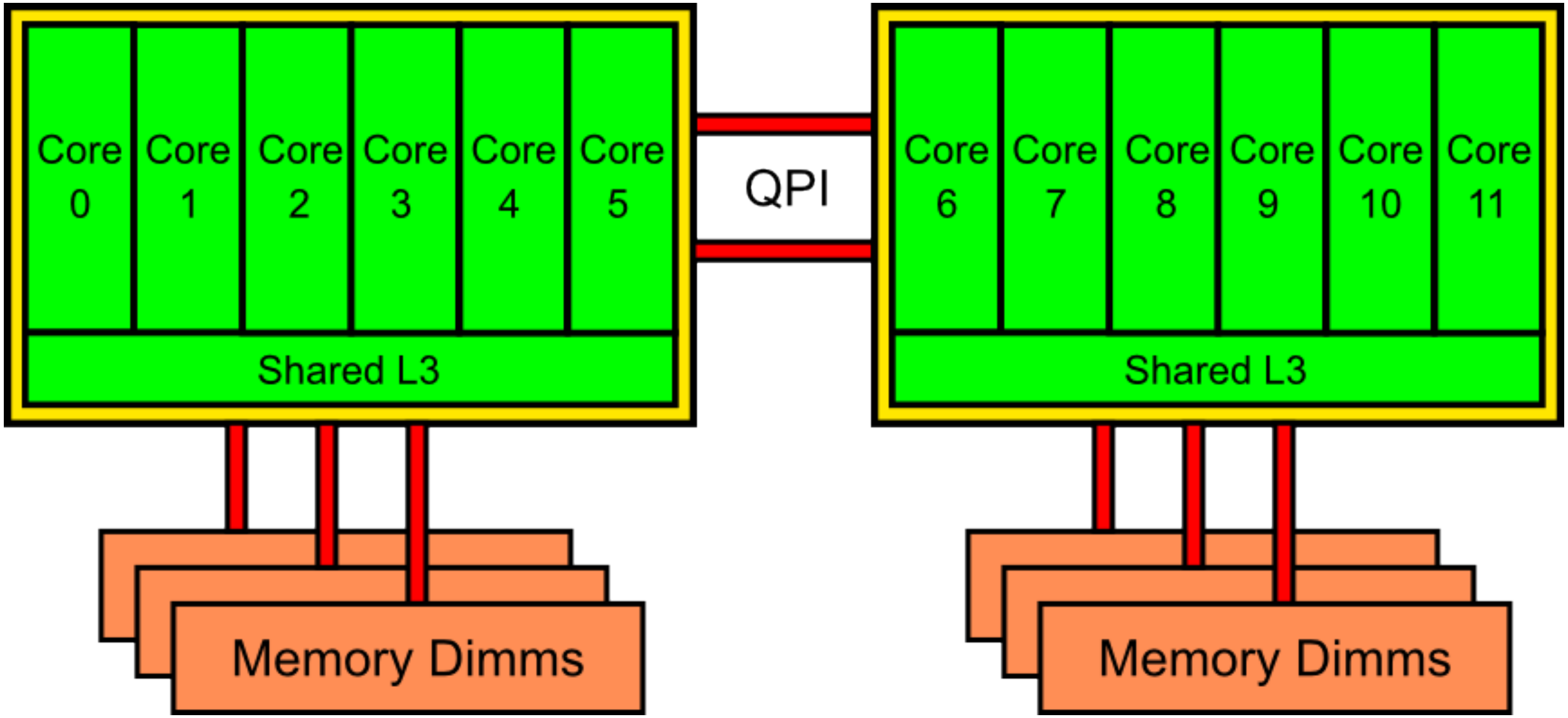
The Abdus Salam
International Centre
for Theoretical Physics

