Parallel Approaches To Lattice Boltzman Methods

Sebastiano Fabio Schifano

University of Ferrara and INFN-Ferrara

Introductory School on Parallel Programming and
Parallel Architecture for High-Performance Computing

October 10th, 2016

ICTP Trieste, Italy

Outline

Focus

Parallel Approaches To Lattice Boltzman Methods.

1. introduction
2. multi- and many-core processors
3. programming issues
4. Lattice-Boltzmann as case study
5. Portability
6. Energy Issues

the one million dollar question

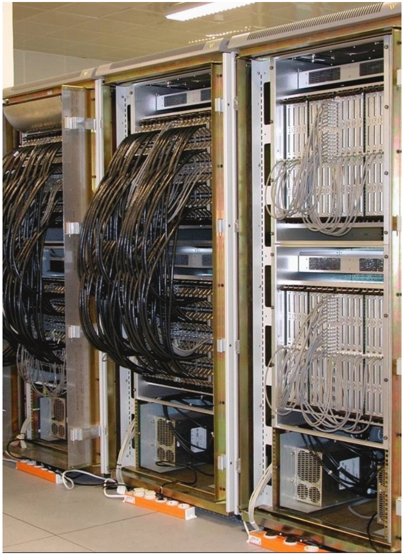
which is the best computing system to use ?

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: FPGA-based system for spin-glass simulations
- QPACE: Cell-based machine, mainly LQCD
- AuroraScience: multi-core based machine

APEmille e apeNEXT (2000 and 2004)



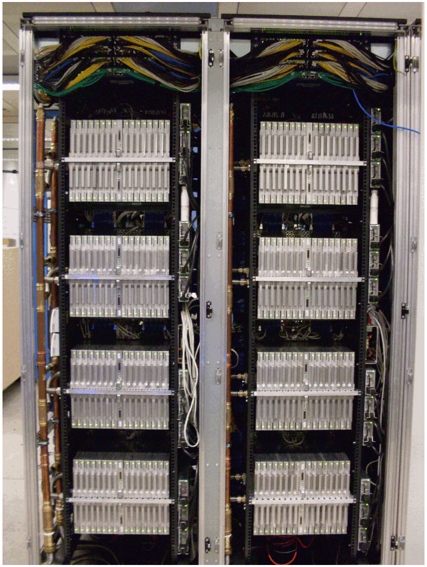
Janus (2007)

- 256 FPGAs
- 16 boards
- 8 host PC

QPACE Machine (2008)

- 8 backplanes per rack
- 256 nodes (2048 cores)
- 16 root-cards
- 8 cold-plates

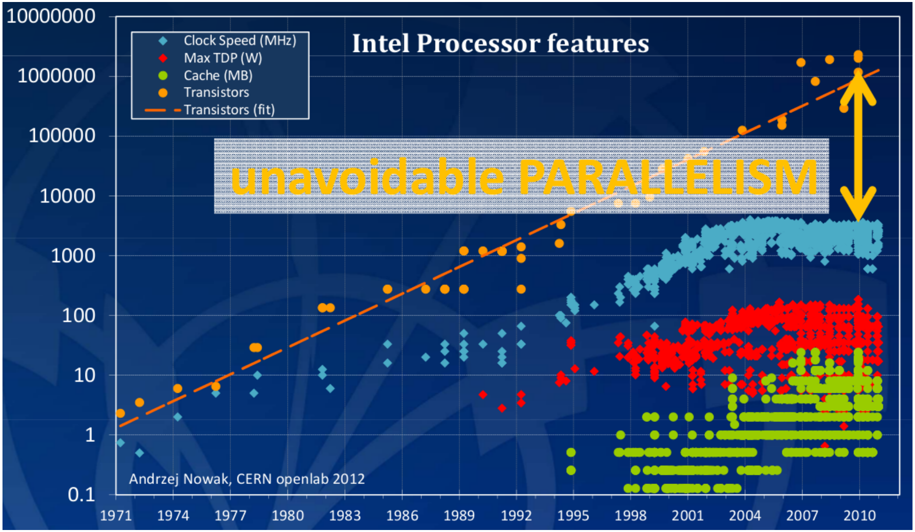
- 26 Tflops peak double-precision
- 35 KWatt maximum power consumption
- **750 MFLOPS / Watt**
- TOP-GREEN 500 in Nov.'09 and July'10



Use of recent processors

- QPACE has been the first attempt (in our community) to use a **commodity** processor interconnected by a **custom** network
- what I would like to discuss now is how and how well we can use recent developed processors for our applications
- which issues we have to face out ?
- how to program them ?

Hardware Evolution

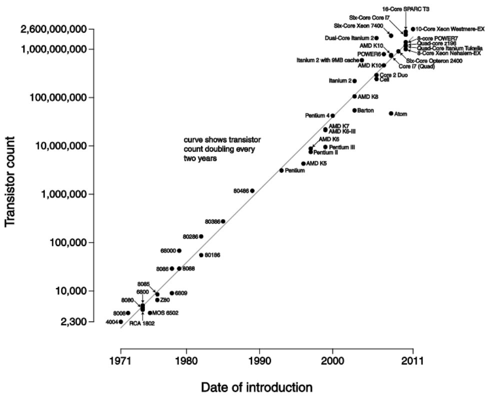


The Multi-core processors era begins !

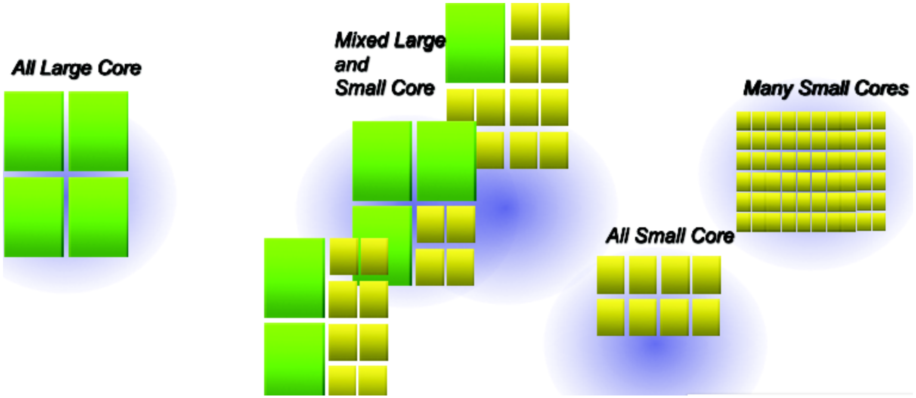
Multi-core architecture allows CPU performances to scale according to Moore's law.

