

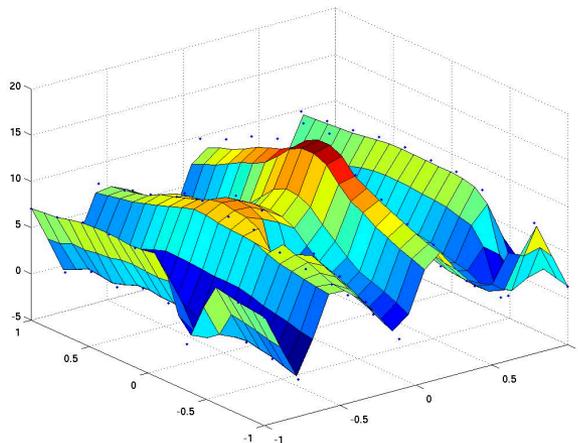
# High Performance Computing

*Synopsis of Technical and Programming Concepts*



**Claude TADONKI**

**MINES ParisTech – PSL Research University**  
**Paris - France**



# INTEL BROADWELL

Intel® Xeon® Processor E5-2699 v4  
Released in April 2016

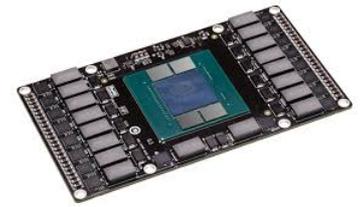
- 22x2 = 44 cores
- 2.2 Ghz/core
- 3.6 GHz Boost
- Hyperthreading
- 256-bit vectors
- 256 Gb RAM
- 76.8 Gb/s
- 500 Gb disk
- 1.54 Tflops SP
- 0.78 Tflops DP

<b>Hardware</b>	
CPU Name:	Intel Xeon E5-2699 v4
CPU Characteristics:	Intel Turbo Boost Technology up to 3.60 GHz
CPU MHz:	2200
FPU:	Integrated
CPU(s) enabled:	44 cores, 2 chips, 22 cores/chip, 2 threads/core
CPU(s) orderable:	1,2 chip
Primary Cache:	32 KB I + 32 KB D on chip per core
Secondary Cache:	256 KB I+D on chip per core
L3 Cache:	55 MB I+D on chip per chip
Other Cache:	None
Memory:	256 GB (16 x 16 GB 2Rx4 PC4-2400T)
Disk Subsystem:	1 x SATA, 500 GB, 7200 RPM
Other Hardware:	None

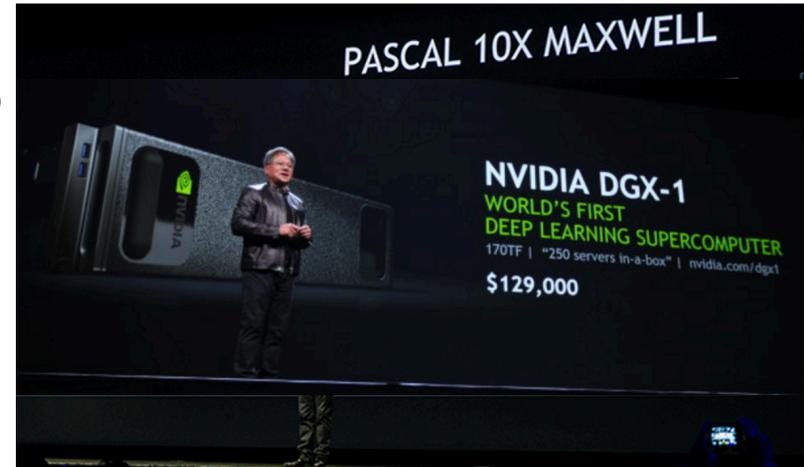
- **Tflops** is 1000 000 000 000 (1 billion) floating point operations per seconds

# NVIDIA DGX-1

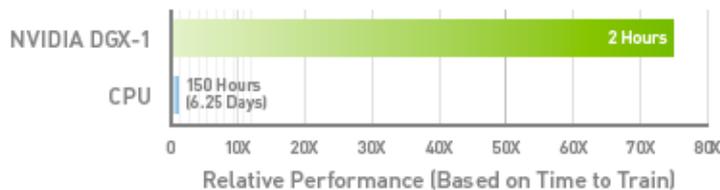
\$129,000 US  
Released in April 2016



- NVIDIA supercomputing solution
- **8 Tesla P100 GPUs** (Pascal GPU based)
- Dual **Intel Xeon** processors (host)
- **170 Tflops FP16** peak perf
- **7 Tb** of SSD Storage
- Aggregate bandwidth **768 Gb/s**
- Perf throughput **250 x86 servers**
- **Pascal GPU**: 3584 CUDA Cores; 1480 MHz; 16 GB RAM at 720 Gb/s 5thGen
- We should understand that GPU is specialized for specific tasks where it is likely to show up noticeable performances

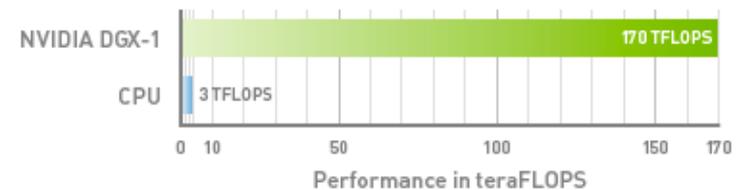


## NVIDIA DGX-1 Delivers 75X Faster Training



CPU is dual socket Intel Xeon E5-2697 v3

## NVIDIA DGX-1 Delivers 56X More Performance



CPU is dual socket Intel Xeon E5-2697 v3

# N°1 SUPERCOMPUTER

Top500 - Nov 2015

## TIANHE-2 (MILKYWAY-2)

- In China
- Intel Xeon E5
- 260, 000 nodes
- 3 million cores
- 54 PFlops peak
- 33 PFlops (61%)

Site:

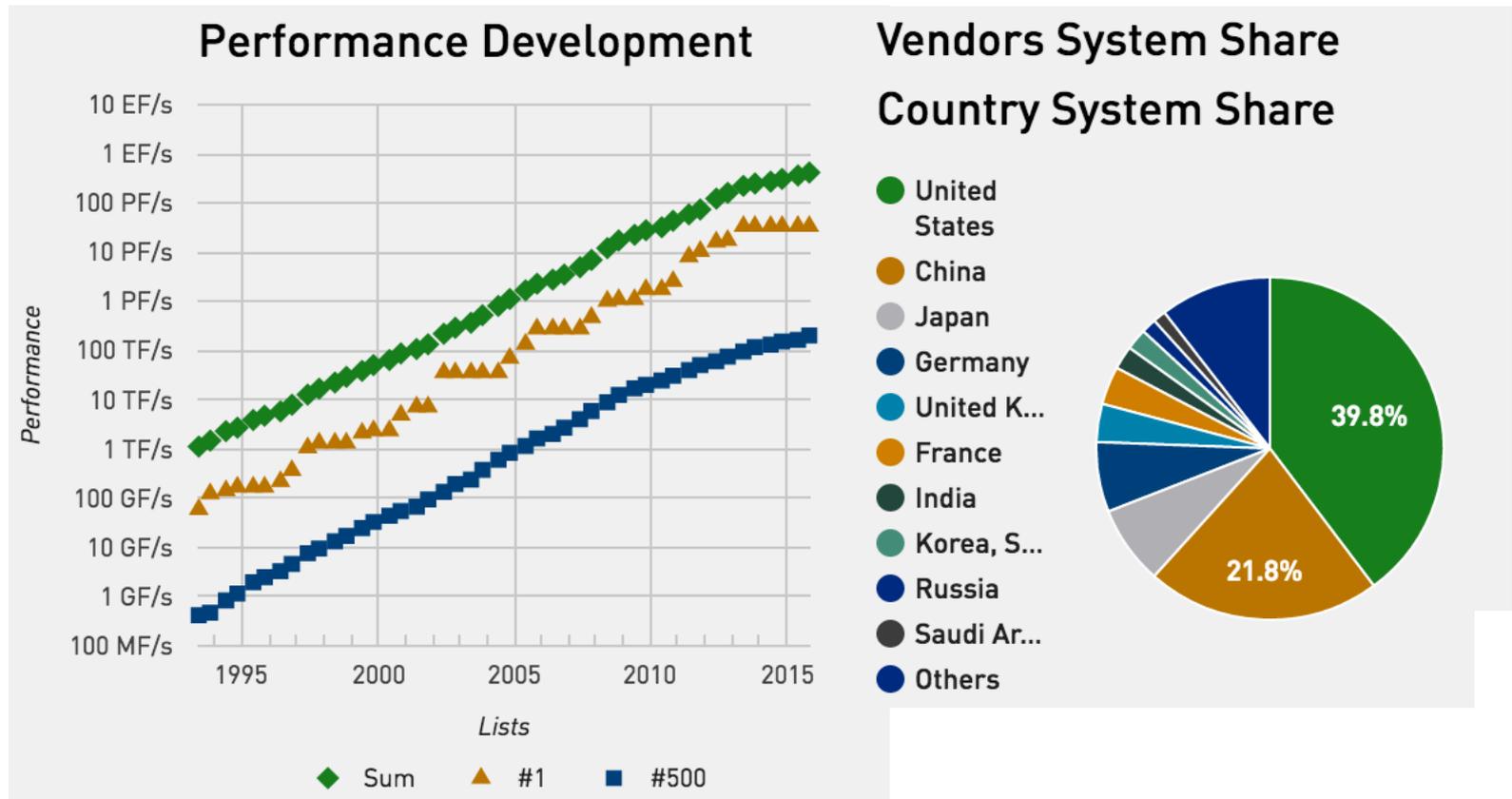
National Super Computer Center in Guangzhou



MPI:

MPICH2 with a customized GLEX channel

# Performances Evolution



- We are moving toward **ExaFlops** ( $E = Exa = 10^{18}$ )

# TOP500

Top 5 sites - Top500 - Nov 2015

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	<b>Tianhe-2 (MilkyWay-2)</b> - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	<b>Titan</b> - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	<b>Sequoia</b> - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	<b>Mira</b> - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

# Peak Performance Evaluation

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808

## Getting Tianhe-2 RPEAK:

- CPU-core frequency: 2.2 Ghz = 2.2 GFlops
- Considering the vector capability (256-bit wide - 4 DP):  $4 \times 2.2 = 8.8$  GFlops
- Given the CPU can do ADD and MUL in one cycle (FMA):  $2 \times 8.8 = 17.6$  GFlops
- Finally the total number of cpu-cores:  $3,120,000 \times 17.6$  Ghz = 54.912 PFlops

***Clearly, we should exploit all levels of parallelism, if we need to harvest an acceptable fraction of the peak performance.***

# Peak vs Sustained

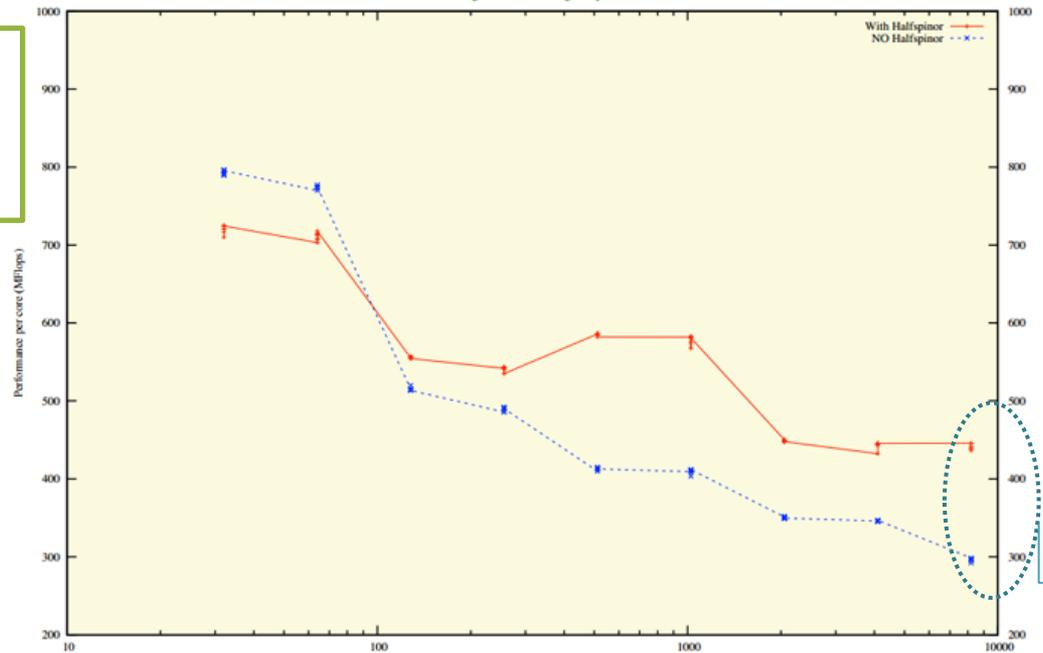
Not counted in peak performance:

- Memory accesses
- Interprocessor communications

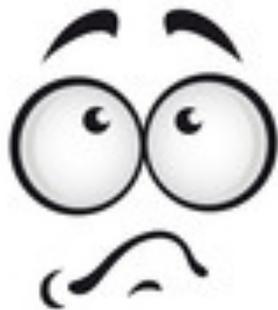
## Curie Fat performance (weak scaling)

Page 500: Version 7.0 G.Grosdidier 0703/11 - TGCC-Curie (Last Update Time-stamp: <11/03/07 15:08:22 grosdid>)

Fig 501: Curie Scaling Study



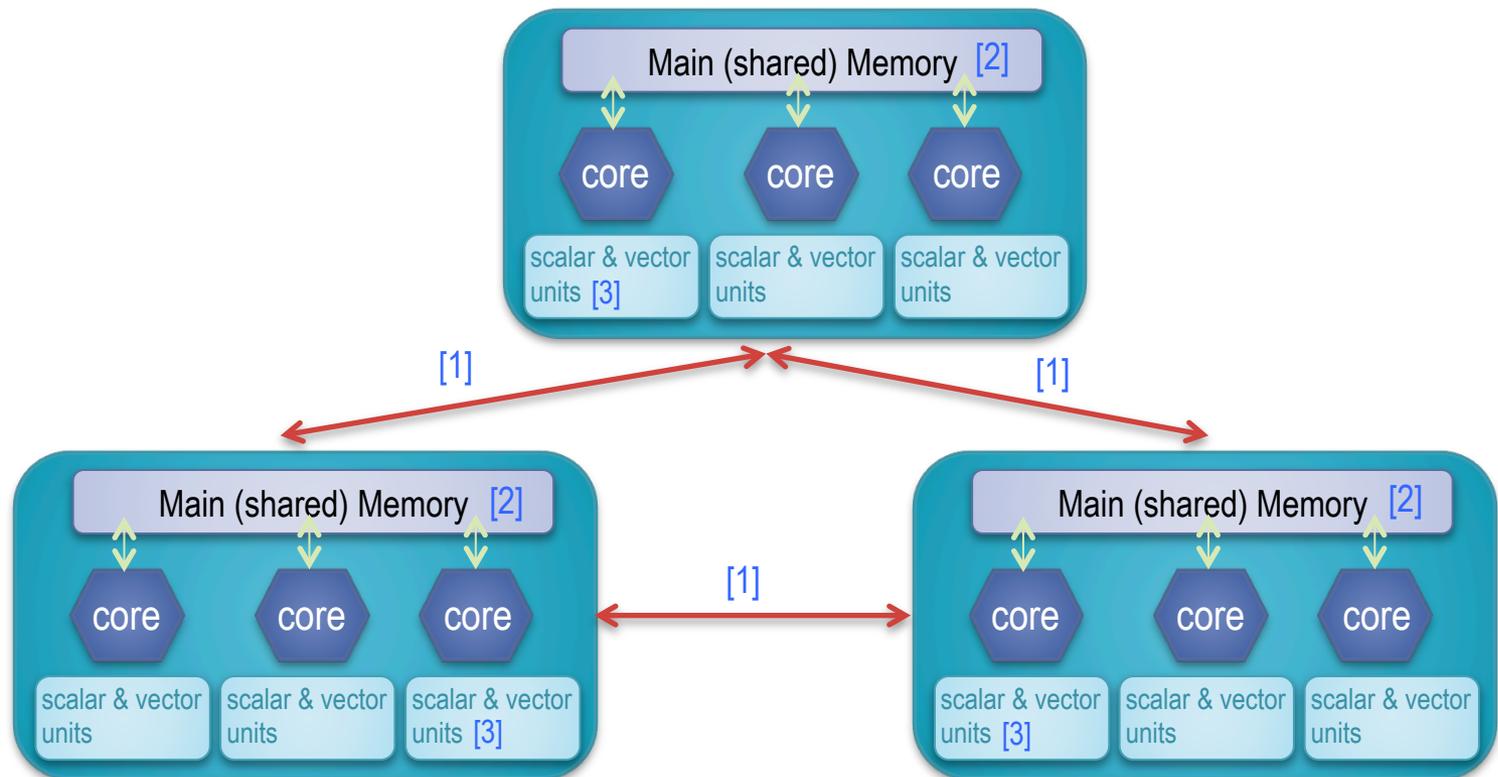
We got here a sustained performance per core of **500 Mflops over 9 GFlops**



G.Grosdidier, « Scaling stories », PetaQCD Final Review Meeting, Orsay, Sept. 27<sup>th</sup> – 28<sup>th</sup> 2012 **500 Mflops/core**

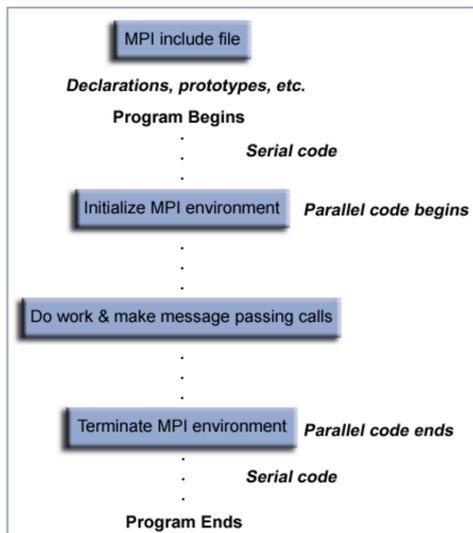
# How to Program a Supercomputer

- **Message passing** between nodes (MPI, ...) [1]
- **Shared memory** between cores (Pthreads, OpenMP, ...) [2]
- **Vector computing** inside a core (SSE, AVX, ...) [3]



