Multi- and many-core computing for Physics applications

S. F. Schifano

University of Ferrara and INFN-Ferrara

X Seminar on Nuclear, Subnuclear and Applied Physics

June 2-8, 2013

Porto Conte, Alghero, Italy

Outline

Efficient programming of high-performance processors for physics applications.

1. aspects of parallel computing (in short)
2. multi- and many-core architectures:
   “classic” CPUs, GP-GPUs, Xeon-Phi
3. programming issues and performance results:
   Lattice-Boltzmann as case study
4. Multicore and FPGA-based systems for L0-trigger processors

the one million dollar question

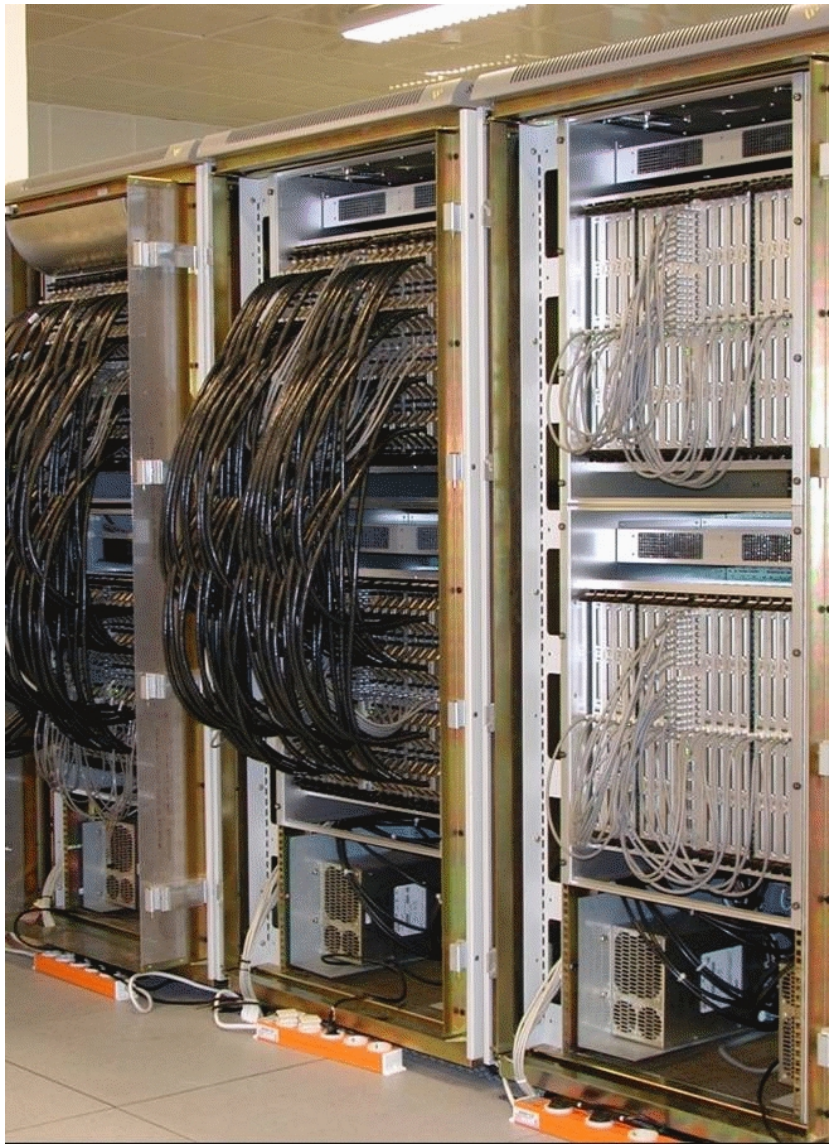
which is the best computing system to use ?

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

From custom to commodity computing systems

- making the own processor is not a easy job
- commodity system have evolved in the **right** directions

Designing an ideal LQCD-machine

Assuming computation and I/O can be performed in parallel, an **ideal** LQCD-machine should satisfy the equation:

$$w(n)/F = I(n,m)/b$$

where:

- $w(n)$ is the number of floating point operations performed by the processor when execute a computation of size n (i.e. when execute the Dirac operator on a sublattice size of n)
- F is the number of floating point operations performed by the processor per clock cycle
- $I(n,m)$ is the **information exchange function**
- b is the number of bits exchanged by the processor with the rest of the system per clock cycle

G. Bilardi, A. Pietracaprina, G. Pucci, F. Schifano, R. Tripiccione,
The Potential of On-Chip Multiprocessing for QCD Machines, HiPC 2005, LNCS vol. 3769.

Information Exchange Function

The $I(n, m)$ function is the **information exchange function** which defines:

> *the amount of of bit echanged by the processor with the rest of the system, assuming it has on board a storage element of m bits (register file, memory on-chip, cache, . . .) and performs a computation of size n.*

On a parallel machine the $I(n, m)$ function is composed by two parts:

$$I(n, m) = I_{lc}(n, m) + I_{nb}(n, m)$$

- $I_{lc}(n, m)$: the quantity of bits exchanged with the **local** memory
- $I_{nb}(n, m)$: the quantity of bits exchanged with the **remote** memories accessed via **network**

Processors Comparison

Using the balance equation we have evaluated different processor architecture

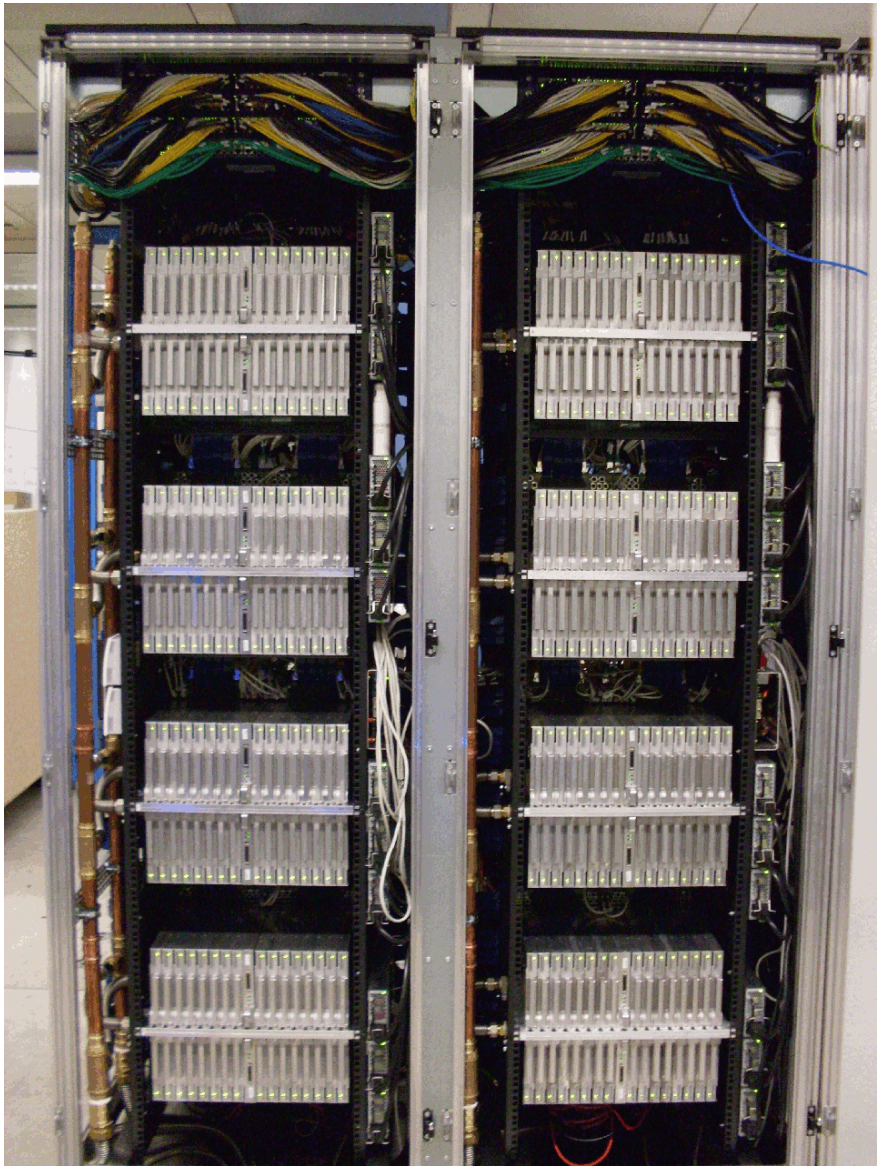
	apeNEXT	**BG/L**	**Cell**	**ITANIUM2**
frequency	200Mhz	700Mhz	3.2Ghz	1.6Ghz
λ	180nm	130nm	90nm	90nm
L_w	64	32/64	32/64	64
F	8	4	64/8	4
m	32kb	32Mb	20Mb	72Mb
b_{lc}	128	62.85	64	x
b_{nb}	48	24	192	$32 - x$
ξ_{LM}	n.a.	6.07/2.28	2.27/6.06	4.04
ξ_{MM}	**0.76**	9.53/4.77	**0.61**/2.42	2.20

ξ is a measure of how well the processor is balanced between computing and I/O for QCD application

QPACE Machine (2008)

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

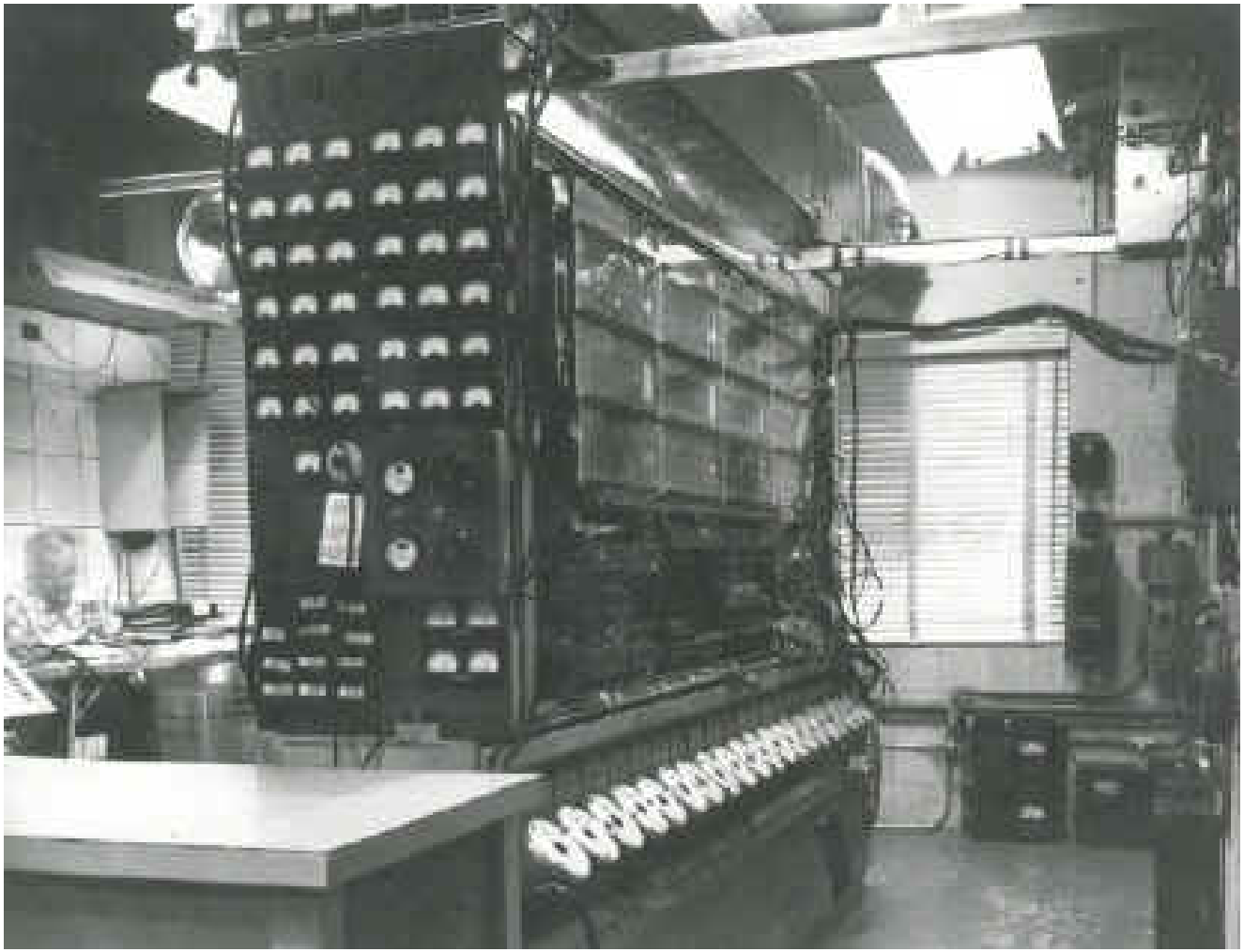
- 26 Tflops peak double-precision
- 35 KWatt maximum power consumption
- **750 MFLOPS / Watt**
- TOP-GREEN 500 in Nov.'09 and July'10



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 ?

A “modern” CPU architecture: my point of view !



. . . YES . . . (the core of) a modern CPU is **still** based on the 1950 Von Neumann model !!

The Von-Neumann Architecture

1. **instruction fetch**: obtain instruction from program-storage
2. **instruction decode**: identify the operation to execute
3. **operand fetch**: locate and get operand-data
4. **execute**: compute result value (or status)
5. **result store**: write result into storage (for later use)
6. **next instruction**: identify next instruction to fetch
7. go to step 1

J. Backus

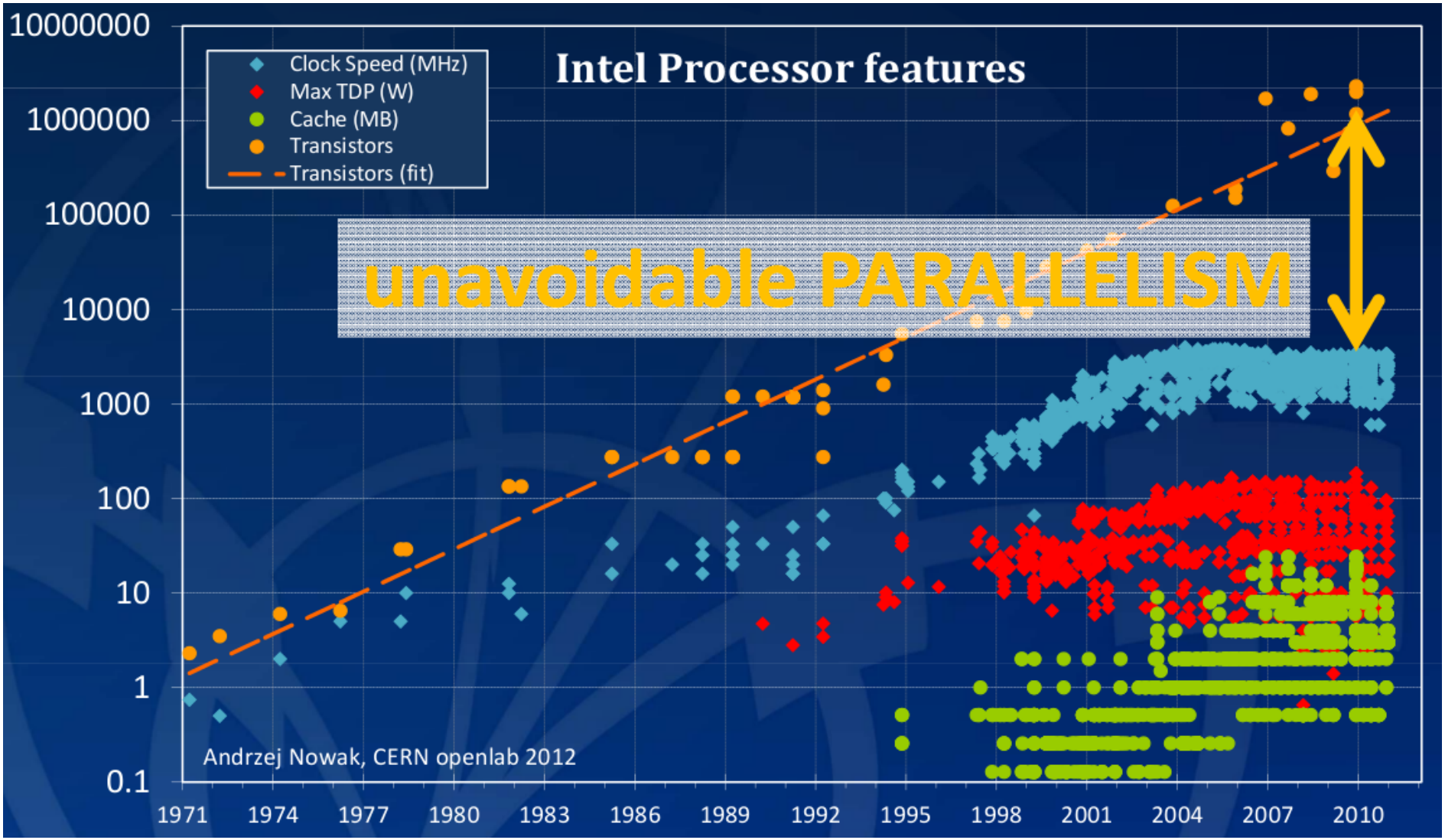
. . . thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.

CPU performances

At beginning, CPU performances have heavily relied on hardware:

- clock frequency
- supports to optimized memory time access: e.g. levels of caches, ROB, ...
- supports to increase ILP: brach-predictors, out-of-order execution, ...