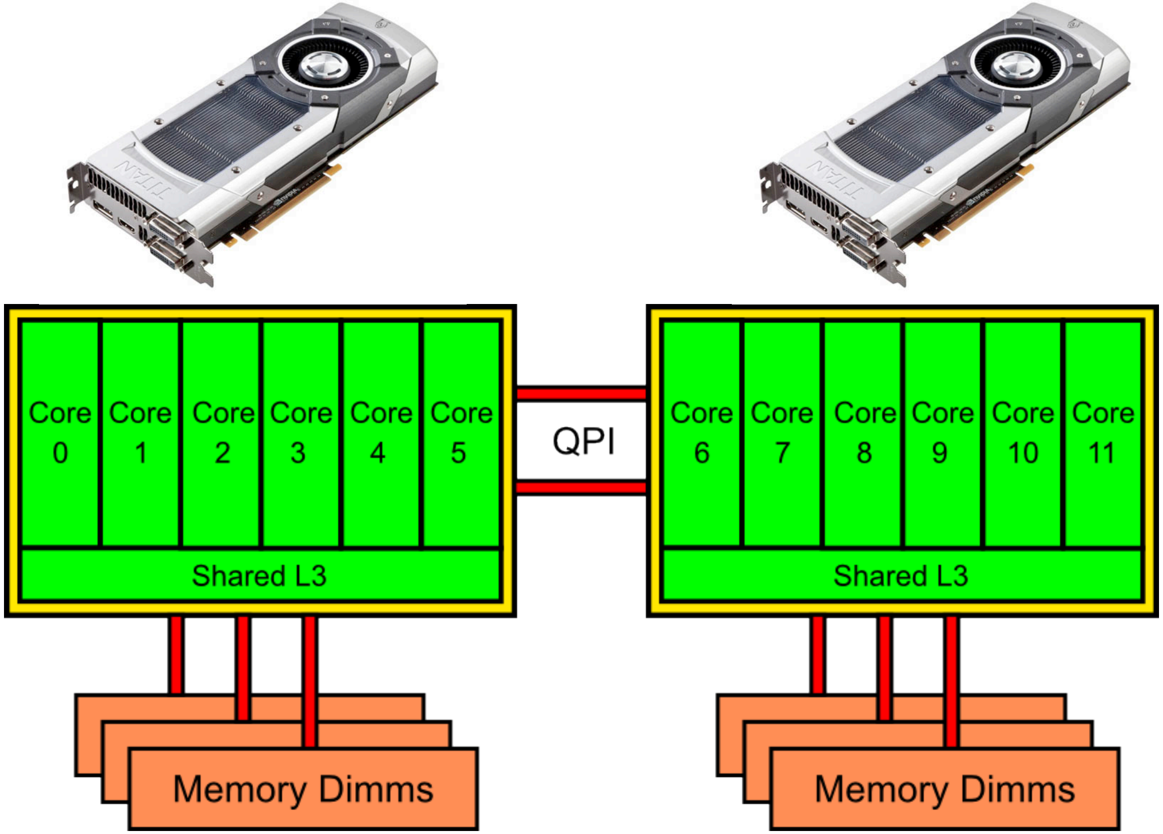
Overview on GPU Accelerators and Programming Paradigms

Ivan Girotto – igirotto@ictp.it
Information & Communication Technology Section (ICTS)
International Centre for Theoretical Physics (ICTP)

