CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction to HPC

HPC Summer School 2015

Claudio Gheller
cgheller@cscs.ch

School scope, objectives, program

The purpose of the school is to give you an introduction to the the main approaches to exploit modern High Performance Computing systems

- Understand the big picture, the ideas, the concept
- Understand the programming models
- Experiment specific solutions

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Introducing ourselves

- *Claudio Gheller* – School coordinator (everyday)
- *Ben Cumming* – OpenMP, CUDA (mon, tue, first week; mon, tue, second week)
- *Jean-Guillaume Piccinali* – Tools and practicals (first week)
- *Maxime Martinasso* – MPI (wed, thur, first week)
- *Jean Favre* – I/O and Visualization (fri, first week)
- *Markus Wetzstein* – OpenACC (wed, second week)
- *Radim Janalík* – Practicals (everyday)
- *Juraj Kardos* – Practicals (everyday)
- *Andreas Jocksch* - Practicals (second week)
- *Patrick Sanan* - Scientific Libraries (fri, second week)
- *Tatjana Ruefli* - Event manager (on call)

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

The Program, Week 1: MPI & OpenMP

	Monday July 20, 2015	**Tuesday July 21, 2015**	**Wednesday July 22, 2015**	**Thursday July 23, 2015**	**Friday July 24, 2015**
9:00 – 10:30	Introduction to HPC *Claudio Gheller*	OpenMP *Ben Cumming*	MPI *Maxime Martinasso*	MPI *Maxime Martinasso*	Efficient I/O and visualization *Jean Favre*
Break					
11:00 – 12:30	Access to CSCS systems *Jean-Guillaume Piccinali*	OpenMP *Ben Cumming*	MPI *Maxime Martinasso*	MPI *Maxime Martinasso*	Efficient I/O and visualization *Jean Favre*
Break					
14:00 – 15:30	Introduction to the Miniapp *Ben Cumming*	Miniapp (Adding OpenMP parallelism)	MPI *Maxime Martinasso*	Miniapp (adding MPI parallelism)	Miniapp (experiment with Viz)
Break		VISIT to CSCS Dinner at Ristorante *Il Cantinone*			
16:00 – 17:30	OpenMP *Ben Cumming*		MPI *Maxime Martinasso*	Miniapp (adding MPI parallelism)	Miniapp (experiment with Viz)
Goal	Familiarize with CSCS systems and Miniapp, and introduction to OpenMP	Introduction to OpenMP and application to Miniapp	Introduction to MPI	Approaching hybrid computing	Introduction to I/O and visualization

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

The Program, Week 2: GPU programming

	Monday **July 27, 2015**	**Tuesday** **July 28, 2015**	**Wednesday** **July 29, 2015**	**Thursday** **July 30, 2015**
9:00 – 10:30	Introduction to GPU architecture *Ben Cumming*	CUDA *Ben Cumming*	OpenACC *Markus Wetzstein*	Scientific libraries *Patrick Sanan*
Break				
11:00 – 12:30	CUDA *Ben Cumming*	CUDA *Ben Cumming*	OpenACC *Markus Wetzstein*	Scientific libraries *Patrick Sanan*
Break				Wrap-up of the accomplished work
14:00 – 15:30	CUDA *Ben Cumming*	Miniapp (Implementing Miniapp with CUDA)	Miniapp (Implementing Miniapp with OpenACC)	
Break				
16:00 – 17:30	Miniapp CUDA setup and simple kernels implementation	Miniapp (Implementing Miniapp with CUDA)	Miniapp (Implementing Miniapp with OpenACC)	
Goal	Introduction to GPUs and CUDA	Developing CUDA knowledge	Introduction to OpenACC	Introduction to scientific libraries and wrap up benchmarks and results for miniapp

Why HPC…

HPC helps with

- **Huge data**
  - Data management in memory
  - Data management on disk
- **Complex problems**
  - Time consuming algorithms for describing the behavior of a system
  - Machine learning
  - Data mining and analysis
  - Visualization

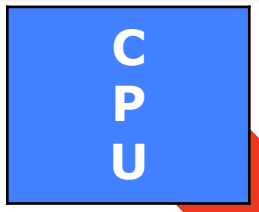
Requires

- **Special purpose solutions, in terms of**
  - Processors
  - Networks
  - Storage
  - Software
  - Applications

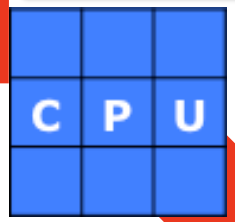
HPC in a nutshell
1: The Hardware

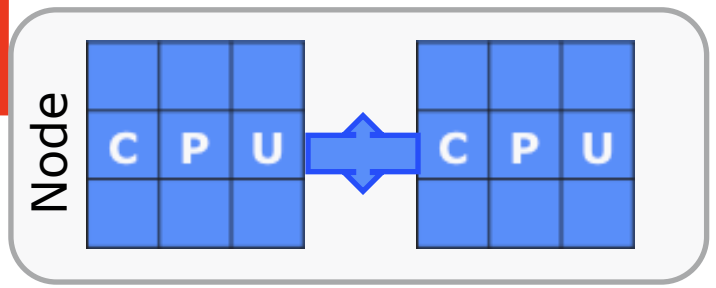
HPC systems: the building blocks

Single powerful processing unit



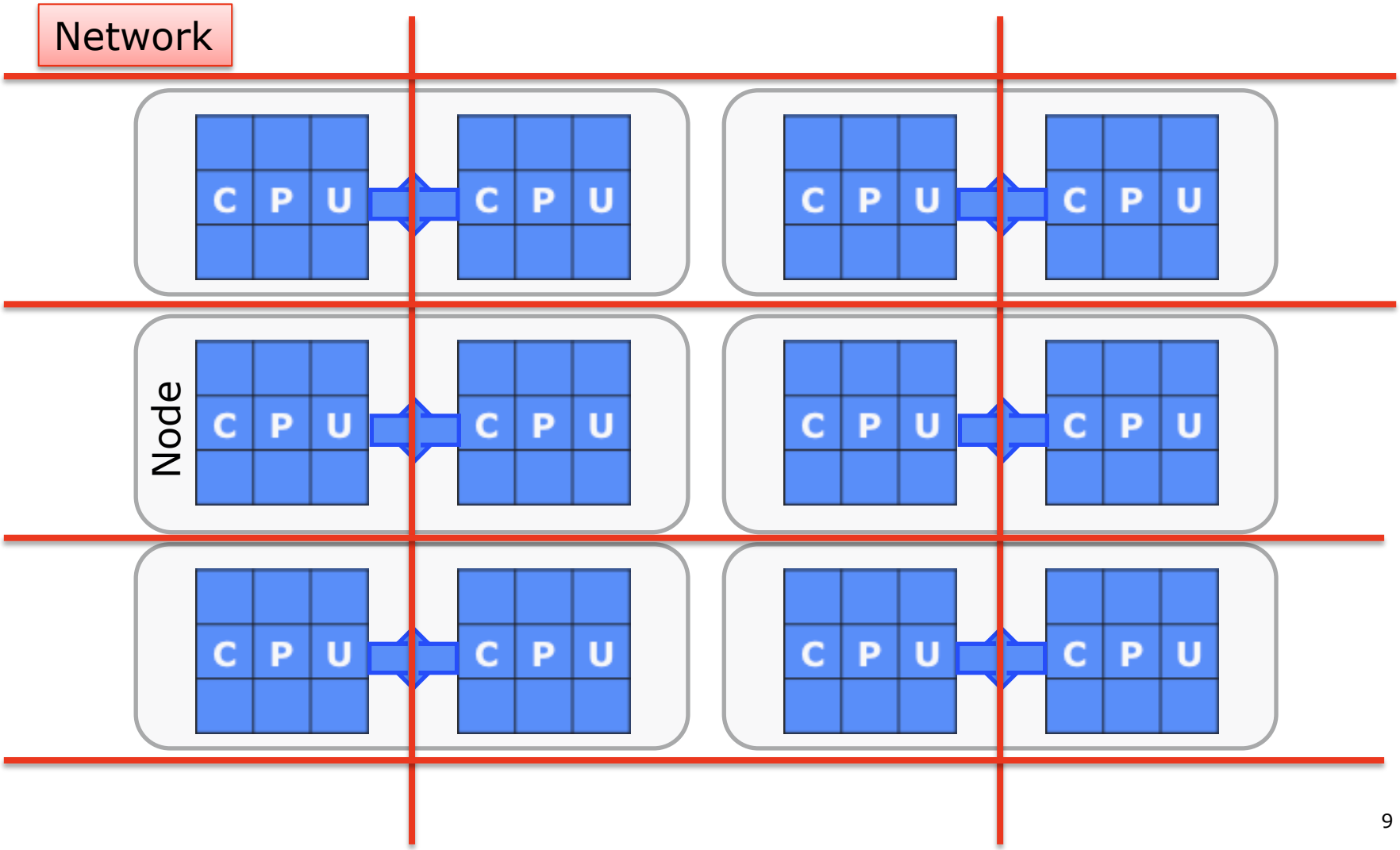
Multi-core, multi-threaded processor with shared memory