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

Microprocessor Transistor Counts 1971-2011 & Moore's Law

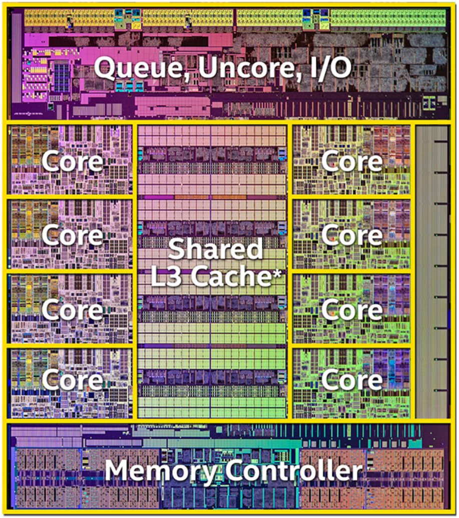


Many Different Cores



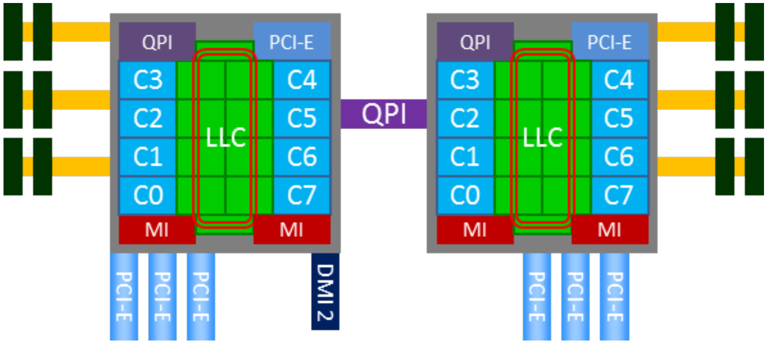
- all large core: multi-core Intel x86 CPUs
- many small core: NVIDIA GPUs accelerators
- all small cores: MIC architectures, Intel Xeon Phi accellerator
- mixed large and small cores: Cell, AMD-Fusion, NVIDIA-Denver

“Classic” Intel multi-core CPU Architectures



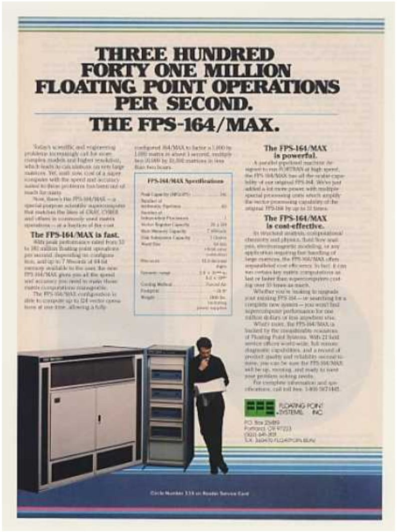
4-8+ cores, 1 shared L3-cache (*nehalem, sandybridge, haswell, skylake, . . .*)

Numa SMP Multi-socket Multi-core Systems



- *Symmetric Multi-processor Architecture* (SMP)
- *Non Uniform Memory Architecture* (NUMA)

Many-core Accelerators: Is this really a new concept ?



THREE HUNDRED FORTY ONE MILLION FLOATING POINT OPERATIONS PER SECOND.

THE FPS-164/MAX.

Today's scientific and engineering problems increasingly call for more complex models and higher resolution, which leads to calculations on very large matrices. Yet, until now, cost of a supercomputer with the speed and accuracy suited to these problems has been out of reach for many.

Now, there's the FPS-164/MAX — a special-purpose scientific supercomputer that matches the likes of CRAY, CYBER and others in commonly used matrix operations — at a fraction of the cost.

The FPS-164/MAX is fast.

With peak performance rated from 33 to 341 million floating point operations per second, depending on configuration, and up to 7 Mwords of 64-bit memory available to the user, the new FPS-164/MAX gives you all the speed and accuracy you need to make those matrix computations manageable.

The FPS-164/MAX configuration is able to compute up to 124 vector operations at one time, allowing a fully configured 164/MAX to factor a 1,000 by 1,000 matrix in about 1 second, multiply two 10,000 by 10,000 matrices in less than two hours.

FPS-164/MAX Specifications

Peak Capacity (MFLOPS)	341
Number of Arithmetic Pipelines	[illegible]
Number of Independent Processors	1
Vector Register Capacity	[illegible]
Main Memory Capacity	7 MWords
Disk Subsystem Capacity	[illegible]
Word Size	64 bits +8-bit error correction
Precision	15.9 decimal digits
Dynamic range	[illegible]
Cooling Method	Forced Air
Footprint	[illegible]
Weight	1800 lbs. including power supplies

The FPS-164/MAX is powerful.

A parallel pipelined machine designed to run FORTRAN at high speed, the FPS-164/MAX has all the scalar capability of our original FPS-164. We've just added a lot more power with multiple special processing units which amplify the vector processing capability of the original FPS-164 by up to 31 times.

The FPS-164/MAX is cost-effective.

In structural analysis, computational chemistry and physics, fluid flow analysis, electromagnetic modeling, or any application requiring fast handling of large matrices, the FPS-164/MAX offers unparalleled cost-efficiency. In fact, it can run certain key matrix computations as fast or faster than supercomputers costing over 10 times as much.

Whether you're looking to upgrade your existing FPS-164 — or searching for a complete new system — you won't find supercomputer performance for one million dollars or less anywhere else.

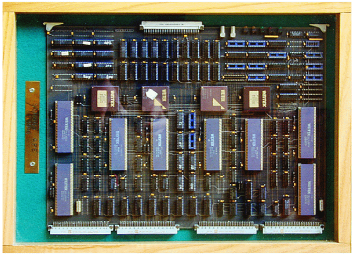
What's more, the FPS-164/MAX is backed by the considerable resources of Floating Point Systems. With 21 field service offices world-wide, full remote diagnostic capabilities, and a record of product quality and reliability second to none, you can be sure the FPS-164/MAX will be up, running, and ready to meet your problem solving needs.

For complete information and specifications, call toll free: 1-800-547-1445.

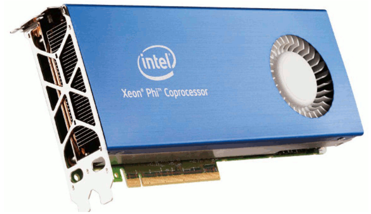
FLOATING POINT SYSTEMS INC

P.O. Box 23489
Portland, OR 97223
(503) 641-3151
TLX: 360470 FLOATPOINT BEAV

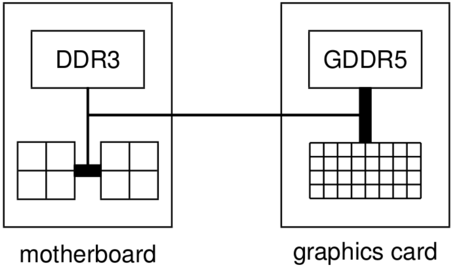
Circle Number 333 on Reader Service Card



Accelerators: today they look like much better !



A common configuration is:



Accelerators as Sausage Machines

A processor is like a sausage machine:

- . . . no input-meat . . . no output-sausage !!
- . . . it produces results if you provide enough input-data !!

Performance Evaluation: Amdhal's Law

How much can I accelerate my application ?