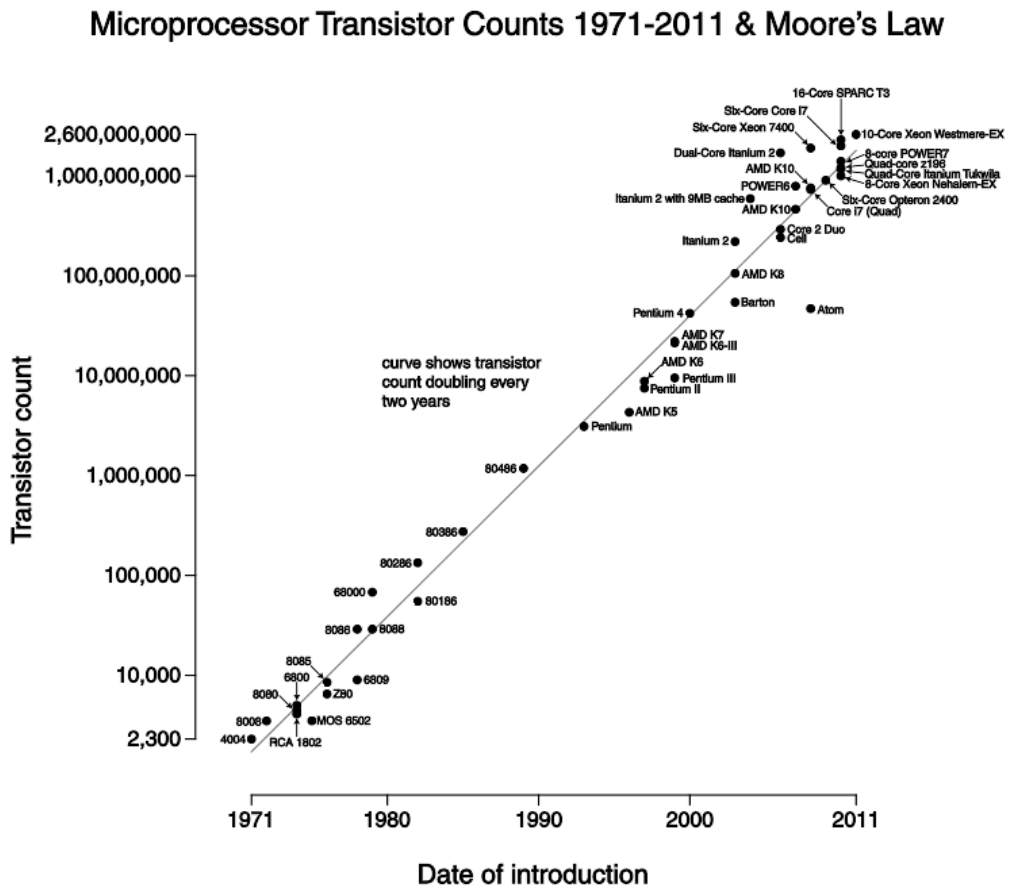
Hardware Evolution



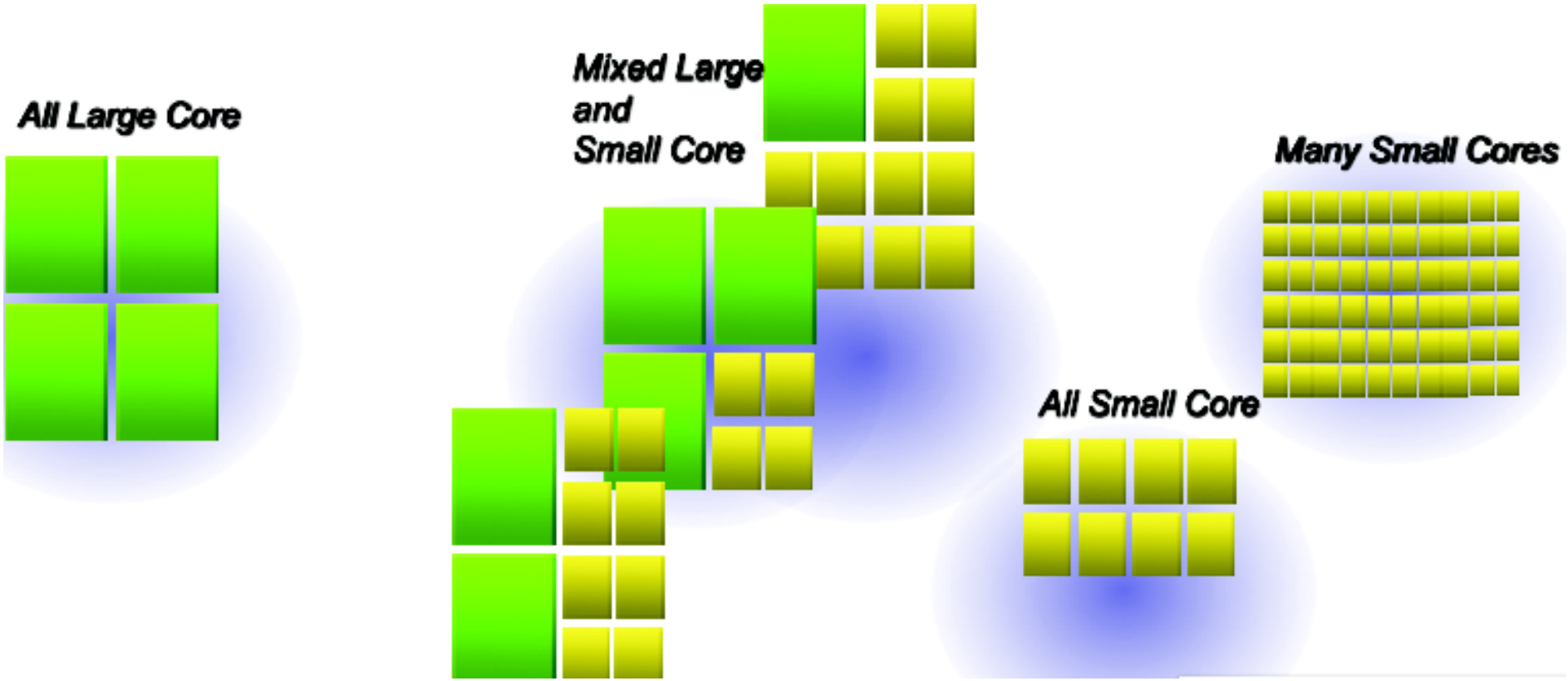
The Multi-core processors era begins !

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

- increase frequency beyond $\approx$ 3 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 ✘



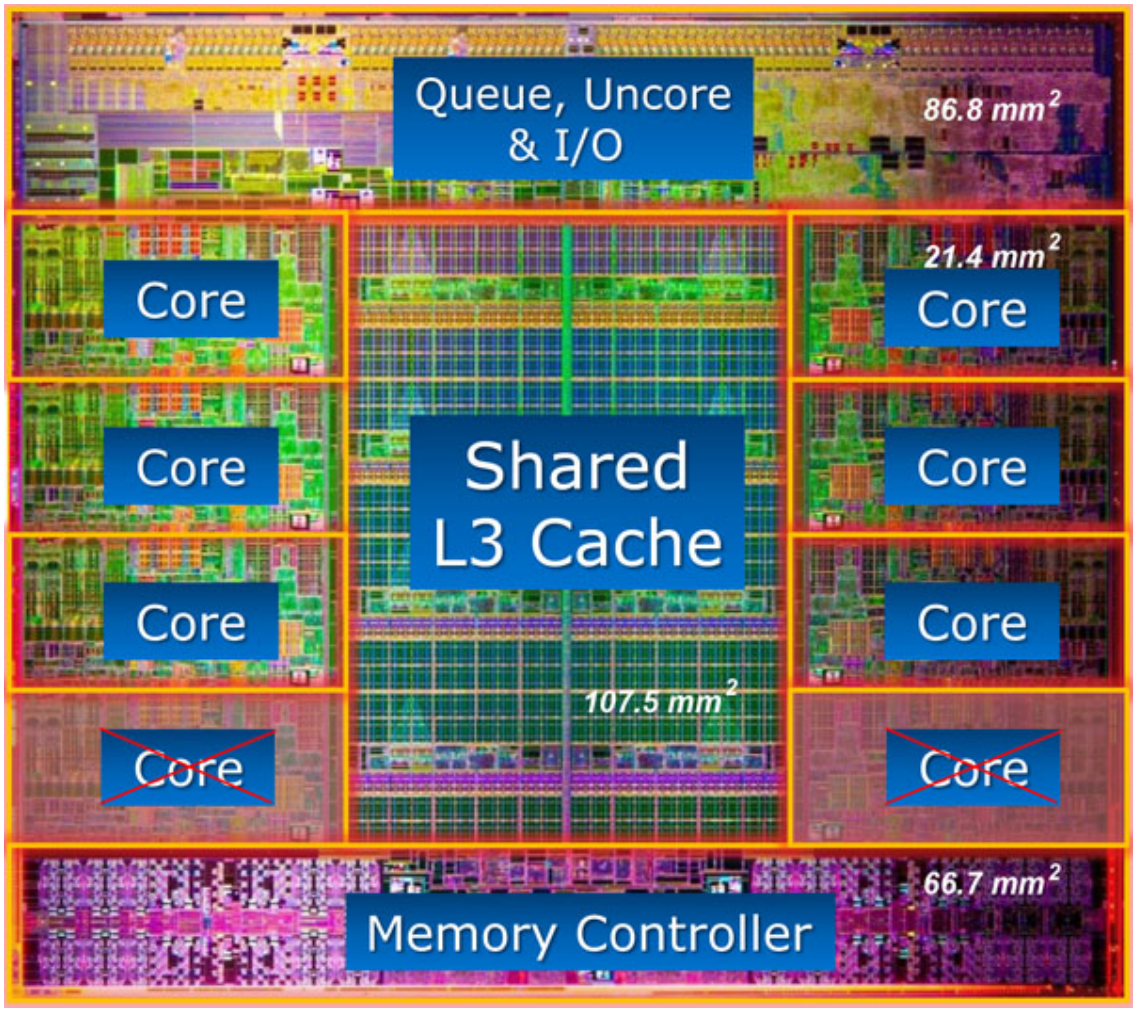
Many Different Cores



- 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” CPU Architectures

The most recent Intel *classic* CPU micro-architectures is the *Sandybridge*:



8 cores, 1 shared L3-cache

“Classic” CPU Architectures

Main features:

- 6-10 (and soon more) cores
- frequency $\approx$ 3 GHz
- 3 levels of caches, 2 withing a core and 1 shared
- support for SIMD execution: AVX 256-bits
- e.g.: Xeon E5-2680 Sandybridge: 691.2/345.6 GFlops SP/DP

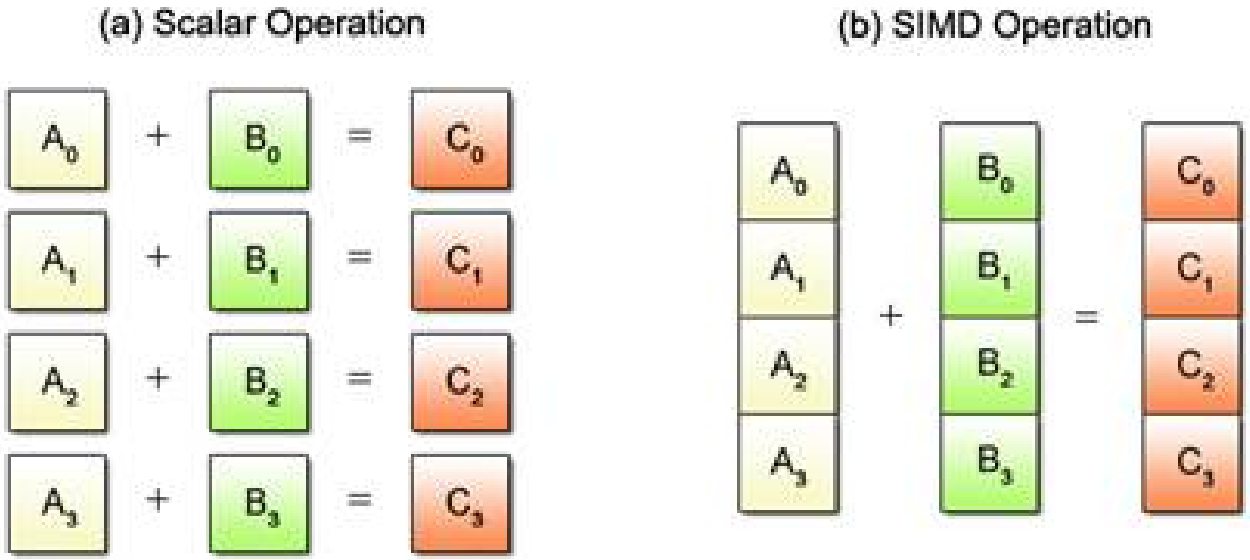
Programming issues:

- **core parallelism**
- **data parallelism**
- **cache optimizations**
- **Non Uniform Memory Architecture** (NUMA)

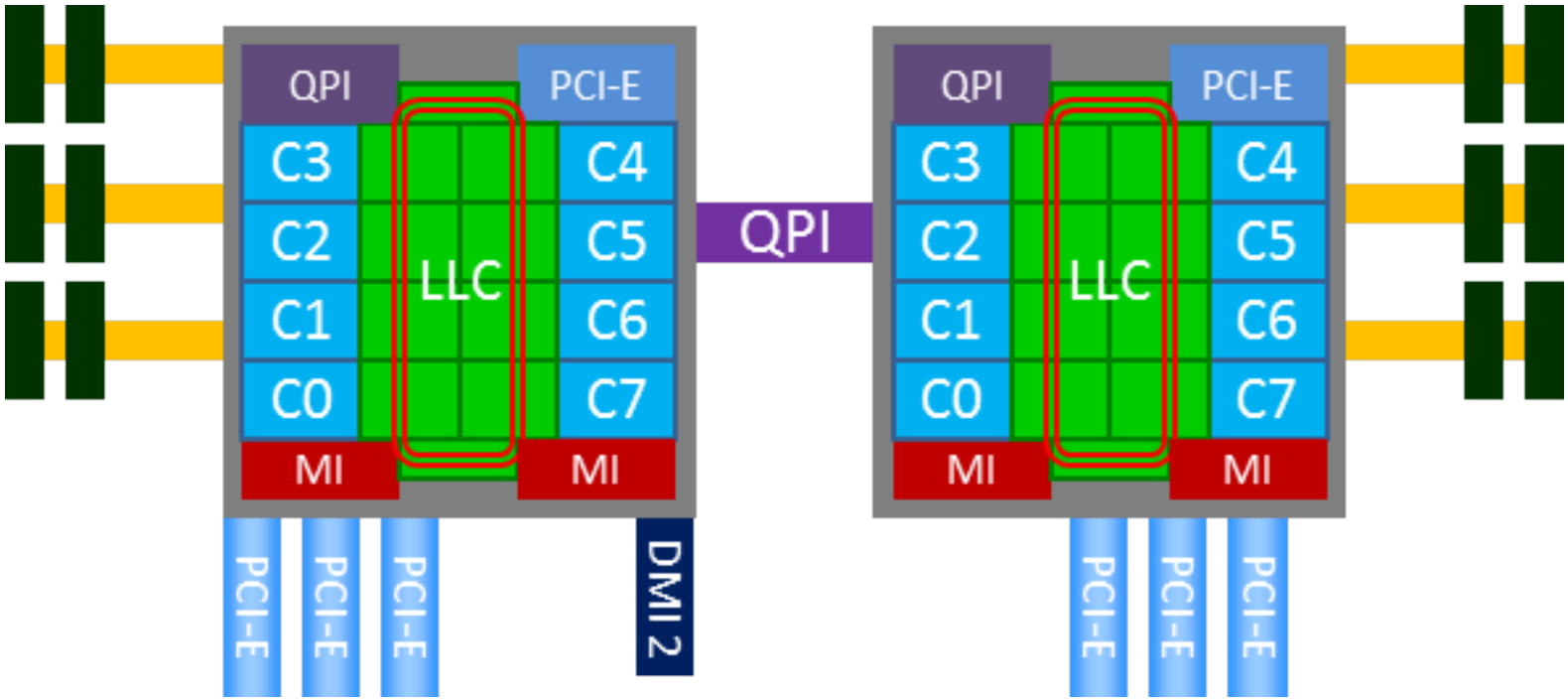
“Classic” CPU Architectures: Performances

- $c = 8$ cores
- SIMD instructions on 256-bit operands:
  each vector register can pack $n = 4(8)$ double (single) precision numbers
- each core can execute two operations per clock-cycle:
  one *add* and one *mul*

$$P = f \times 2 \times n \times c$$



Numa SMP Multi-socket Multi-core Systems



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

Accelerator: Is this a really new concept ?

THREE HUNDRED FORTY ONE MILLION FLOATING POINT OPERATIONS PER SECOND.

THE FPS-164/MAX.

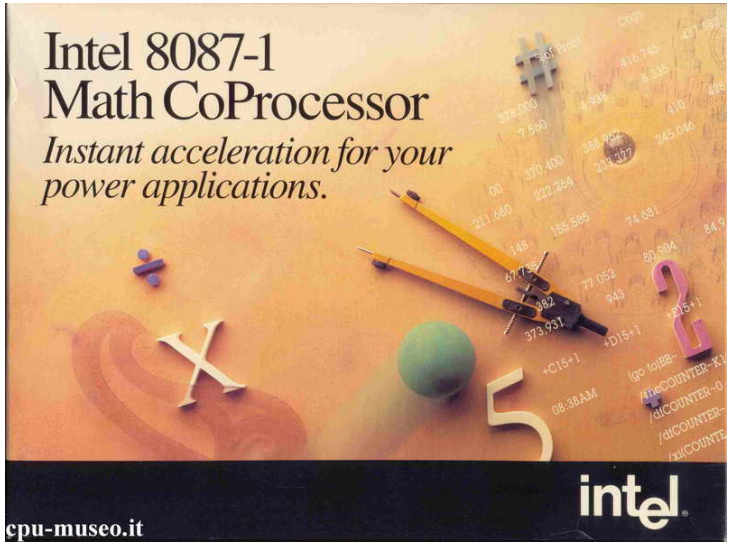
The FPS-164/MAX is fast.

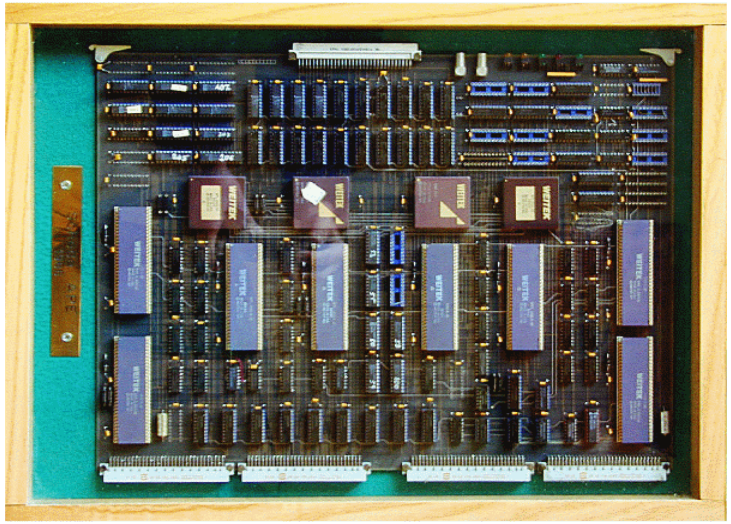
The FPS-164/MAX is powerful.

The FPS-164/MAX is cost-effective.