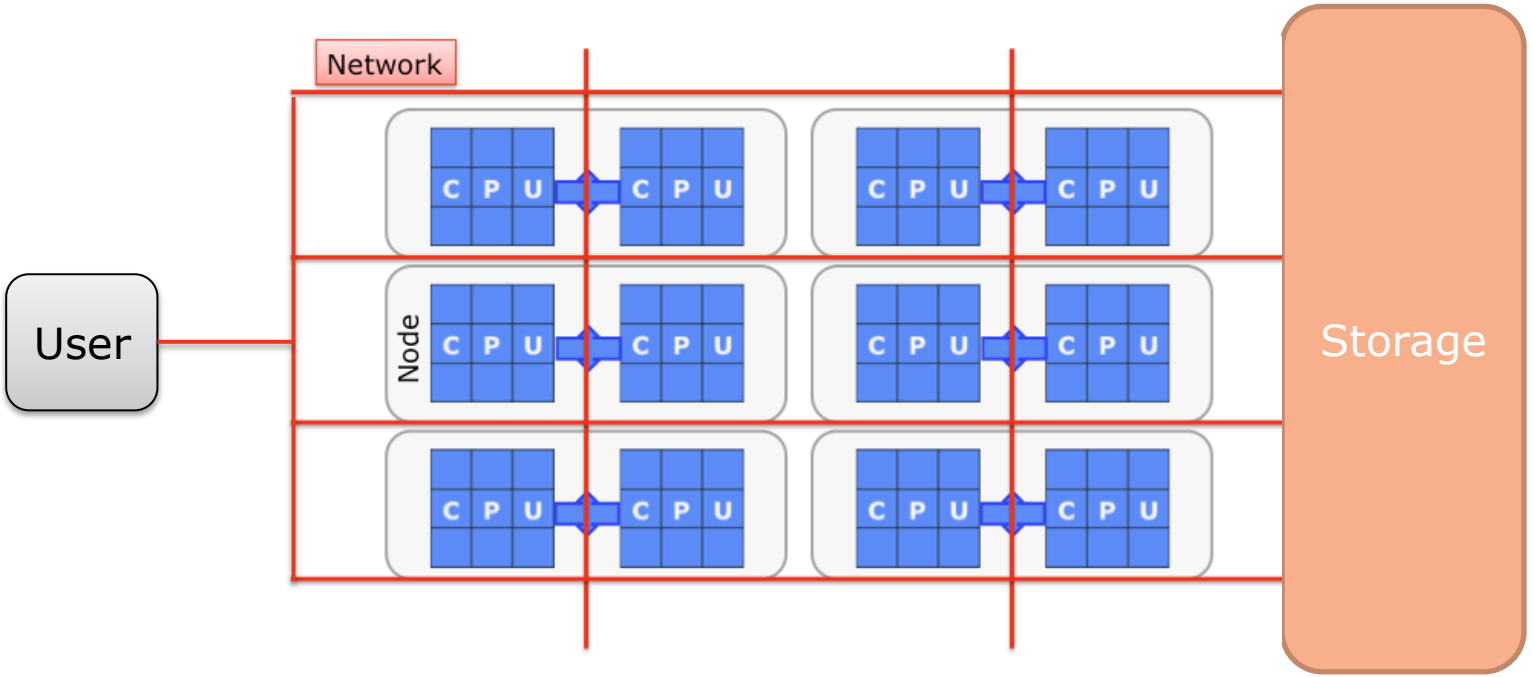


Combination of multi-core processors with NUMA shared memory and shared components

They need to communicate...



…and also to give feedbacks…



Two problems with this model

1. Systems are becoming too expensive

- **Power consumption**
- Cooling and infrastructural costs
- Costs of technology

2. It is hard to increase the performance

- **Speed-up the processor**
- Increase network bandwidth
- Reduce latencies
- Improve connectivity
- Make I/O faster
- …

