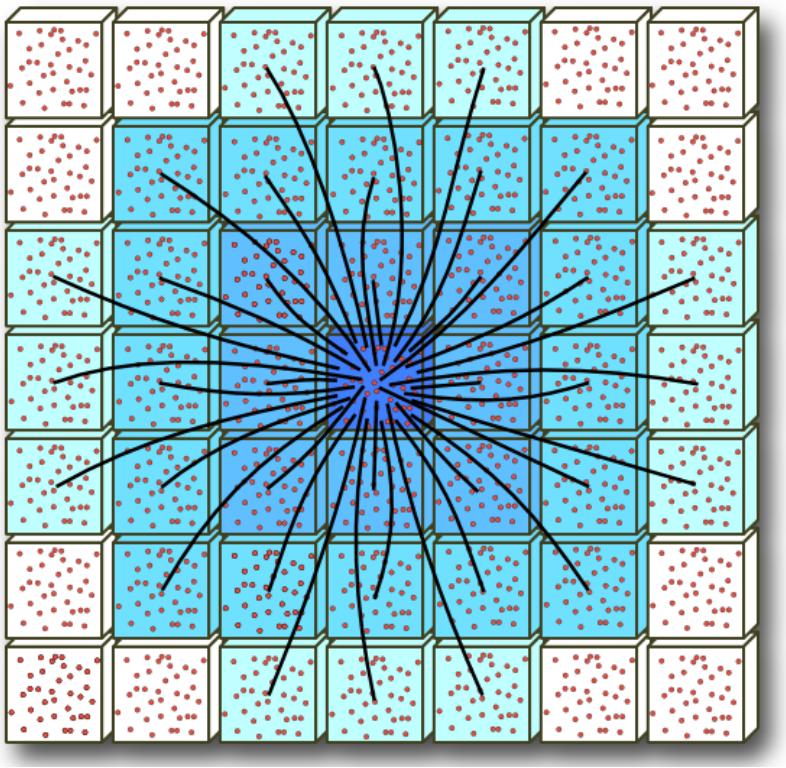


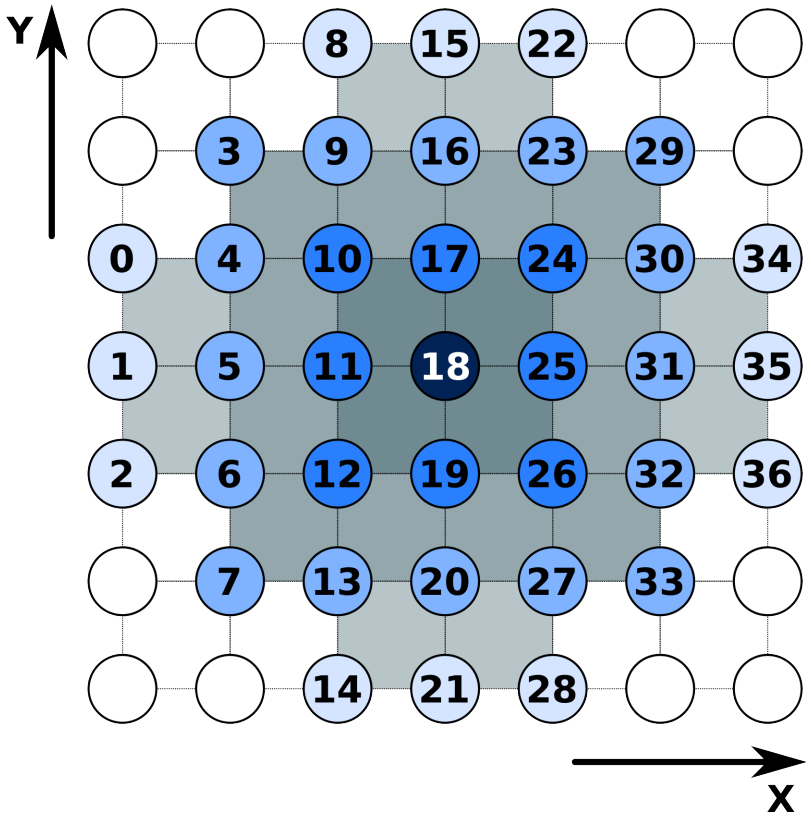


D2Q37: pseudo-code

```
foreach time-step

  foreach lattice-point
    propagate();
  endfor

  boundary_conditions();

  foreach lattice-point
    collide();
  endfor

endfor
```

D2Q37: propagation scheme

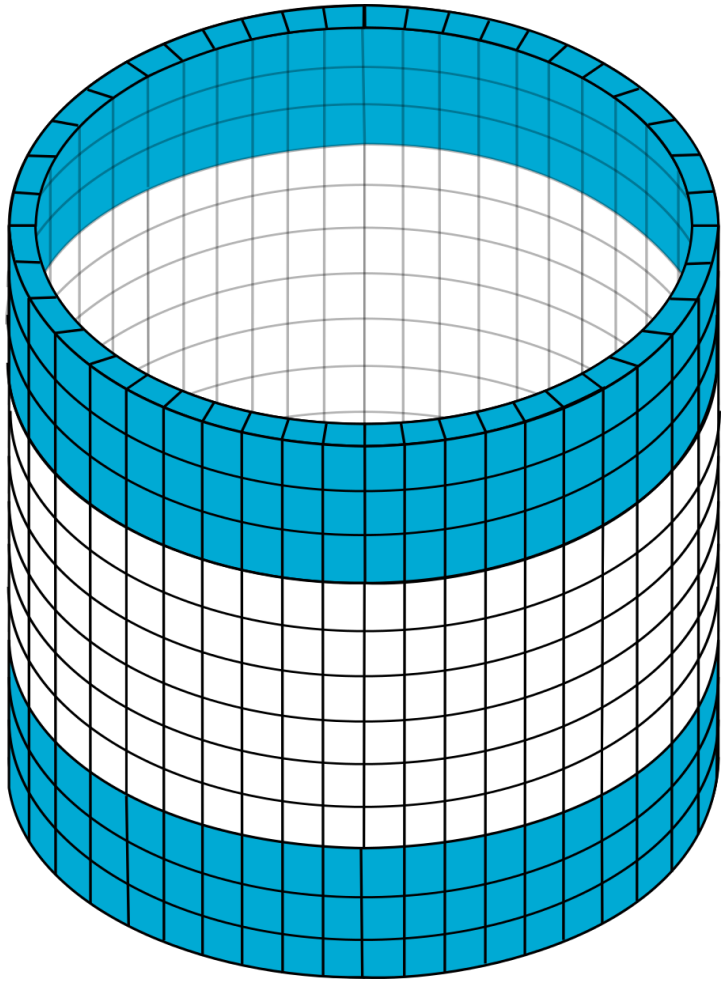




- require to access neighbours cells at distance 1,2, and 3,
- generate memory-accesses with **sparse** addressing patterns.

D2Q37: boundary-conditions

- we simulate a 2D lattice with periodic-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



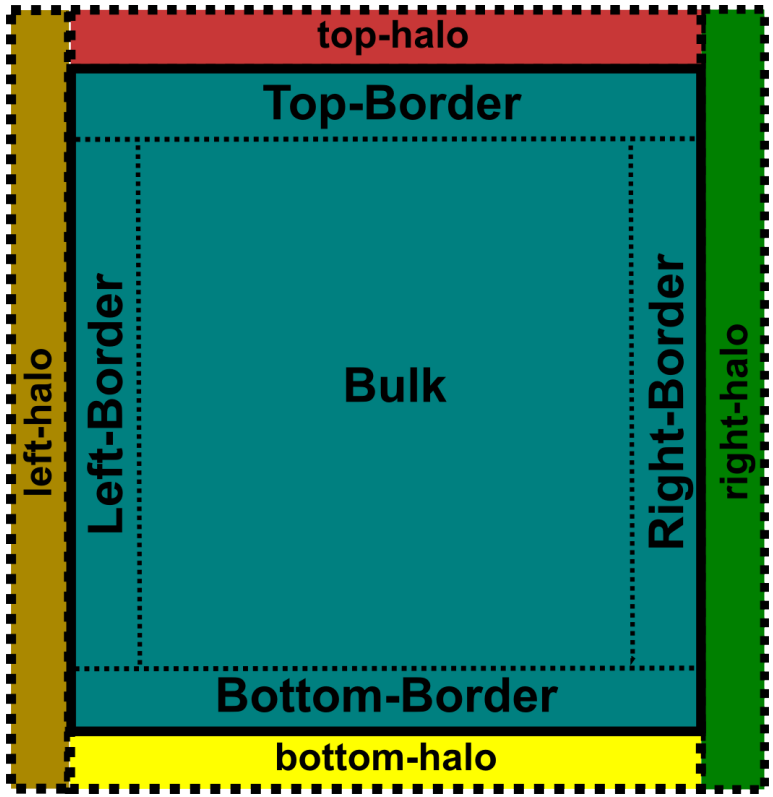
This step (bc) is computed before the collision step.