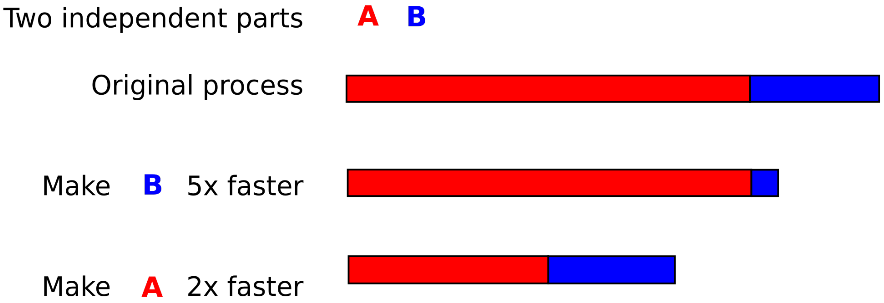
Amdahl's Law approximately states:

> *Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph.*
>
> *No matter how fast you drive the last half: it is impossible to achieve 90 mph average before reaching the second city.*
>
> *Since it has already taken you 1 hour and you only have a distance of 60 miles total; going* **infinitely** *fast you would only achieve 60 mph.*

Accelerator and the Amdahl's Law

Amdahl's Law

The speedup of an accelerated program is limited by the time needed for the host fraction of the program.



Accelerator Issues: the Amdahl's law

Let assume that:

- a computation has a execution time:

$$T_s = t_A + t_B, \quad t_A = P \cdot T_s, \quad t_B = (1 - P) \cdot T_s$$

- execution time of portion *A* can be improved by a factor *N* using an **accelerated** version of the code
- execution time of portion *B* is run on the host and remain un-parallelized.

Under this assumptions the execution time of the new code is

$$T_p = t_A/N + t_B = (P \cdot T_s)/N + (1 - P) \cdot T_s$$

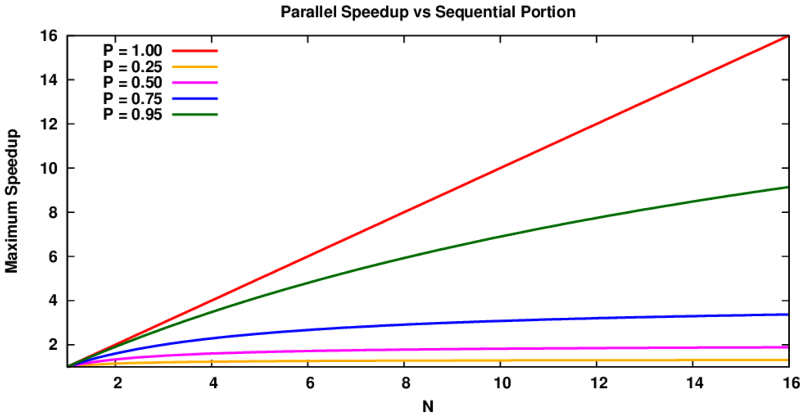
Then the **speedup** (a measure of how fast is the new code) is:

$$S(n) = T_s/T_p = \frac{T_s}{((P \cdot T_s)/N + (1-P) \cdot T_s)} = \frac{1}{(\frac{P}{N} + (1-P))}$$

where *P* is the fraction of code accelerated, and *N* is the improving factor.

Accelerator Issues: the Amdahl's law

Plotting the speed-up as function of N:



even if I improve the 3/4 of my code by large values of N the maximum speedup I can achieve is limited to 4 !!!

Accelerator Issues: Host-Device Latency

. . . bandwidth problems can be cured with money.

Latency problems are harder because the speed of light is fixed and we can't bribe Nature.

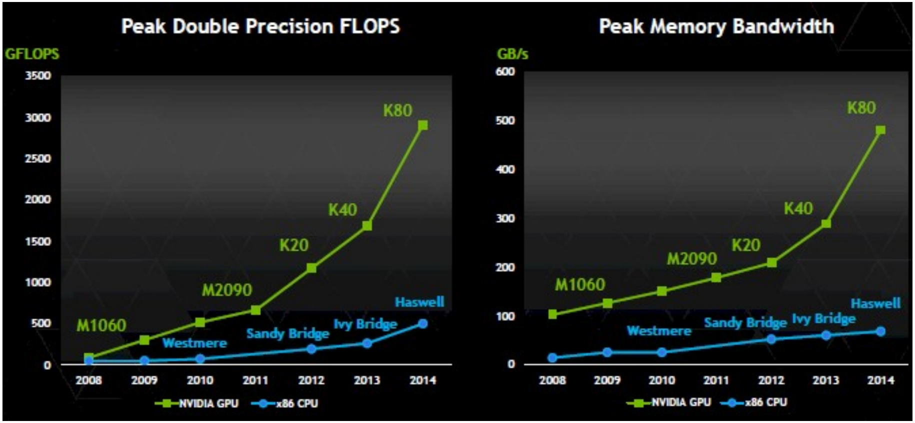
Anonymous

Moving data between Host and GPU is limited by **bandwidth** and **latency**:

$$T(n) = l + n/B$$

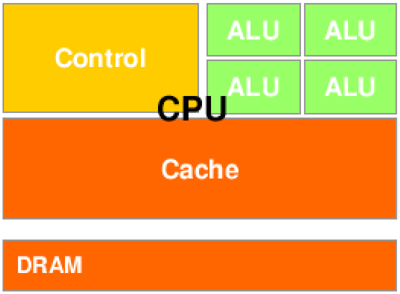
- accelerator processor clock period is $\mathcal{O}(1)$ns
- PciE latency is $\mathcal{O}(1)\mu$s

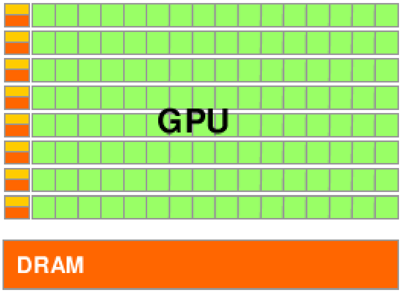
Let’s look on GPUs: GPU evolution



- GPUs evolve much faster in terms of raw-computing power
- Fast-growing video-game market forces innovation

GPUs vs CPUs architecture



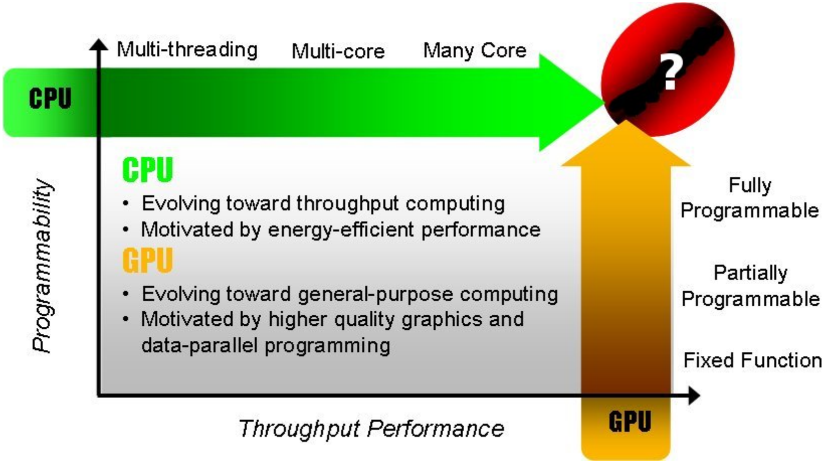


- GPUs specialized for highly data-parallel and intensive computation (exactly what rendering is about)
- more transistors devoted to **data-processing** rather than **data caching** and **flow-control**

GPU Programming Issues

- host-to-device latency:
  Amdhal's law
- memory access latency:
  $\mathcal{O}(10^3)$ processor cycles, run many threads to hide memory-latency
- high-data parallelism:
  many threads-per-block and many blocks-per-grid

Where we are going ?