Multiple Socket CPUs

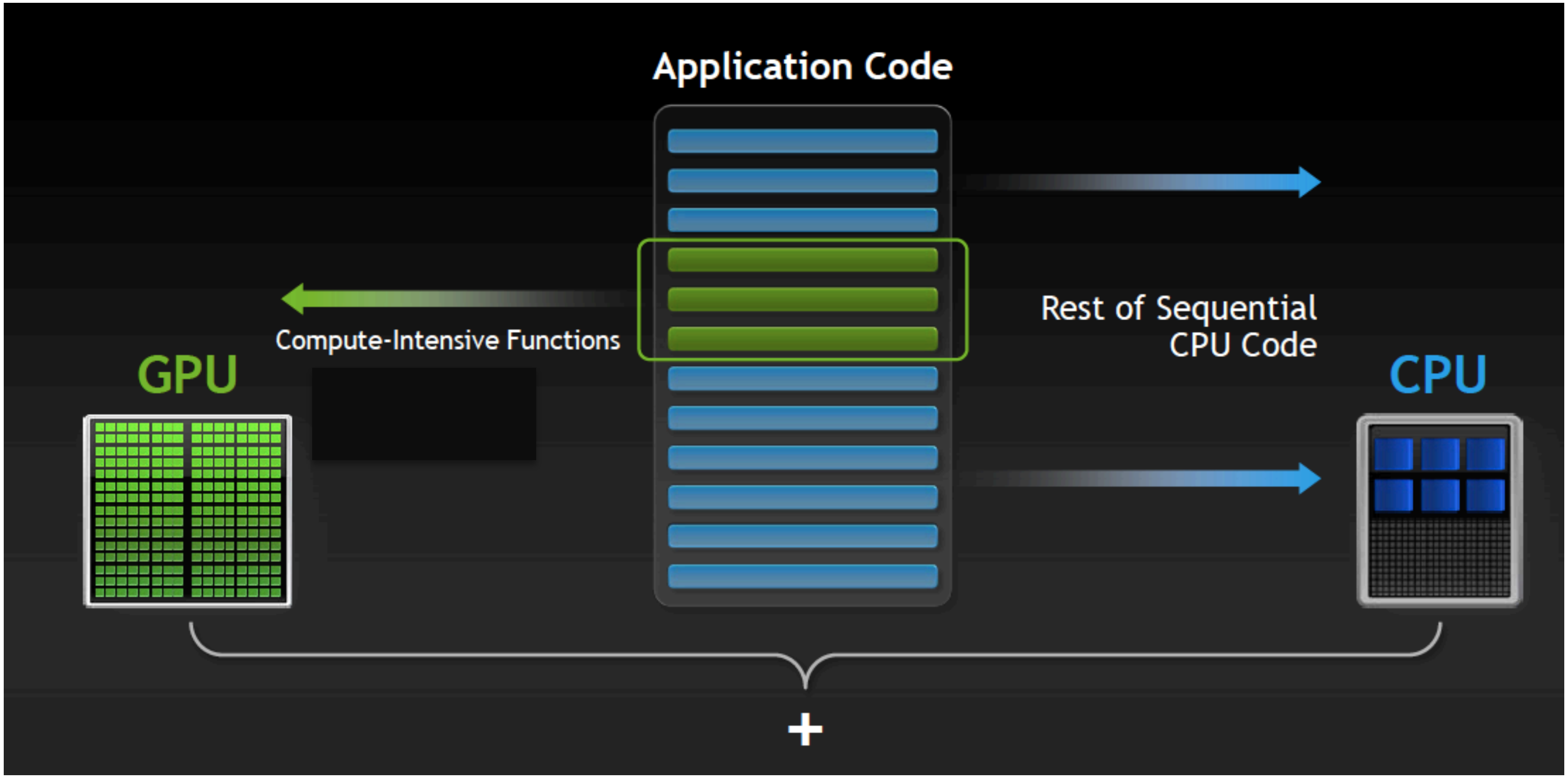


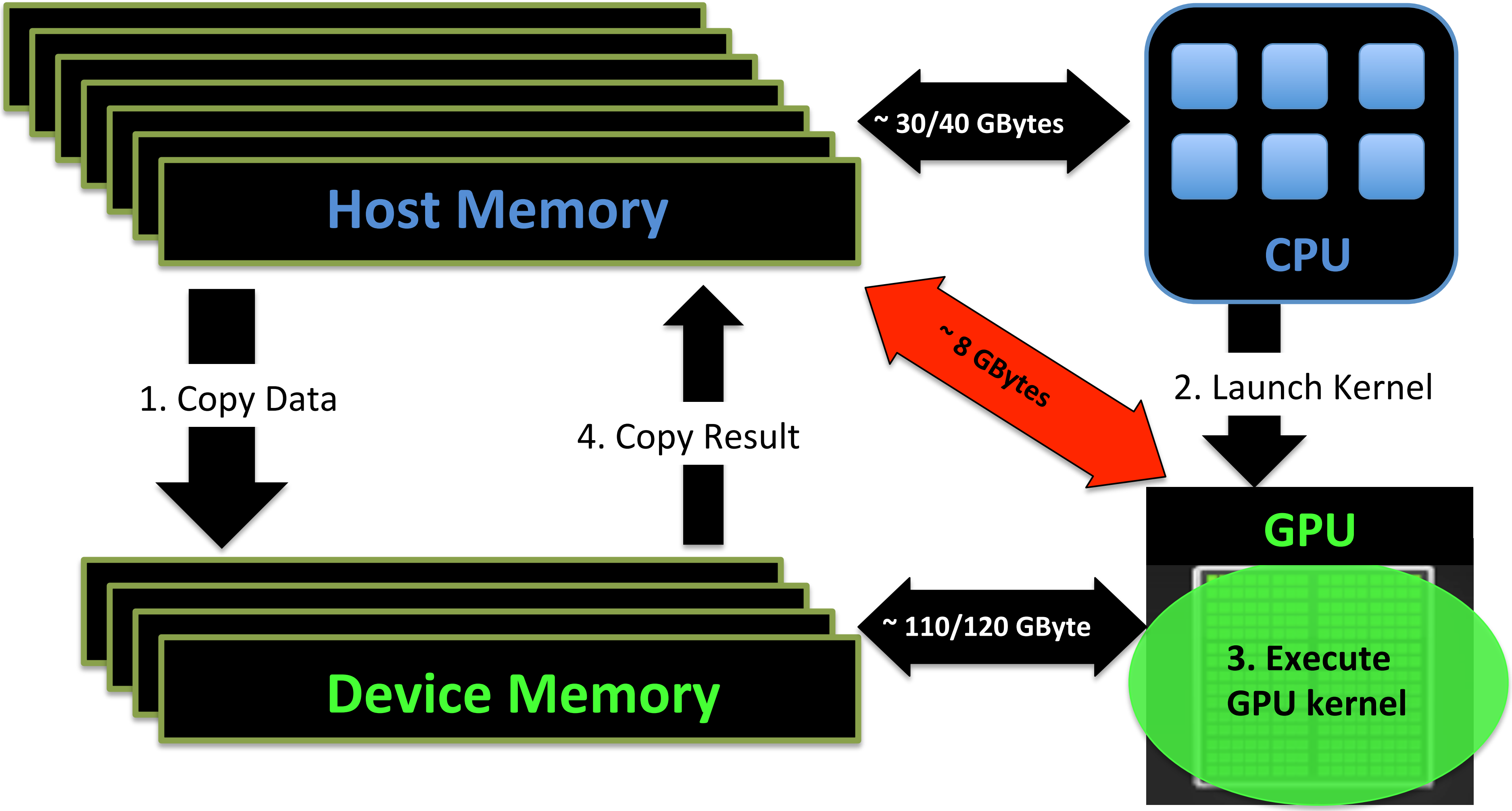
Multiple Socket CPUs + Accelerators



The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

The General Concept of Accelerated Computing

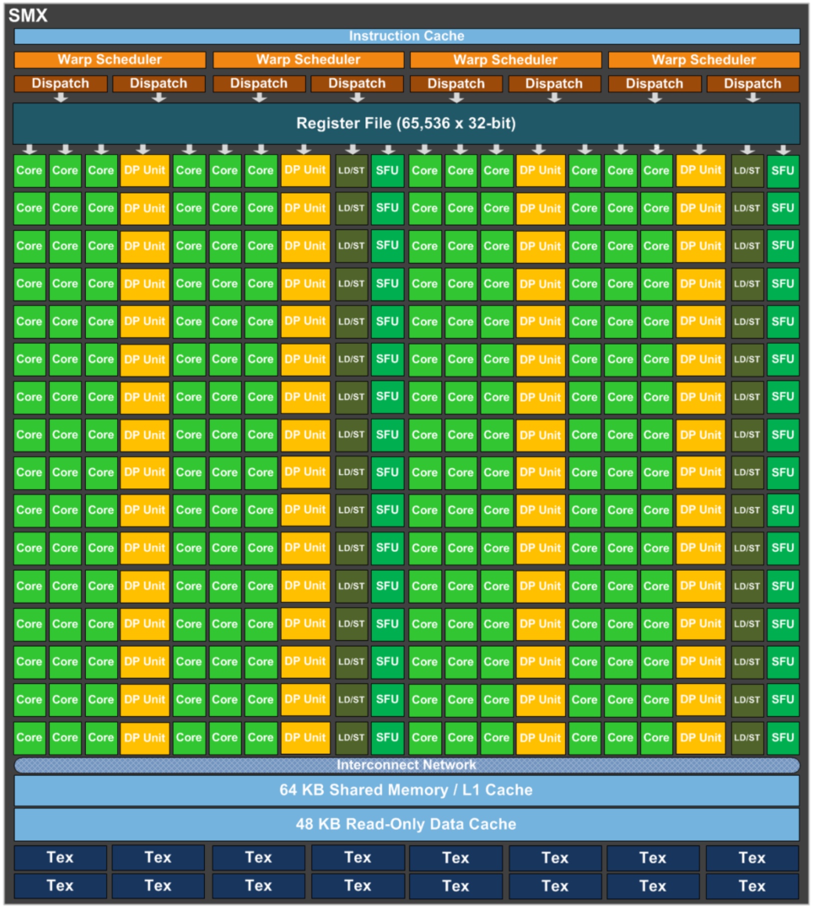


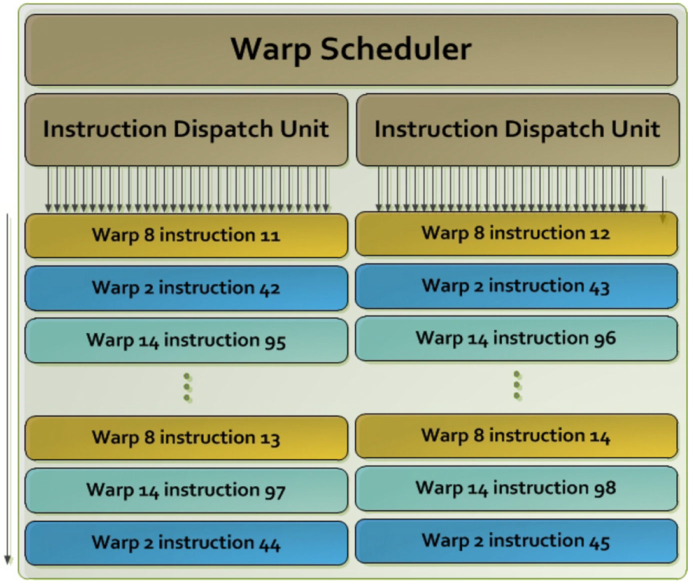

Host Memory
~ 30/40 GBytes
CPU
~ 8 GBytes
1. Copy Data
4. Copy Result
2. Launch Kernel
GPU
~ 110/120 GByte
3. Execute GPU kernel
Device Memory

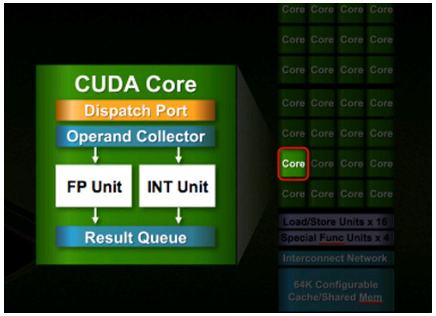
Why Does GPU Accelerate Computing?

- Highly scalable design
- Higher aggregate memory bandwidth
- Huge number of low frequency cores
- Higher aggregate computational power
- Massively parallel processors for data processing

SMX Processor & Warp Scheduler & Core


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex





Why Does GPU Not Accelerate Computing?

- PCI Bus bottleneck
- Synchronization weakness
- Extremely slow serialized execution
- High complexity
  - SPMD(T) + SIMD & Memory Model
- People forget about the Amdahl's law
  - accelerating only the 50% of the original code, the expected speedup can get at most a value of 2!!

What is CUDA?

- **NVIDIA** compute architecture
- Software development capability provided free of charge by NVIDIA
- C and C++ programming language extension that simplifies creation of efficient applications for CUDA-enabled GPGPUs
- Available for Linux, Windows and Mac OS X

Vector Sum on GPU

```
#define N  (2048 * 2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;                   // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;   // device copies of a, b, c
    int size = N * sizeof( int ); // we need space for N integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );

   // copy inputs to device
   cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
   cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );
   // launch add() kernel with blocks and threads
   add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(dev_a, dev_b, dev_c);
   // copy device result back to host copy of c
   cudaMemcpy( c, dev_c, size,   cudaMemcpyDeviceToHost );
   free( a ); free( b ); free( c );
   cudaFree( dev_a );
   cudaFree( dev_b );
   cudaFree( dev_c );
   return 0;
}
```