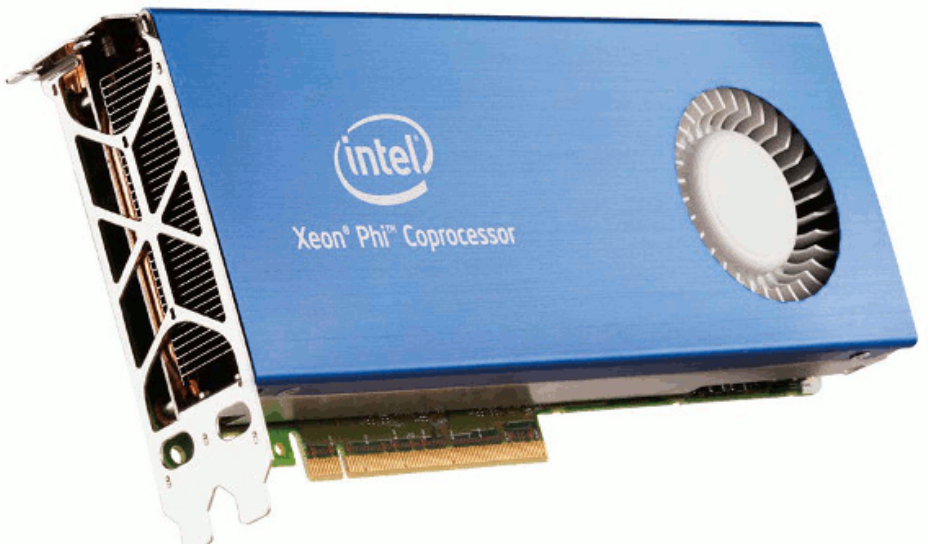
FPS-164/MAX Specifications

FLOATING POINT SYSTEMS INC



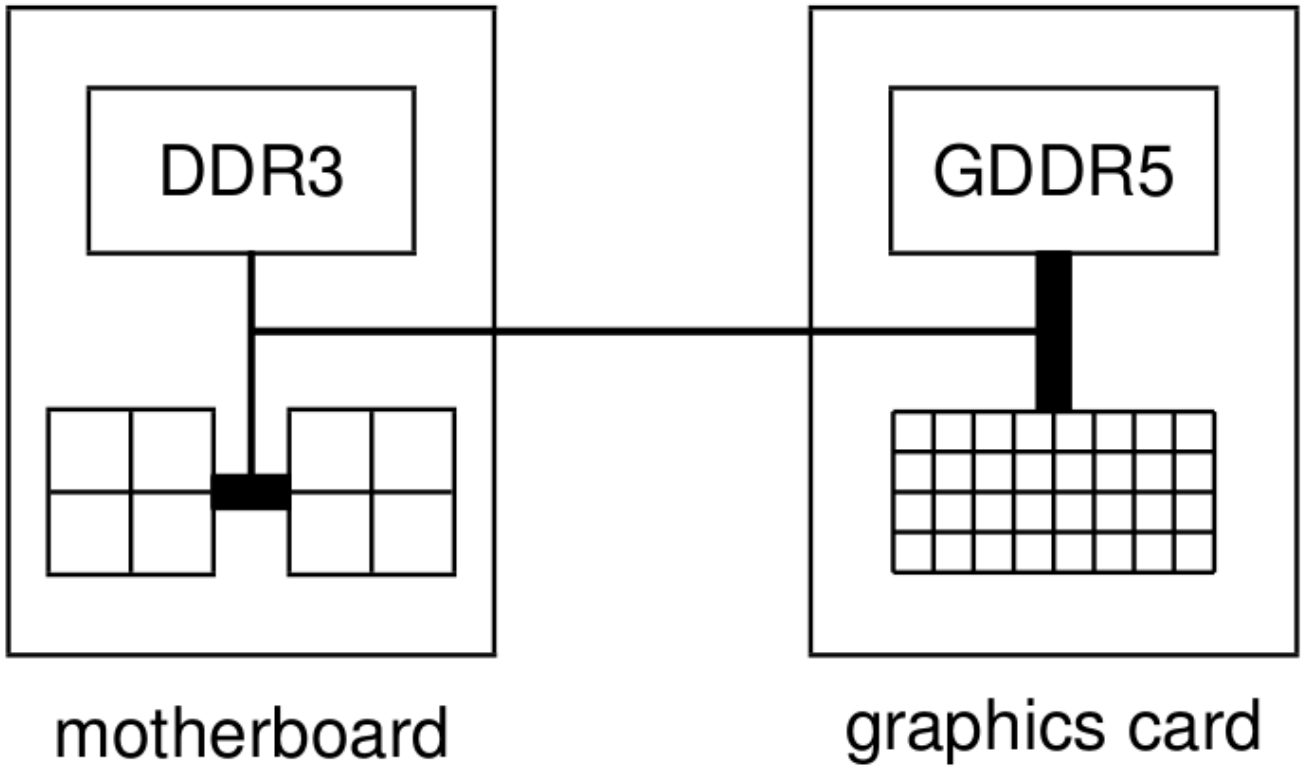


Accelerator: today they appear much better !



Accelerator-based systems: hardware view

At top-level a PCIe accelarator-card (GP-GPU, . . .) sits inside a standard commodity server with one or two multi-core CPUs.



The Sausage Machine Model

A computer is like a sausage machine:

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

Are accellerators good sausage machines ?

FPS-164 and VAX (1976):

- Floating Point: F = 11 Mflop/s, IO Rate: B = 44 MB/s
- Ratio of flops to bytes of data movement: R = 0.25 Flops / Byte
- Host-device latency: $\mathcal{O}(1)$ clock-cycle

Nvidia Kepler K20 and PciE (2012):

- Floating Point: F = 1170 Gflop/s (DP), IO Rate: B = 8 GB/s
- Ratio of flops to bytes of data movement: R = **146.25** Flops / Byte
- Host-device latency: $\mathcal{O}(10 - 100)$ clock-cycles

- Flop/s are cheap, so are provisioned in excess,
- data needs to be re-used and processed several times by the FPUs,
- smart programming techniques to hide data movement latency, e.g. recompute data instead of access memory.

Performance Evaluation: Amdhal's Law

How much can I accelerate my application ?

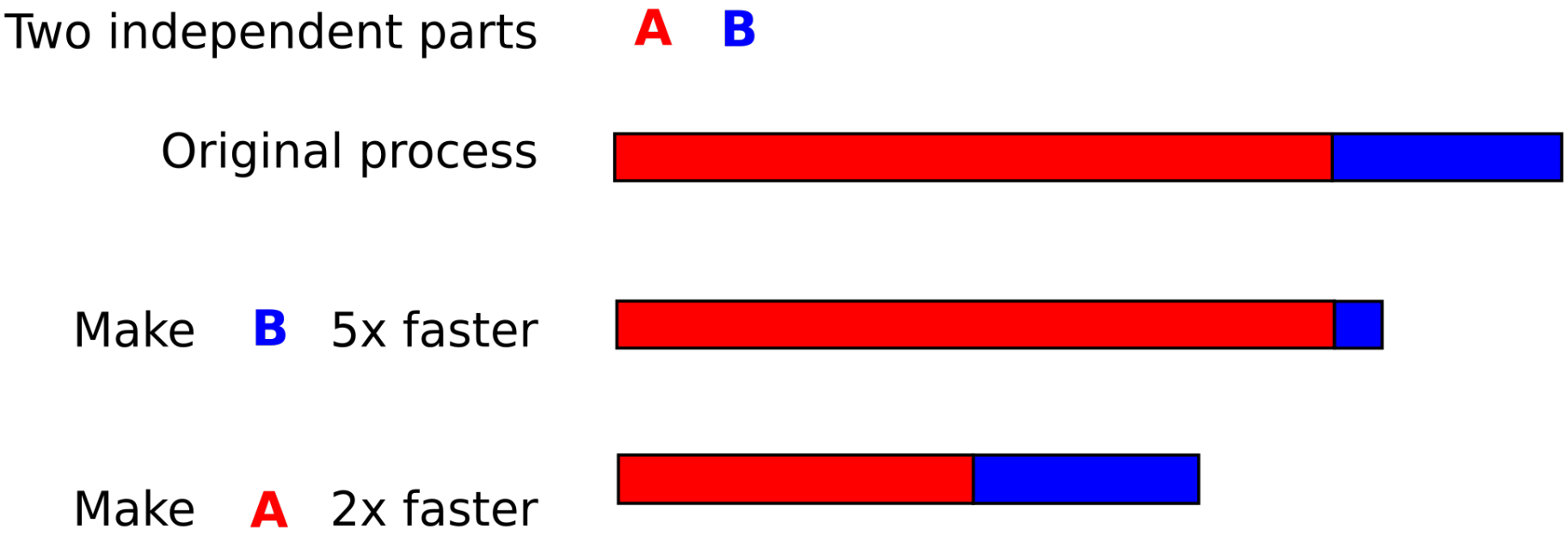
Amdahl's Law approximately states:

> *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

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

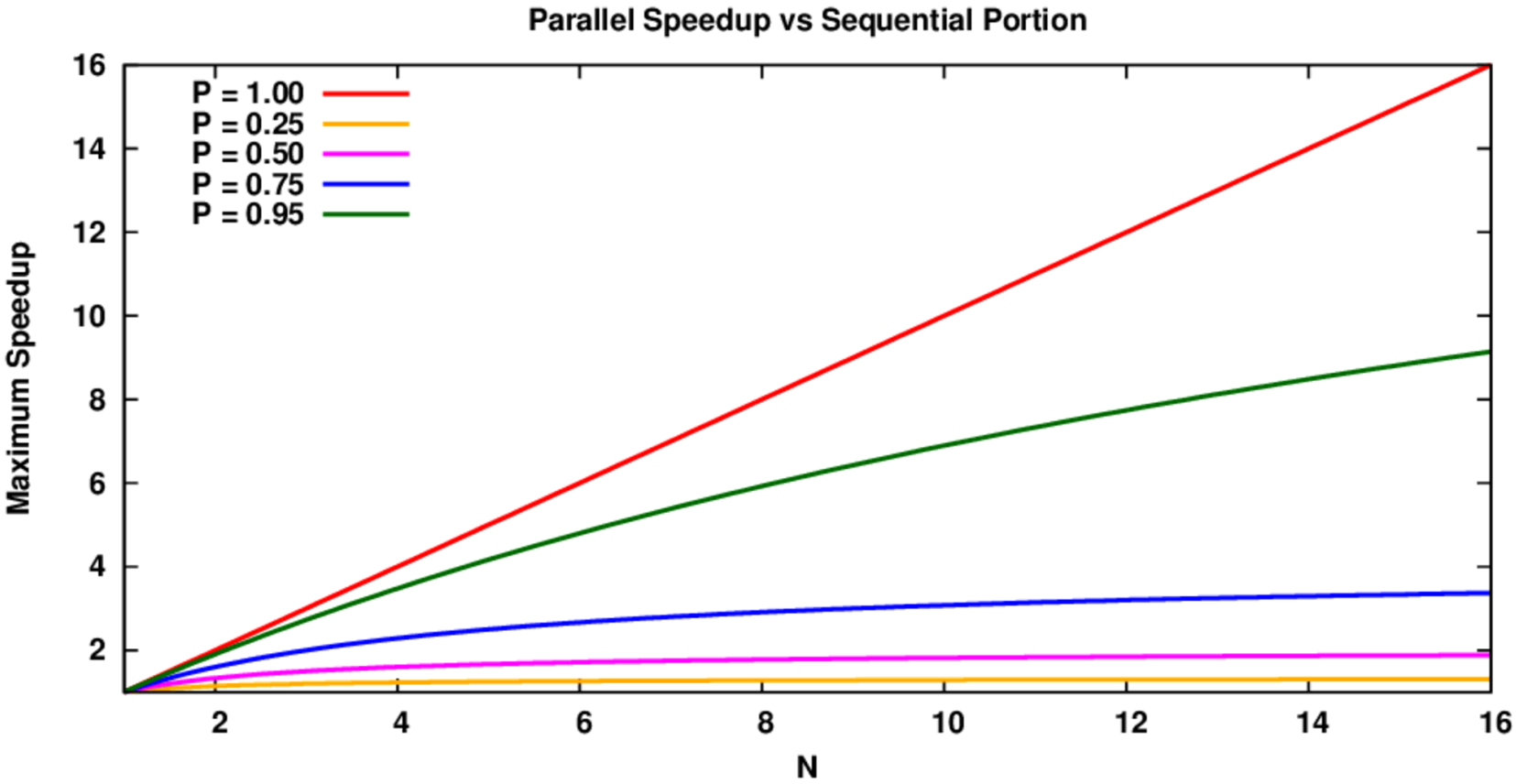
Then the **speedup** (a measure of how fast is the new code) is:

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

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 - you can't bribe God.

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

So ... what's better ? Multi-core CPUs or Accelerators

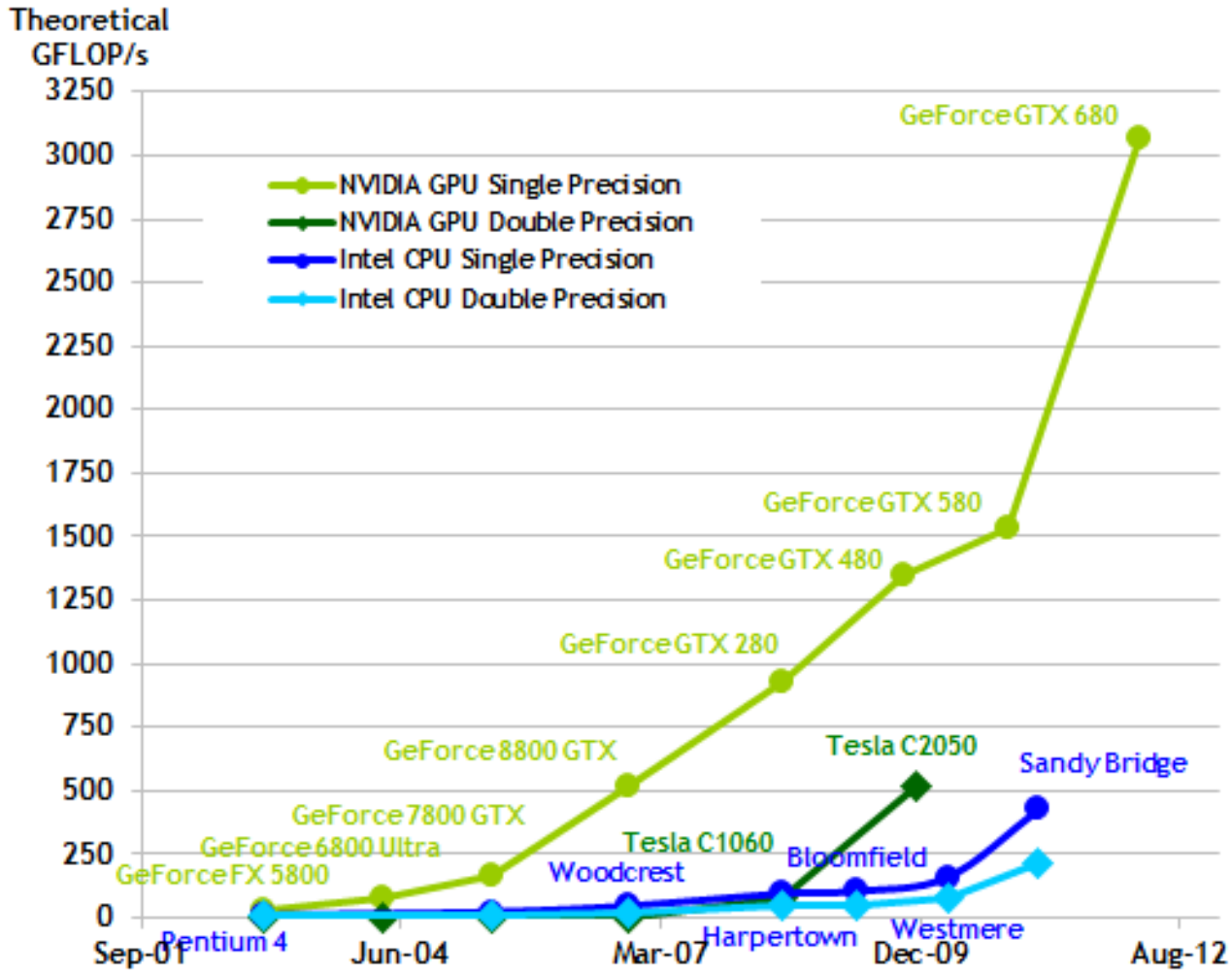
what's better to plow a ground ?

It depends on what do we need. As rule of thumb:

- low-latency and reasonable throughput: left
- high-througput and reasonable-latency: right

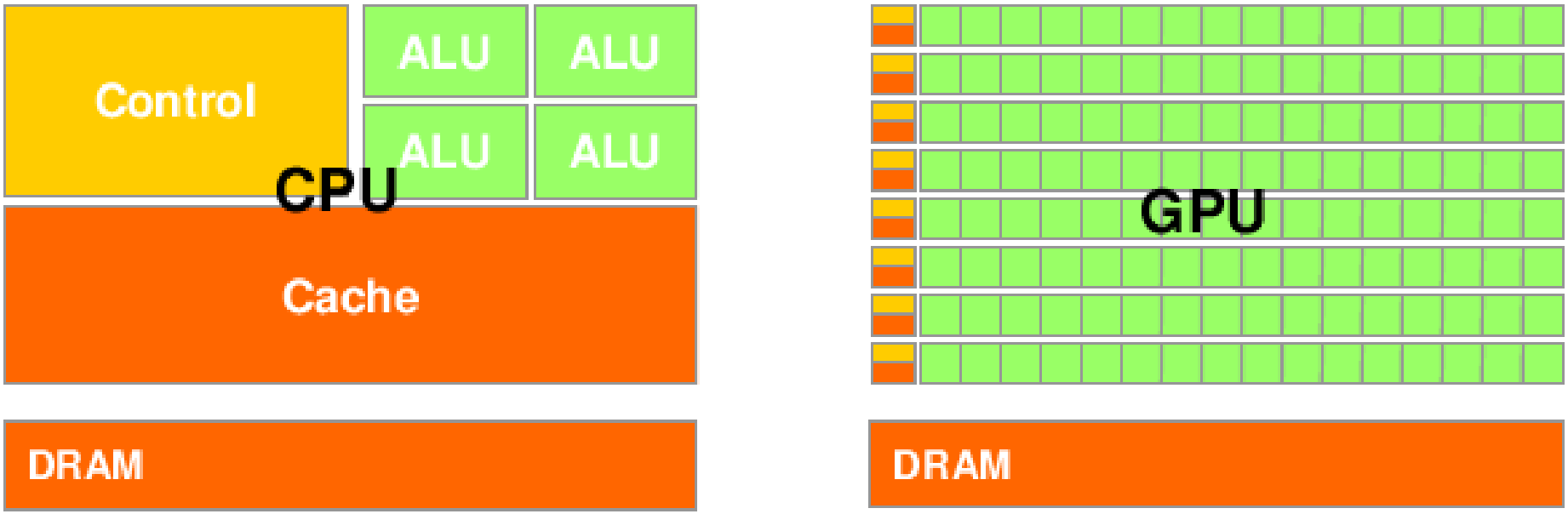
Better if you can use both !!! May be hard to program and get good efficiency !