HPC systems power consumption

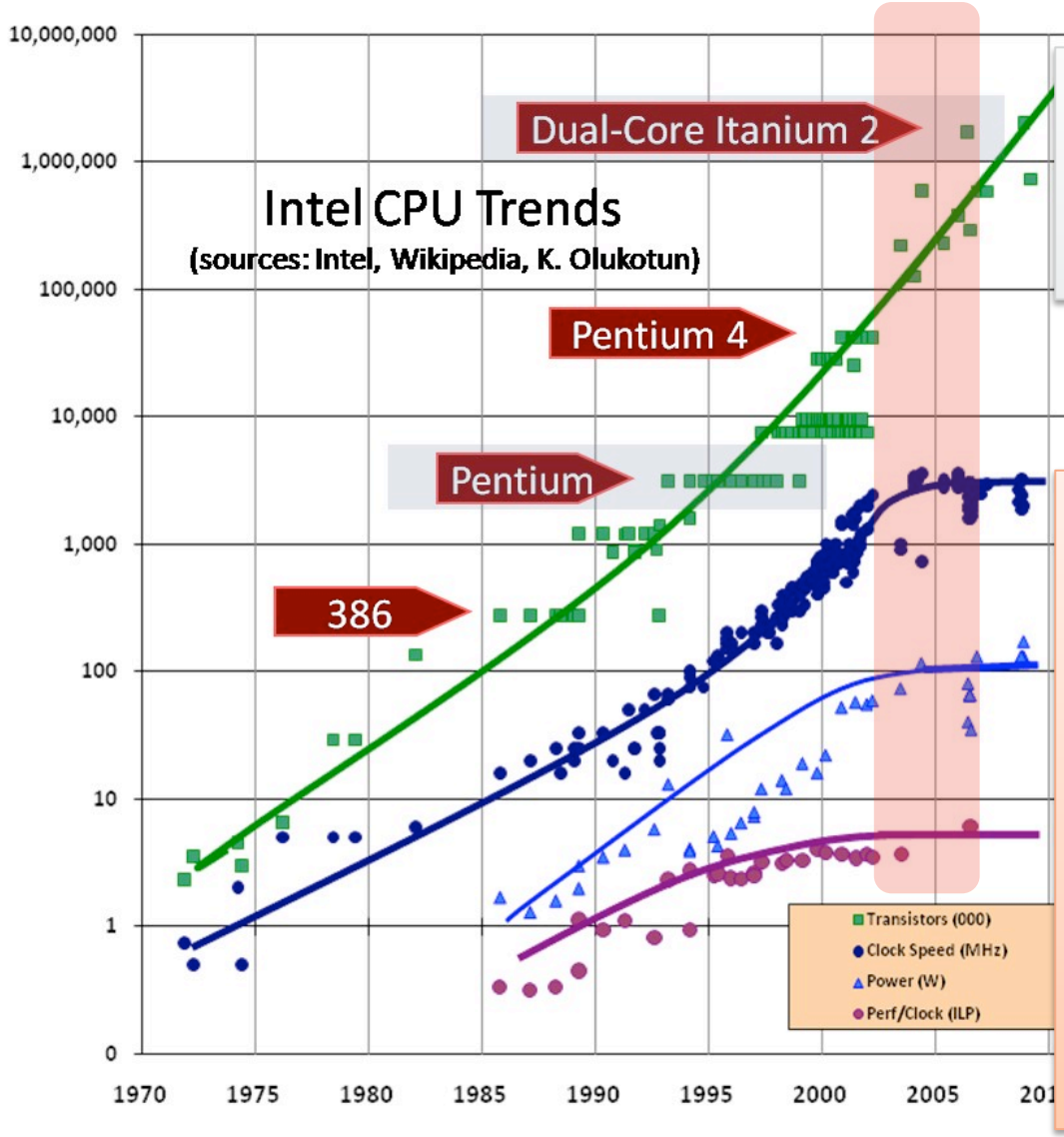
Coming HPC systems are anticipated to draw enormous amounts of electrical power…

Example: N.1 Top 500

Year (Nov.)	System	Performance (Tflop/s)	Electrical power (KW)
2002	Hearth Simulator	41	3200
2005	Blue Gene/L	367	1433
2008	Roadrunner	1105	2483
2009	Jaguar	2331	6950
2011	K computer	11280	12659
2012	Titan	27112	8209
2014	Tianhe-2	54902	17808
→2020	????????	1000000	>100000 (100 MW!!!)

THIS IS NOT SUSTAINABLE!!!

Processor performance: The end of the "Free Performance Lunch" era



Moore's Law does not help anymore in increasing the performance

Clock speed cannot increase more:

1. heat (too much of it and too hard to dissipate)
2. current leakage
3. power consumption (too high – also memory must be considered!!!)

Moore's law

- **Processor performance doubles every 12/18/24 months**
  - Still true for the number of transistor
  - Not true for clock speed (W ~ f^3 + hardware failures)
  - Not true for memory and storage

- **But, pocessor's performance depends mostly on the frequency, less from the number of transistors!**

- **...what about using the "spare" transistors in a different way?**

Improving performance: multi-cores

Adding more and more cores can increase the computing power without having to increase the clock speed. However:

- **Hardware strongly limits the scalability**
  - Remote shared memory access
  - Cache coherency
  - Shared hardware components
  - …
- **Applications are not parallel or not fully parallel (Amhdal's law)**

The typical number of cores is between 8 and 16, the actual applications' scalability is often lower

Improving further: many-cores

Progressively, a new generation of processors with less general, more specialized architecture, capable of superior performance on a narrower range of problems, is growing:

Many-cores accelerators:

- **Nvidia Graphics Processor Units (GPU)**
- **Intel XeonPHI Many Integrated Cores (MIC) architecture**
- **AMD Accelerated Processing Units (APU)**

They all support a massively parallel computing paradigm.

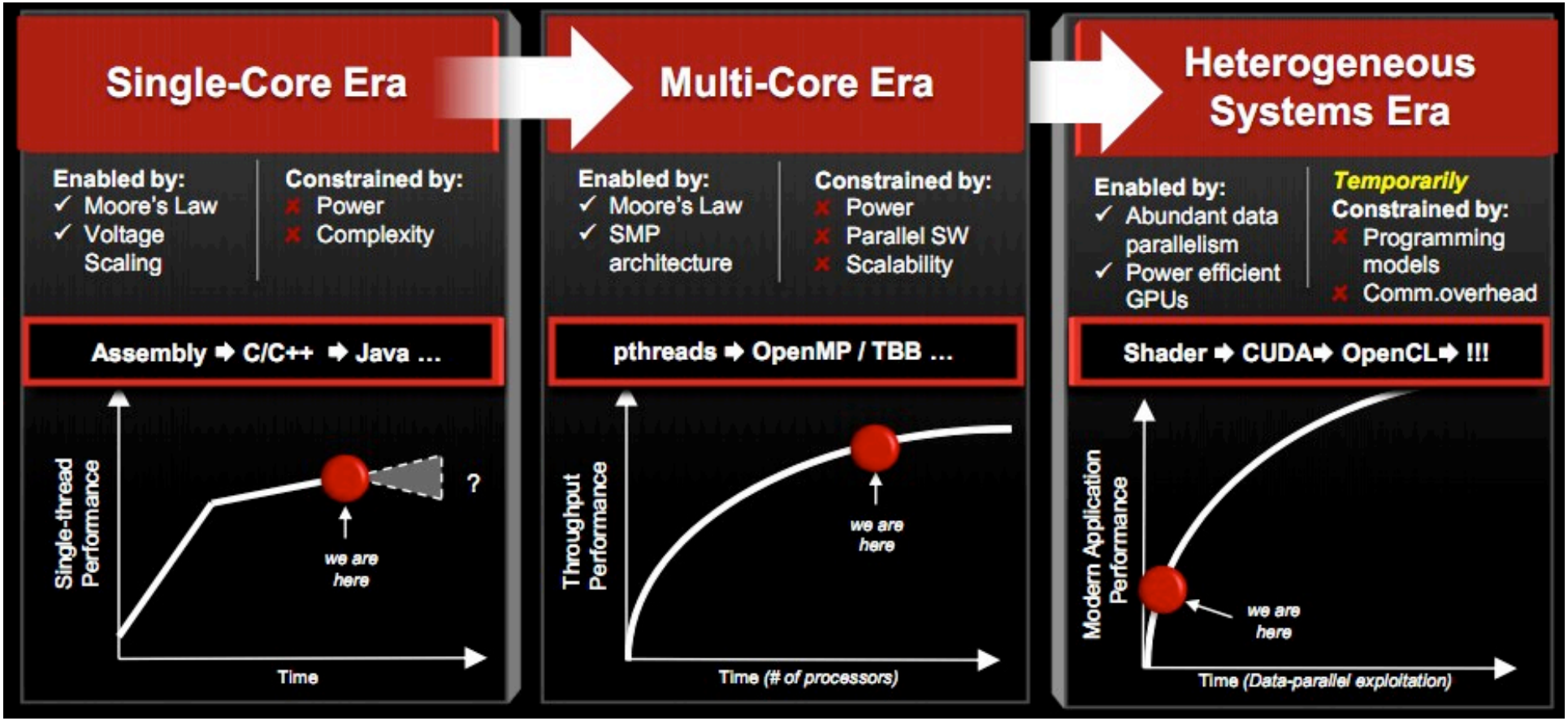
Why accelerators are more efficient than CPUs