D2Q37 collision

- collision is computed at each lattice-cell site
- computational intensive:
  for the D2Q37 model requires $\approx$ 6500 DP floating point operations
- computation is completely **local**:
  arithmetic operations require only the populations associated to the site

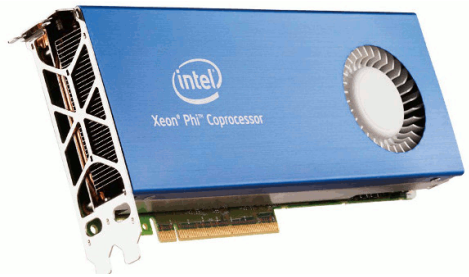
D2Q37 pseudo-code

```
foreach time-step

  propagate_and_collide_bulk()

  update_halos_LR_halos();

  propagate_and_collide_LR_borders()

  update_halos_TB_halos();

  propagate_top_and_bot();

  boundary_conditions();

  collide_top_and_top();

endfor
```



Panorama of Computing Architectures and Programming Tools

Panorama of “modern” processors ... neglecting many details



	apeNEXT Ape	Xeon E5-2697 v4 Broadwell	Xeon 8160 Skylake	Xeon Phy 7230 KNL	P100 Pascal	V100 Volta
Year	2002	2016	2017	2016	2016	2017
f [GHZ]	0.2	2.3	2.1	1.3	1.3	1.3
#cores / SMs	1	18	24	64	56	80
#threads / CUDA-cores	1	36	48	256	3584	5120
$\mathcal{P}_{DP}$ [GFlops]	1.6	650	1533	2662	4759	7000
β_{mem} [GB/s]	3.2	76.8	119.21	400	732	900
β_{net} [GB/s]	1.2	12.5	12.5	12.5	12.5	12.5
Watt	4	145	150	215	250	250
$\mathcal{P}/W$	0.4	4.5	10	12	19	28
$\mathcal{P}/\beta_{mem}$	0.5	8.5	13	6.6	6.5	7.7
$\mathcal{P}/\beta_{net}$	1.3	52	123	213	381	560
$\mathcal{P} \times \lambda$	240	331’000	766’000	1’331’000	2’379’000	3’500’000

β_{net} for apeNEXT is 200 MB/s x 6, $\lambda = 150$ ns

β_{net} for P100 and V100 is Mellanox 4X EDR $\approx$ 12.5 GB/s, $\lambda = 500$ ns

β_{net} for Xeon is OmniPath 100 Gbit/s = 12.5 GB/s, $\lambda = 500$ ns

- several levels of parallelism: $\mathcal{P} = f \times$ #cores $\times$ #opPerCycle $\times$ #flopPerOp
- memory layout plays an important role also for computing performances.

Programming Tools

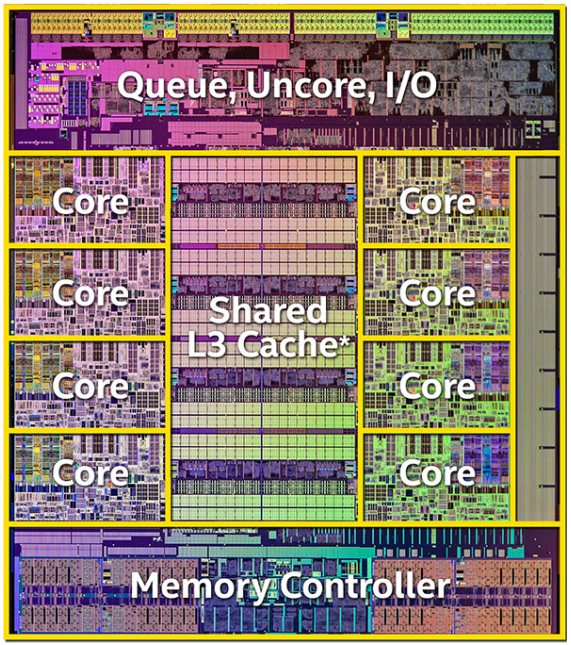
Together with different processors several programming tools are available:

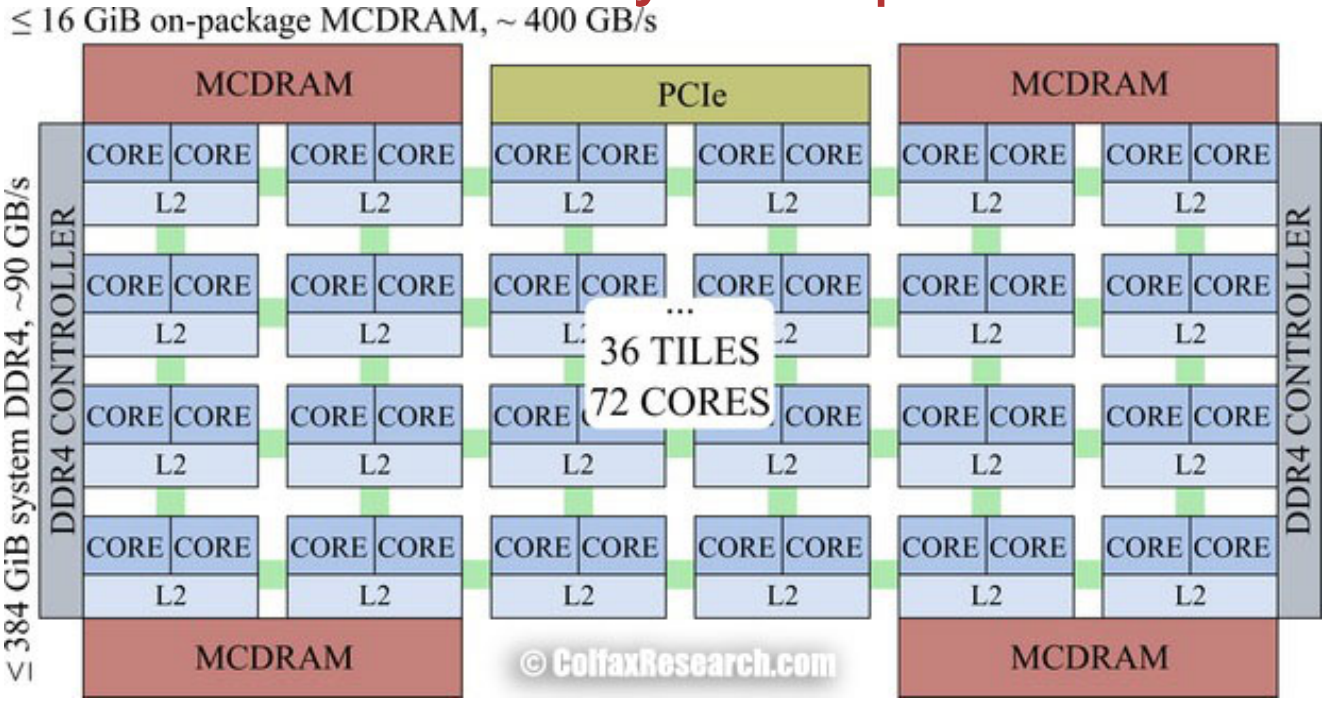
	Target	Model	Comment
pThreads	CPU,MIC	library	OS dependent
OpenMP	CPU,MIC,(GPU)	directives	supported by major compilers
OpenACC	GPU,(CPU)	directives	supported by PGI, Cray and GCC
CUDA	GPU	extensions	supported by NVIDIA compiler
TargetDP	CPU,MIC,GPU	defines	C defines
GRID	CPU,MIC	extensions	C++ extensions + intrinsics

- none fully support all architectures
- TargetDP and GRID a programming frameworks developed by academic research groups

Computing on multi- and many-core Processors

Architecture of Intel multi- and many-core processors





- several cores supporting each 2, 4 threads
- several hierarchy of caches (L1, L2, L3)
- large shared LL cache ($\mathcal{O}(10)$ MB for CPUs, **16 GB** for KNL)
- large VPUs (256-bit, 512-bit)
- **out-of-order** execution (CPU and KNL)