Let's look on GPUs in more detail: GPU evolution



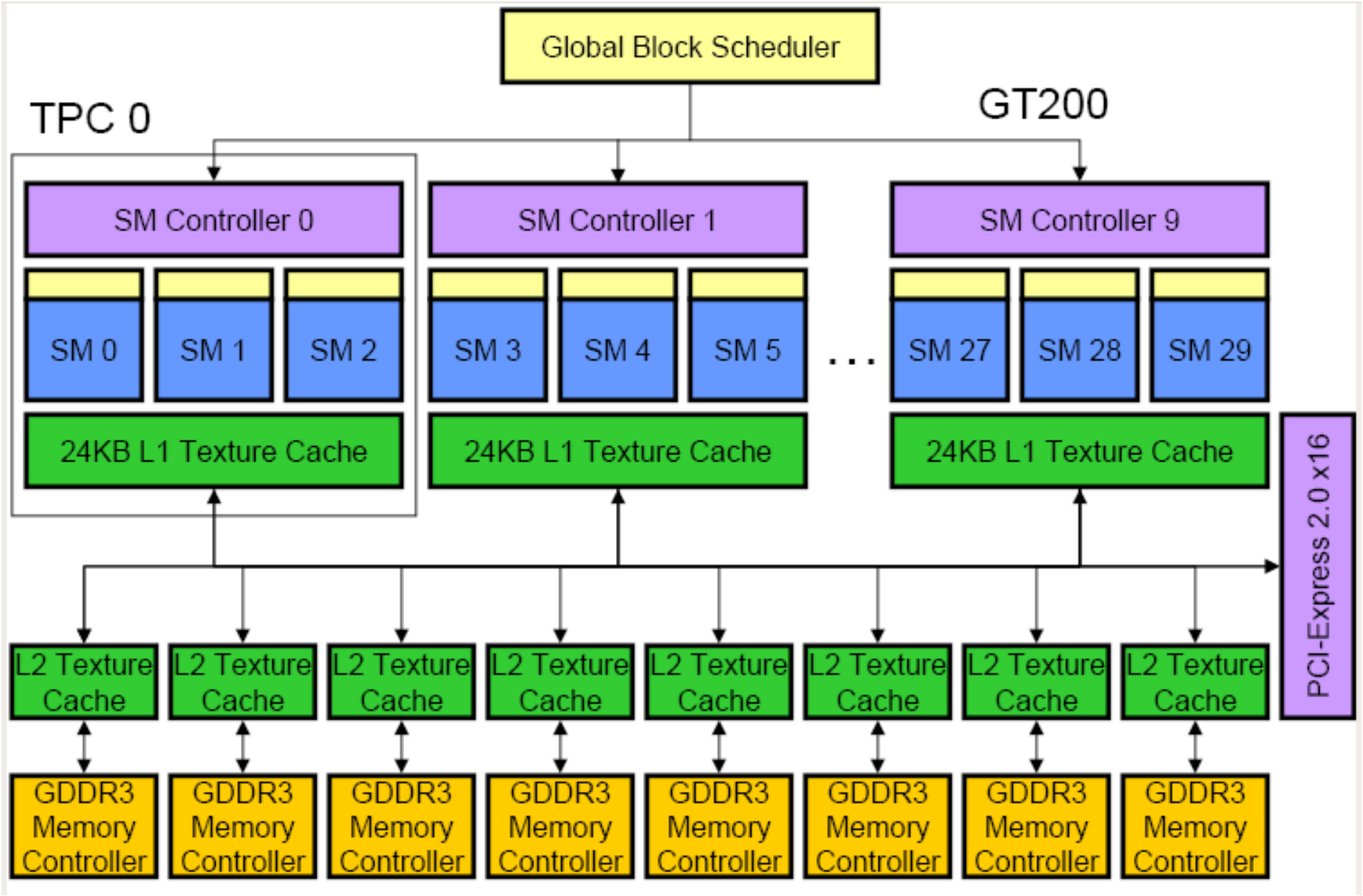
- GPUs evolve much faster in terms of raw-computing power
- Fast-growing video-game market forces innovation

GPUs vs CPUs architecture



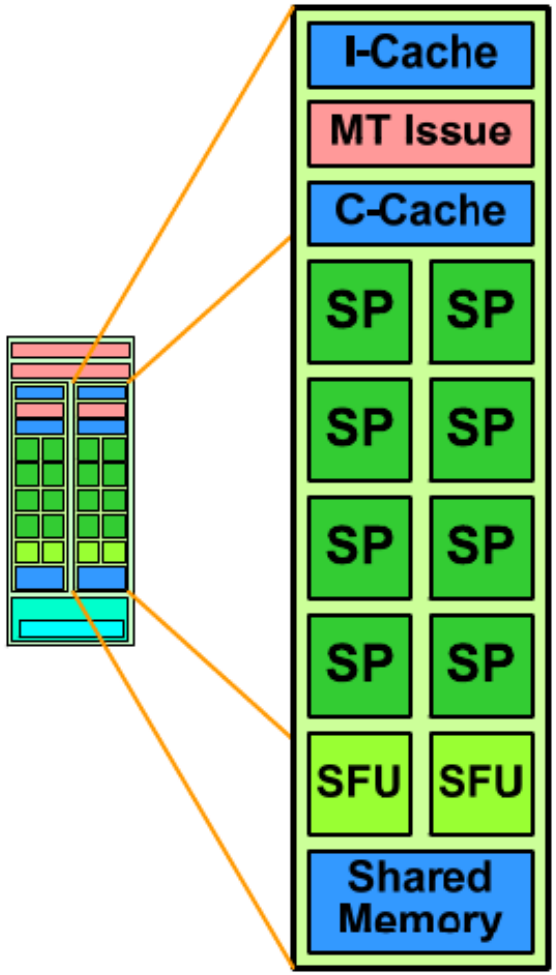
- GPUs specialized for highly data-parallel and intensive computation (exactly what rendering is about)
- more transistors devoted to **data-processing** rather than **data caching** and **flow-control**

Architecture of GP-GPU: NVIDIA GT200



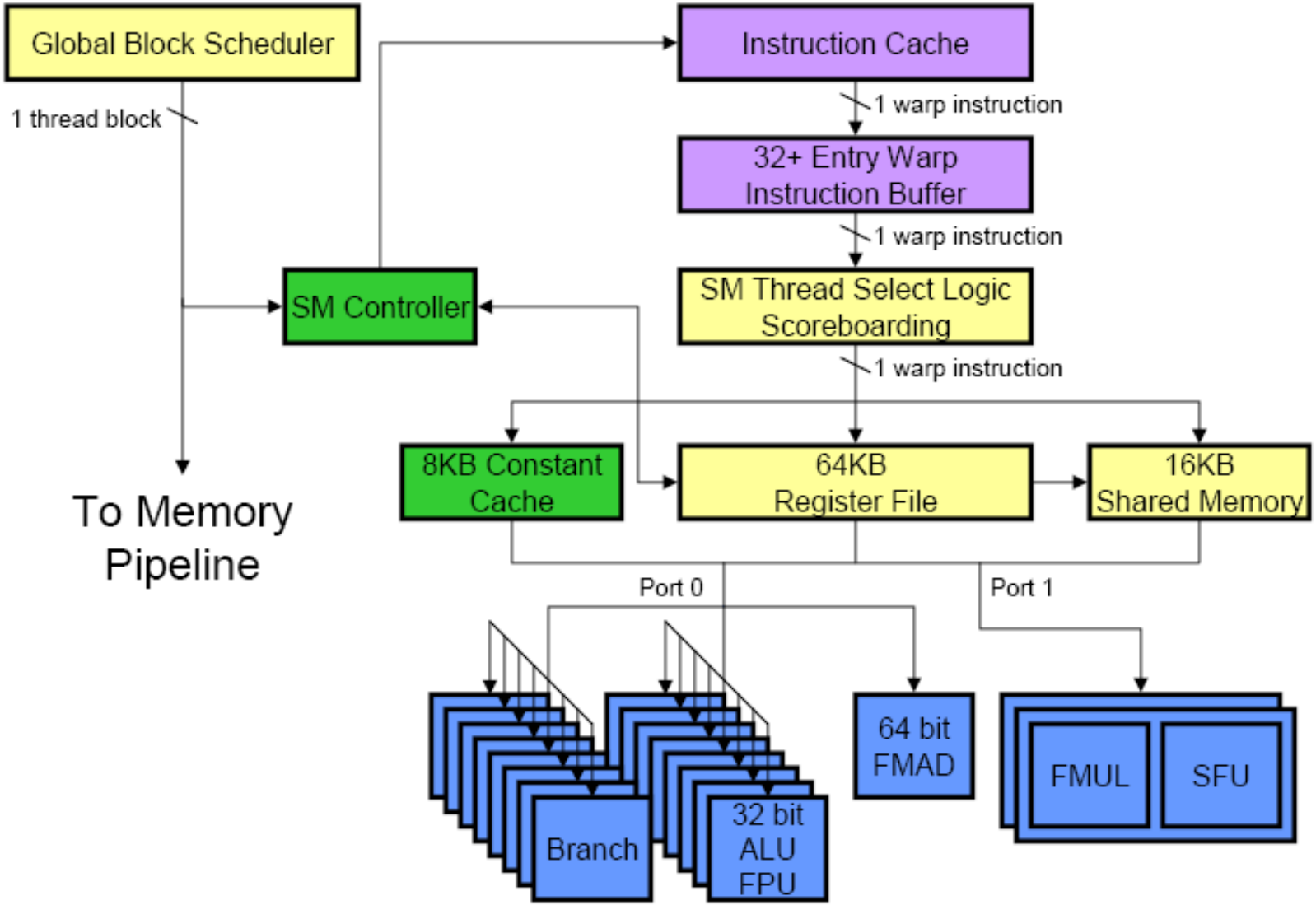
NVIDIA GT200: SM Architecture

- 64KB register file (2KB for each SP)
- 16KB shared memory
- 8KB constant cache
- 8 32-bit ALU/FPU
- 1 64-bit FMAD
- 8 branch units



1 SM executes in parallel up to 8 thread and manages up to 1024 threads

NVIDIA GT200: SM Architecture



NVIDIA GT200 Specs

Processor:

- 1.3 GHz, **936 Gflops SP**, 78 Gflops DP
- 10 **Texture Processor Cluster** (TPC)
- 1 TPC includes 3 **Streaming Multiprocessor** (SM) + 1 Texture Memory
- 1 SM include 8 **Streaming Processor** (SP)
  loosely corresponding to a modern CPU-core with 8-way SIMD computing-unit
- a total of $10 \times 3 \times 8 = 240$ **threads**

Memory:

- 4GB Global Memory, 512-bit, 102.4 GB/s, 400 . . . 600 cycles of latency
- Constant memory 64 KB (RO)
- Texture memory 256 KB (RO, two dimensional locality)

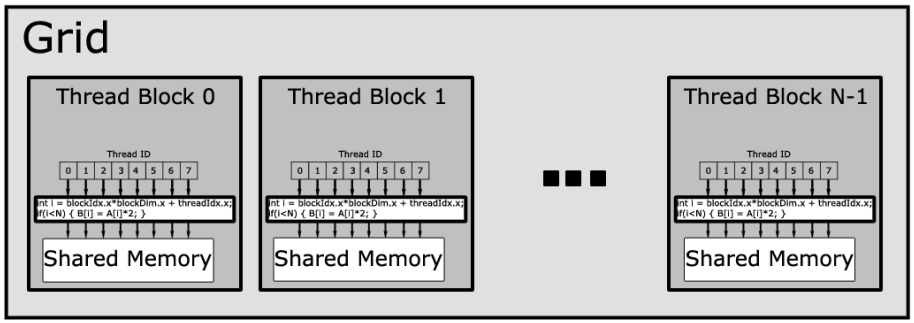
160 Watt (typical, w/Memory), $<$ 6 Gflops/Watt (SP, w/MUL), 1.5 K €

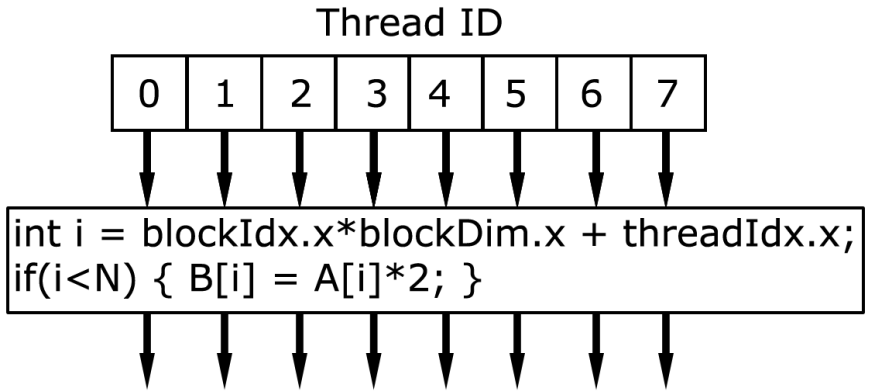
NVIDIA GPU Architecture Evolution

	GT200	GF100	GK110
#SM	15	14	14
#cuda-core	240	448	2688
GHz	1.2	1.15	800
GF/s (SP/DP)	936 / 78	1030 / 515	3950 / 1320
mem. bits	512	384	384
mem. MHz	1107	1500	1500
mem. GB/s (ECC-off)	141.7	144	250
Watt	160	215	235

GPU Programming Model

- execution has an hierarchical structure:
  - a grid of blocks
  - each block is a 1-2-3 D array of threads
- host launches a **grid** of thread-blocks
- a CUDA kernel (program executed on the device) is executed by an array of **threads**





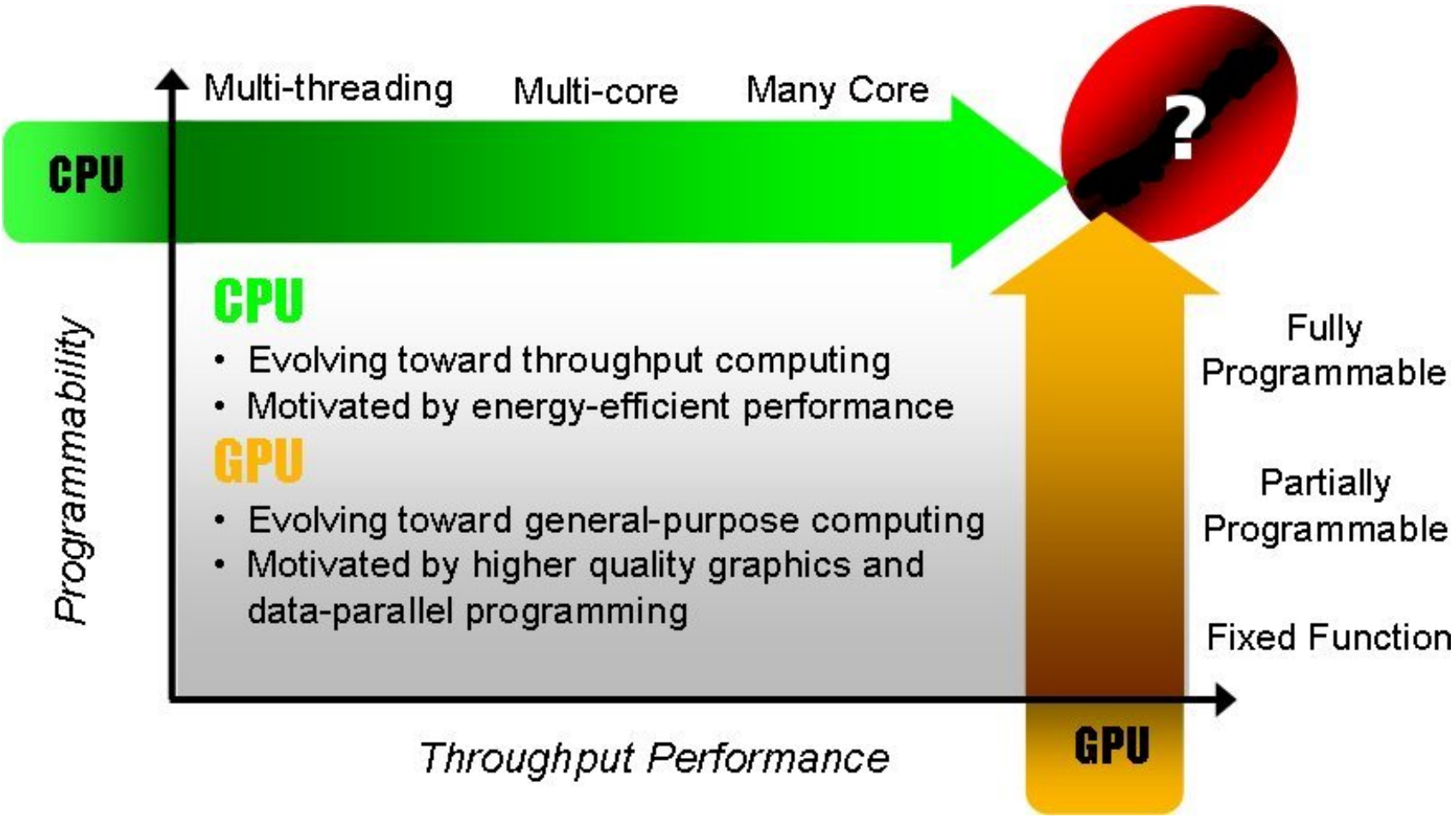
Vector sum example

```
// device code
__global__ void vadd ( double * A, double * B, double * C ) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  C[i] = A[i] + B[i];
}

int main () {
  double A_h[N], B_h[N], C_h[N];
  double * A_d, * B_d, * C_d;

  srand48();
  vinit(double *A, double *B, double *C);

  // allocate and copy data on the device
  cudaMalloc((void**) &A_d); cudaMalloc((void**) &B_d); cudaMalloc((void**) &C_d);
  cudaMemcpy(A_d, A_h, N, H2D); cudaMemcpy(B_d, B_h, N, H2D); cudaMemcpy(C_d, C_h, N, H2D);

  dim3 dimBlock(64,  1) ; // size of thread-block
  dim3 dimGrid(N/64, 1) ; // size of block-grid

  // run kernel
  vadd <<< dimGrid, dimBlock >>> (A,B,C);

  cudaThreadSynchronize(); // wait until kernel terminates !!!!

  // copy results back to host
  cudaMemcpy(C_h, C_d, N, D2H);

  // feee memory device
  cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}
```

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 ?



. . . towards a convergence between CPU and GPU architectures

First attempt to merge GPU and CPU concepts: MIC architectures

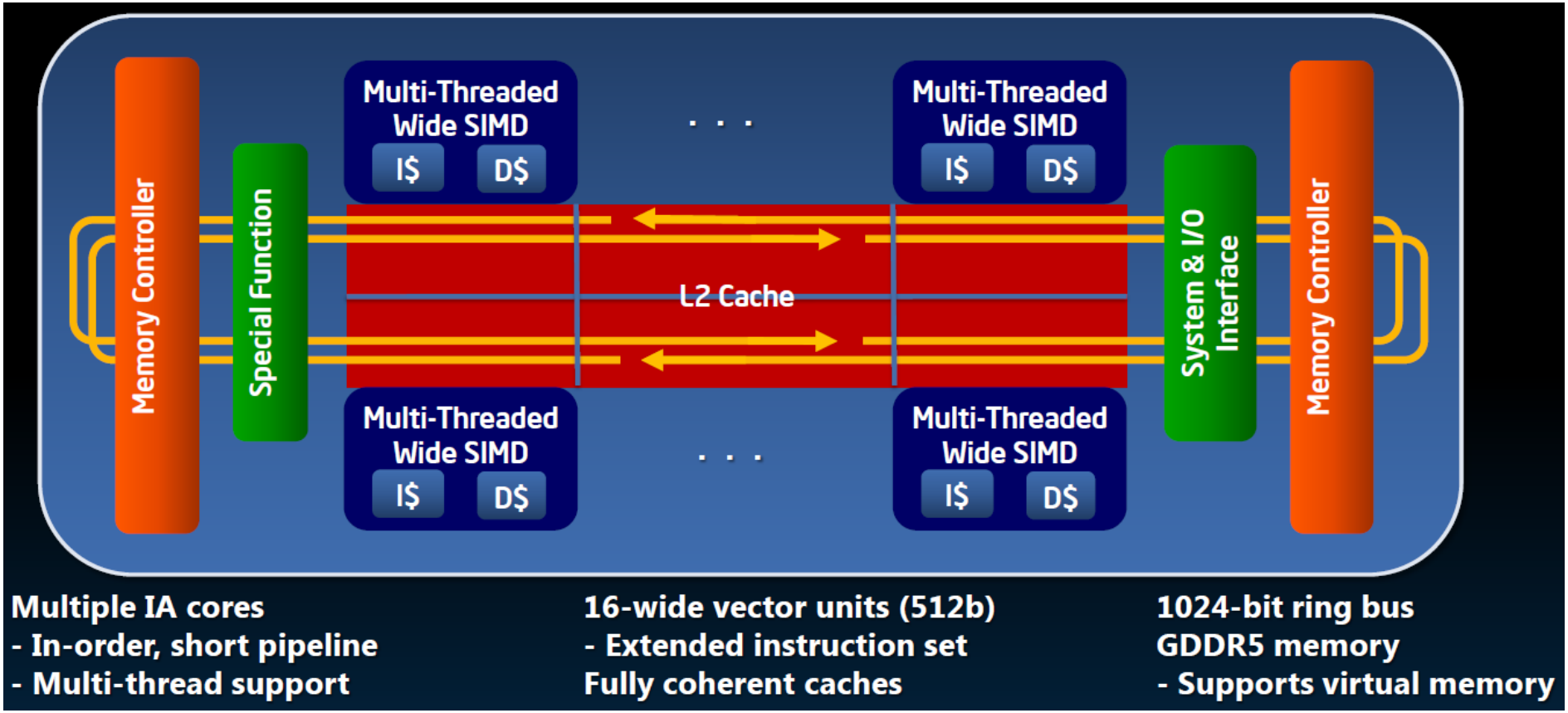
MIC: Many Integrated Core Architecture

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

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

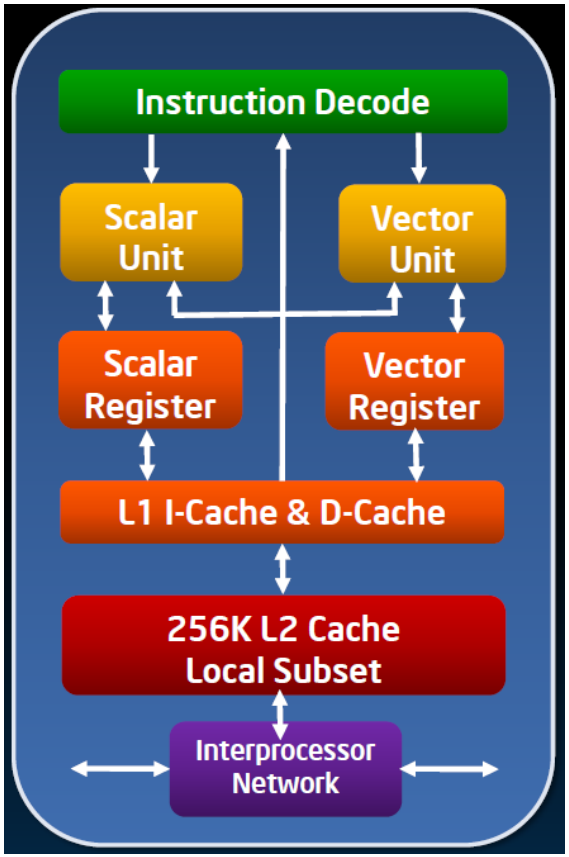
MIC Architectures



- 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 Model

- native:
  `icc -mmic pippo.c -o pippo`
- offload:
  using approriate `pragmas` to mark code that will be *transparently* executed onto the MIC board

Programming is well integrated with many languages:

- openMP
- TBB
- Cilk
- ...