A few main reasons:

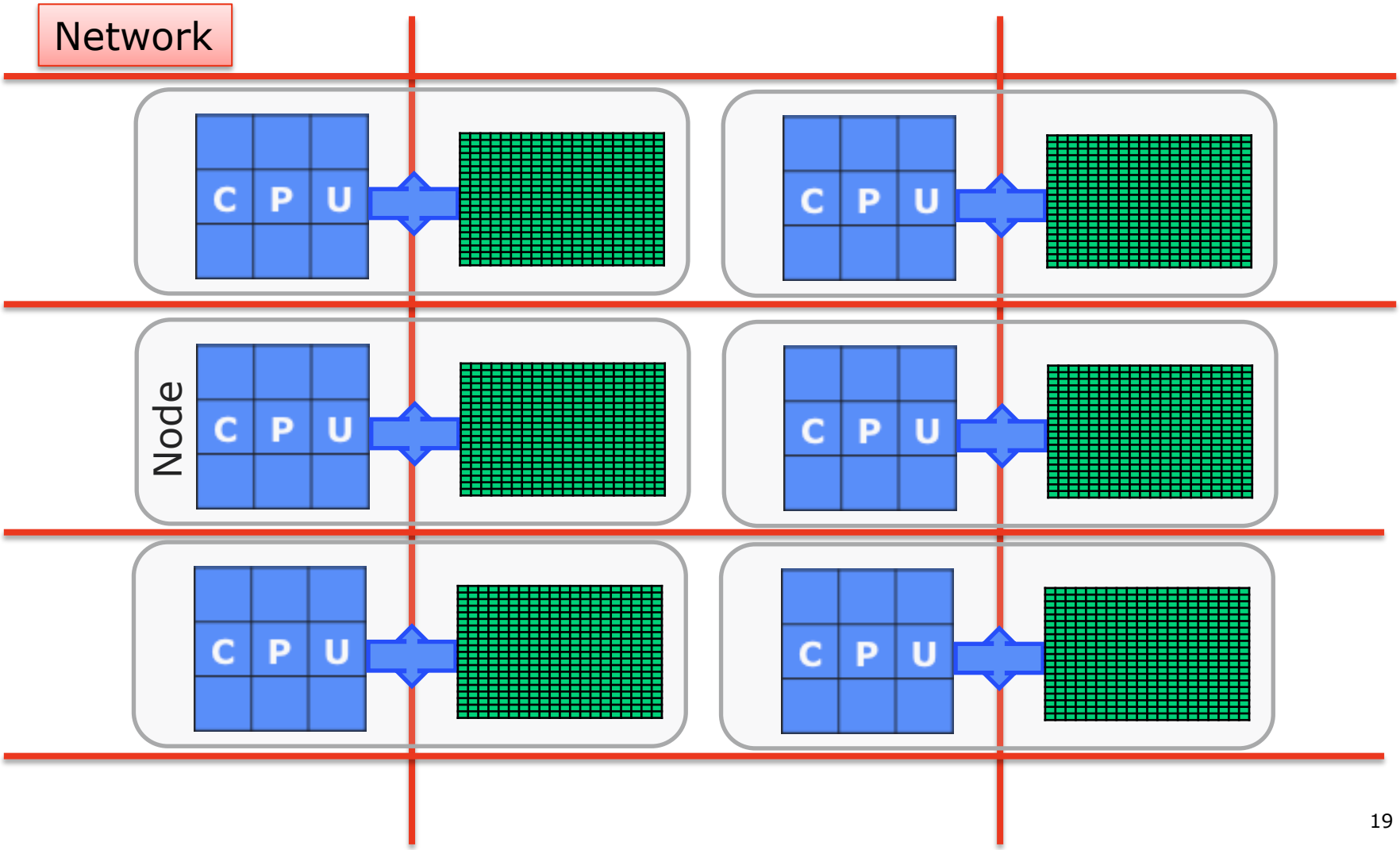
- **lower frequencies of GPU clocks**
- **more of the transistors in a gpu are actually working on the computation. CPUs are more general purpose. They require energy to support such "flexibility"**
- **Single Instruction Multiple Data (SIMD) approach, which leads to a simpler architecture and instruction set**

Summing up

Heterogeneous systems are the next future. Right now, they represent the only viable solution for efficient high performance computing



Putting all together... the current HPC systems



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Piz Daint

"Piz Daint" CRAY XC30 system @ CSCS (N.6 in Top500)

Nodes:

5272 CPUs 8-core Intel SandyBridge equipped with:

- 32 GB DDR3 memory
- One **NVIDIA Tesla K20X GPU** with 6 GB of GDDR5 memory

Overall system

- 42176 cores and 5272 GPUs
- 170+32 TB
- Interconnect: Aries routing and communications ASIC, and dragonfly network topology
- Peak performance: 7.787 Petaflops

- **User Lab:**

http://www.cscs.ch/user_lab/becoming_a_user/index.html

Efficiency comparison

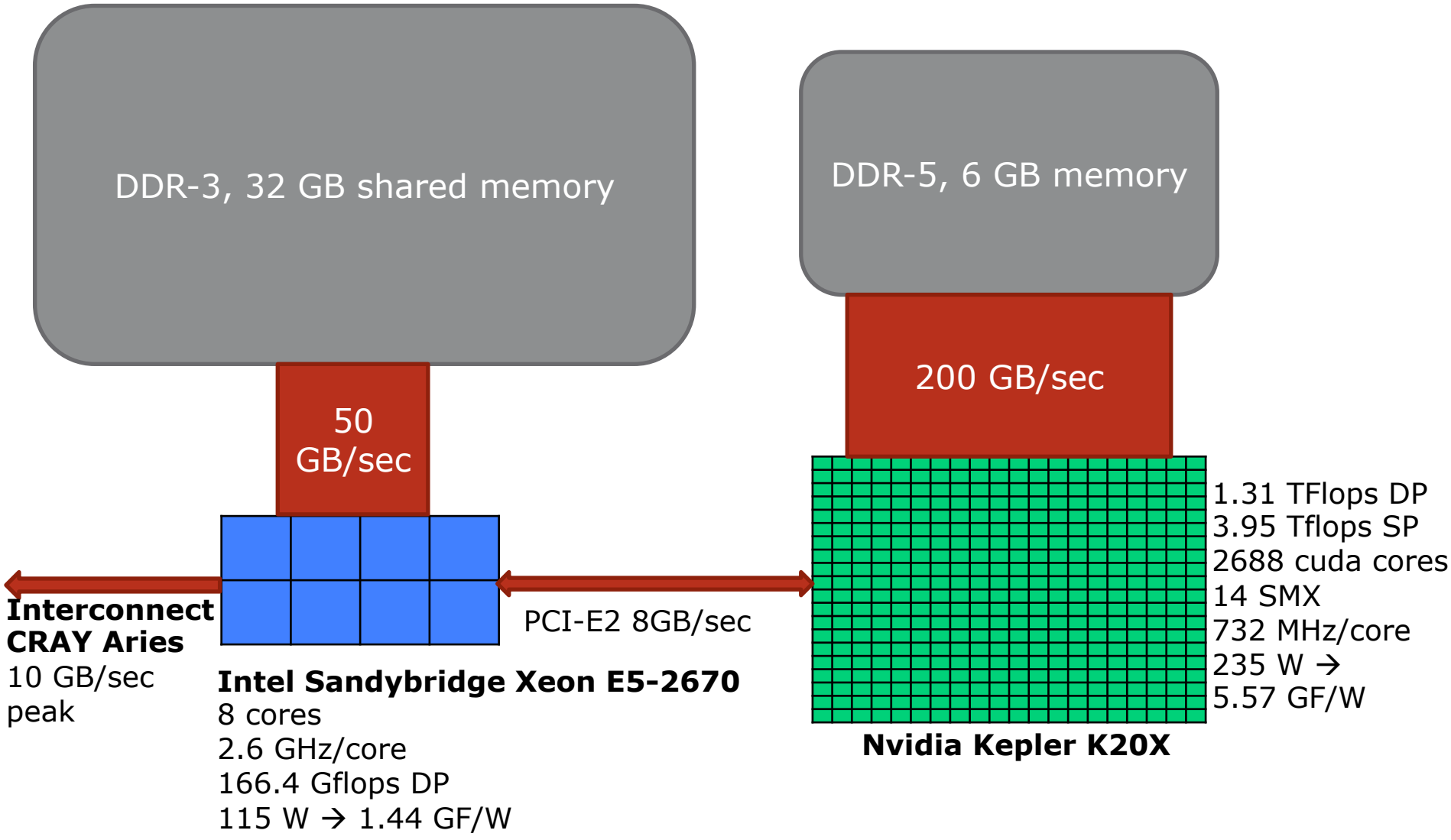
Data from last Top500 (November 2014)