Programming Issues

$$P = f \times \#\text{cores} \times \text{nInstrPerCycle} \times \text{nFlopPerInstr}$$

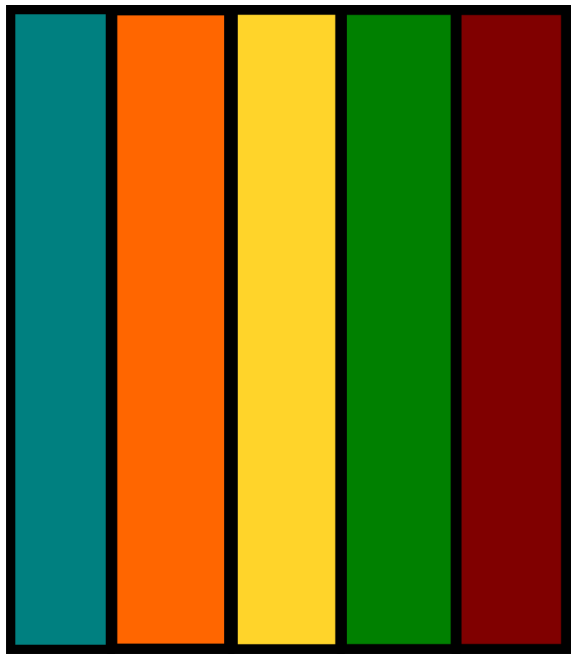
- **core parallelism**:
  keep busy all available cores of a processor;
- **hyper-threading**:
  each core should run several (2, 3, 4) threads to keep busy hardware pipelines and hide memory access latency;
- **vectorization**:
  each core should process several data-elements (4, 8,...) in parallel using vector (streaming) instructions (SIMD parallelism);
- **thread and memory affinity**:
  many systems are dual-processors, threads of same process should be tied to one processor and access the closest memory bank;
- **cache-awareness**:
  exploit cache locality, cache reuse, avoid *reads for ownerships* (RFO). RFO is a read operation issued with intent to write to that memory address;
- **data layout**:
  relevants to expoit efficient vectorization and memory accesses.

Core parallelism and hyperthreading

Run multi-threaded or multi-process application

- single-process multi-threaded application:
  one process spawning several threads
- multi-process application:
  run several processes

Example: for LBM divide/slice the lattice domain among the threads.



Core parallelism: OpenMP

Each core executes a thread processing a sub-lattice:

```
void propagate ( nxt, prv ) {

  #pragma omp parallel for schedule(static)
  for ( xx = XMIN; xx < XMAX; xx++ ) {
    for ( yy = YMIN; yy < YMAX; yy++ ) {
      propagate_site ()
    }
  }

}
```

`xx`-loop is partitioned among threads each executing the inner loop and working on a portion/slice of lattice

OpenMP is a standard broadly supported by several compilers for many architectures.

Core parallelism: MPI

Each process execute the same code on a different sub-lattice

```
for ( step = 0; step < MAXSTEP; step++ ) {
  comm();        // exchange borders
  propagate(); // apply propagate
  bc();          // apply boundary condition to top and bottom rows
  collide();     // compute collide
}
```

MPI vs OpenMP:

- OpenMP is shared-memory, each thread can access data managed by other threads
- MPI not allows shared memory and requires explicit communications to access data managed by other processes.

Vector programming using Intrinsics

Intrinsincs: special functions mapped onto assembly instructions.
Populations of 8 lattice-cells are packed in a AVX vector of 8-doubles

```
struct {
  __m512d vp0;
  __m512d vp1;
  __m512d vp2;
  ...
  __m512d vp36;
} vpop_t;

vpop_t lattice[LX][LY];
```

Intrinsics

$d = a \times b + c \implies$ `d = _m512_fmadd_pd(a,b,c)`

Bad news

Instrinsics are not portable across different architectures !

Vector programming through directives

Directives tell the compiler how to vectorize the code.

```
typedef struct {
  double *p[NPOP];
} pop_soa_t;

  // snippet of propagate code to move population index 0
  #pragma omp parallel for schedule(static)
  for ( xx = XMIN; xx < stopx; xx++ ) {
    #pragma vector nontemporal
    for( yy = YMIN; yy < YMAX; yy++ ) {
      idx = IDX(xx,yy);
      (nxt->p[0])[idx] = (prv->p[0])[idx+OXM3YP1];
    }
  }
```

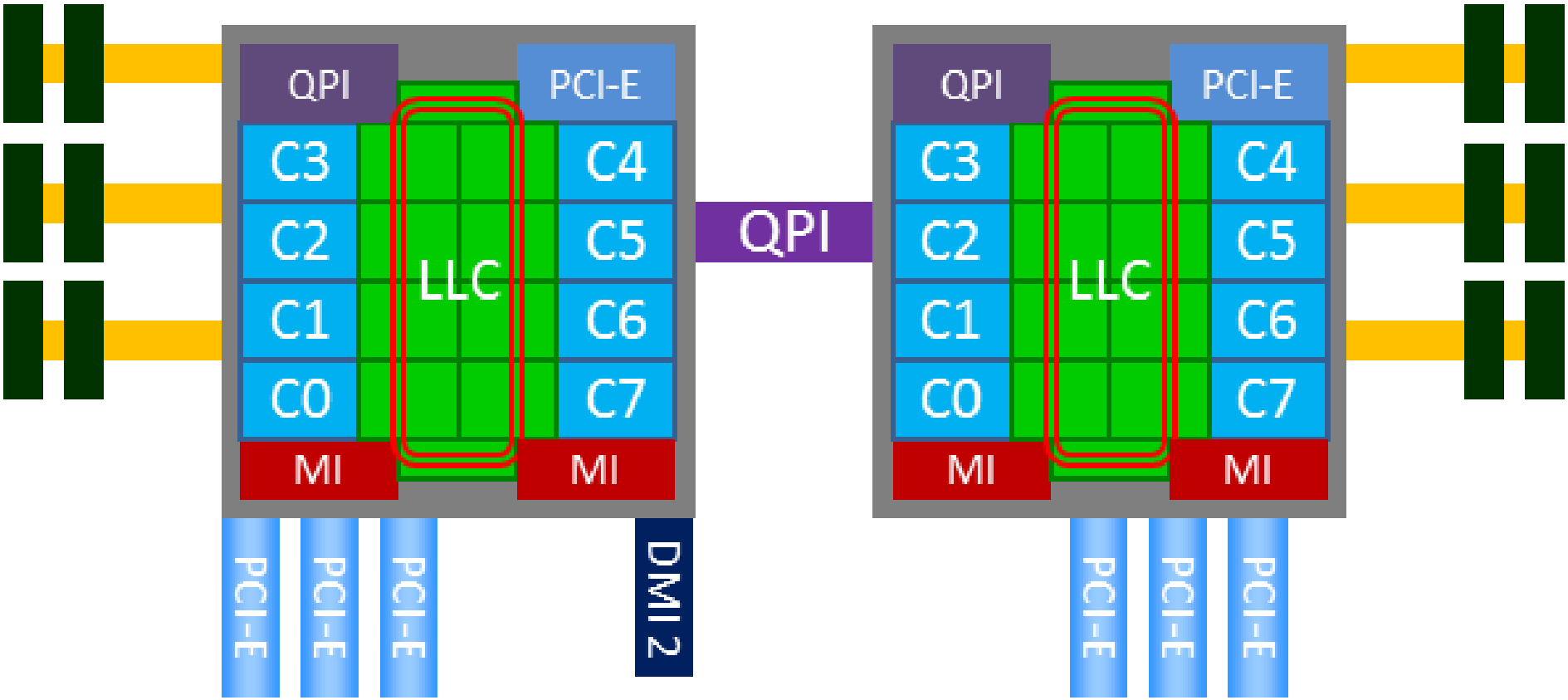
- `pragma vector`: `yy` loop can be vectorized: 2 or more iterations can be executed in parallel using SIMD instructions;
- `nontemporal`: store can by-pass read-for-ownership (RFO).

Good/Bad news

Directives are fully portable across several architectures !
Proprietary directives are not fully portable: Intel vs OpenMP !