# Message Passing Programming

- This is the typical way to execute across several independent compute nodes
- The whole program is decomposed at runtime into several processes
- Processes exchange data among themselves using message passing routines
- The standard programming model is **SPMD (Single Program Multiple Data)**



```
#include <stdio.h> /* printf and BUFSIZ defined there */
#include <stdlib.h> /* exit defined there */
#include <mpi.h> /* all MPI-2 functions defined there */

int main(argc, argv)
int argc;
char *argv[];
{
    int rank, size, length;
    char name[BUFSIZ];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &length);

    printf("%s: hello world from process %d of %d\n", name, rank, size);

    MPI_Finalize();

    exit(0);
}
```

# Message Passing Programming

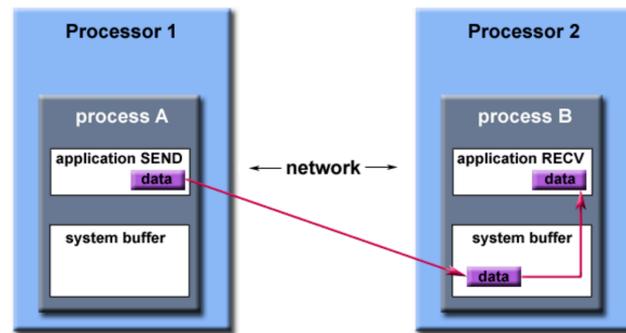
- MPI code is compiled with `mpicc -o myprogram myprogram.c`
- Our MPI program is launched with the command `mpirun myprogram -np 8`
- The value passed through “-np” is the number of processes
- The number of processes can be higher or lower than the number of processors.
- The scalability of your MPI code will mainly depends on data exchanges overhead

```
$ mpiexec -n 8 hellow2
bc89: hello world from process 0 of 8
bc31: hello world from process 2 of 8
bc29: hello world from process 1 of 8
bc33: hello world from process 3 of 8
bc34: hello world from process 5 of 8
bc30: hello world from process 4 of 8
bc35: hello world from process 6 of 8
bc32: hello world from process 7 of 8
```

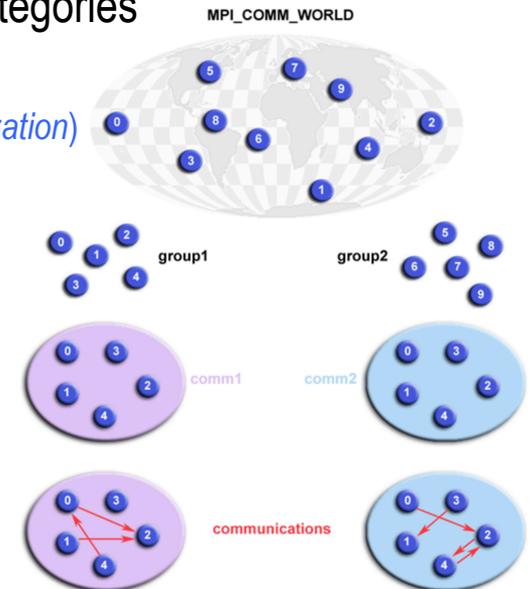
- Every MPI command starts with the prefix “MPI\_”
- There several implementations and versions of MPI, but portability is preserved

# Message Passing Programming

- MPI commands can be roughly grouped into three categories
  - Environment Management Routines
  - Communication Routines (*point-to-point – collective - synchronization*)
  - Group Communicator Management Routines

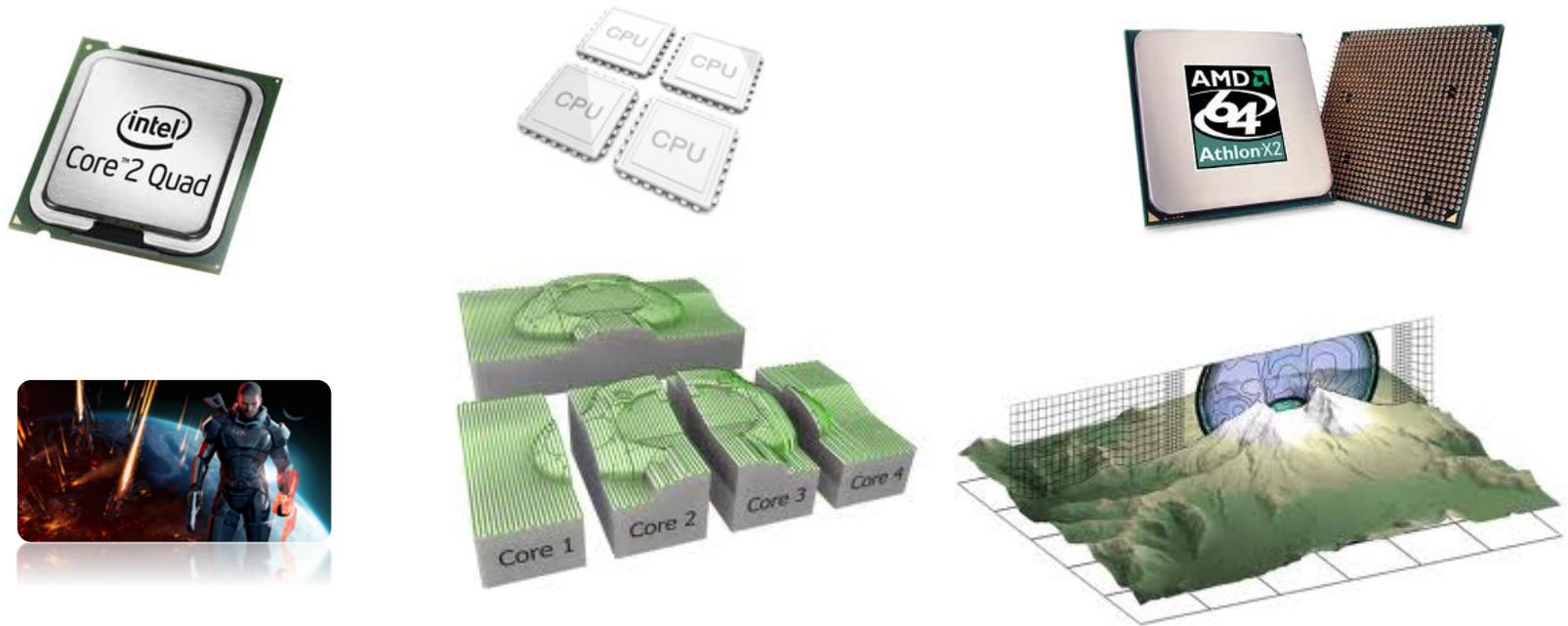


Path of a message buffered at the receiving process



- From here you just need to delve into MPI documentation for details & specific needs
- The global performance of your program will depend on both the parallel algorithm behind and the quality of the corresponding parallel program → **2 skills involved!!!**