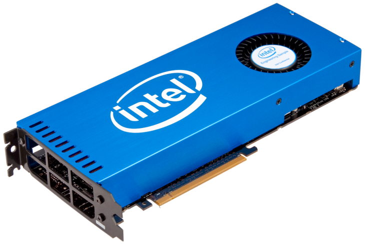
CPU
- Evolving toward throughput computing
- Motivated by energy-efficient performance

GPU
- Evolving toward general-purpose computing
- Motivated by higher quality graphics and data-parallel programming

…towards a convergence between CPU and GPU architectures

First attempt to merge GPU and CPU concepts: MIC architectures

MIC: Many Integrated Core Architecture

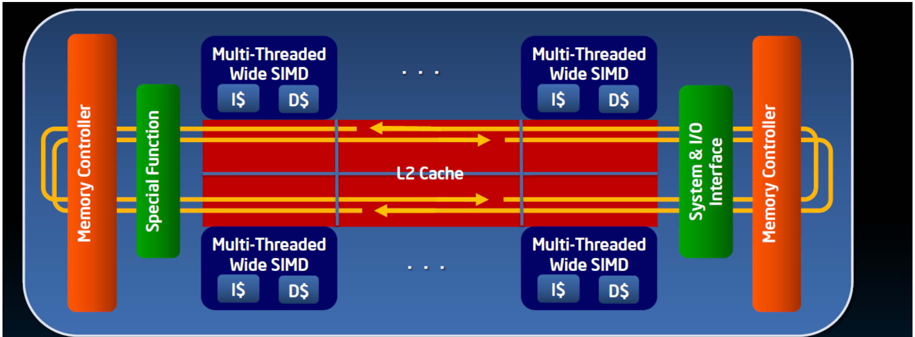


- Knights Ferry: development board
- Knights Corners: production board
- Intel Xeon-Phi: commercial board
- Knights Landing: latest development

Intel MIC Systems: Knights Corners

- Yet another accelerator board
- PCIe interface
- Knights Corners: 61 x86 core @ 1.2 GHz
- each core has 32KB L1 instruction cache, 32KB L1 data cache, and 256KB L2 cache
- 512-bit SIMD unit: 16 SP, 8 DP
- multithreading: 4 threads / core
- 8 MB L3 shared coherent cache
- 4-6 GB GDDR5

MIC Architectures



Multiple IA cores
- In-order, short pipeline
- Multi-thread support

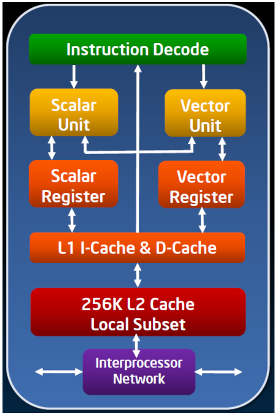
16-wide vector units (512b)
- Extended instruction set
Fully coherent caches

1024-bit ring bus
GDDR5 memory
- Supports virtual memory

- cores based on Pentium architecures
- $\approx$ 60 cores
- in-order architecture
- wide SIMD instructions

Core Architectures

- Scalar pipeline derived from the dual-issue Pentium processor
- Fully coherent cache structure
- 4 execution threads per core
- Separate register sets per thread
- Fast access to its 256KB local subset of a coherent L2 cache.
- 32KB instruction-cache and 32KB data-cache per core
- 3-operand, 16-wide vector processing unit (VPU)
- VPU executes integer, single-precision float, and double precision
- 1024 bits wide, bi-directional (512 bits in each direction)



MIC Programming Issues

- **core parallelism**:
  - keep all 60 cores (1 reserver for OS) busy
  - runs 2-3 (up-to) 4 threads/core is necessary to hide memory latency
- **vector parallelism**:
  - enable data-parallelism
  - enable use of 512-bit vector instructions
- **Amdhal's law**:
  - transfer time between host and MIC-board not negligible
  - hide transfer time overlapping computation and processing

Case Study

Lattice Boltzmann application

- “classic” multi-core: Sandybridge
- GP-GPU
- Xeon-Phi

Lattice Boltzmann Methods

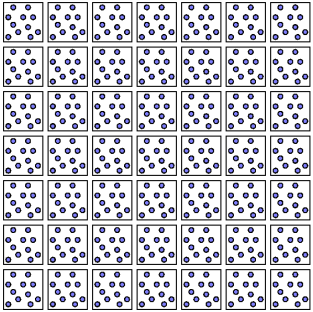
The D2Q37 Lattice Boltzmann Model

- 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
```



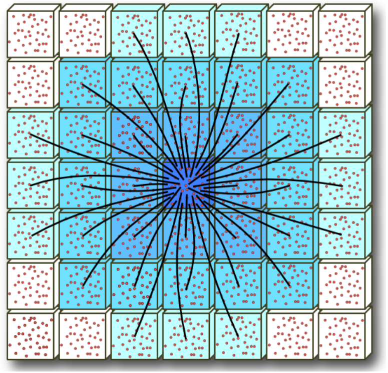
Embarassing parallelism

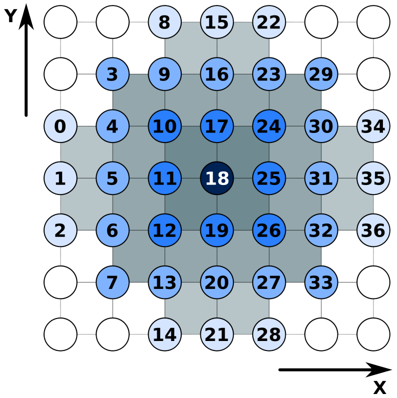
All sites can be processed in parallel applying in sequence propagate and collide.

Challenge

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

D2Q37: propagation scheme

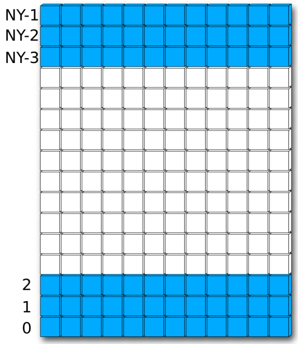




- require to access neighbours cells at distance 1,2, and 3,
- generate memory-accesses with **sparse** addressing patterns.

D2Q37: boundary-conditions

- we simulate a 2D lattice with 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

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

D2Q37: version 1 and 2

We have developed two versions of the code:

- **Version 1**:
  - computes propagation and collision in two separate steps;
  - is used if reactive dynamics is enable
  - requires computing of the divergence of the velocity field between the two steps; to do so, we need a further step in which data is gathered from memory.
- **Version 2**:
  - merges computation of propagation and collision in just one single step;
  - saves to access memory twice and improves performances.

Implementation on multi-core CPUs

Implementation on Multicore CPUs

Sandybridge architecture

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 16-32 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.

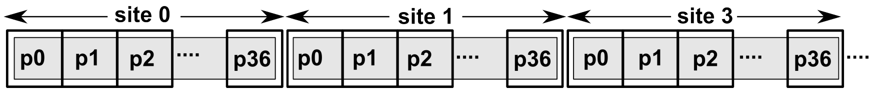
$$T_{exe} \geq \max\left(\frac{W}{F}, \frac{I}{B}\right) = \max\left(\frac{7666}{345.2}, \frac{592}{85.312}\right) \text{ ns} = \max(22.2, 6.94) \text{ ns}$$

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.

Memory layout for LB: Array of Structure