System	RPEAK (TFlop/sec)	Power (MW)	Efficiency (MF/sec/W)
Piz Daint (GPU - 6)	7788	2.325	3349
Thiane-2 (MIC - 1)	54902	17.808	3083
Titan (GPU - 2)	27112	8.209	3302
Sequoia (BGQ - 3)	20132	7.890	2551
K computer (Sparc - 4)	11280	12.66	891
Pleiades (SGI ICE X - 11)	3988	3.141	1270

Nvidia K20X GPU architecture

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

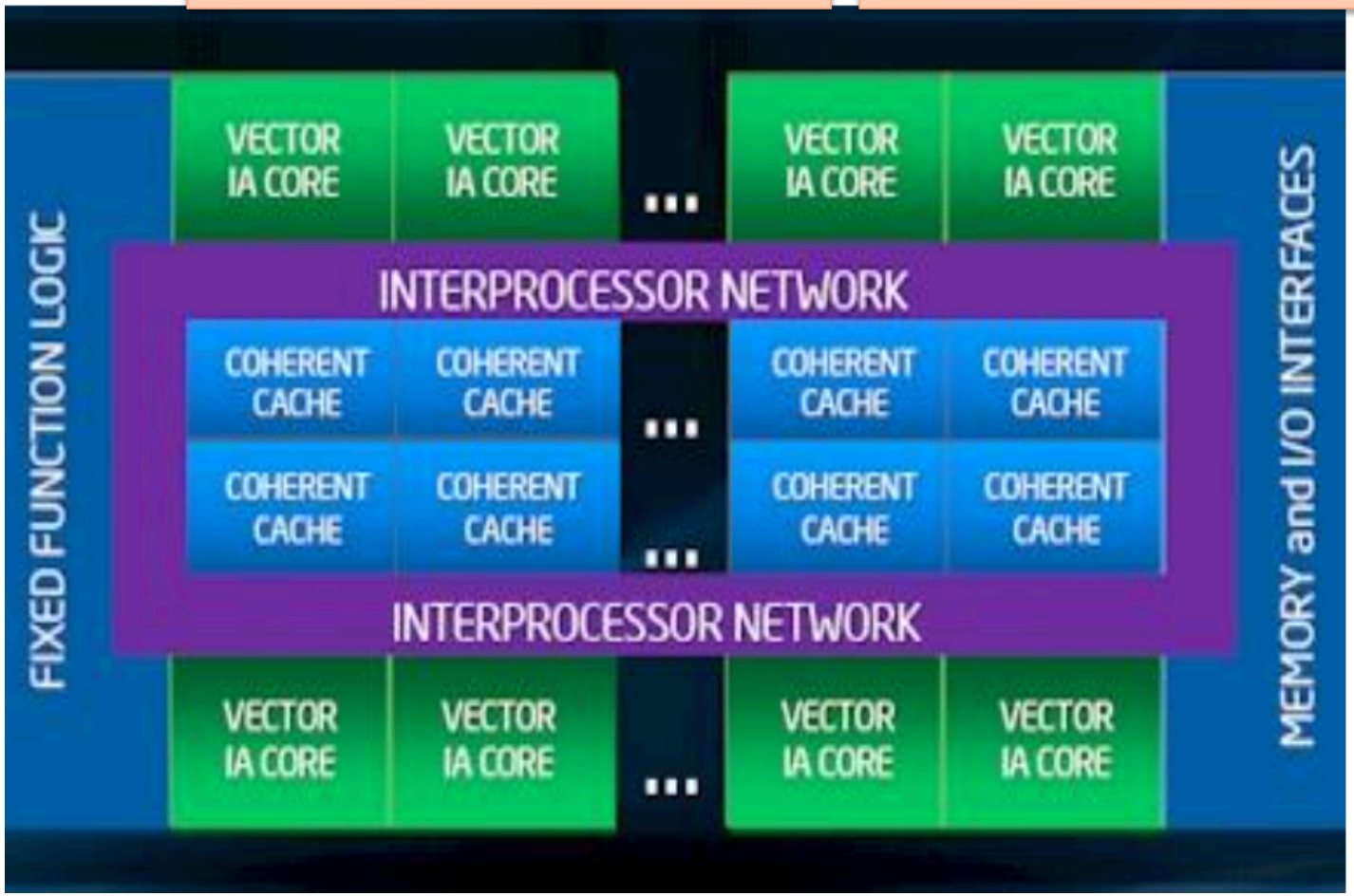
Processor architecture (Piz Daint)