Thread and Memory Affinity

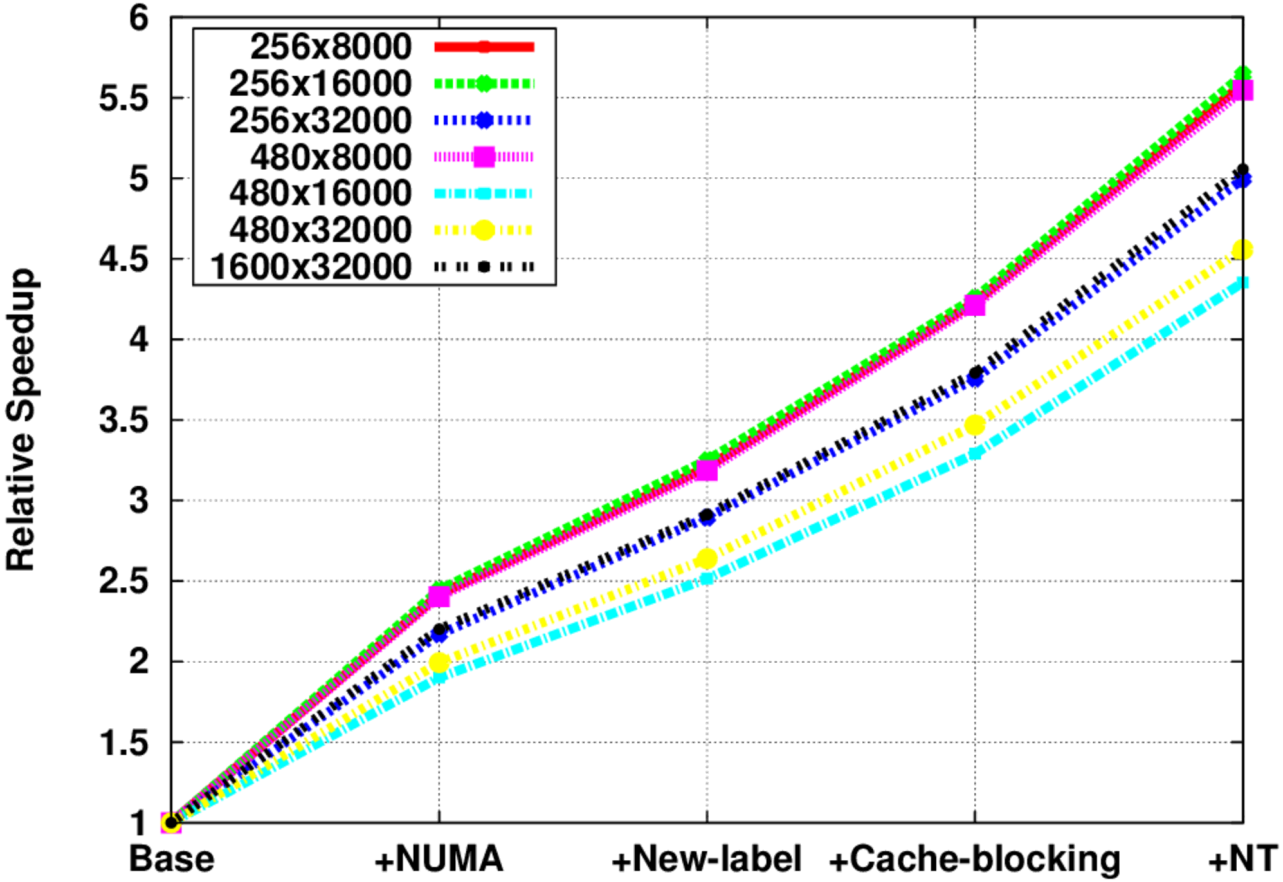
Typical architecture of a standard cluster node:



- affinity of threads and memory allocation
- control affinity with `numactl` or throught specific flags of `mpirun` or job submission system.

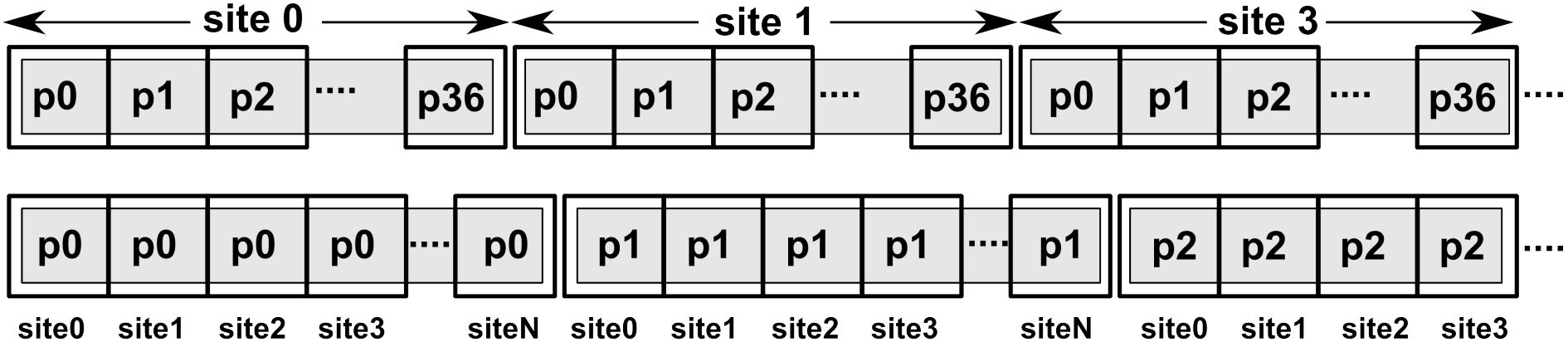
Optimization of Propagate

Running on a dual-socket Xeon E5-2680 Sandybridge:



Version including all optimizations performs at $\approx$ 58 MB/s, $\approx$ 68% of peak (85.3 MB/s) and very close to memory-copy (68.5 MB/s).

Memory Layout for vectorizing LBM: AoS vs SoA

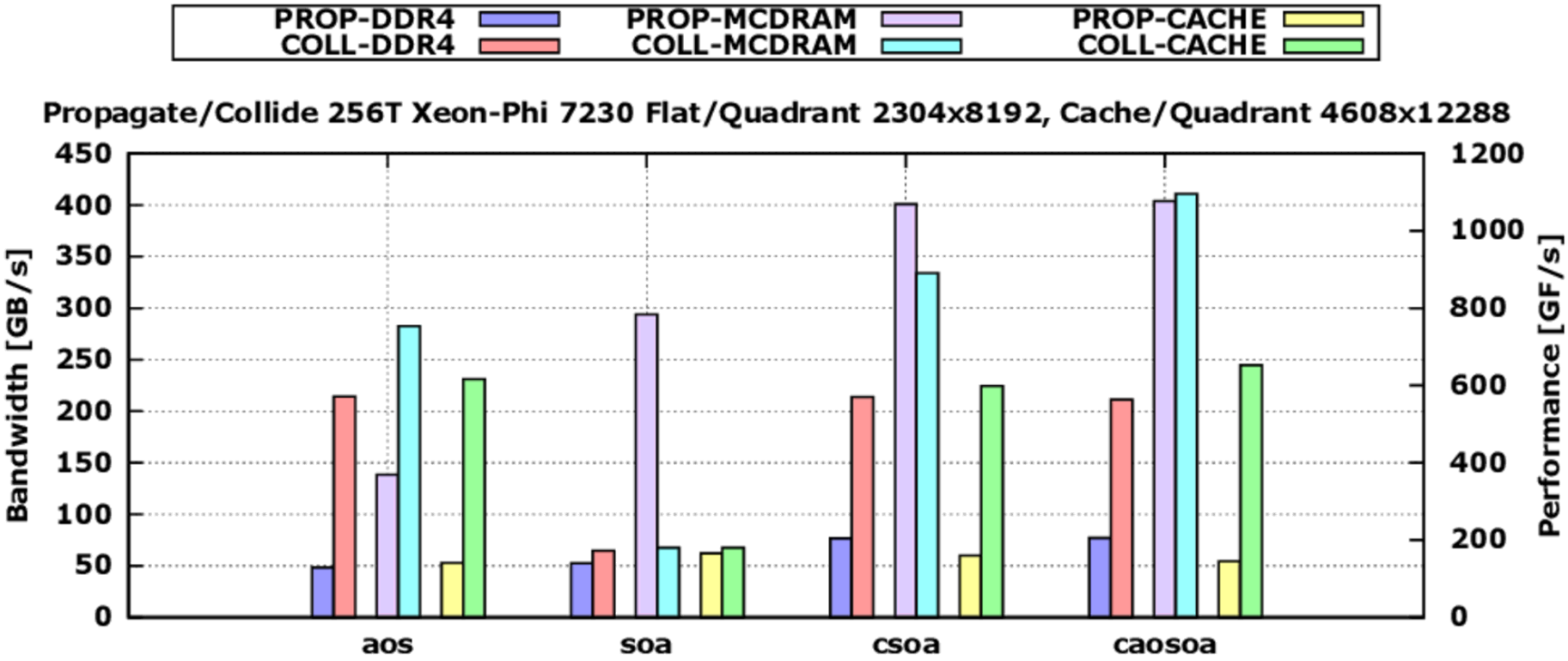



```
#define N (LX*LY)
typedef struct {
  double  p1; // population  1
  double  p2; // population  2
  ...
  double p37; // population 37
} pop_t;

pop_t lattice[N];
```

```
#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;
```