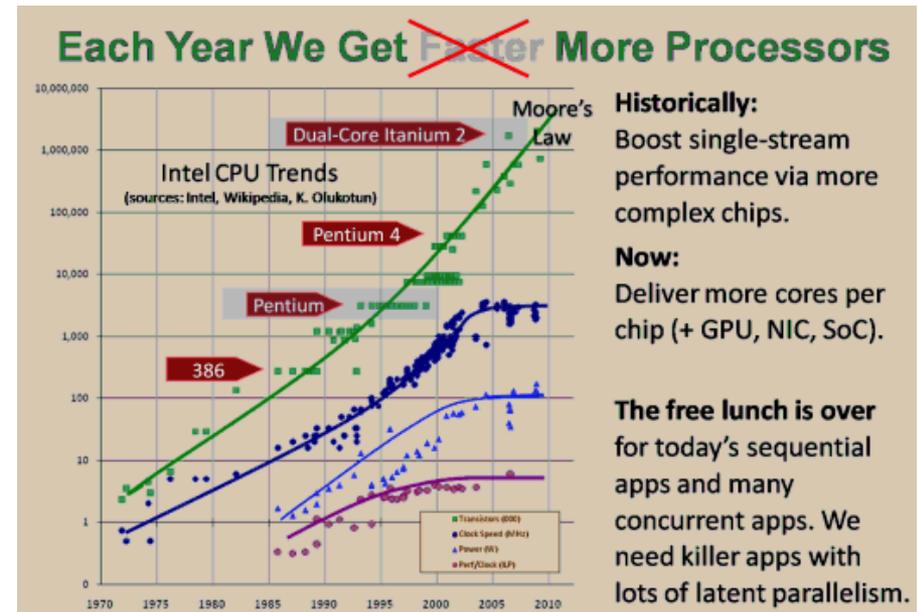
# Multithreaded Programming



*Although we can still use the message passing approach for multicore machines, it is important to know that there is a specific paradigm for this context.*

# Multithreaded Programming

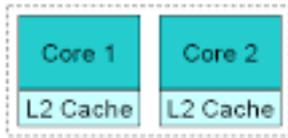
## HISTORICAL CONTEXT AND TREND



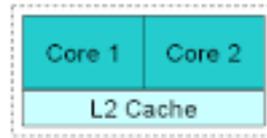
- ▶ We observe a stagnation of the processor frequency (tends to decrease)
- ▶ We need to keep following the trend of Moore's Law (transistors count)
- ▶ In order to scale up with processor speed, we need more cores per chip
- ▶ The number of cores per chip is increasing, but with complex memory system

# Multithreaded Programming

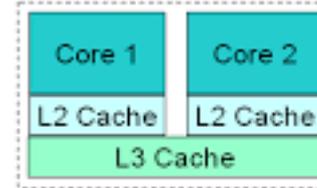
## PACKAGING & HIERARCHICAL MEMORY



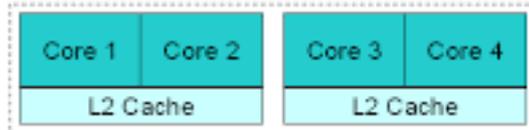
Separated Caches  
• AMD CPUs  
• Pentium D



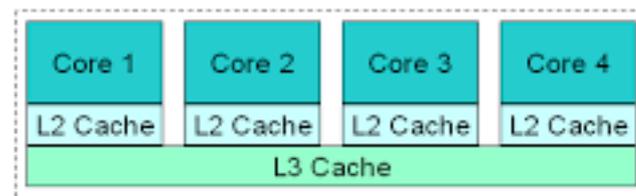
Shared Cache  
• Core Duo  
• Core 2 Duo



K10-based dual-core CPU



Current Intel Quad-Core CPUs  
• Core 2 Quad  
• Core 2 Extreme QX

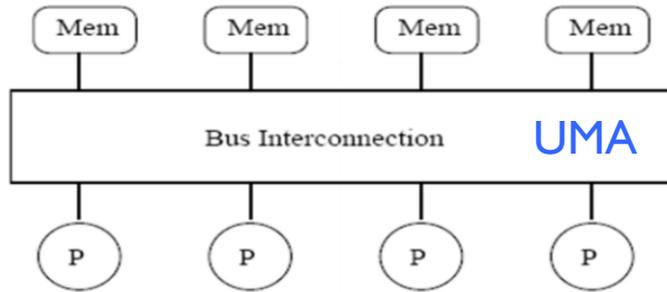


K10-based quad-core CPU

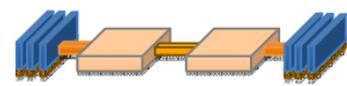
- ▶ The cores always share the main memory and there are different cache levels
- ▶ Cache memories are distributed among the cores depending on the packaging
- ▶ A given core might be able to get data from non-local unshared caches
- ▶ Cache coherency is guarantee by the hardware and associated protocols

# Multithreaded Programming

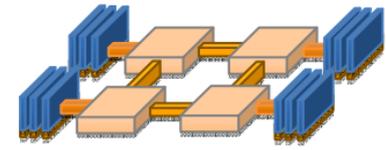
## PACKAGING AND NUMA CONSIDERATION



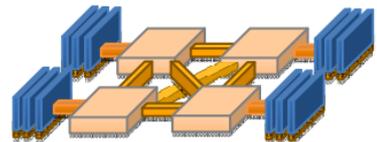
2-socket



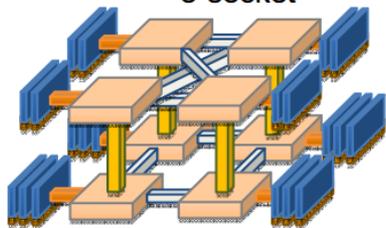
4-socket (a)



4-socket (b)



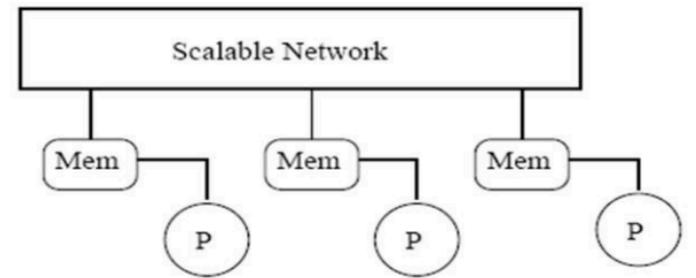
8-socket



Yinan Li et al.