```
typedef struct {
  double  p1; // population 1
  double  p2; // population 2
  ...
  double p37; // population 37
} pop_t;

pop_t lattice2D[SIZEX*SIZEY];
```

- AoS exploits cache-locality of populations: relevant for computing collision and suitable for CPUs.
- Two copies of the lattice are kept in memory: each step read from **prv** and write onto **nxt**.

Code Optimizations

- **core parallelism**:
  - lattice split over the cores
  - **pthreads** library to handle parallelism
  - **NUMA** library to control allocations of data and threads
- **instruction parallelism**:
  - exploiting vector instructions (AVX)
  - process 4 lattice-sites in parallel

Core Parallelism

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

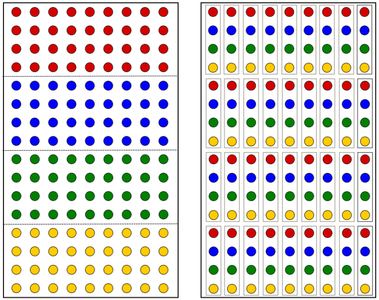
```
for ( step = 0; step < MAXSTEP; step++ ) {

  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(..);
}
```

Vector Programming

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

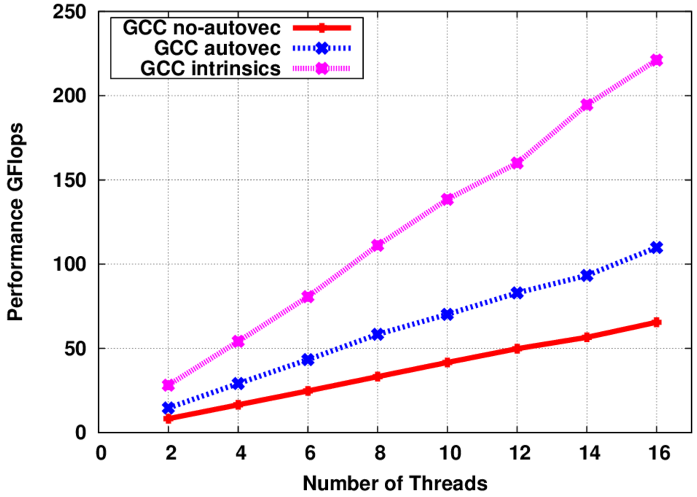
GCC and ICC vectorization by

- 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)



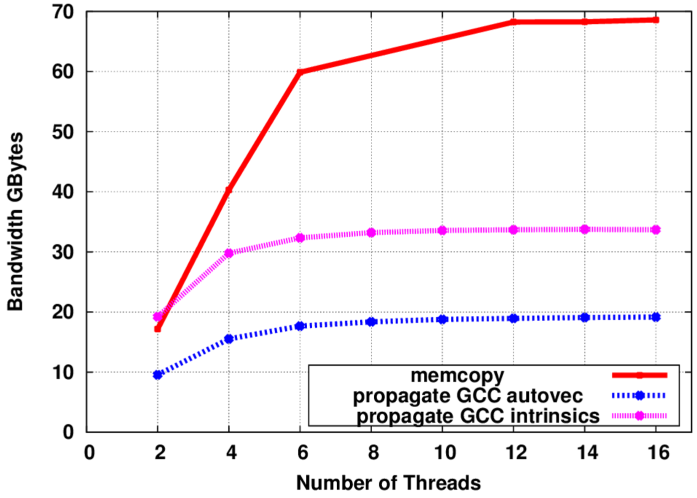
```
typedef double fourD __attribute__ ((vector_size(4*sizeof(double))));

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

Collide Performance



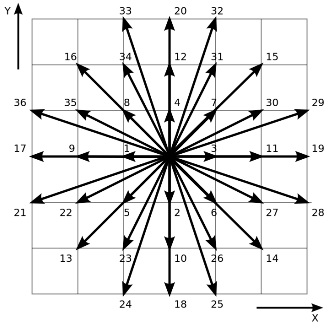
- GCC no-autovec: 18% of peak
- GCC autovec: 31% of peak
- GCC intrinsics: 62% of peak

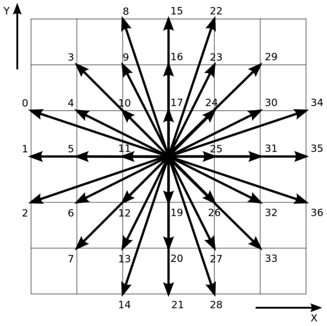
Propagate Performance



- memcopy: 80% of peak
- GCC autovec: 22% of peak
- GCC intrinsics: 40% of peak

Optimization of Propagate





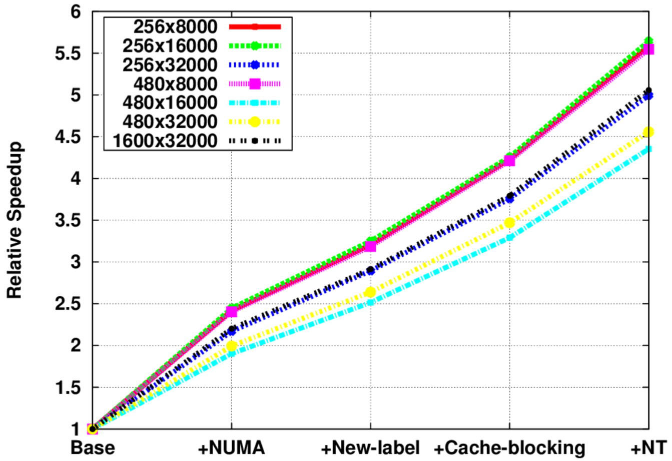
- **cache data-reuse**: reordering of populations allows a better CACHE-reuse and improves performances of propagate;
- **NUMA control**: using the NUMA library to control data and thread allocation avoids overheads in accessing memory;
- **cache blocking**: load the cache with a small data-subset and 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