MIC architecture

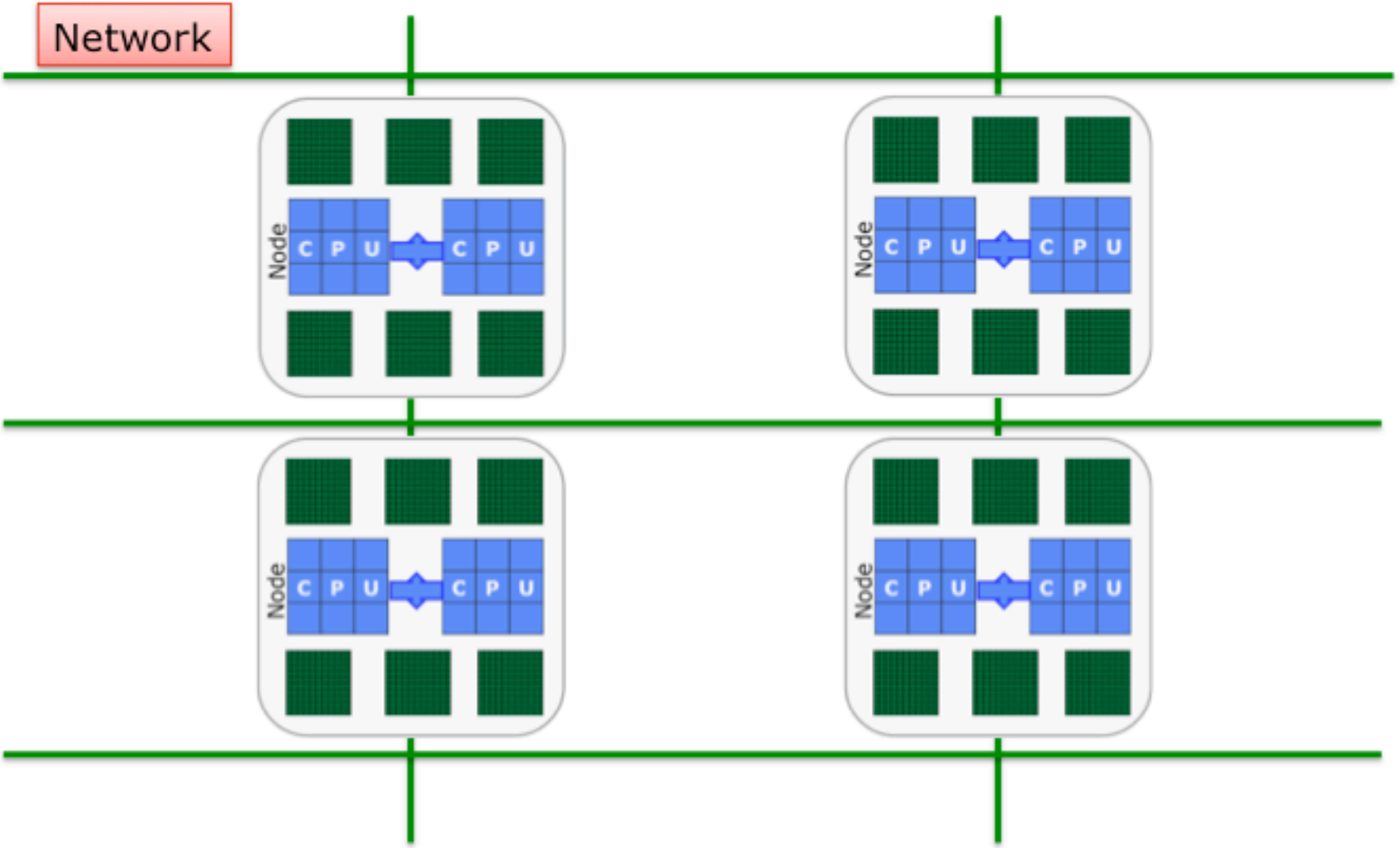
Based on Larrabee (2006)
512 bit vector units
Cache coherent

~60 cores
4-way multi-threading
X86 instruction set



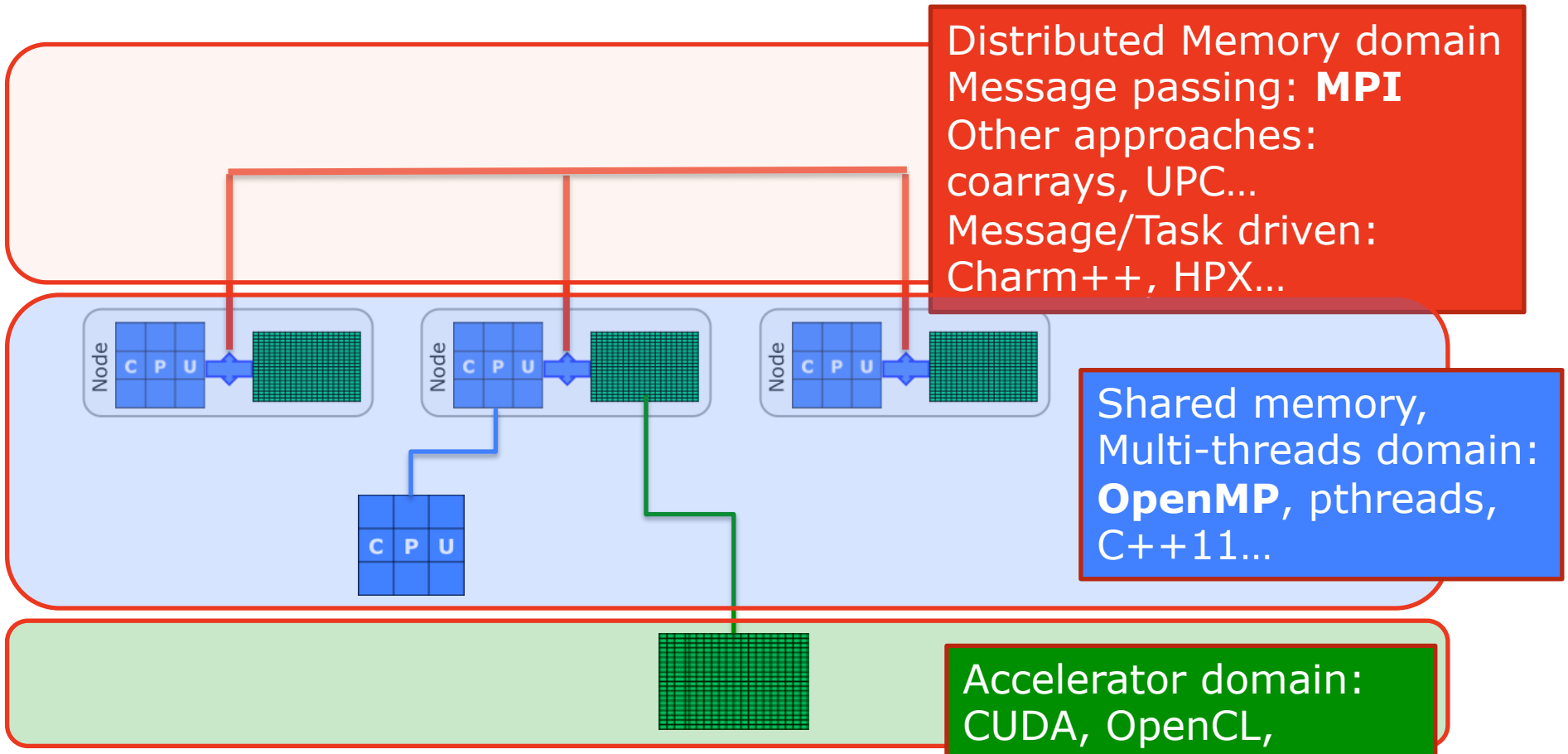
- Many cores on the die
- L1 and L2 cache
- Bidirectional ring network for L2
- Memory and PCIe connection

The systems of the next future



HPC in a nutshell
2. Programming Models

Overall picture

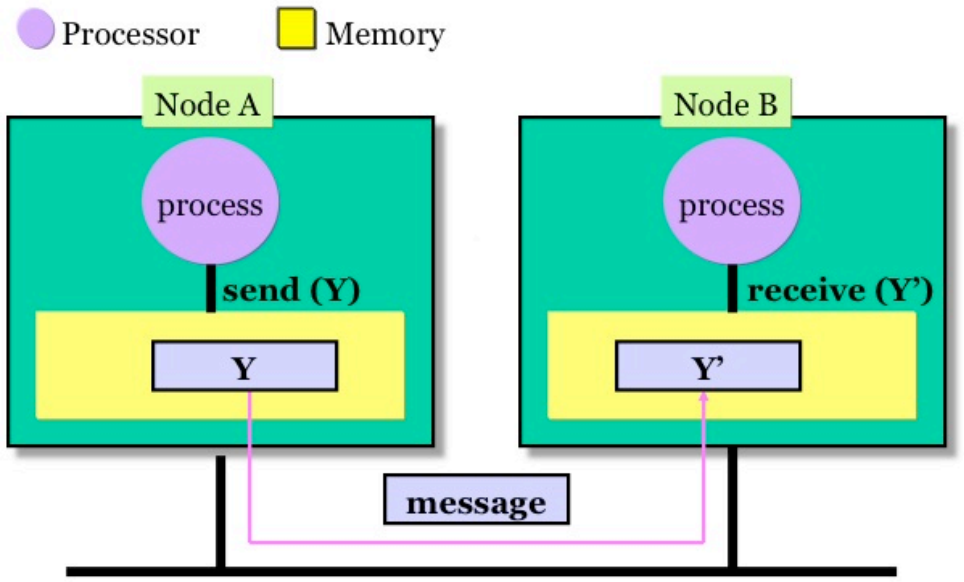


Message Passing Paradigm

- **Resources are LOCAL (differently from shared memory model)**
- **Each process runs in a "isolated" environment. Interactions requires Messages Exchange**
- **Messages can be: instructions, data, synchronization**
- **Message Passing works also in a Shared Memory system**
- **Time to exchange messages is much larger than accessing local memory**