Parallelism management

- offload a code that spanws threads
- use openMP

```
for(t = 0; t < NTHREAD; t++) {
  pthread_create(&threads[t], NULL, threadFunc, (void *) &tData[t]);
}

for(t = 0; t < NTHREAD; t++) {
  pthread_join(threads[t], NULL);
}
```

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  theadFunc((void *) &targv[tid]);
}
```

Example: saxpy

$$C = s \times A + B$$

```
#define N 1717

void vinit (double *A, double *C) {
  int i;
  for (i=0; i<N; i++){
    A[i] = drand48(); B[i] = drand48(); C[i]= 0.0;
  }
}

int main () {
  double A[N], B[N], C[N];
  double s;
  srand48();
  vinit(double *A, double *B, double *C);
  s = rand48();

  #pragma offload target(mic:-1) in(A,B:lenght(N)) in(s) inout(C:lenght(N))
  {
    #pragma omp parallel for private(i)
    for ( i=0; i < N; i++ )
      C[i] = s* A[i] + B[i]
  }

  ...

}
```

Example: vector sum

```
#define N 1717

void __attribute__((target(mic))) vadd ( double *A , double *B , double *C )

void vinit (double *A, double *B, double *C) {
  int i;
  for (i=0; i<N; i++){
    A[i] = drand48(); B[i] = drand48(); C[i]= 0.0;
  }
}

int main () {
  double A[N], B[N], C[N];
  srand48();
  vinit(double *A, double *B, double *C);
  ....
  #pragma offload target(mic:0) in(A,B:lenght(N)) inout(C:lenght(N))
  {
    vadd (A,B,C);
  }
  ....
}

void vadd (double *A, double *B, double *C) {
#ifdef __MIC__
  int i;
  for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
#else
  fprint(stderr, "This code is running on the host\n");
#endif
}
```

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

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.

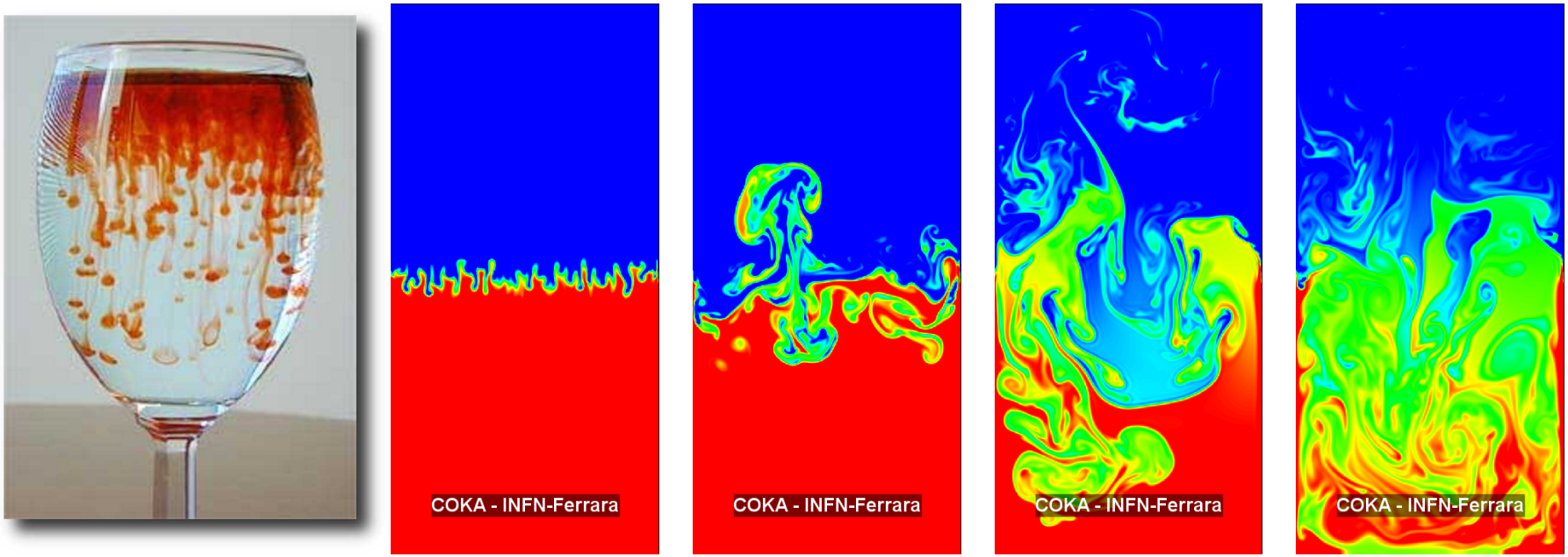
The D2Q37 Lattice Boltzmann Model

- 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.

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).

LBM Computational Scheme

```
foreach time-step
  foreach lattice-point

    propagate();

    collide();

  endfor
endfor
```

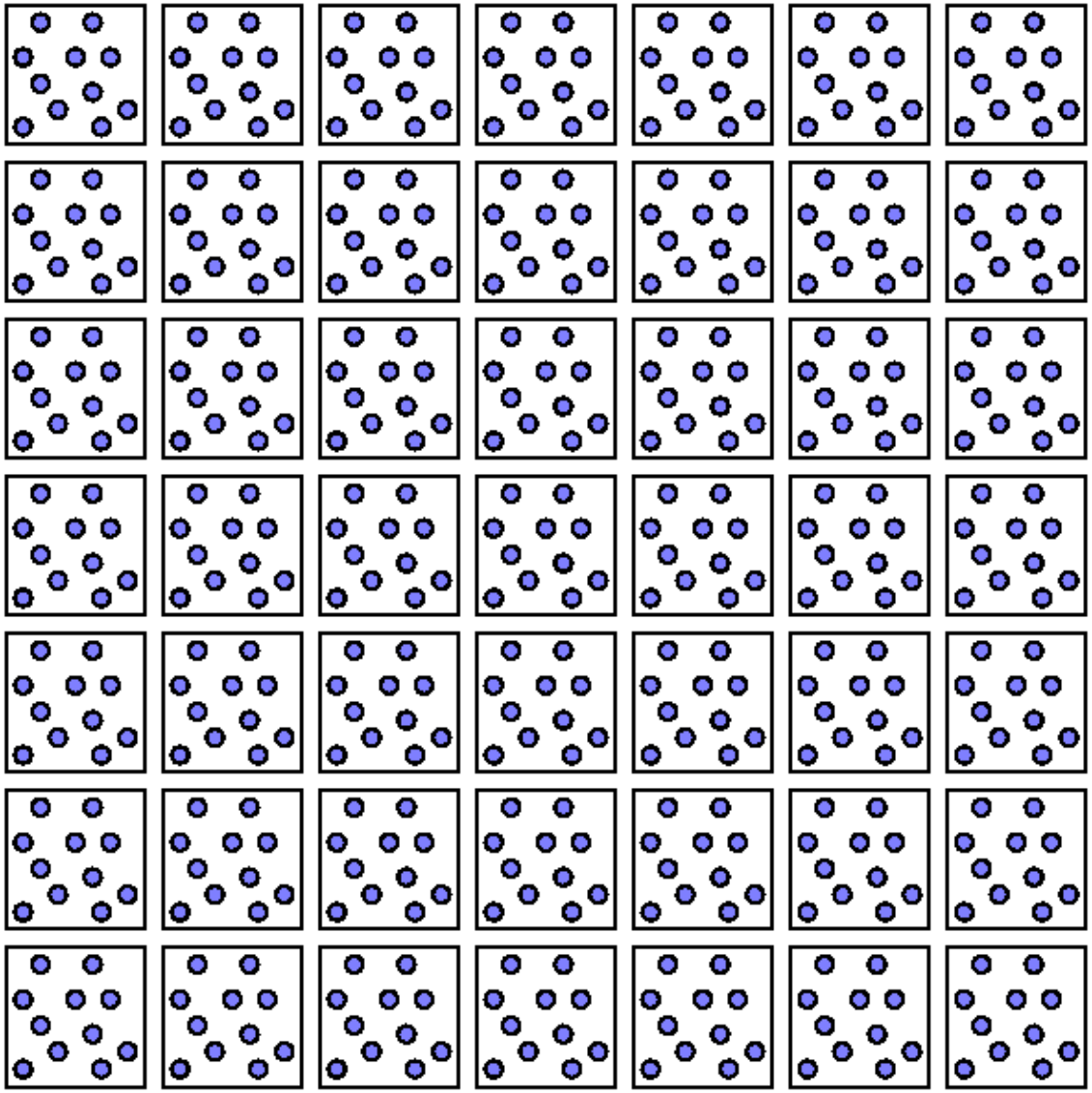
Embarrassing parallelism

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

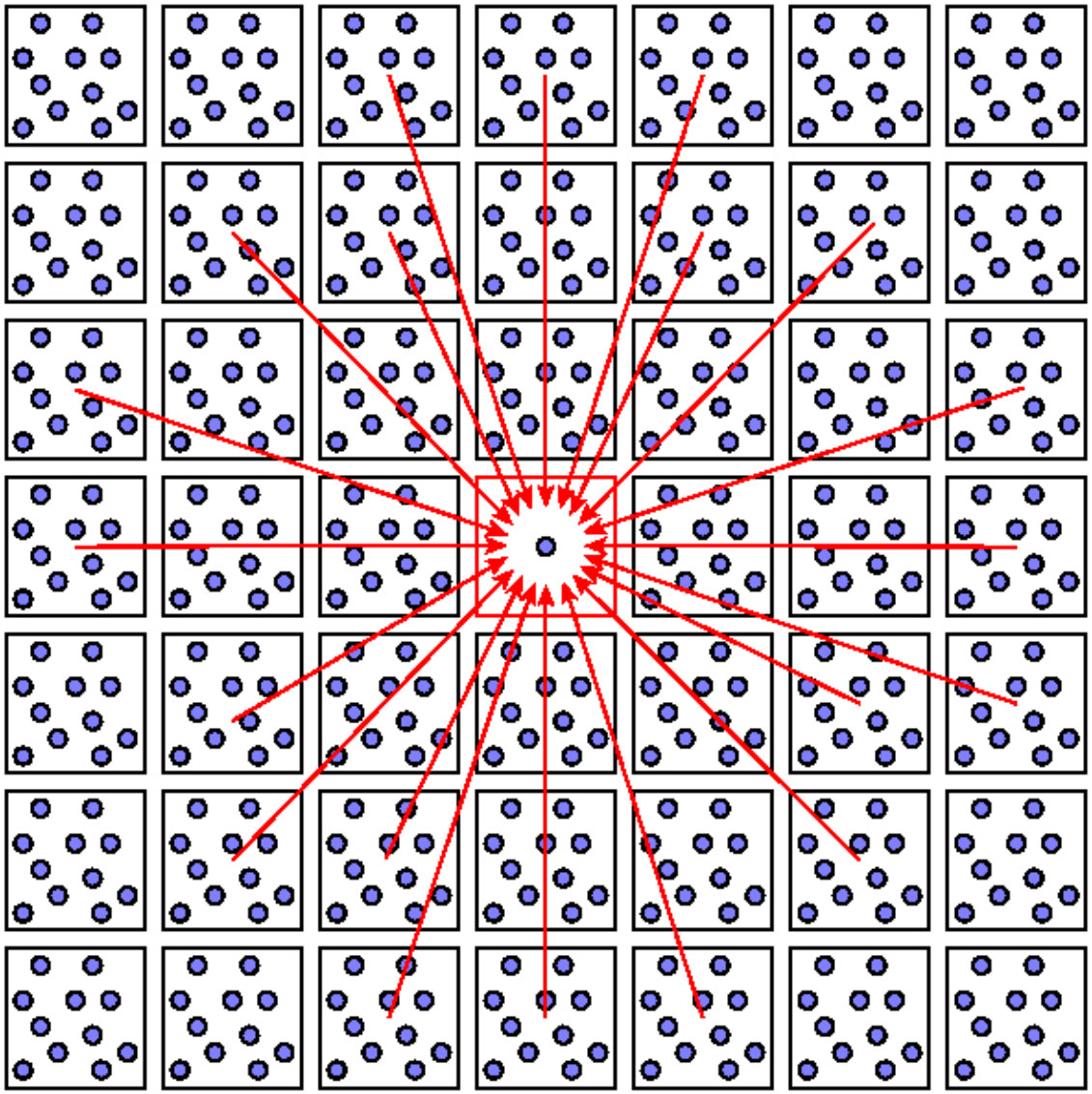
Challenge

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

D2Q37 propagation scheme

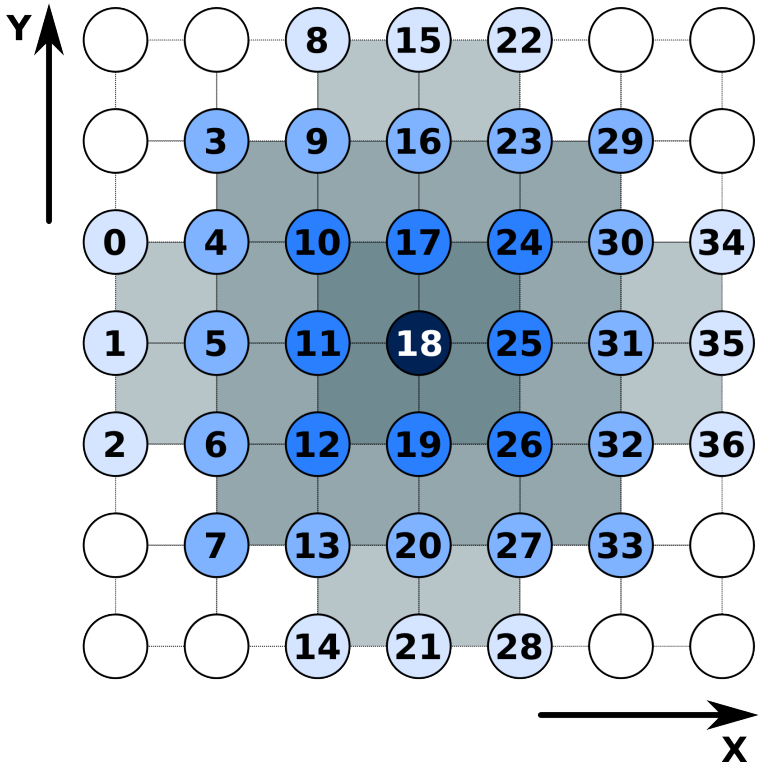


D2Q37 propagation scheme



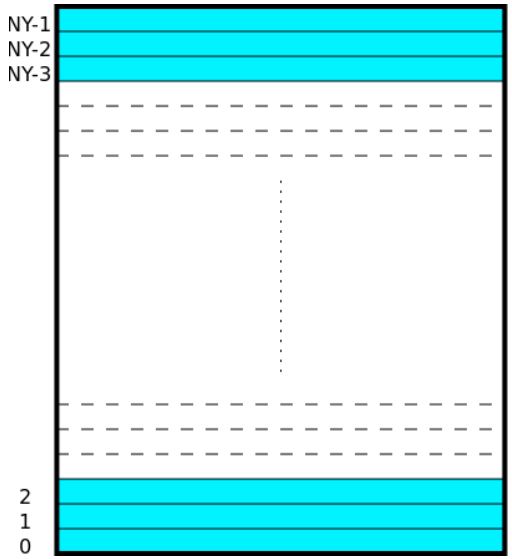
Gather 37 populations from 37 different lattice-sites.

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: boundary-conditions



- we simulate a 2D lattice with periodic-boundaries along x-direction
- 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 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

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 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.

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];
```

- 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;
```

- AoS exploits cache-locality of populations of as site: relevant for computing collision
- SoA exploits data locality of corresponding populations of sites: suitable for GPUs.

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:

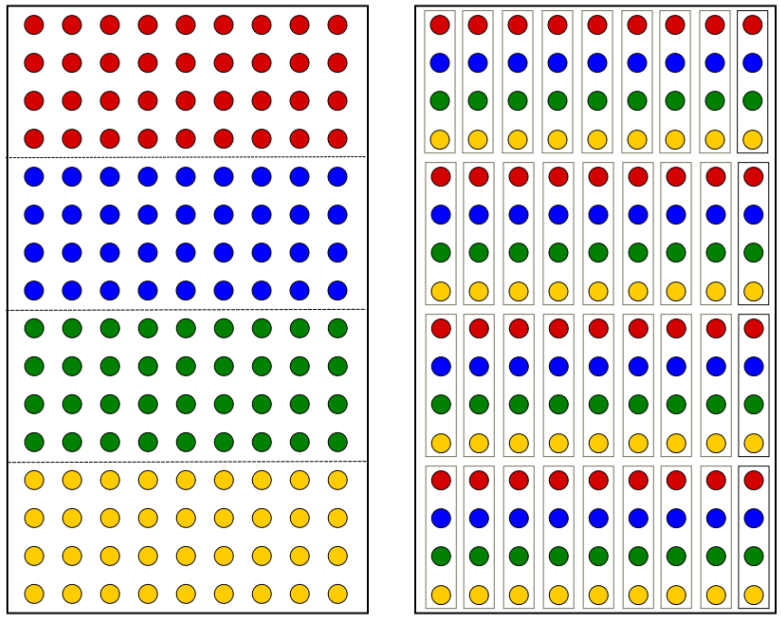
```
for ( step = 0; step < MAXSTEP; step++ ) {