Sandybridge Execution Time (ms)						
$L_x \times L_y$	Size (GB)	Base	+NUMA Ctrl	+New-Labelling	+Cache-Blocking	+NT
256 × 8000	0.56	116.08	47.84	36.22	27.54	20.86
256 × 16000	1.13	234.44	95.90	72.14	55.16	41.62
256 × 32000	2.26	414.32	190.95	143.13	110.34	82.97
480 × 8000	1.06	215.92	89.83	67.76	51.42	39.04
480 × 16000	2.12	338.96	178.34	134.77	103.18	77.99
480 × 32000	4.23	711.62	356.87	269.64	205.28	156.23
1680 × 16000	7.41	1376.55	625.31	472.54	372.34	279.16

Sandybridge Bandwidth (GB/s))						
$L_x \times L_y$	Size (GB)	Base	+NUMA Ctrl	+New-Labelling	+Cache-Blocking	+NT
256 × 8000	0.56	10.44	25.34	33.48	44.16	58.31
256 × 16000	1.13	10.34	25.29	33.61	44.02	58.35
256 × 32000	2.26	11.71	25.40	33.88	43.99	58.49
480 × 8000	1.06	10.53	25.31	33.55	44.34	58.40
480 × 16000	2.12	13.41	25.49	33.73	44.13	58.38
480 × 32000	4.23	12.78	25.48	33.72	44.33	58.25
1680 × 16000	7.41	11.56	25.45	33.68	43.83	58.46

Optimization of Propagate



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

Full Code Performance Version 1

Lattice size: $\approx 250 \times 16000$ cells

	NVIDIA C2050	2-WS	2-SB
propagate	29.11 ms	140.00 ms	42.12 ms
collide	154.10 ms	360.00 ms	146.00 ms
propagate	84 GB/s	17.5 GB/s	60 GB/s
collide	205.4 GF/s	88 GF/s	220 GF/s
T/site	44 ns	130 ns	46 ns
MLUps	22	7.7	21.7
P	172 GF/s	60 GF/s	166 GF/s
R_{max}	**33%**	**38%**	**48%**
ξ (collide)	–	1.19	1.27

- NVIDIA Tesla C2050, $\approx$ 500 GF DP, $\approx$ 144 GB/s peak (PARCFD'11)
- 2-WS: Intel dual 6-core (Westmere), $\approx$ 160 GF DP, $\approx$ 60 GB/s peak (ICCS'11)
- 2-SB: Intel dual 8-core (Sandybridge), $\approx$ 345 GF DP, $\approx$ 85.3 GB/s peak

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

Full Code Performance Version 2

Execution of `propagate` and `collide` performed in a single step.

Lattice size: $\approx 250 \times 16000$ cells.

	NVIDIA C2050	2-WS	2-SB
propagateCollide	167.2 ms	410.0 ms	144.0 ms
propagateCollide	190 GF/s	77 GF/s	224 GF/s
T/site	40 ns	110 ns	35 ns
MLUps	25	9.3	28.2
P	188 GF/s	72 GF/s	216 GF/s
R_{max}	**36%**	**45%**	**62%**
ξ (propColl)	–	1.05	1.29

- NVIDIA Tesla C2050, $\approx$ 500 GF DP, $\approx$ 144 GB/s peak (PARCFD'11)
- 2-WS: Intel dual 6-core (Westmere), $\approx$ 160 GF DP, $\approx$ 60 GB/s peak (ICCS'11)
- 2-SB: Intel dual 8-core (Sandybridge), $\approx$ 345 GF DP, $\approx$ 85.3 GB/s peak

Difference of ξ might be accounted to different speed of memory-controllers.

Results

LBM code on CPUs supporting the new AVX instructions carefully exploiting:

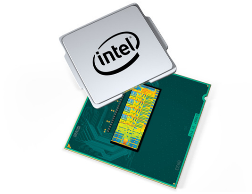
- **core parallelism**
- **vector/streaming parallelism**
- **cache blocking**, **cache data-reuse** and **not-temporal instruction**

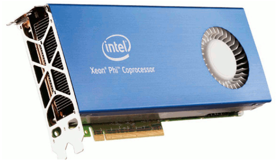
Results:

- AVX version improves performances of `collide` and `propagate` by a factor $\approx$ 2X w.r.t. the SSE
- efficiency is high: 45% – 62% for the dual-socket

Code and Performance Portability

Code and Performance Portability





Beside multi-core CPUs, use of accelerator based systems is today a common option for HPC: Intel Xeon-Phi and AMD and NVIDIA GPUs.

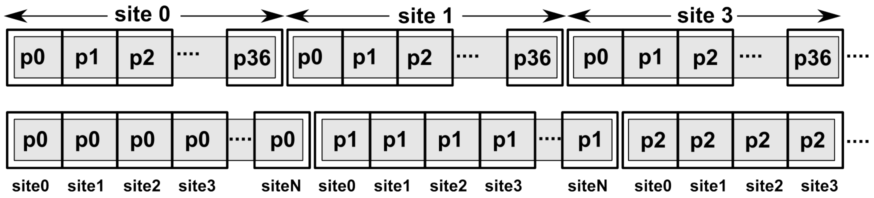
- different programming languages and frameworks: C, CUDA, OpenCL, OpenMP, OpenACC, . . .
- specific code-tuning and optimization

Multi- and Many-core Processors

	Xeon E5-2630 v3	Xeon-Phi 7120P	Tesla K40
#physical-cores	8	61	15 SMX
#logical-cores	16	244	2880
clock (GHz)	2.4	1.238	0.745
GFLOPS (DP/SP)	307/614	1208/2416	1430/4290
SIMD	AVX2 256-bit	AVX2 512-bit	N/A
cache (MB)	20	30.5	1.68
#Mem. Channels	4	16	–
Max Memory (GB)	768	16	12
Mem BW (GB/s)	59	352	288
ECC	YES	YES	YES

- 1 Tflops DP in one device ✔
- nothing is for free ✘
  - manage high number of threads and several levels of parallelism
  - hide latency host-device (Amdhal law)
  - rewrite of code

AoS vs SoA Memory Layout

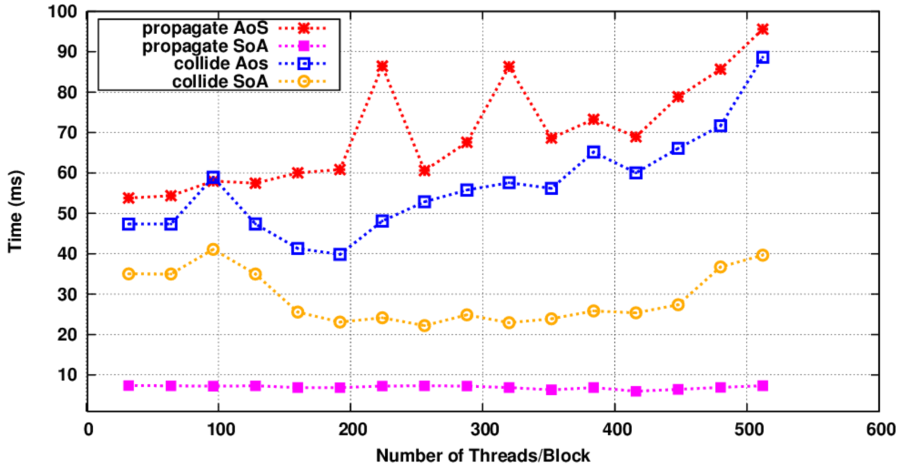


```
#define N (LX*LY)
typedef struct {
  double  p1; // population  1
  double  p2; // population  2
  ...
  double p37; // population 37
} pop_t;

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;
```