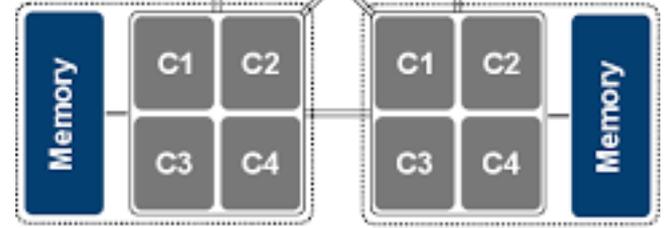
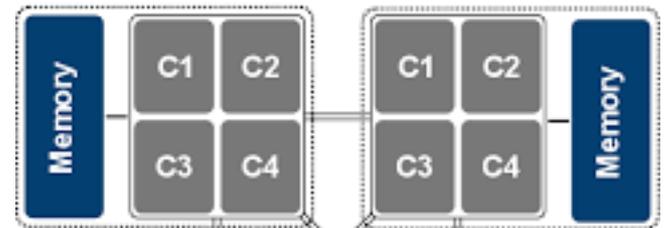
► **Serious source of scalability issues**

## Shared Memory Architecture – NUMA



NUMA Node 1

NUMA Node 2



NUMA Node 3

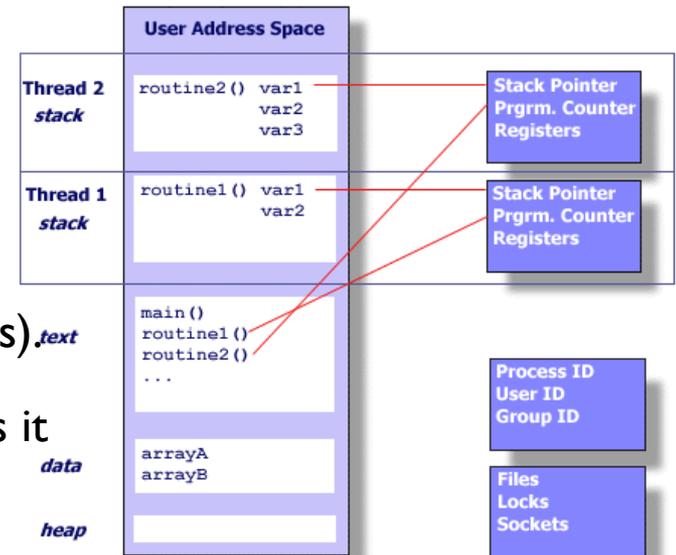
NUMA Node 4

Link

# Multithreaded Programming

## THREAD

- 🎯 In a program, an independent section or a routine can be executed as a thread.
- 🎯 A *multi-threaded* program is a program that contains **several concurrent threads**.
- 🎯 A *thread* can be seen as a **lightweight process** (memory is shared among threads).*.text*
- 🎯 A *thread* is a child of a (OS) process. Thus it uses the main resources of the process (shared between all running threads), while keeping its own
  - ✓ Stack pointer
  - ✓ Registers
  - ✓ Scheduling properties (policy ,priority)
  - ✓ Set of pending and blocked signals
  - ✓ Thread specific data.

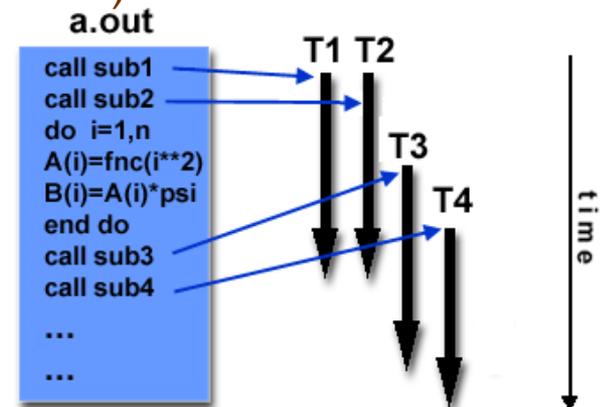


# Multithreaded Programming

A **threaded** program is built from a **classical** program by embedding the execution of **some** of its subroutines within the framework of **associated threads**.

▶ Typical scenario to design a threaded program implies

- calls to a specialized library (thread implementation)
- programming directives for threads creation
- appropriate compiler directives



▶ There are **several (incompatible) implementations of threads** depending on the target architecture (vendors) or the operating system.

This impacts on programs portability.

▶ Two standard implementations of threads are: *POSIX Threads* and *OpenMP*.

# Multithreaded Programming

## OpenMP

- ▶ Directives oriented compiler for multithreaded programming

```
PROGRAM HELLO
!$OMP PARALLEL
PRINT *, "Hello World"
!$OMP END PARALLEL
STOP
END
```

```
#include <iostream>
#include "omp.h"
int main() {
#pragma omp parallel
{
    std::cout << "Hello World\n"
}
return 0;
}
```

OMP COMPILER DIRECTIVES

```
intel: ifort -openmp -o hi.x hello.f
pgi:  pgfortran -mp -o hi.x hello.f
gnu:  gfortran -fopenmp -o hi.x hello.f
```

```
intel: icc -openmp -o hi.x hello.f
pgi:  pgcpp -mp -o hi.x hello.f
gnu:  g++ -fopenmp -o hi.x hello.f
```

```
Export OMP_NUM_THREADS=4
./hi.x
```

OMP ENVIRONMENTAL VARIABLE

NOTE: example hello.f90

# Multithreaded Programming

## Pthread

Pthread library contains hundred of routines that can be grouped into 4 categories:

- ▶ **Thread management:** Routines to create, terminate, and manage the threads.
- ▶ **Mutexes:** Routines for synchronization (through a “*mutex*”  $\approx$  *mutual exclusion* ).
- ▶ **Condition variables:** Routines for communications between threads that share a mutex.
- ▶ **Synchronization:** Routines for the management of read/write locks and barriers.