Directive Based Approaches: OpenACC

- Implementations available now from PGI, Cray, and GCC
- Same source can be used to generate code for CPU and GPU
- Easier development
- Less flexibility

Vector Sum on GPU

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( int argc, char* argv[] )
{
    int n = 10000;

    double *restrict a;
    double *restrict b;
    double *restrict c;

    size_t bytes = n*sizeof(double);
    a = (double*)malloc(bytes);
    b = (double*)malloc(bytes);
    c = (double*)malloc(bytes);

    // Initialize content of input vectors, vector a[i] = sin(i)^2 vector b[i] = cos(i)^2
    int i;
    for(i=0; i<n; i++) {
        a[i] = sin(i)*sin(i);
        b[i] = cos(i)*cos(i);
    }

    // sum component wise and save result into vector c
    #pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])
    for(i=0; i<n; i++) {
        c[i] = a[i] + b[i];
    }
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Directive Based Approaches: OpenMP

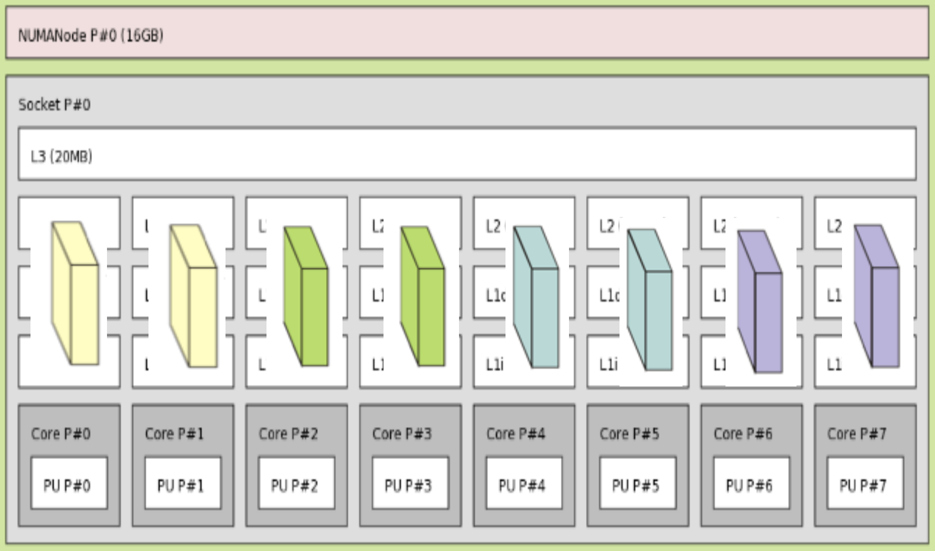
- The API V4.5 describes OpenMP pragma for GPUs
- At the moment IBM is the only main compiler supporting it (see http://www.openmp.org/resources/openmp-compilers/)
- Ideally works with same model of OpenACC

CUDA Fortran

- PGI / NVIDIA collaboration
- Same CUDA programming model as CUDA-C with Fortran syntax
- Variables with device-type reside in GPUmemory
- Use standard allocate, deallocate
- Copy between CPU and GPU with assignment statements:

  ```
  GPU_array = CPU_array
  ```
- Kernel loop directives (CUF Kernels) to parallelize loops with device data