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

AoS vs SoA Memory Layout



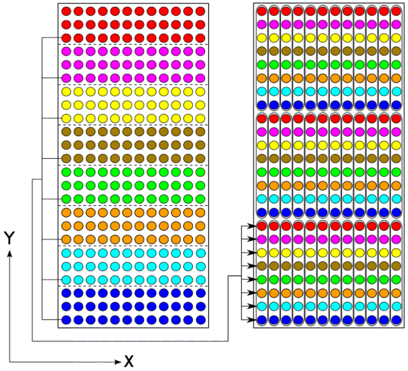
E.g.: on NVIDIA K40 GPU:

- propagate ≈ 10X faster
- collide ≈ 2X faster

Intrinsics programming is not portable

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)`

Vector programming throuhg 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
  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];
    }
  }
```

- snippet of sample code for `propagate`, moving population f_0
- `OXM3YP1`: memory address offset associated to the population hop;
- `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).

Implementation: using SoA on all architectures

- On cache-based processors (Xeon-Phi e x86-CPUs), using SoA data-layout scheme code for collide is correclty vectorized but performance are low
- investigating this on Xeon-Phi with the Intel profiler we found out that:

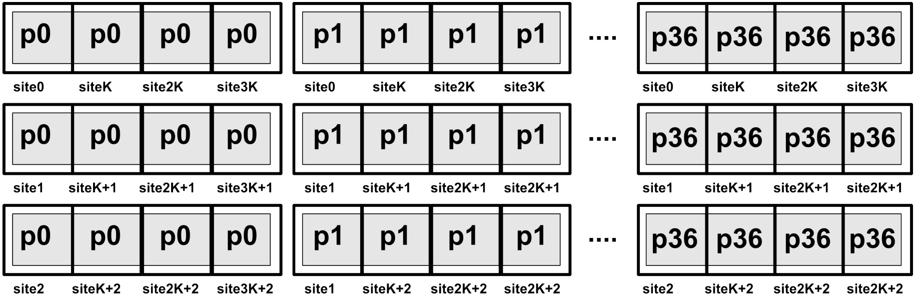
Metric	Measured	Threshold
L1 TLB Miss Ratio	4.30%	1.0%
L2 TLB Miss Ratio	3.00%	0.1%

- many TLB misses in executing the collide kernel are caused by several strided memory accesses to load all data populations.

Implementation: CAoSoA

Clustered Array of Structure of Array:

- allows vectorization of inner structures (clusters) of size VL,
- improves locality of populations keeping them closer in memory



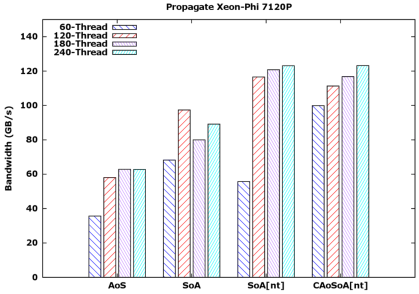
p0 p0 p0 p0
site0 siteK site2K site3K
p1 p1 p1 p1
site0 siteK site2K site3K
....
p36 p36 p36 p36
site0 siteK site2K site3K
p0 p0 p0 p0
site1 siteK+1 site2K+1 site3K+1
p1 p1 p1 p1
site1 siteK+1 site2K+1 site2K+1
....
p36 p36 p36 p36
site1 siteK+1 site2K+1 site2K+1
p0 p0 p0 p0
site2 siteK+2 site2K+2 site3K+2
p1 p1 p1 p1
site1 siteK+2 site2K+2 site2K+2
....
p36 p36 p36 p36
site2 siteK+2 site2K+2 site2K+2

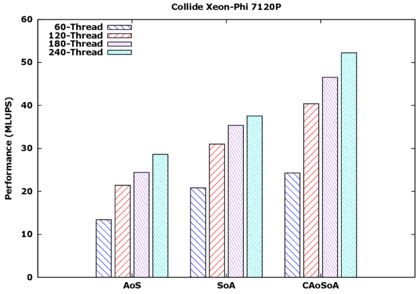
```
typedef struct { double c[VL]; }     vdata_t;   // cluster type definition
typedef struct { vdata_t p[NPOP]; } caosoa_t;  // CAoSoA type definition
```

Implementation: CAoSoA

```c
typedef struct { double c[VL]; }    vdata_t;  // cluster type definition
typedef struct { vdata_t p[NPOP]; } caosoa_t; // CAoSoA type definition

// snippet of propagate code to move population index 0
for ( xx = startx; xx < stopx; xx++ ) {
  for ( yy = 0; yy < SIZEYOVL; yy++, idx++ ) {
    idx =  IDX(xx,yy);
#pragma vector aligned nontemporal
    for(tt = 0; tt < VL; tt++) {
      nxt[idx].p[0].c[tt] = prv[idx+OPOVL0].p[0].c[tt];
    }
  }
}

// snippet of part of collide code to compute density rho
vdata_t rho;