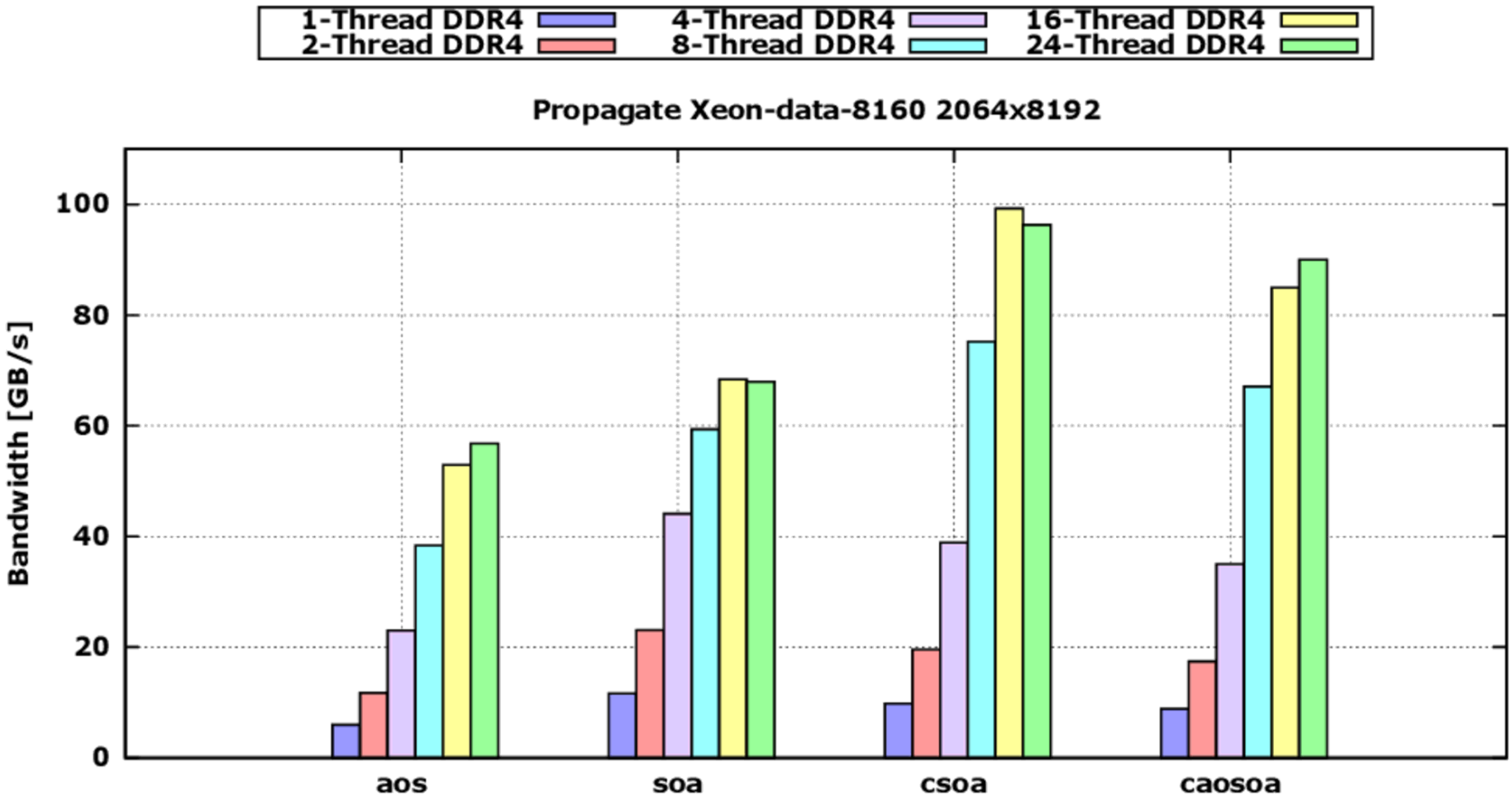
- AoS (upper) site pop. data of each site close in memory
- SoA (lower) same-index pop. data of different site close in memory

Results: Propagate and Collide Performance on KNL



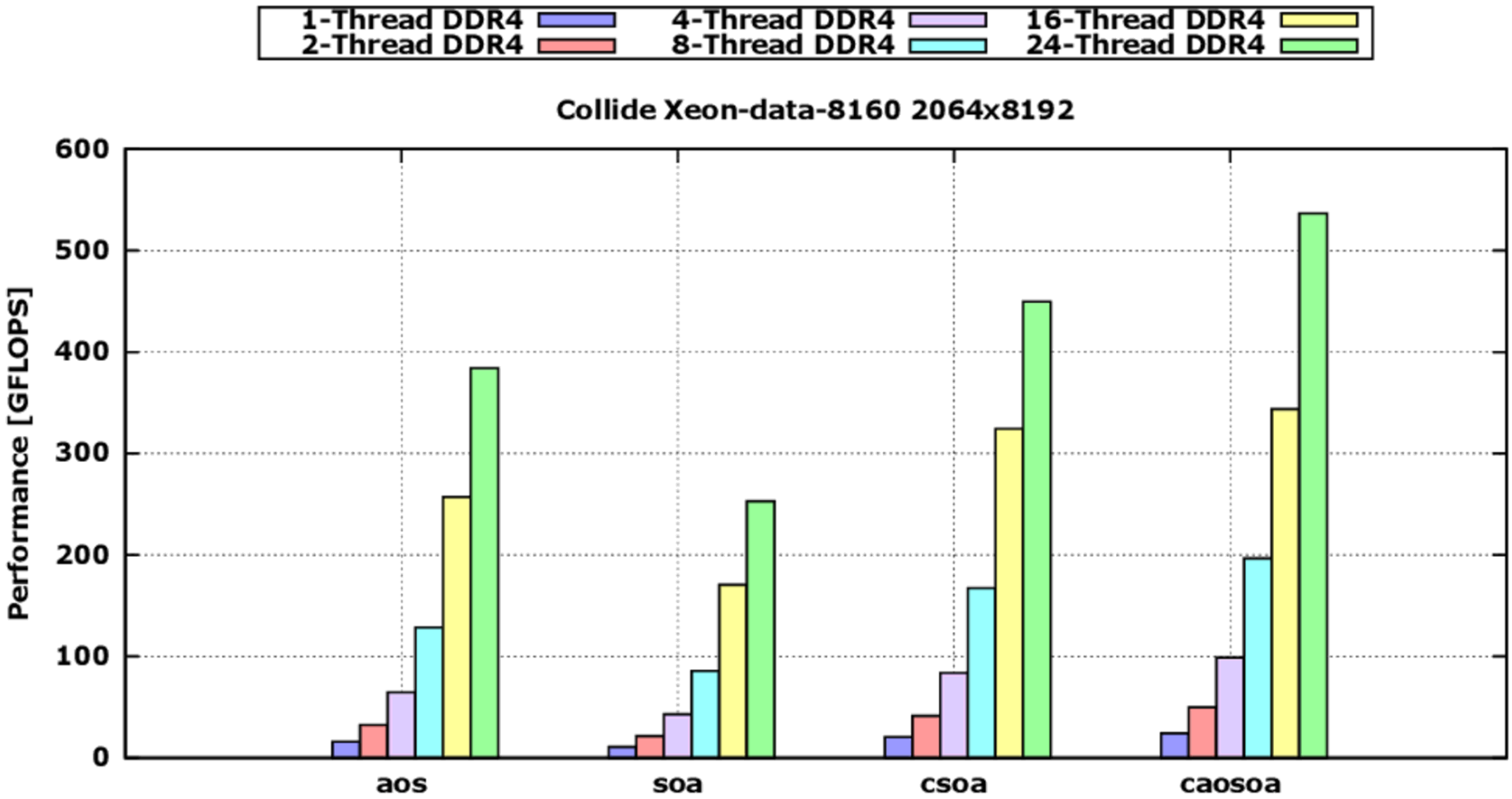
- Propagate:
  - FLAT modes: performances increases from $AoS \rightarrow SoA \rightarrow CSoA$
  - MCDRAM: $\approx$ 400 GB/s, DDR4: $\approx 80 GB/s$, CACHE: $\approx 60 GB/s$
- Collide:
  - FLAT-MCDRAM: performance increases from $AoS \rightarrow CSoA \rightarrow CAoSoA$
  - $CAoSoA$: sustained performance of 1 Tflops ($\approx$ 30% of raw peak)
  - FLAT-DDR4 and Cache modes: performances are limited by memory bandwidth.