  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

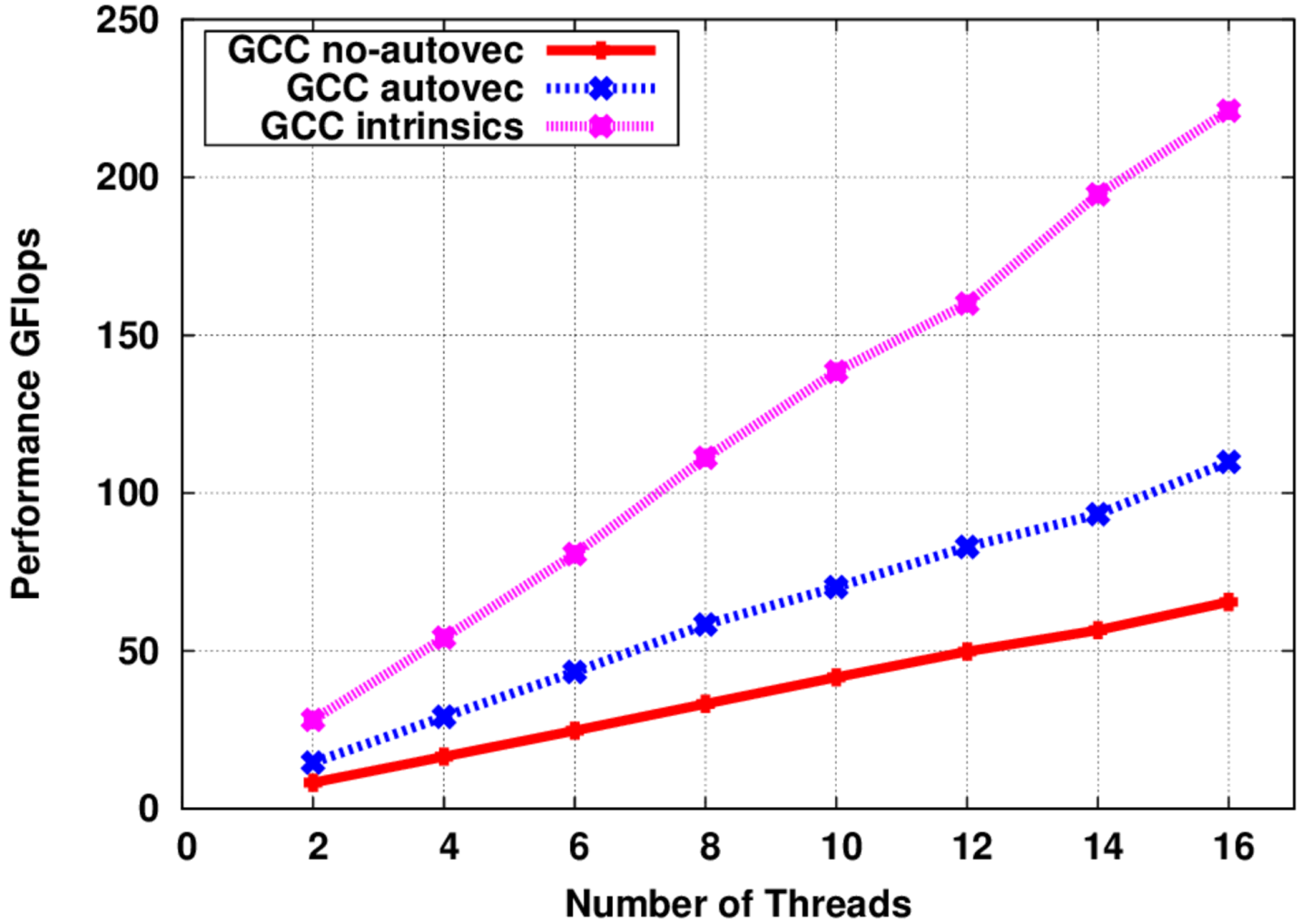
GCC and ICC vectorization by

- 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)



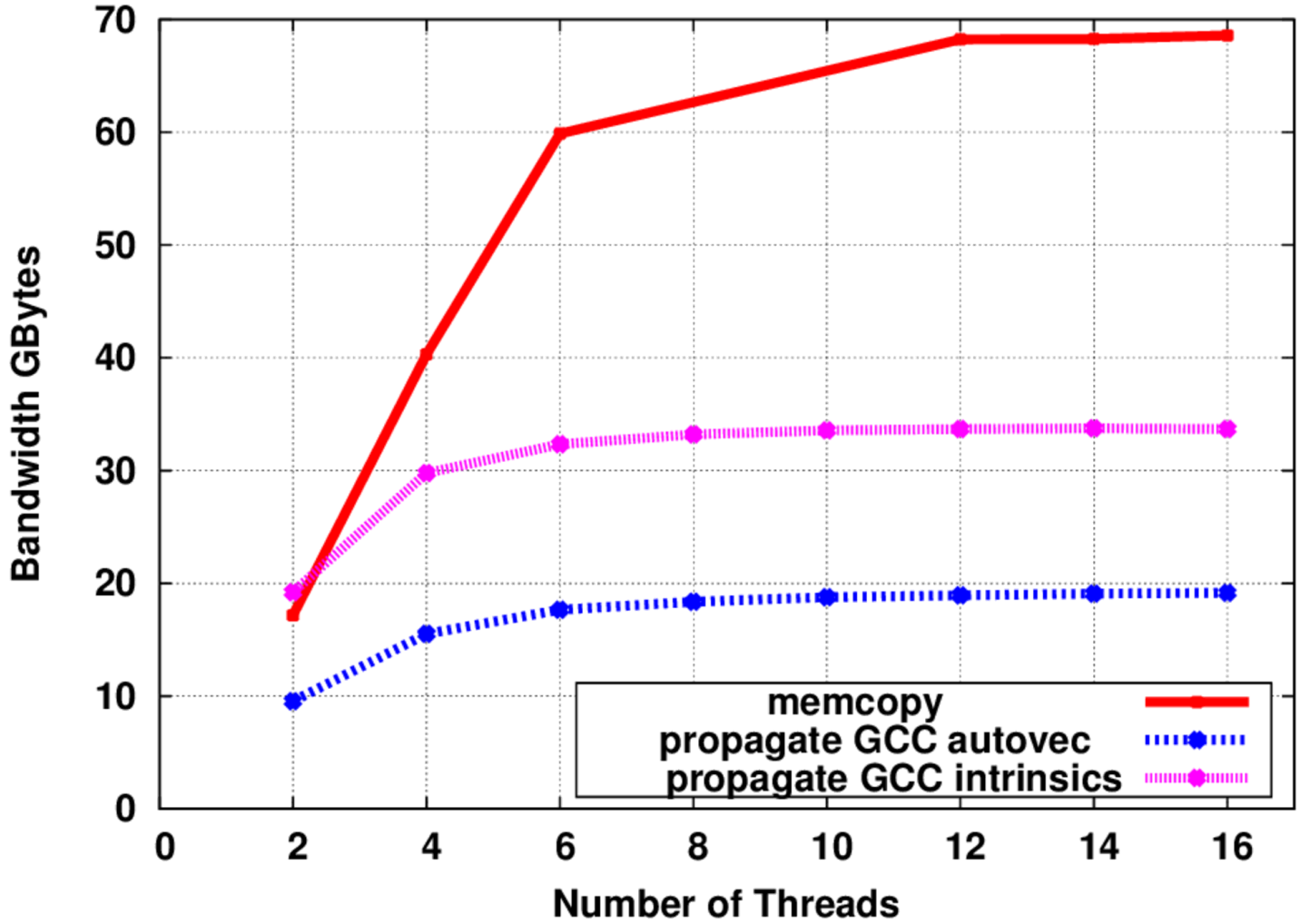
```
typedef double fourD __attribute__ ((vector_size(4*sizeof(double))));

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

Collide Performance



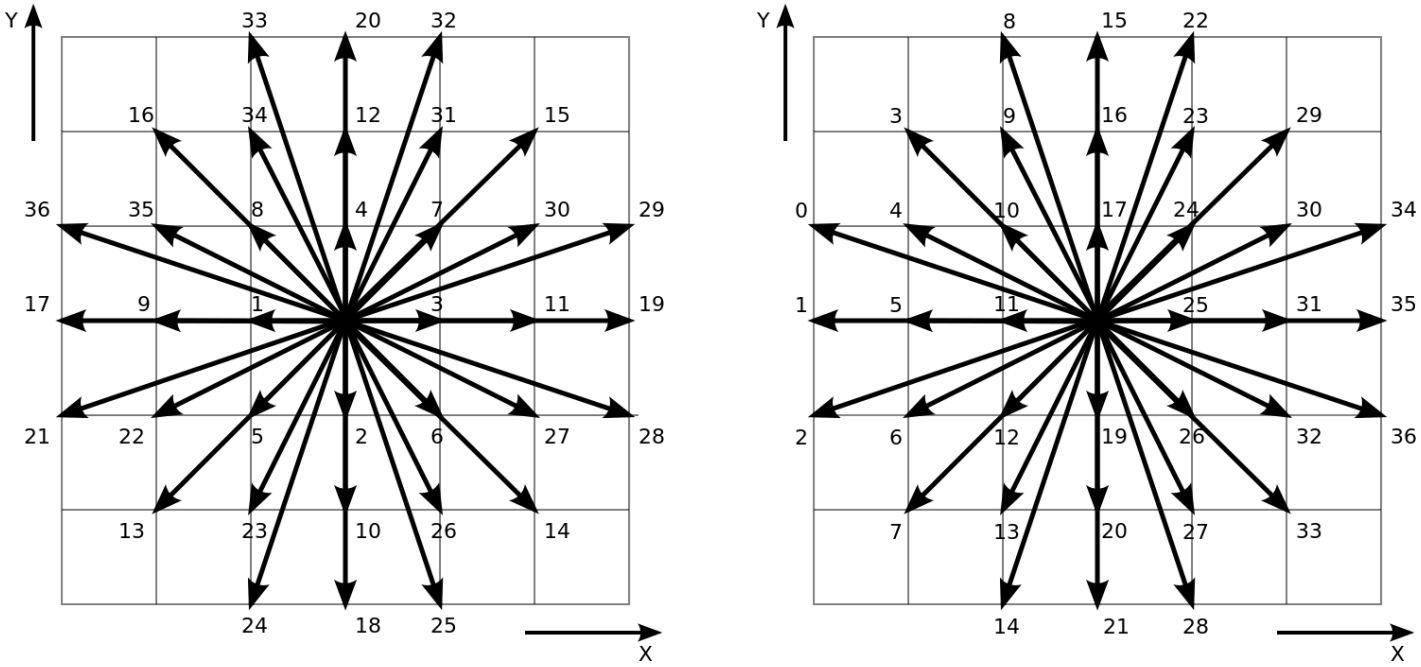
- GCC no-autovec: 18% of peak
- GCC autovec: 31% of peak
- GCC intrinsics: 62% of peak

Propagate Performance



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

Optimization of Propagate



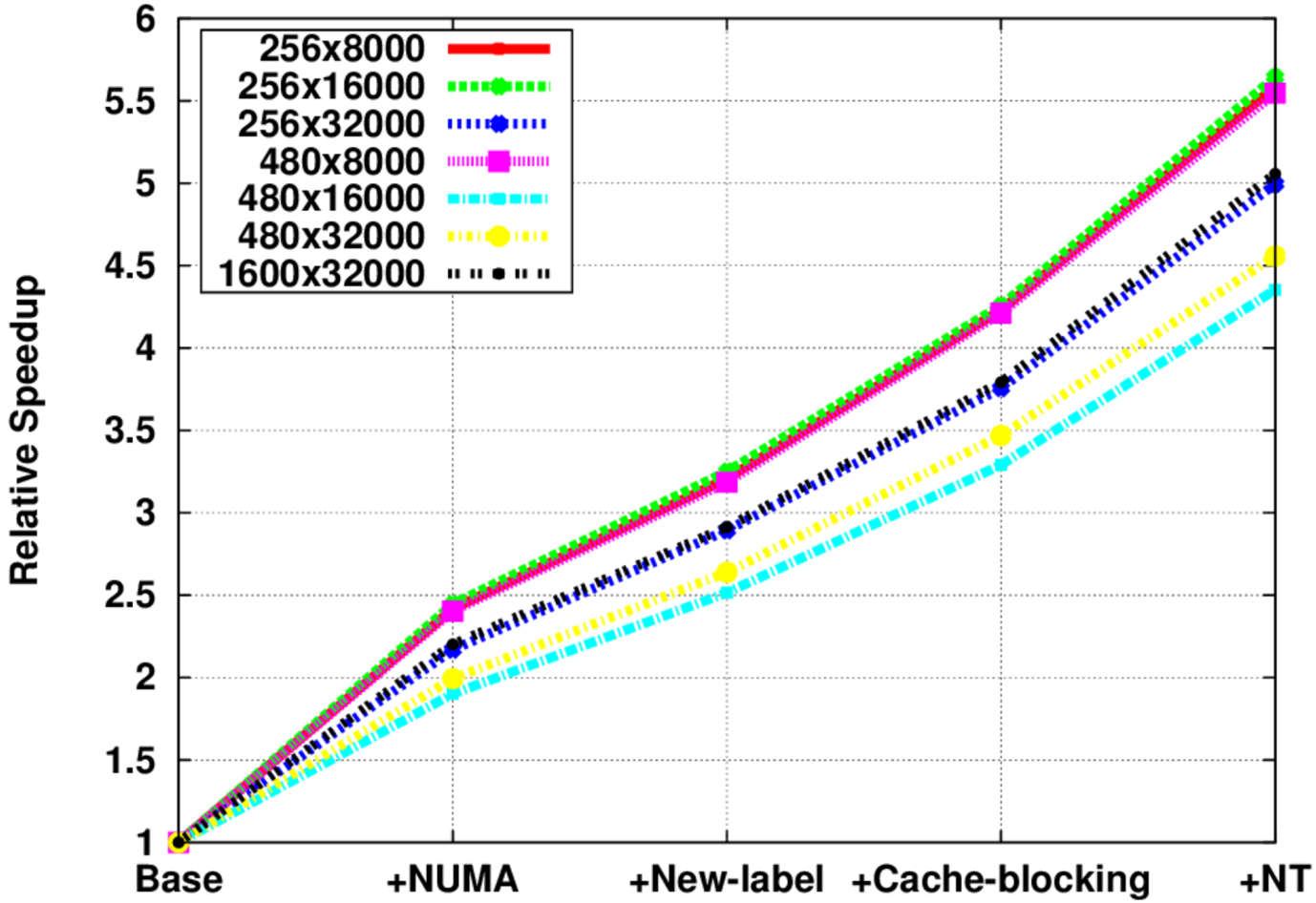
- **cache data-reuse**: reordering of populations allows a better CACHE-reuse and improves performances of propagate;
- **NUMA control**: using the NUMA library to control data and thread allocation avoids overheads in accessing memory;
- **cache blocking**: load the cache with a small data-subset and 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 MB/s, $\approx$ 67% of peak and very close to memory-copy (68.5 MB/s).

Full Code Performance Version 1

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

	NVIDIA C2050	2-WS	2-SB
propagate collide	29.11 ms 154.10 ms	140.00 ms 360.00 ms	42.12 ms 146.00 ms
propagate collide	84 GB/s 205.4 GF/s	17.5 GB/s 88 GF/s	60 GB/s 220 GF/s
T/site MLUps	44 ns 22	130 ns 7.7	46 ns 21.7
P R_{max}	172 GF/s **33%**	60 GF/s **38%**	166 GF/s **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 MLUps	40 ns 25	110 ns 9.3	35 ns 28.2
P R_{max}	188 GF/s **36%**	72 GF/s **45%**	216 GF/s **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:

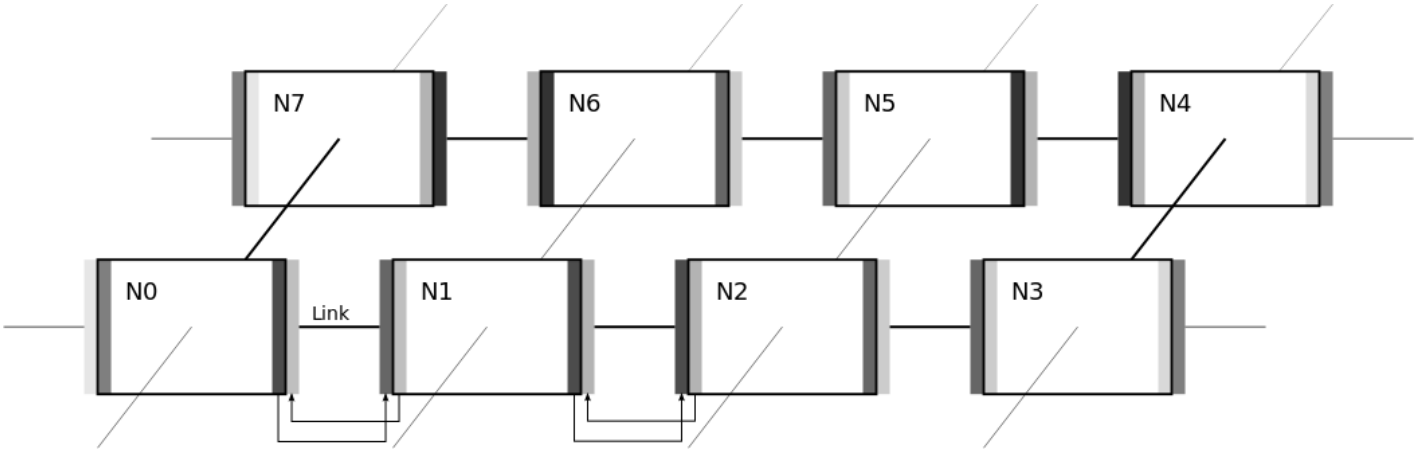
- AVX version improves performances of `collide` and `propagate` by a factor $\approx$ 2X w.r.t. the SSE
- efficiency is high: 45% – 62% for the dual-socket

GPU Implementation

- JUDGE - JÜlich Dedicated Gpu Environment
- PLX system at CINECA

- **Compute Nodes**:
  - 54 Compute nodes IBM System x iDataPlex dx360 M3
  - node: 2 Intel Xeon X5650(Westmere) 6-core processor 2,66 GHz
  - Main memory: 96 GB
  - Network: IB QDR HBA
  - GPU: 2 NVIDIA Tesla M2050 (Fermi) 1,15 GHz (448 cores), 3 GB memory
- **Complete System**:
  - 648 CPU cores
  - 108 GPU cards
  - 5,1 TB main memory
  - 62,5 Teraflops peak performance

Node Parallelism



- lattice $L_x \times L_y$ is split into sub-lattices $\frac{L_x}{N_p} \times L_y$ along X-direction
- on each node *borders* of neighbor sub-lattices are replicated (ghost borders)
- on each node the sub-lattice is further split on GPUs
- nodes have been logically arranged in a ring
- X-splitting requires to exchange Y-borders with neighbours
- make easy parallelization w/o bad impacts on performance
- ... other splitting can be used.

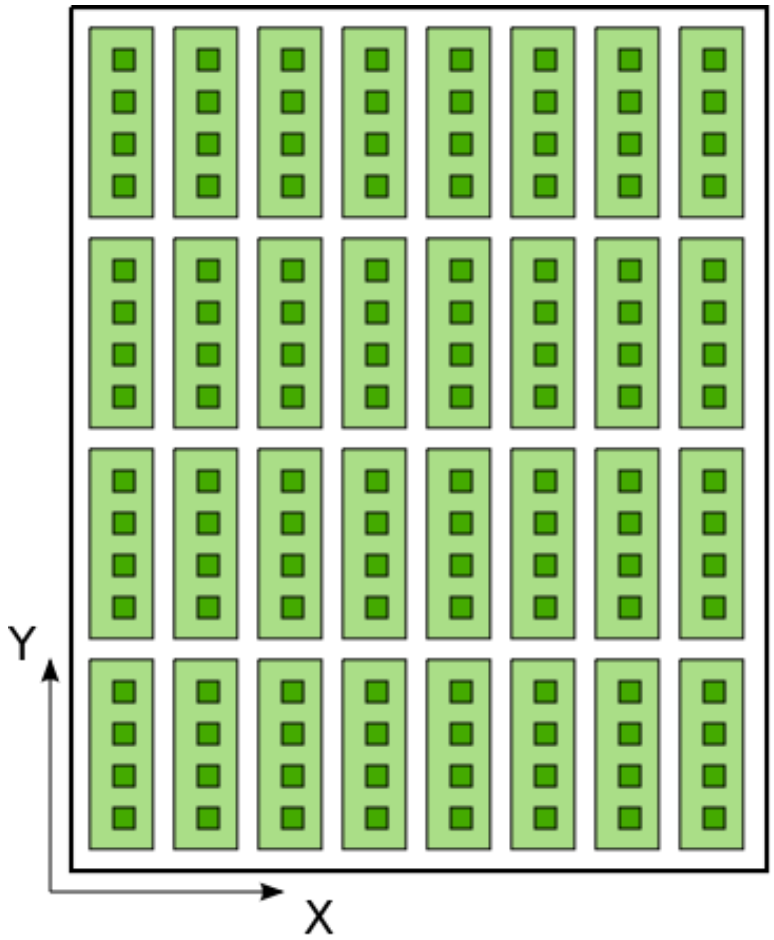
Host Program