 All identifiers of the Pthreads routines and data types are prefixed with « `pthread_` »  
Example: `pthread_create`, `thread_join`, `pthread_t`, ...

 For portability, the `pthread.h` header file should be included in each source file

 The generic compile command is  
« `cc -lpthread` » or « `cc -pthread` »,  
cc = compiler

# Multithreaded Programming



```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

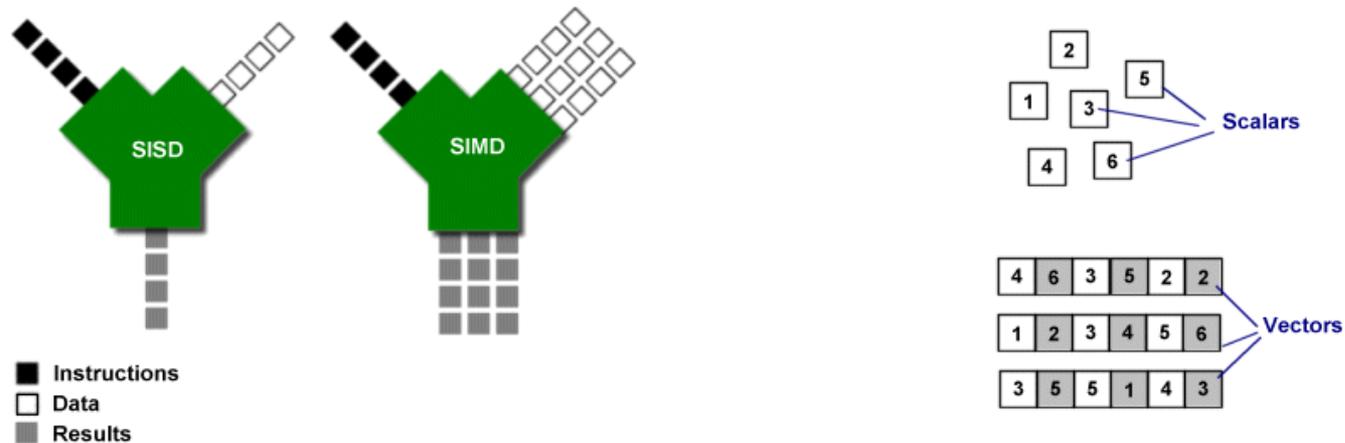
void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

# Vector Programming

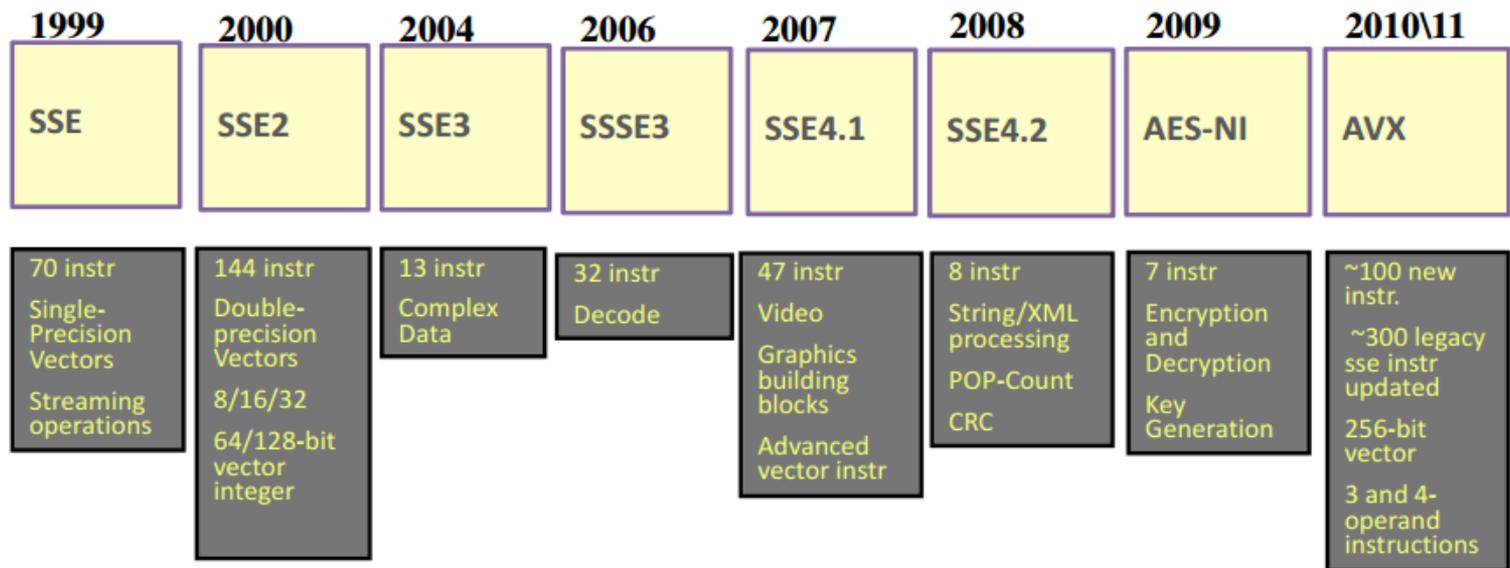


- ▶ A **SIMD** machine simultaneously operates on tuples of atomic data (*one instruction*).
- ▶ **SIMD** is opposed to **SCALAR** (the traditional mechanism).
- ▶ **SIMD** is about exploiting parallelism in the data stream (**DLP**), while superscalar **SISD** is about exploiting parallelism in the instruction stream (**ILP**).
- ▶ **SIMD** is usually referred as **VECTOR COMPUTING**, since its basic unit is the *vector*.
- ▶ Vectors are represented in what is called *packed data format* stored into *vector registers*.
- ▶ On a given machine, the length/number of the vector registers are fixed
- ▶ SIMD can be implemented on using specific extensions **MMX**, **SSE**, **AVX**, ...

# Vector Programming

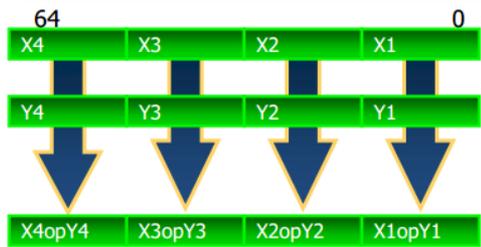
## SIMD Implementation

### SIMD: Continuous Evolution



- ▶ Then AVX2, MIC, ...
- ▶ Vector instructions can be used from their native form or through intrinsics

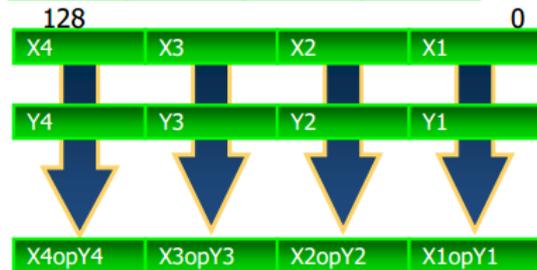
# Vector Programming



## MMX™

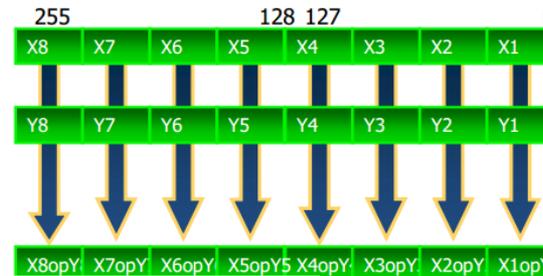
Vector size: 64bit  
 Data types: 8, 16 and 32 bit integers  
 VL: 2,4,8  
 For sample on the left: Xi, Yi 16 bit integers

**MMX = MultiMedia eXtension**  
**SSE = Streaming SIMD Extension**  
**AVX = Advanced Vector Extensions**  
**MIC = Many Integrated Core**



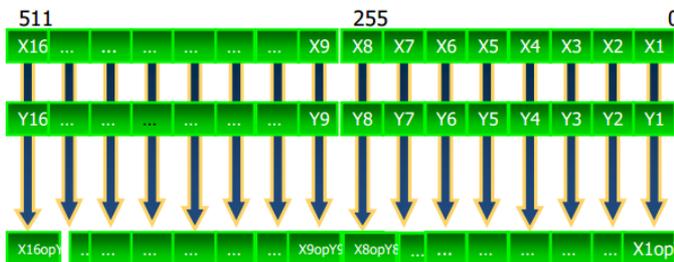
## Intel® SSE

Vector size: 128bit  
 Data types:  
 8,16,32,64 bit integers  
 32 and 64bit floats  
 VL: 2,4,8,16  
 Sample: Xi, Yi bit 32 int / float



## Intel® AVX

Vector size: 256bit  
 Data types: 32 and 64 bit floats  
 VL: 4, 8, 16  
 Sample: Xi, Yi 32 bit int or float



## Intel® MIC

Vector size: 512bit  
 Data types:  
 32 and 64 bit integers  
 32 and 64bit floats  
 (some support for 16 bits floats)  
 VL: 8,16  
 Sample: 32 bit float

# Vector Programming

- ▶ **SSE = Streaming SIMD Extensions**
- ▶ SSE programming can be done either through (inline) assembly or from a high-level language (C and C++) using **intrinsics**.
- ▶ The `{x,e,p}mmintrin.h` header file contains the declarations for the SSE<sub>x</sub> instructions intrinsics.
  - `xmmintrin.h` -> SSE
  - `emmintrin.h` -> SSE2
  - `pmmmintrin.h` -> SSE3
- ▶ SSE instruction sets can be enabled or disabled. If disabled, SSE instructions will not be possible. It is recommended to leave this BIOS feature enabled by default. In any case MMX (MultiMedia eXtensions) will still be available.
- ▶ Compile your SSE code with "`gcc -o vector vector.c -msse -msse2 -msse3`"
- ▶ SSE intrinsics use types `__m128` (*float*), `__m128i` (*int, short, char*), and `__m128d` (*double*)
- ▶ Variable of type `__m128`, `__m128i`, and `__m128d` (exclusive use) maps to the XMM[0-7] registers (**128 bits**), and automatically aligned on 16-byte boundaries.
- ▶ Vector registers are `xmm0`, `xmm1`, ..., `xmm7`. Initially, they could only be used for single precision computation. Since SSE2, they can be used for any primitive data type.

# Vector Programming

## SSE (Connecting vectors to scalar data)

- ▶ Vector variables can be connected to scalar variables (arrays) using one of the following ways