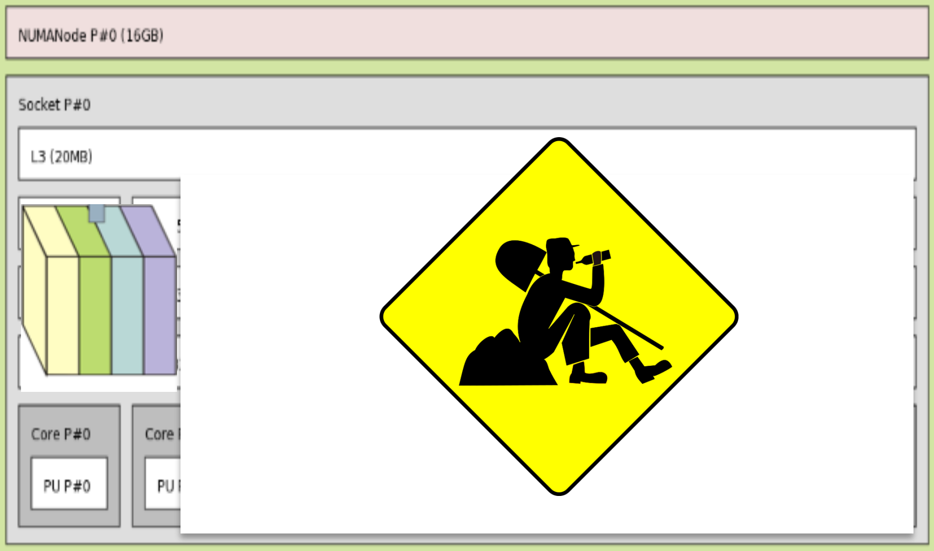
CPU & GPU



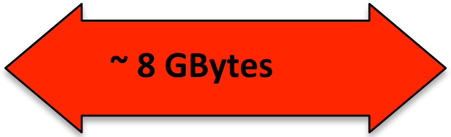
The Intel Xeon E5-2665
Sandy Bridge-EP 2.4GHz

~ 8 GBytes

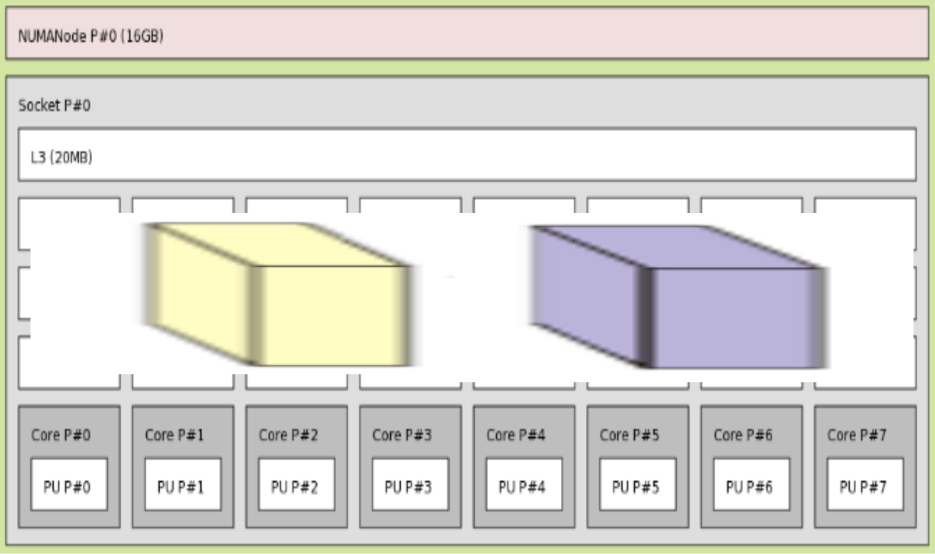
CPU & GPU



The Intel Xeon E5-2665
Sandy Bridge-EP 2.4GHz



CPU & GPU



NUMANode P#0 (16GB)

Socket P#0

L3 (20MB)

Core P#0 PU P#0

Core P#1 PU P#1

Core P#2 PU P#2

Core P#3 PU P#3

Core P#4 PU P#4

Core P#5 PU P#5

Core P#6 PU P#6

Core P#7 PU P#7

The Intel Xeon E5-2665
Sandy Bridge-EP 2.4GHz

~ 8 GBytes

GPUs platforms for HPC

- Deploy balanced and cost effective GPUs based platforms is still really hard these days
- Management, usage and SW development for add on accelerated platform requires skills and expertise
- The NVLINK promises delivers high bandwidth between GPUs but only IBM supports NVILINK connection GPU/CPU
- General purpose high-density GPU based solution are limited to specific cases

GPU SW Development and Applications

- GPU based technology platforms evolve rapidly
- New features are often disruptive and requires effort for software optimization
- Efficient GPU code requires constant update and maintenance (today really much true for CPU SW too)
- Few remarks on GPU based SW for scientific computing