Results: Propagate Performance on SkyLake



propagate $\approx$ 100 GB/s, *approx*85% of raw peak.

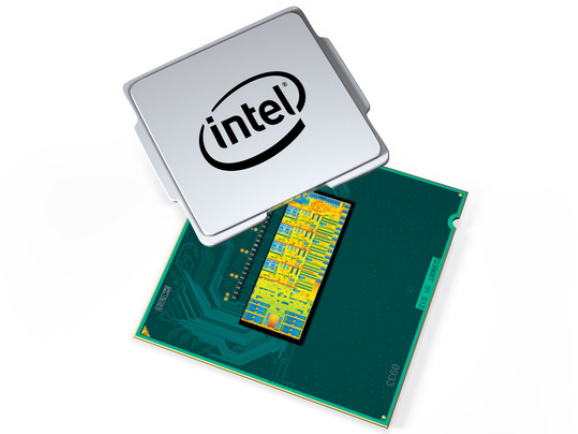
Results: Collide Performance on Skylake

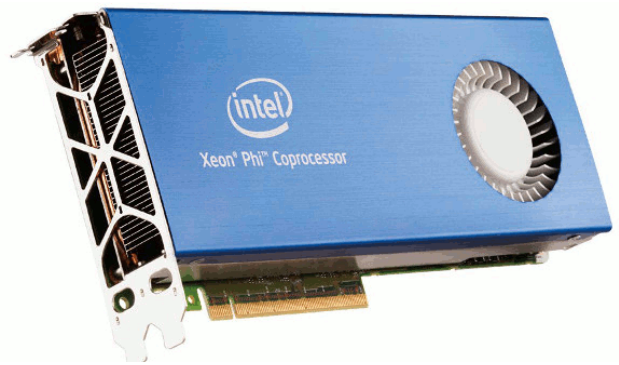


collide $\approx$ 530 GFlops. $\approx$ 35% of raw peak.

Computing on GP-GPUs

GP-GPUs: why are they interesting ?



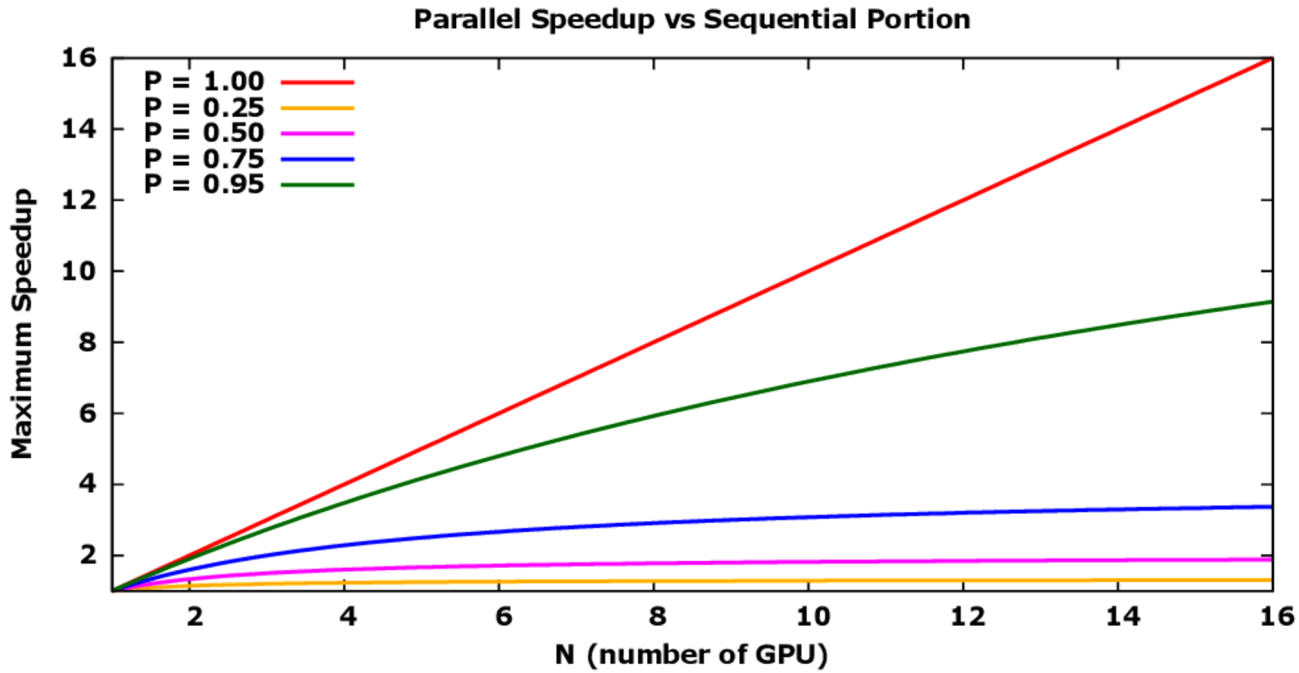


	Xeon E5-2697 v4	Xeon 8160	Xeon Phy 7230	P100	V100
Year	2016	2017	2016	2016	2017
$\mathcal{P}_{DP}$ [GFlops]	650	1533	2662	4759	7000
β_{mem} [GB/s]	76.8	119.21	400	732	900
$\mathcal{P}/W$	4.5	10	12	19	28

Why may not be interesting ?

Amdahl's Law

Speedup of an accelerated program is limited by the fraction of time run on the host.



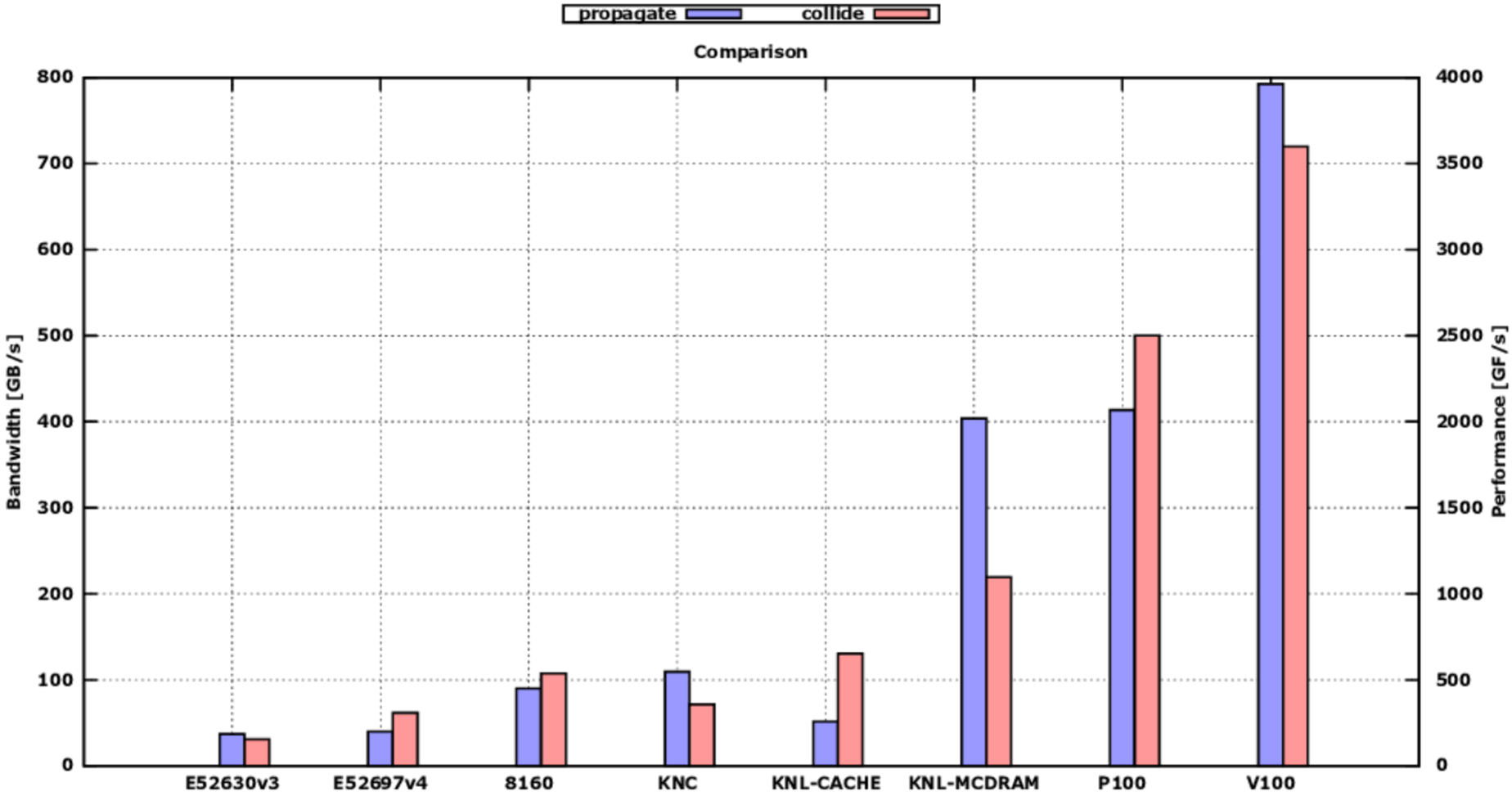
Accelerating the 3/4 of the code, the maximum theoretical achievable speedup is limited to 4 !!!