Message Passing is a COOPERATIVE approach, based on 3 basic operations:

- **SEND (a message)**
- **RECEIVE (a message)**
- **SYNCRONIZE**



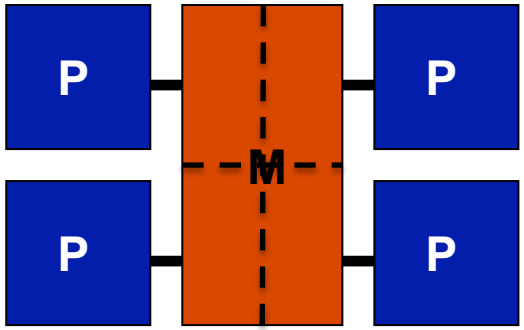
Why (and why not...) Message Passing

- **Advantages**
  - Communications is the most important part of high-performance parallel computing: it can be highly optimized
  - Message-passing paradigm is portable
  - Many current applications/libraries use message-passing.
  - (Message passing can be used for distributed processing.)
- **Drawbacks**
  - Explicit nature of message-passing is error-prone. . .
  - and discourages frequent communications. . .
  - and hard to do MIMD (task parallel) programming. . .

Message Passing Interface - MPI

- **MPI is standard defined in a set of documents compiled by a consortium of organizations: http://www.mpi-forum.org/**
- **In particular the MPI documents define the APIs (application interfaces) for C, C++, FORTRAN77 and FORTRAN 90.**
- **Bindings available for Perl, Python, Java…**
- **The actual implementation of the standard is demanded to the software developers of the different systems**
- **In all systems MPI is implemented as a library of subroutines/functions over the network drivers and primitives**

Shared memory systems



- **Processors can access the same GLOBAL memory**
- **Uniform Memory Access (UMA) model <=> SMP: Symmetric Multi Processors**
  - Access time to memory is uniform
  - Local cache, all other peripherals are shared
- **Non Uniform Memory Access (NUMA) model**
  - Memory is physically distributed among processors
  - Global virtual address space accessible from all processors
  - HW support for remote access
  - Access time to local and remote data is different

- **OpenMP standard "de facto", but other solutions available (and effective)**

OpenMP

- **Implemented as a set of directives to orchestrate the parallel work**
- **All variables are "Shared"**
- **Dynamic parallelism: tasks are created (fork) and destroyed (join) during the program execution**

```
program myprogram
real a,b, ....

a = 0.0
...(1)...
!$omp parallel do ...
do i=1,n
  ...(2)...
enddo
!$omp end parallel do
...(3)...
```

(1) Master executing serially; Slaves sleeping

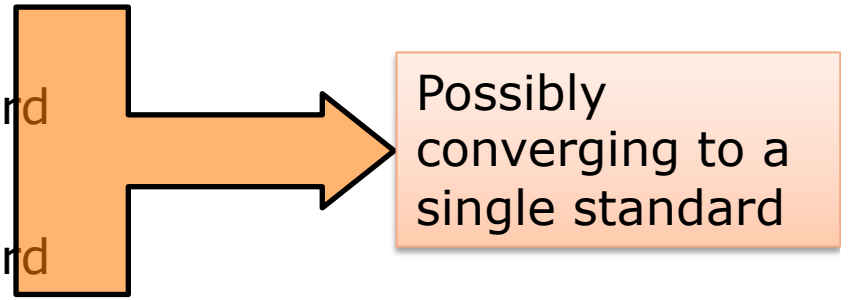
(2) FORK – Master + Slaves executing the loop in parallel