```
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
}
```

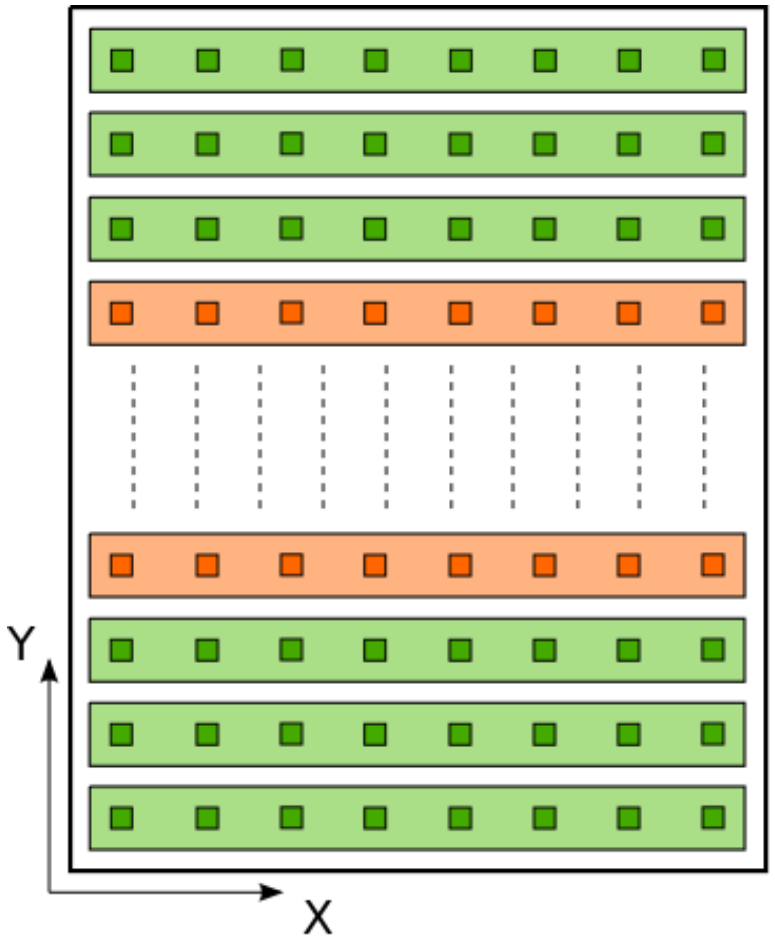
The lattice is stored as a Structure of Arrays (SOA) to exploit data-coalescing.

CUDA Grids Layouts

Physical lattice of 8×16 sites: each CUDA-thread process a lattice-point.



`propagate()` and `collide()`



`bc()`.

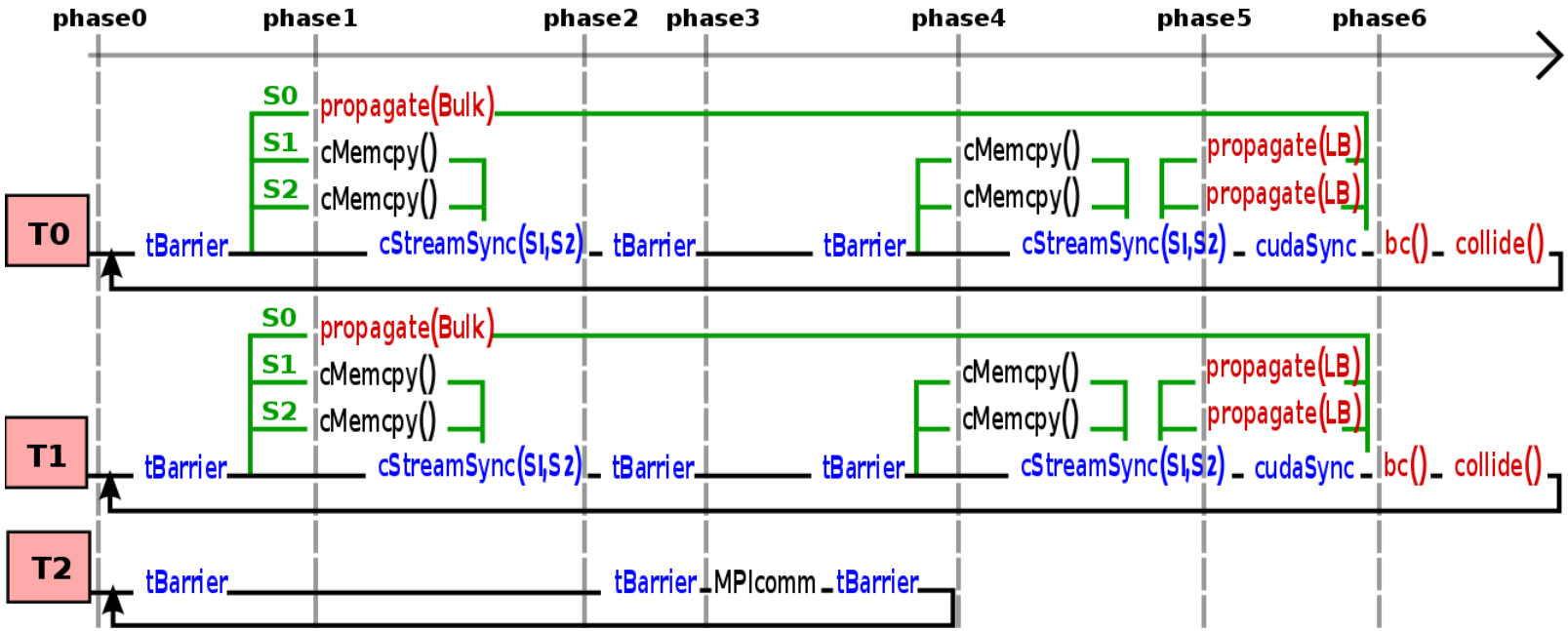
Execution Times

Lattice size: 252×16384

#threads/block	32	64	128	256	512
ECC Enabled					
propagate() ms	33.6	28.5	29.1	29.5	29.2
propagate() GB/s	72.7	85.8	84.0	82.8	83.6
collide() ms	196.4	157.0	156.0	157.4	164.7
collide() GFLOPS	161.1	201.6	202.9	201.0	192.1
ECC NOT Enabled					
propagate() ms	30.6	21.4	22.1	22.5	22.3
propagate() GB/s	79.8	114.1	110.4	108.5	109.4
collide() ms	192.4	151.0	150.5	151.7	158.9
collide() GFLOPS	164.5	209.7	210.3	208.6	199.1

ECC is crucial for the reliability of large production runs.

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

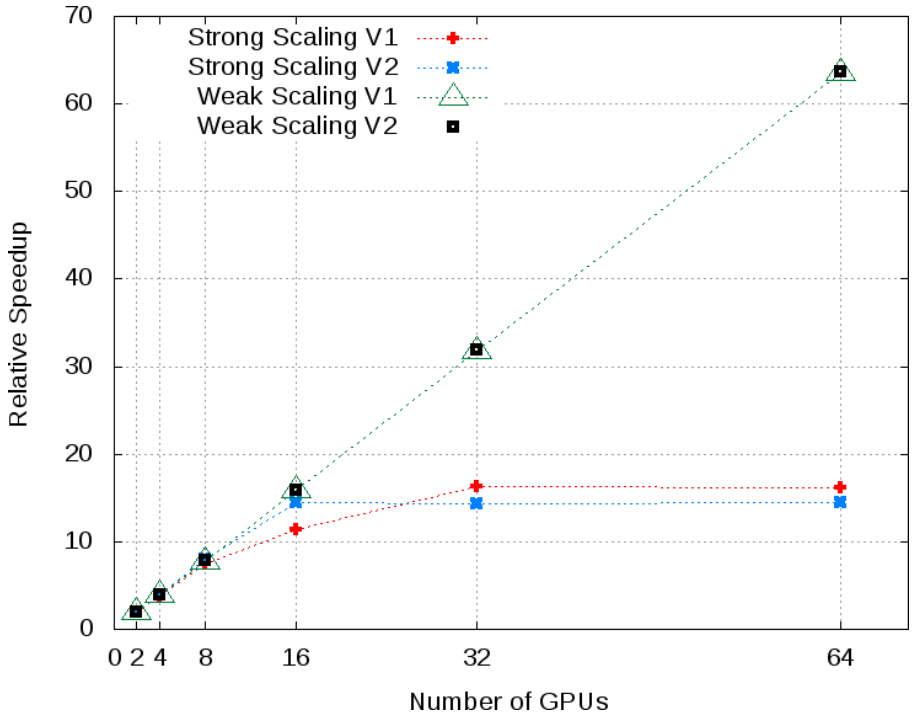
Results: Single GPU Performance

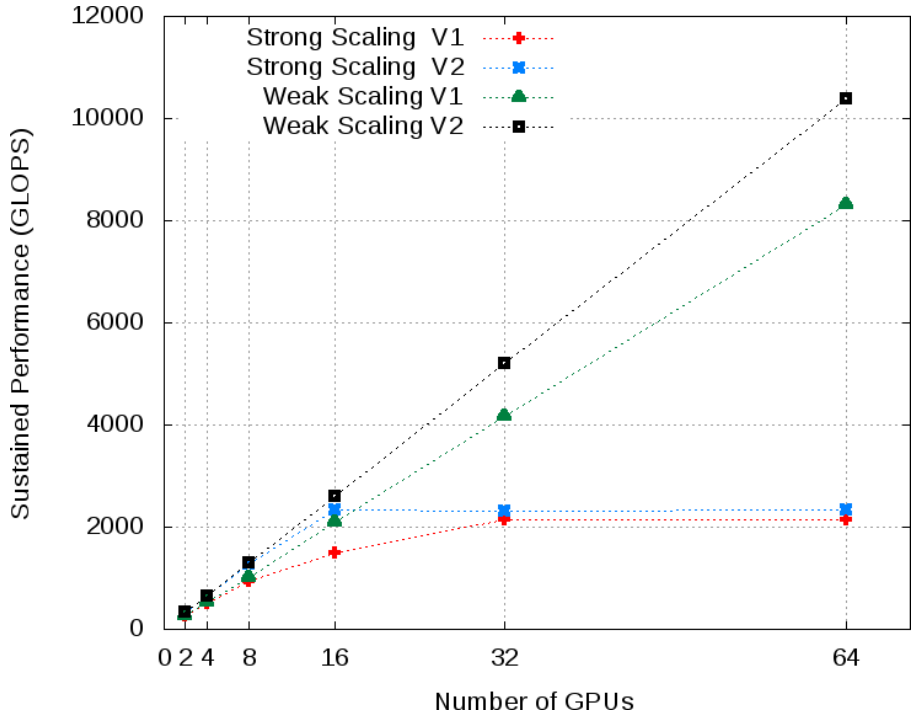
D2Q37 Version V1			
	GPU	CPU-W	CPU-S
GFLOPS	172	60	166
R_{max}	**33%**	**38%**	**48%**
T/site ns	44	130	46
MLUps	22	7.7	21.7

D2Q37 Version V2			
	GPU	CPU-W	CPU-S
GFLOPS	188	72	216
R_{max}	**36%**	**45%**	**62%**
T/site ns	40	110	35
MLUps	25	9.3	28.2

- GPU: NVIDIA Tesla C2050 card, $\approx$ 500 Gflops DP peak-performance
- CPU-W: dual six-core Westmere system, $\approx$ 160 Gflops DP peak-performance
- CPU-S: dual eight-core Sandybridge system, $\approx$ 345 Gflops DP peak-performance

Results: Multi-GPU Performance





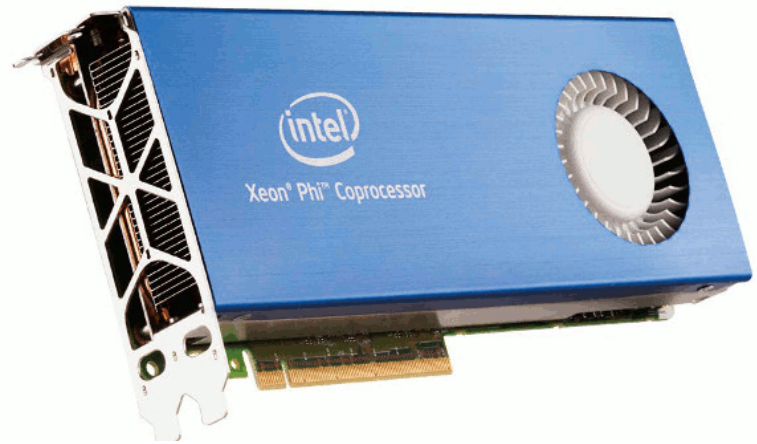
- strong regime: code runs on a lattice size $L_x \times L_y = 1024 \times 7168$
- weak-regime: sub-lattice size on each node is $L_x \times L_y = 254 \times 14464$

Results

- **performance**: the single GPU-code performs
  - a factor $\approx 2 - 3$ better than a dual-socket system based on 6-core Westmere CPUs;
  - equal to a just launched system based on 8-core Sandybridge CPUs (Launch Date Q1'12).
- **efficiency**: on GPUs is lower than CPUs, but still high for a production ready code $\approx 30 - 40\%$ of peak performance.
- **strong-regime scalability**: is good as the size of the local lattice is large enough to hide communication and memory-copy overheads.
- **weak-regime scalability**: is linear as the communications has been hidden with computation of propagate phase.
- **programmability**: fine-tuning CPU-program has required accurate programming efforts while on GP-GPU CUDA allows to exploit data-parallelism.

Xeon-Phi 5110P co-processor

#cores	61
frequency	1090 MHz
memory	8 GB GDDR5
L1-cache / core	32 KB
L2-cache / core	512 KB
Peak Perf. SP/DP	$\approx 2/1$ TFlops
Peak Memory Bw	320 GBytes



- PCIe 16x Gen2 card (8 GB/s)
- 240 threads
- 512-bit vector FPU
- L2-cache blocks are shared among the cores
- 512-bit *Advanced Vector Extensions* (AVX)

$$T_{exe} \geq \max\left(\frac{W}{F}, \frac{I}{B}\right) = \max\left(\frac{7666}{1064}, \frac{592}{320}\right) \text{ ns} = \max(7.20, 1.85) \text{ ns}$$

Relevant Optimization

Xeon-Phi peak floating-point throughput:

$$P = f \times \#\text{cores} \times \text{NopPerCycle} \times \text{NflopPerOp}$$

In our case we have:

- $f = 1.090$ GHz
- #cores = 61
- NopPerCycle = 2, one fused-multiply-add per cloc-cycle
- NflopPerOp = 16 single-, 8 -double precision

Relevant Optimization

Applications running on Xeon-Phi can approach peak performance if codes exploits relevant hardware features:

- **core parallelism**:
  all cores has to be kept active and working in parallel, e.g. running different functions or working on different data-sets (MIMD/multi-task or SPMD parallelism);

- **hyper-threading**:
  cores have to execute 2 or more threads (up-to 4) to keep hardware pipelines busy and hide memory accesses latency;

- **vector programming**:
  each core has to process data-set using vector (streaming) instructions (SIMD parallelism); in the case of Xeon-Phi up-to 8 double-precision values can be processed by each vector instructions

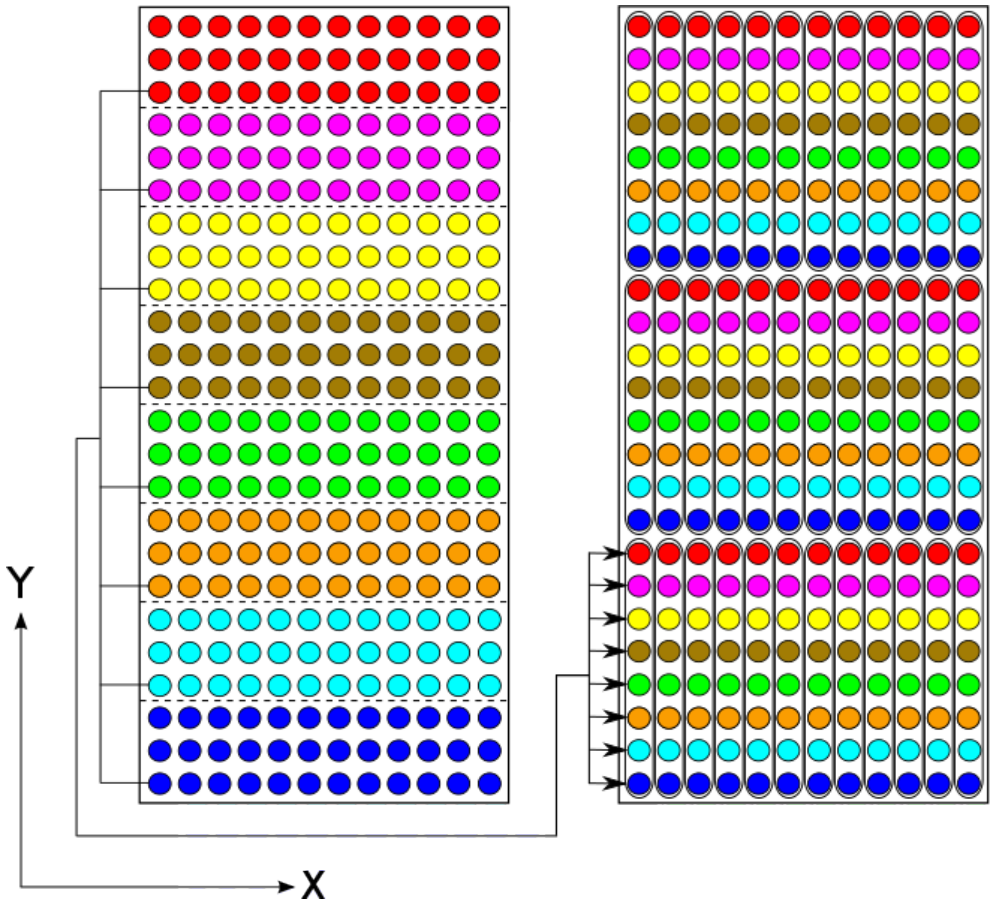
Core Parallelism: Threads Management