GP-GPU architecture: Basics

	C2050 / C2070	K20X	K40	K80	P100	V100
GPU	GF100	GK110	GK110B	GK210 $\times$ 2	P100	V100
Number of SMs	16	14	15	13 $\times$ 2	56	80
Number of CUDA-cores	448	2688	2880	2496 $\times$ 2	3584	5120
Nominal clock frequency (MHz)	1150	735	745	562	1328	1300
Nominal DP performance (Gflops)	515	1310	1430	935 $\times$ 2	4760	7450
Boosted clock frequency (MHz)	–	–	875	875	1480	1530
Boosted DP performance(Gflops)	–	–	1660	1455 $\times$ 2	5304	7800
Total available memory (GB)	3 / 6	6	12	12 $\times$ 2	16	16
Memory bus width (bit)	384	384	384	384 $\times$ 2	4096	4096
Peak mem. BW (ECC-off) (GB/s)	144	250	288	240 $\times$ 2	732	900
Max Power (Watt)	215	235	235	300	300	300

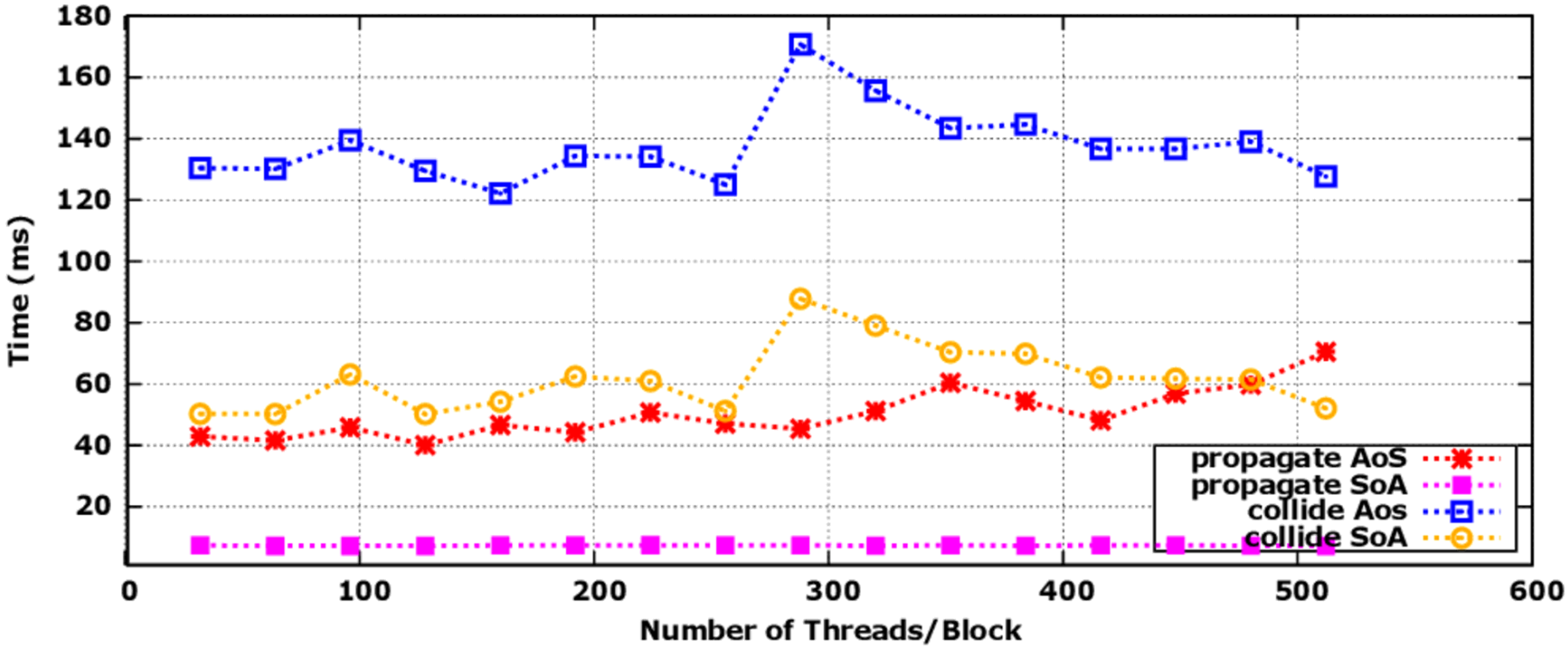
- multi-core processors
- processing units are called *Streaming Multiprocessors* (SM on Fermi and SMX on Kepler)
- each processing unit has several compute-units called *CUDA-cores*
- at each clock-cycle each SM execute a group (*warp*) of 32 operations called *CUDA-threads*
- CUDA-threads are executed in *Single Instruction Multiple Threads* SIMT fashion
- SIMT execution is similar to SIMD but more flexible, e.g. different threads of a SIMT group are allowed to take different branches at a performance penalty.

Why GPUs can be interesting: LBM Performance Comparison



Memory layout for GPUs: AoS vs SoA

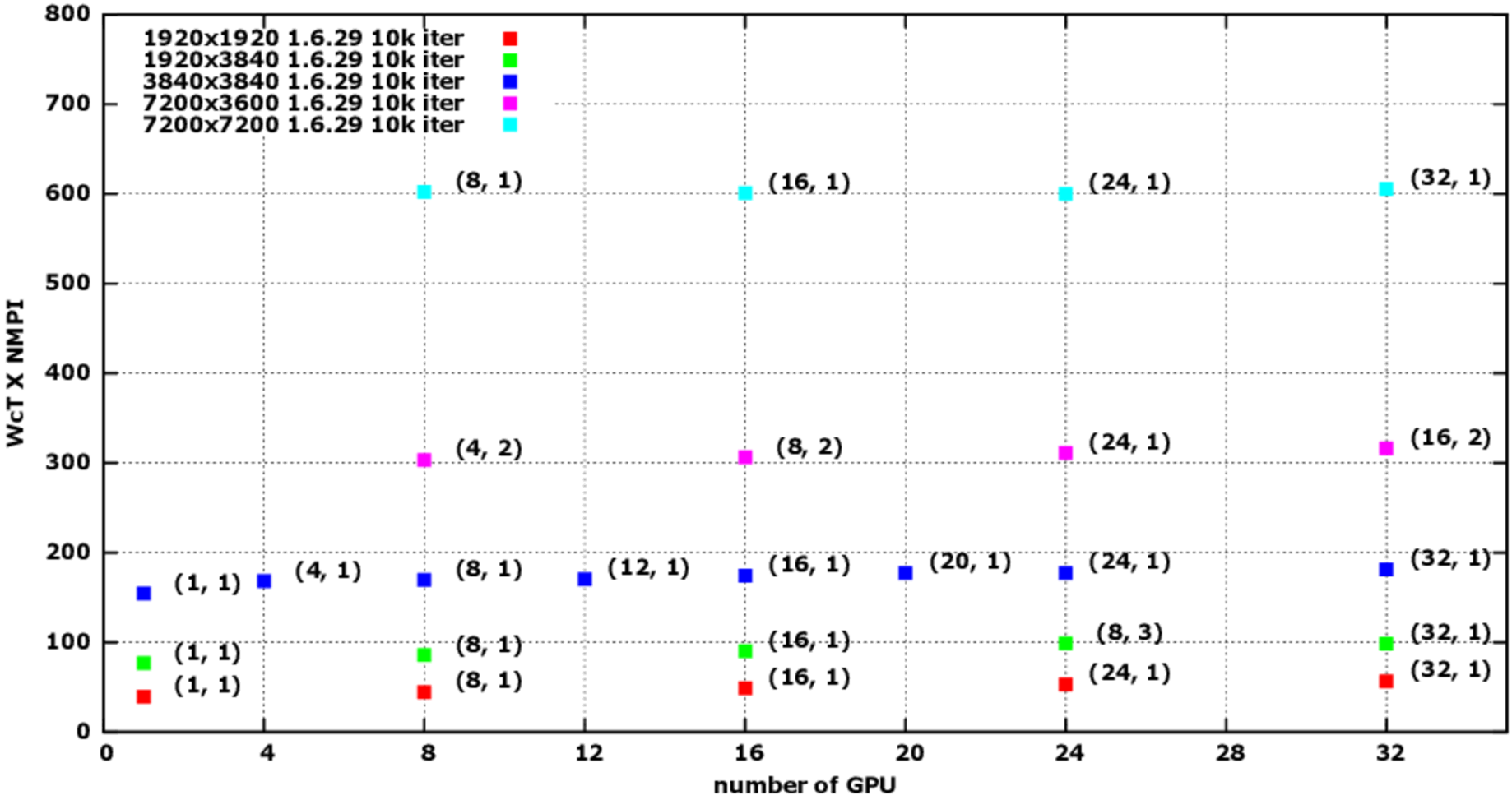
On GPUs memory layout is relevant for performances:



NVIDIA K40 GPU

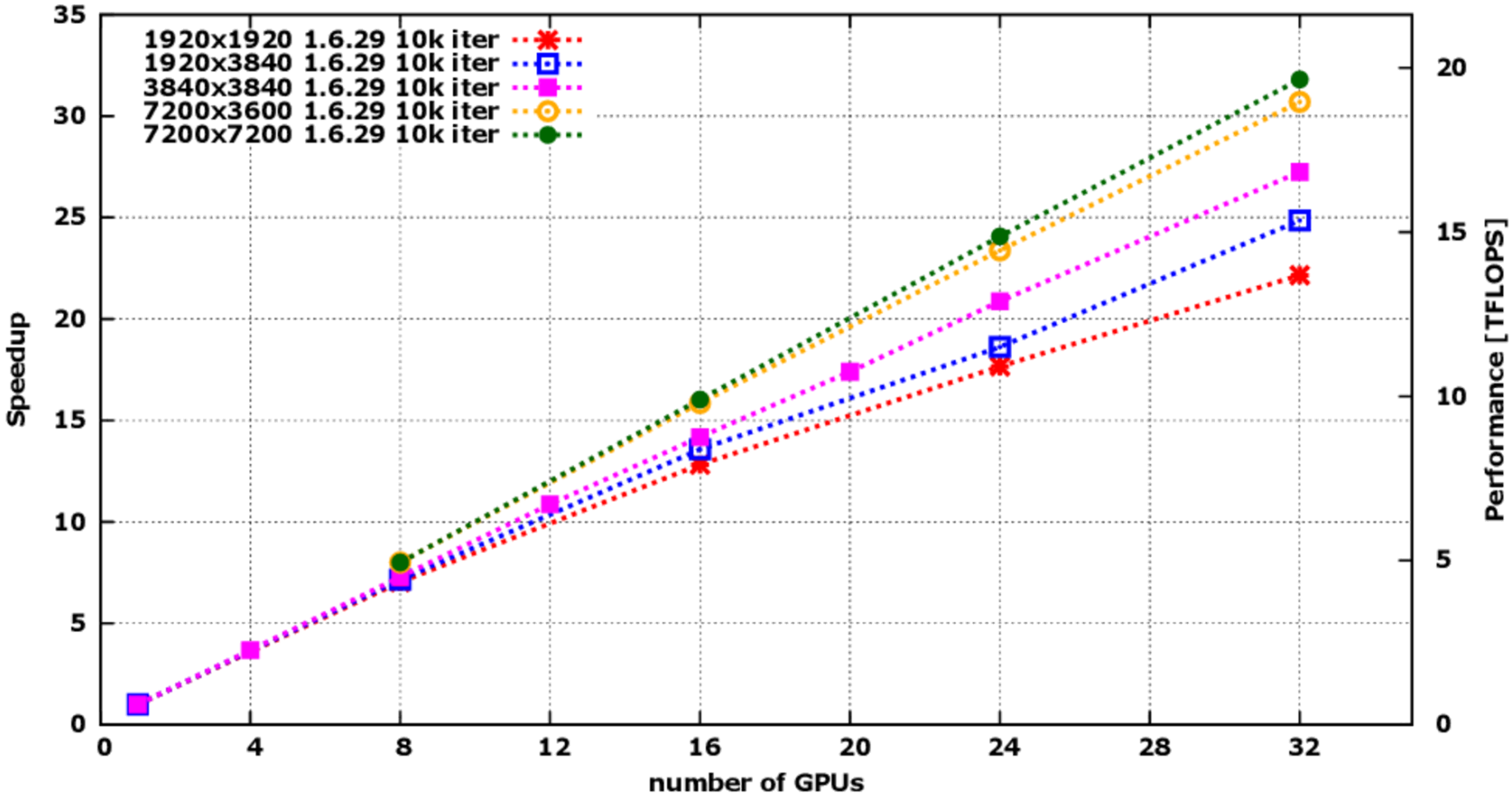
- propagate $\approx$ 10X faster, collide $\approx$ 2X faster
- SoA gives the best results because enable vector processing and coalescing access to memory.

Performance: Scalability



- for each number of GPUs – in most cases – the lowest time corresponds to 1D-tiling
- scalability is limited to a small number of GPUs

Strong Scaling and Aggregate Performance



Summary and Conclusions

Summary and Conclusions

- several different architectures in the panorama of processors, make it difficult to develop and maintain application codes
- each processor architecture needs specific optimizations.

Two major hardware choices:

- GP-GPU systems
  - offer high computing performance
  - require specific programming frameworks
  - CUDA codes are not portable across architectures
- multi- and many-core x86 processors
  - offer more flexibility
  - backward compatible
  - good computing performances
  - all levels of parallelism need to be exploited
  - MIC architecture (KNH) . . . has been abandoned (!?).

Summary and Conclusions: optimizations

- vectorization:
  - intrinsic operations: good performance, not portable;
  - directive-based approaches like OpenMP and OpenACC:
    - directives can be ignored,
    - directives make the code more portable,
    - specific compiler directives are not portable.
  - OpenACC:
    - an easy way to use GPUs
    - code does not need many changes
    - also targets standard multi-core processors
  - OpenMP: standard support also accelerators
- core/thread parallelism
  - OpenMP is a standard supported by all major compilers
  - OpenMP allow easy distribution of workload across cores/threads
  - MPI is a standard but does not exploit shared-memory
- data layout: crucial for performance, different kernels may require different data layouts

Conclusions

Multi and many-core architectures have a big inpact on programming.

- Efficient programming requires to exploit all features of hardware systems: core parallelism, data parallelism, cache optimizations, NUMA (Non Uniform Memory Architecture) system
- several architectures and several ways to program them
- data structure have a big impact on performance
- portability of code and performance is necessary
- energy efficiency is a big issue

Programming commodity HPC system require deep knowledge !

Acknowledgments

- Luca Biferale, Mauro Sbragaglia, Patrizio Ripesi
  University of Tor Vergata and INFN Roma, Italy
- Andrea Scagliarini
  CNR, Istituto per le Applicazioni del Calcolo, Rome, Italy
- Filippo Mantovani
  Barcelona Supercomputer Center, Spain
- Enrico Calore, Alessandro Gabbana, Marcello Pivanti, Sebastiano Fabio Schifano, Raffaele Tripiccione
  University and INFN of Ferrara, Italy
- Federico Toschi
  Eindhoven University of Technology The Netherlands, and CNR-IAC, Roma Italy
- Fabio Pozzati, Alessio Bertazzo, Gianluca Crimi, Elisa Pellegrini, Nicola Demo, Giovanni Pagliarini
  University of Ferrara

Results presented here have been developed in the frameworks of the INFN projects: COKA, COSA and SUMA.

Thanks to CINECA, INFN, BSC and JSC for access to their systems.