(3) Again, serial execution; Slaves sleeping; Master access shared variables

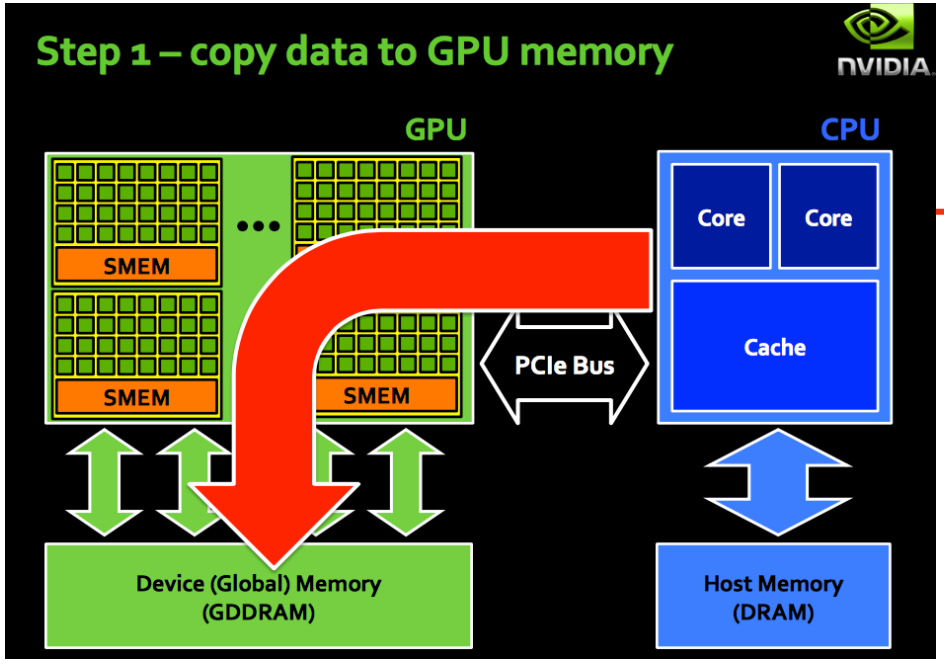
GPU programming

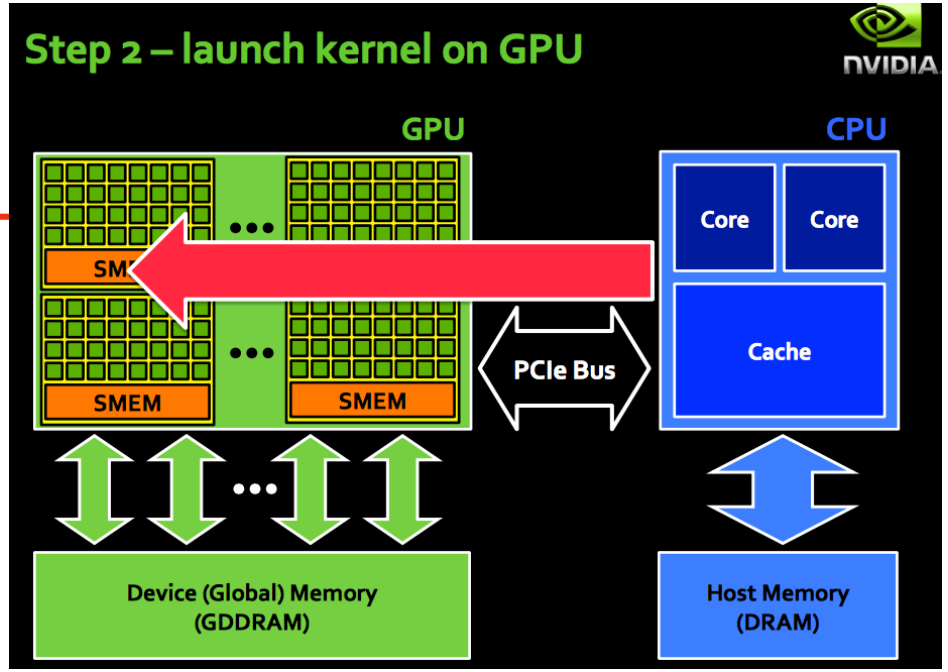
API currently available

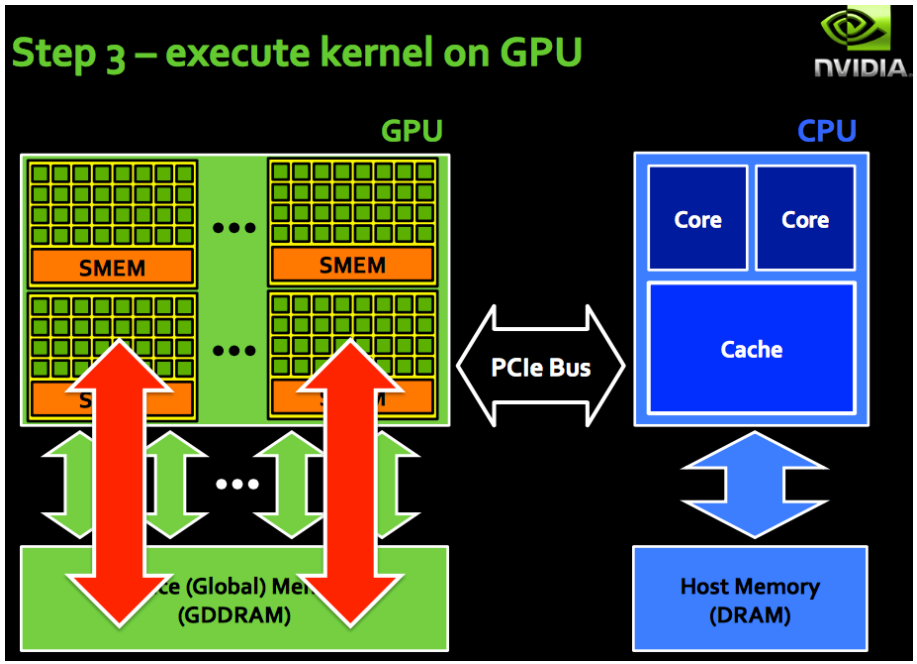
- **CUDA**
  - NVIDIA product, best performance, low portability
- **OpenCL**
  - Open standard, API similar to CUDA, high portability
- **OpenACC**
  - Directive based open standard
- **OpenMP**
  - Directive based open standard
- **Libraries**
  - MAGMA, cuFFT, Thrust, CULA, cuBLAS, cuSOLVER…
- **C++11**
  - High level abstraction

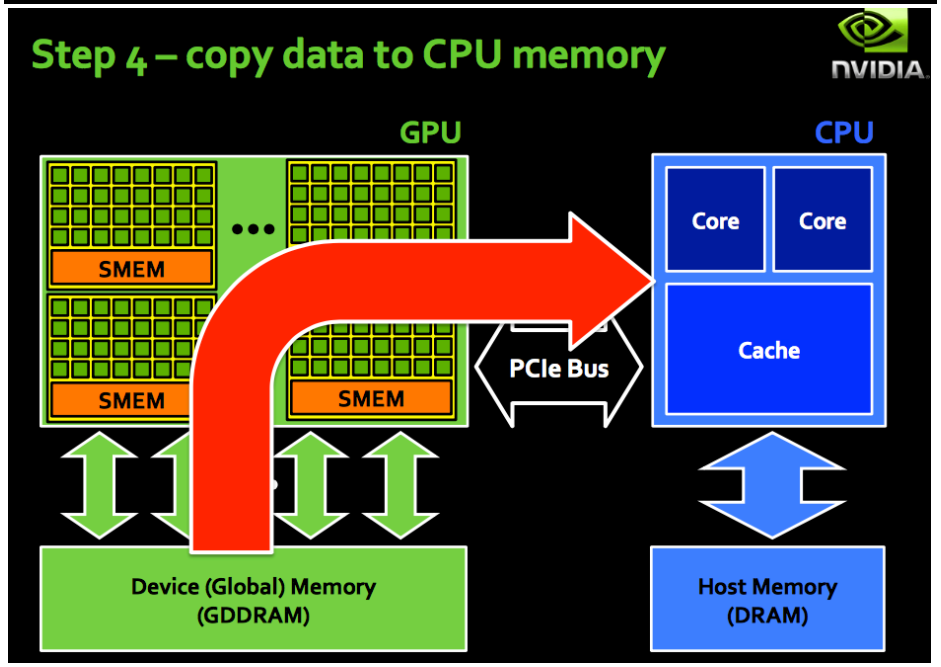


Supported languages: C, C++, Fortran but also Java, Python (not from all the APIs)


Step 1 – copy data to GPU memory
NVIDIA
GPU
CPU
SMEM
SMEM
SMEM
PCIe Bus
Core
Core
Cache
Device (Global) Memory
(GDDRAM)
Host Memory
(DRAM)


Step 2 – launch kernel on GPU
NVIDIA
GPU
CPU
SMEM
SMEM
PCIe Bus
Core
Core
Cache
Device (Global) Memory
(GDDRAM)
Host Memory
(DRAM)


Step 3 – execute kernel on GPU
NVIDIA
GPU
CPU
SMEM
SMEM
PCIe Bus
Core
Core
Cache
(GDDRAM)
Host Memory
(DRAM)


Step 4 – copy data to CPU memory
NVIDIA
GPU
CPU
SMEM
SMEM
PCIe Bus
Core
Core
Cache
Device (Global) Memory
(GDDRAM)
Host Memory
(DRAM)

MIC programming

- **Leverage x86 architecture (CPU with many cores)**
  - x86 cores that are simpler, but allow for more compute throughput
- **Leverage existing x86 programming models**
  - In particular it supports MPI and OpenMP (with proper extensions)
- **MIC can be used**
  - In "Native" mode, compiling and running MPI/OpenMP programs directly on the accelerator
  - In "Offload" mode, offloading, computational demanding, massively parallel parts of the code to the coprocessor, through OpenMP extensions. The hosts acts as a driver.
- **Specific libraries for optimization are available and MUST be used**

WARNING:

- **Time to run on MIC = Time to compile (Native mode)**
- **Time to run FAST on MIC ~ Time to run FAST on a GPU**

Putting all together

- **Current supercomputers are hybrid**
- **Different levels of nested parallelism:**
  - Improve performance
  - Improve scalability
- **Each level should be exploited using the most proper approaches and tools**

- **NOW……. Let's start working on it !!!**