```
for(t = 0; t < NTHREAD; t++) {
  pthread_create(&threads[t], NULL, threadFunc, (void *) &tData[t]);
}

for(t = 0; t < NTHREAD; t++) {
  pthread_join(threads[t], NULL);
}
```

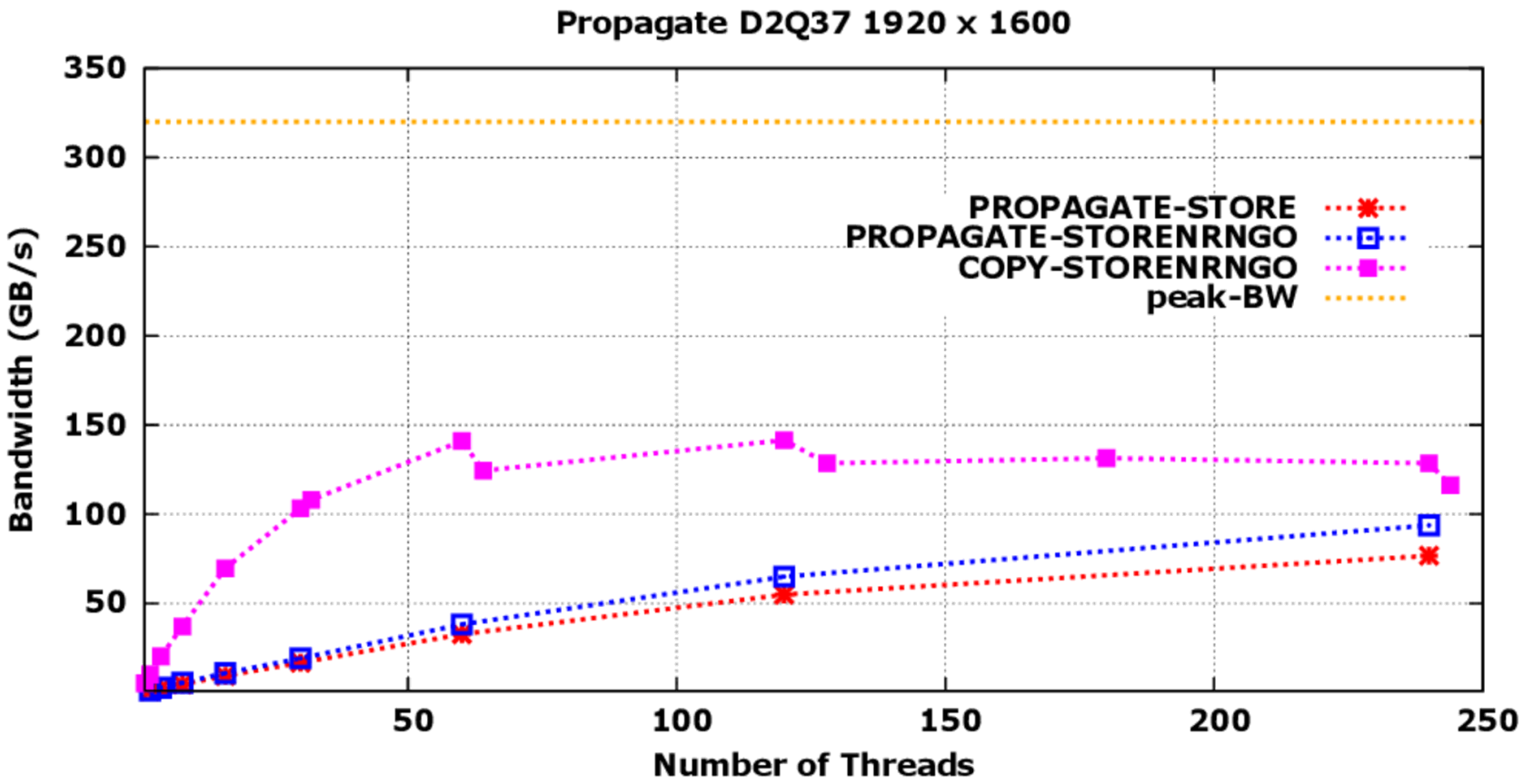
```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  theadFunc((void *) &targv[tid]);
}
```

Vector Parallelism

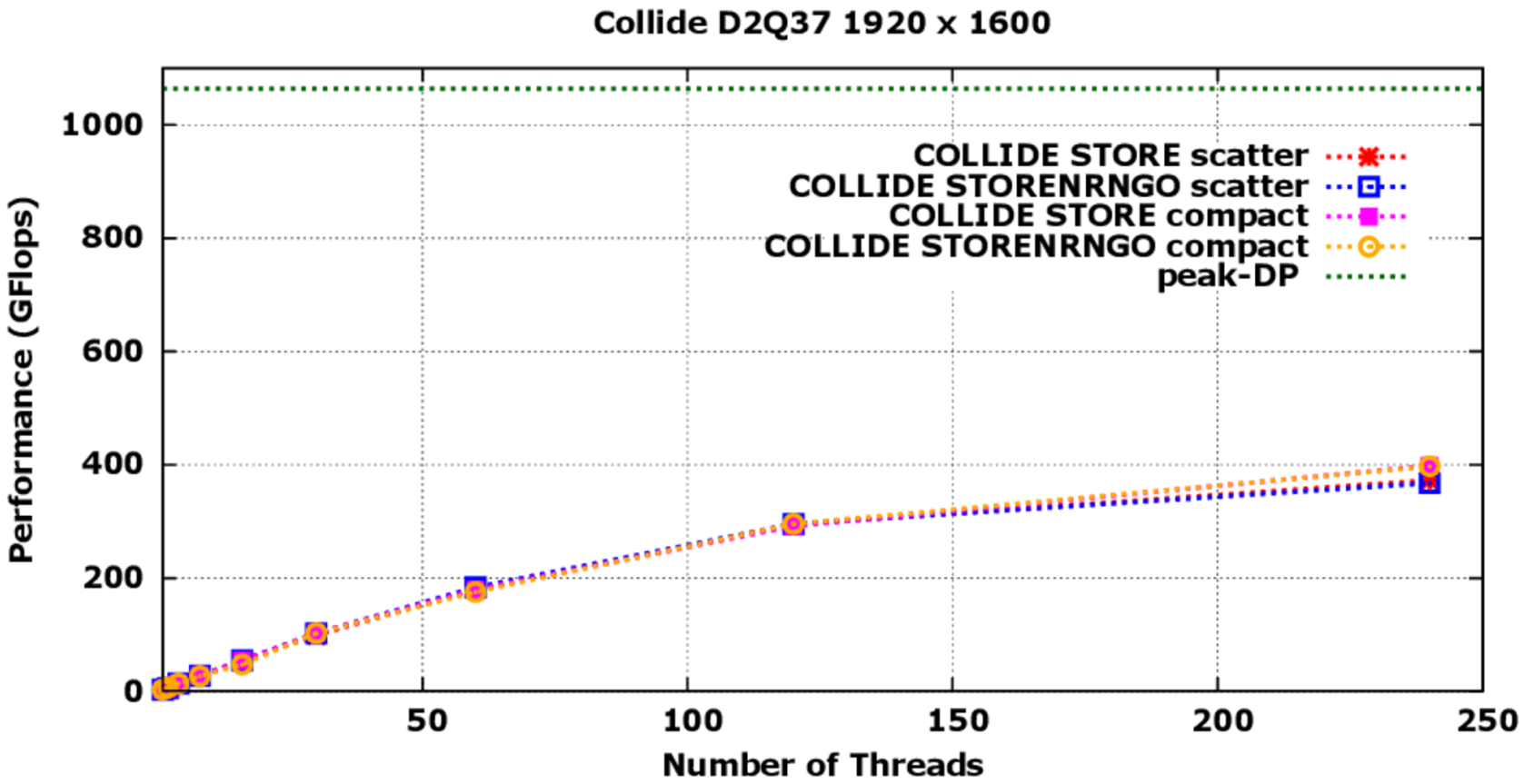
```
struct {
   __m512d vp0;
   __m512d vp1;
   __m512d vp2;
   ...
   __m512d vp36;
} vpop_t;

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



Lattice Boltzmann Propagate Benchmark



Lattice Boltzmann Collide Benchmark



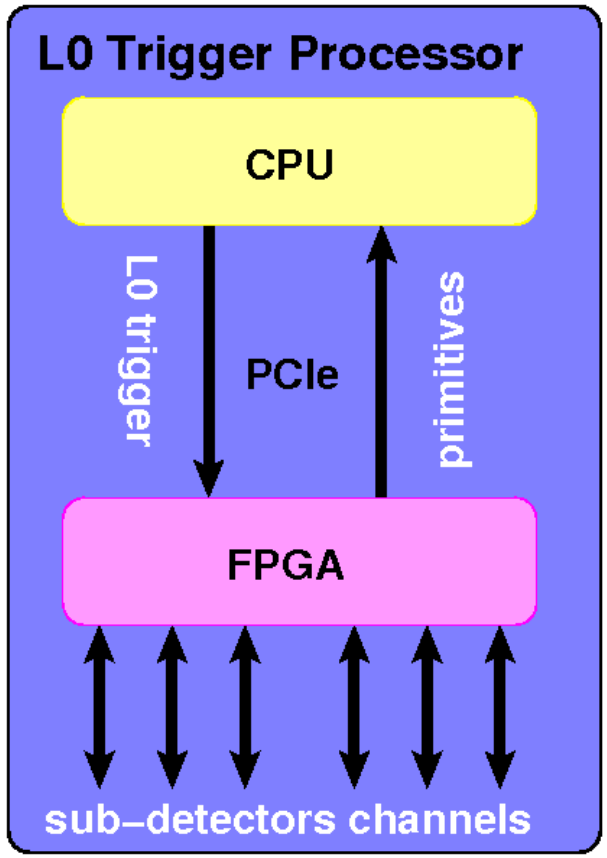
Lattice Boltzmann Summary

	C2050	2-WS	2-SB	Xeon-Phi	K20
propagate GB/s	84	17.5	60	94	160
ϵ	58%	29%	70%	29%	64%
collide GF/s	205.4	88	220	394	506
ϵ	41%	55%	63%	37%	38%
ξ (collide)	–	1.19	1.27	0.76	–

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

PC-based L0 Trigger

L0TP = FPGA + CPU



1. *primitives* $\Rightarrow$ to memory of the CPU
2. CPU elaborates *primitives*
3. CPU sends back *L0 trigger signals*

First Exercise

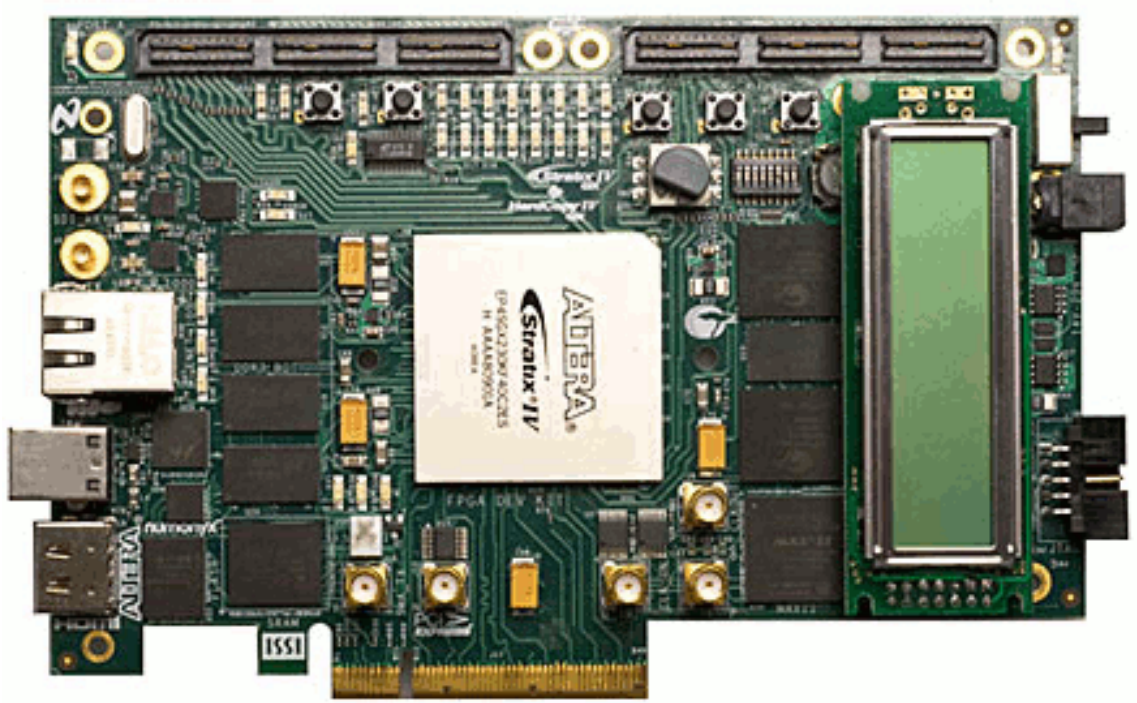
measure the distribution (and maximum) Round-Trip Time (RTT) FPGA-CPU-FPGA (i.e. estimation of $t(L0)$).

Test Setup: PC

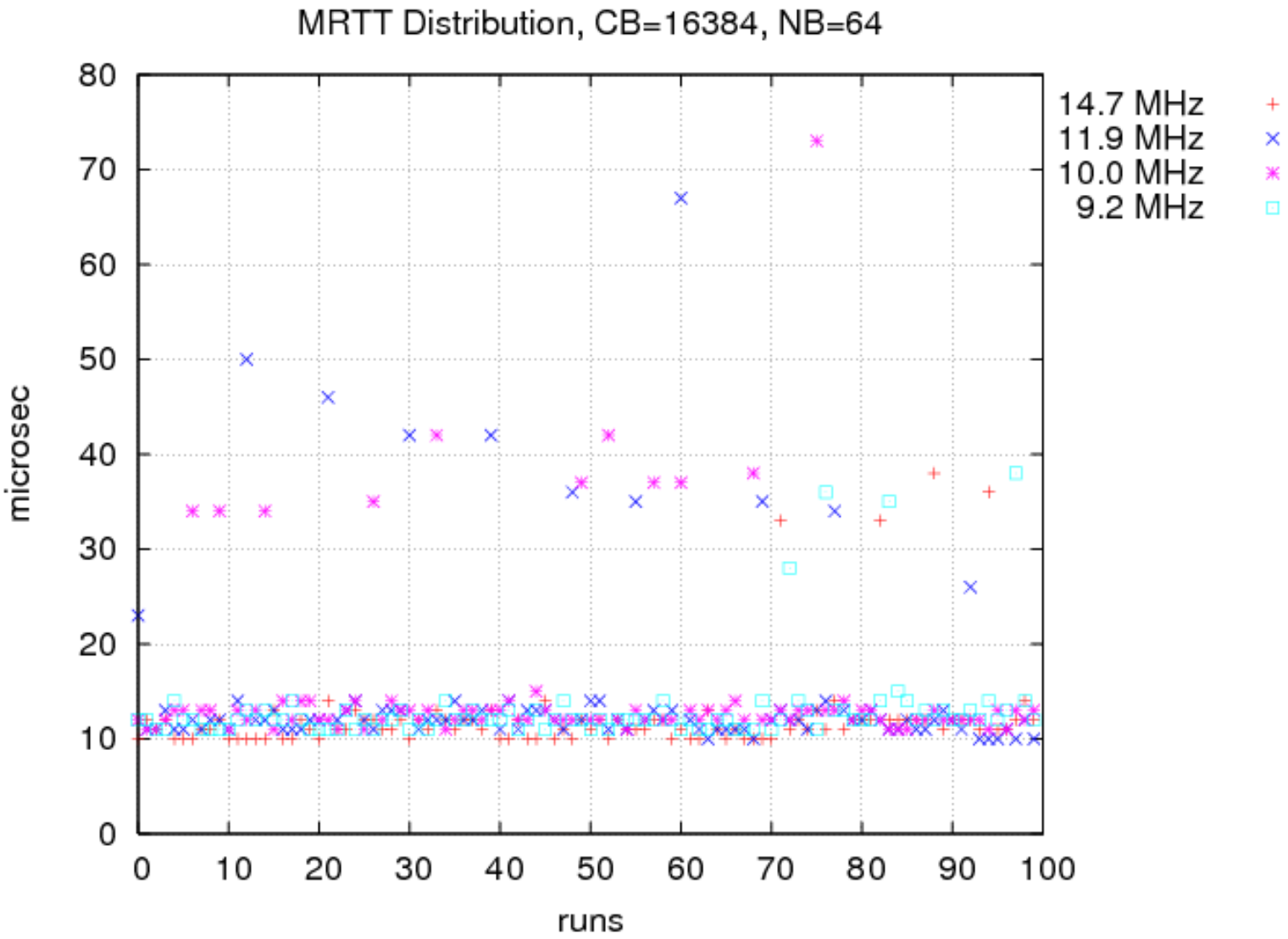
- Intel Core i7 930 2.80 GHz
- 8 MB L3-cache
- 4 core
- hyper threading disabled
- northbridge X58 (rev. 13)
- QPI @ 4.8 GT/s ($\approx$ 22.5 GB/s)
- memory 3 x 2 GB DDR3 1.067 GHz
- OS Fedora core 12, kernel 2.6.32.26-175.fc12.x86_64

Test Setup: FPGA

- devkit altera
- Stratix IV GX 230 (EP4SGX230N) -C2 (fast) speed-grade
- 1 PCI Express 8X IP core



Max RTT Distribution: 20 sec. runs



Max Round-Trip Time (MRTT)

Average/Maximum MRTT over 100 runs of 20 seconds each.

CB	NB	14.7 MHz	12.5 MHz	11.9 MHz	10.0 MHz	9.3 MHz
8192	32	16 / 313	17 / 380	11 / 118	11 / 74	9 / 12
	64	12 / 39	15 / 131	12 / 88	13 / 75	15 / 44
	128	17 / 45	17 / 82	16 / 80	18 / 63	18 / 42
	256	25 / 193	26 / 54	27 / 52	32 / 60	33 / 96
16384	32	13 / 365	18 / 232	12 / 182	12 / 76	12 / 71
	64	12 / 38	15 / 97	14 / 67	15 / 73	13 / 38
	128	22 / 199	19 / 92	16 / 42	18 / 84	21 / 47
	256	26 / 53	28 / 91	29 / 54	33 / 81	31 / 45
32768	64	20 / 275	17 / 150	14 / 78	19 / 74	18 / 76
	128	22 / 209	17 / 54	22 / 79	23 / 81	23 / 85
	256	29 / 209	28 / 54	31 / 90	34 / 93	38 / 90
65536	128	25 / 224	27 / 92	25 / 84	24 / 83	26 / 82
	256	33 / 200	33 / 89	35 / 91	35 / 93	38 / 92

NB and CB in unit of packets (64 Bytes), data in microsec.

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

the one million dollar question

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

Acknowledgments: I would like to thank

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

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

I would like to thank CINECA, INFN-CNAF and JSC for access to their systems.