for (ii =0; ii < NPOP; ii++)
#pragma vector aligned
  for (tt=0; tt < VL; tt++)
    rho.c[tt] = rho.c[tt] + prv[idx].p[ii].c[tt];
```

Results on MIC (2160 × 8192)





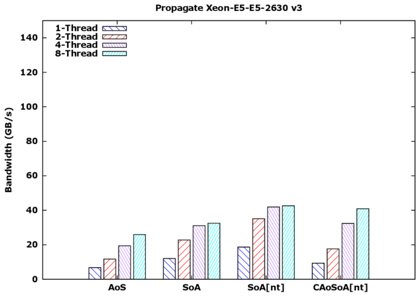
- using the CAoSoA scheme TLB misses have been significantly reduced:

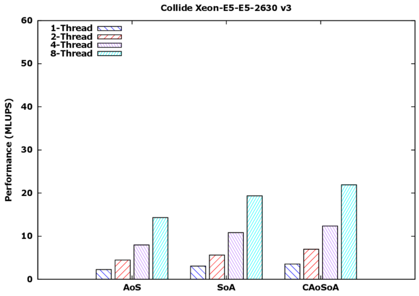
Metric	Measured	Threshold
L1 TLB Miss Ratio	0.06%	1.0%
L2 TLB Miss Ratio	0.00%	0.1%

- performace increases with number of threads

	AoS	SoA	CAoSoA
propagate [GB/s]	62	123	123
collide [MLUPS]	26	37	52

Results on CPU (2160×8192)





- performance increases with number of threads
- improvements using the CAoSoA are smaller w.r.t. Xeon-Phi

	AoS	SoA	CAoSoA
propagate [GB/s]	26	42	41
collide [MLUPS]	14	19	22

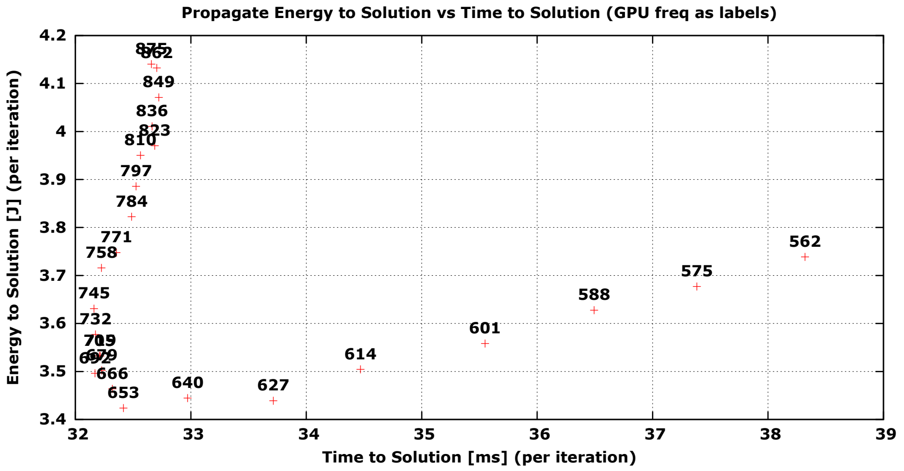
Results on several architectures (2160×8192)

Data Structure	Haswell	Xeon Phi	Tesla K80	AMD Hawaii
		propagate [GB/s]		
AoS	25.67	54.00	32.13	16.14
SoA	12.37	46.76	290.98	183.78
CSoA	42.41	134.30	327.35	232.78
CAoSoA	36.63	117.70	317.43	209.51
		collide [MLUPS]		
AoS	14.36	28.04	23.07	7.80
SoA	10.98	9.96	103.48	17.38
CSoA	18.53	39.76	107.24	39.15
CAoSoA	21.79	54.45	106.59	44.02

- OpenMP on Intel and OpenACC on GPUs and AMD
- CSoA is like SoA but alignement is enforced.

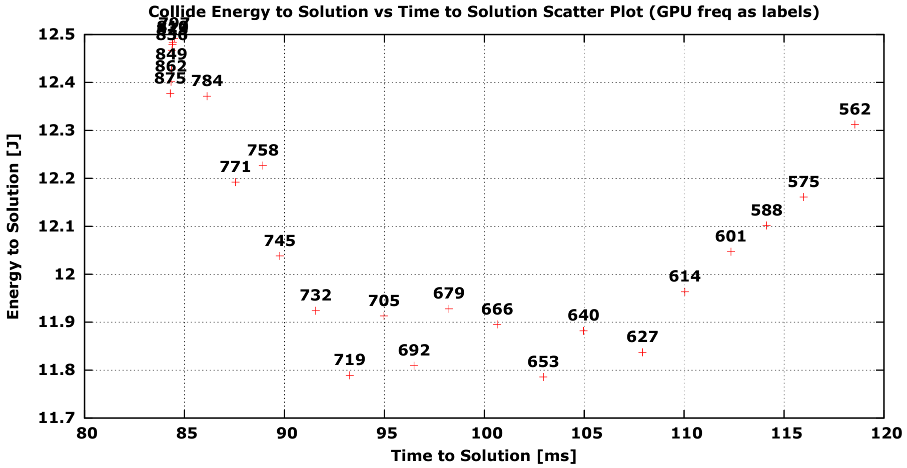
Energy issues

Energy efficiency: Propagate



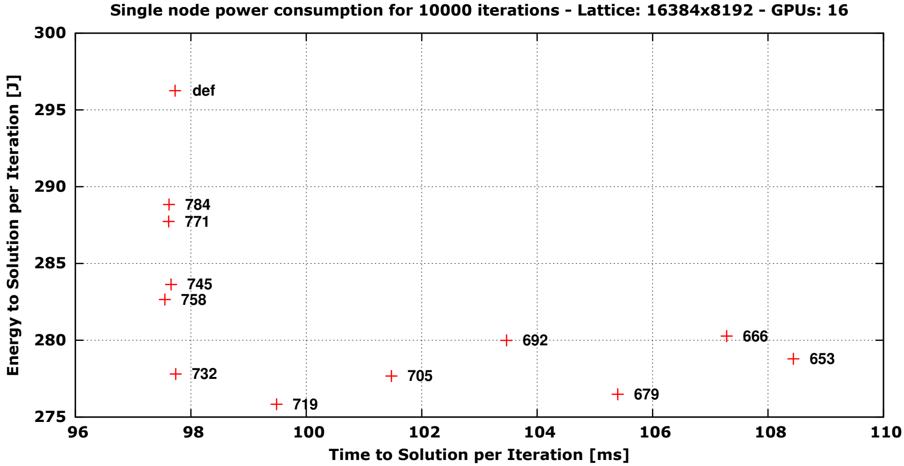
- E_S vs T_S for the propagate and collide functions, measured on the GPU;
- labels are the corresponding clock frequencies f in MHz.

Energy efficiency: Collide



Energy efficiency

Running on 16 GPUs (8 x NVIDIA K80 Dual GPU boards) system:



Power drain of the node measured at PSU through IPMI, during code execution for different GPUs clock frequencies.

Energy efficiency

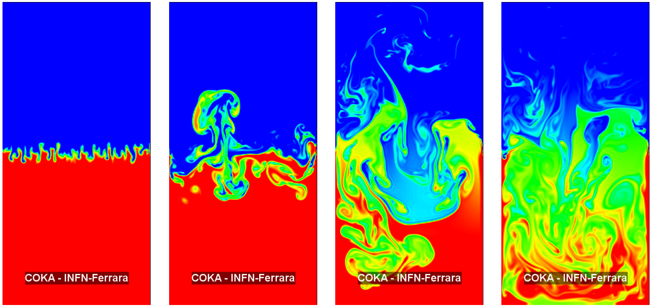
Taking into account the Italian average energy cost of 0.17€/kWh, the savings amount to ≈ 300€/year for each computing node.

Node No.	32	128	1024
k€/year	9.5	38.1	305

Potential saving in k€/year of electricity bill for clusters of different sizes, not taking into account the savings related to the energy dissipated by the cooling system.

Simulation of the Rayleigh-Taylor (RT) Instability

Instability at the interface of two different-density fluids 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).

Conclusions

Multi-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
- Accelerators are not a *panacea*:
  - good for desktop-applications
  - hard to scale on large clusters
- data structure have a big impact on performance
- portability of code and performance is necessary
- energy efficiency is a big issue

coming back to the one million dollar question . . .

so . . . which is the best computing system to use ?

Acknowledgments

- Luca Biferale, Mauro Sbragaglia, Patrizio Ripesi
  University of Tor Vergata and INFN Roma, Italy
- Andrea Scagliarini, University of Barcelona, Spain
- 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
  University of Ferrara

Results presented here are developed in the framework of the INFN COKA, COSA and SUMA projects.

I thank CINECA, INFN and JSC for access to their systems.