```
float a[N] __attribute__((aligned(16)));  
__m128 *ptr = (__m128*)a;
```

**ptr[i]** or **\*(ptr+i)** represents the vector  
**{a[4i], a[4i+1], a[4i+2], a[4i+3]}**

```
float a[N] __attribute__((aligned(16)));  
__m128 mm_a;  
mm_a = _mm_load_pd(&a[4i]); // here we explicitly load data into the vector
```

**mm\_a** represents the vector  
**{a[4i], a[4i+1], a[4i+2], a[4i+3]}**

- ▶ Using the above connections, we can now use SSE instruction to process our data. This can be done through
  - ★ (inline) assembly
  - ★ intrinsics (interface to keep using high-level instructions to perform vector operations)

# Vector Programming

## SSE (illustrations)

```
void scalar_sqrt(float *a){
    int i;
    for(i = 0; i < N; i++)
        a[i] = sqrt(a[i]);
}
```

### Scalar version

```
void sse_sqrt(float *a){
    // We assume N % 4 == 0.
    int nb_iters = N / 4;
    __m128 *ptr = (__m128*)a;
    int i;
    for(i = 0; i < nb_iters; i++, ptr++, a += 4)
        _mm_store_ps(a, _mm_sqrt_ps(*ptr));
}
```

### Vector version (SSE)

```
Tadonki@TADONKI-PC ~/vector
$ ./test
Running time of the scalar code: 0.286017
Running time of the SSE code: 0.031001
```

10 times faster !!!!!!!

# Conclusion

---

HPC is making noticeable progresses, but we still need to skillfully use its elements and concepts in order to reach our performance expectations.

## There is no free lunch



# End

---

Thanks for your attention

