

# Parallel Computing

R. Guivarc'h

Ronan.Guivarch@toulouse-inp.fr

2021 - Sciences du Numérique

## Part II - GPU

# Acknowledgements

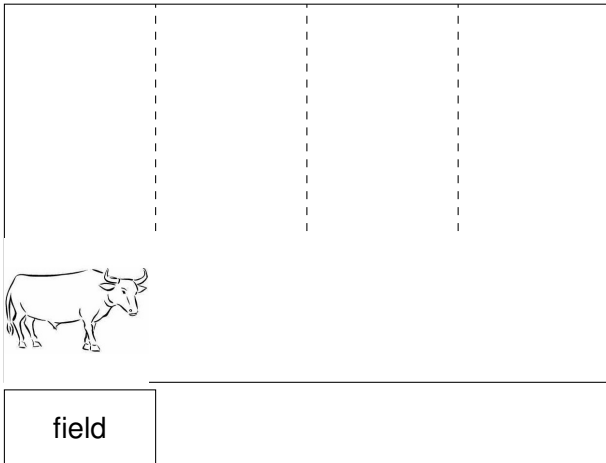
Used materials and examples from

- **Introduction to GPU Programming with the OpenMP API, Dr.-Ing. Michael Klemm, OpenMP Architecture Review Board**
- Introduction aux GPGPU, Amina Guermouche, cours de Télécom SudParis
- OpenMP on GPUs, First Experiences and Best Practices, Jeff Larkin, NVIDIA
- OpenMP 5.0/5.1 Tutorial, Exascale Computing Project
- Introduction to OpenACC and OpenMP GPU, Pierre-François Lavallée, Thibaut Véry, IDRIS
- Introduction to Directive Based Programming on GPU, Helen He, NERSC
- Applications of GPU Computing, Alex Karantza

# Web Pointers

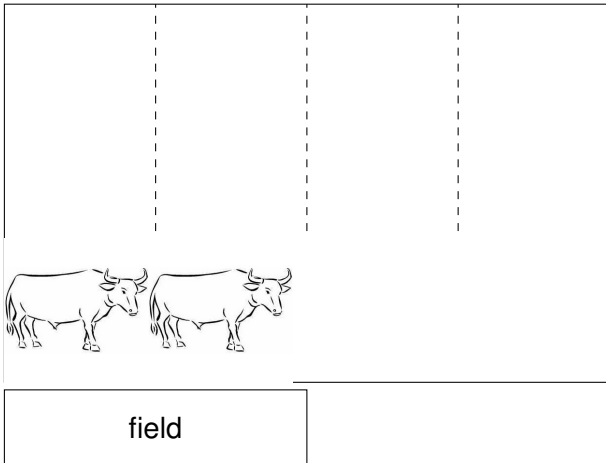
- **Intro to GPU Programming with the OpenMP API (OpenMP Webinar)**
- Introduction to OpenMP - Tim Mattson (Intel)
- GPU, IA et Big Data : comprendre le rôle des cartes graphiques en Data Science
- GPUs explained
- NVIDIA, AMD

# A bit of history



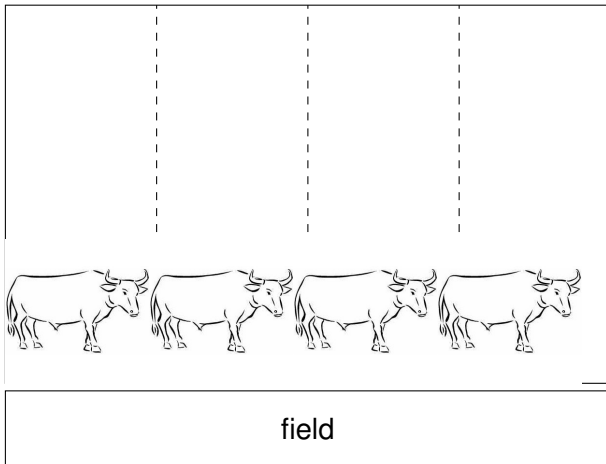
# A bit of history

- Increased number of CPUs
- 😊 Faster

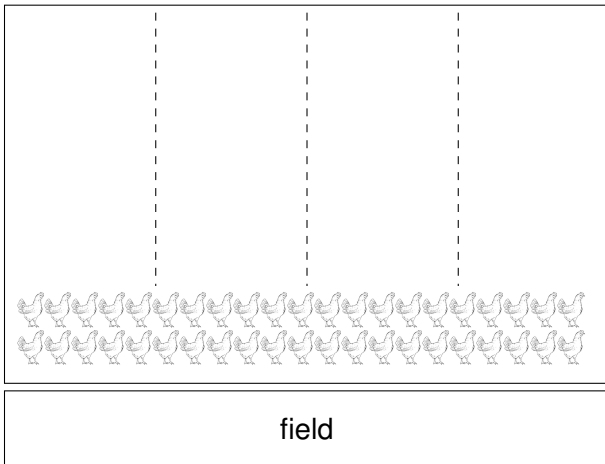


# A bit of history

- Increased number of CPUs
- 😊 Faster
- 😞 More expensive, physical limits



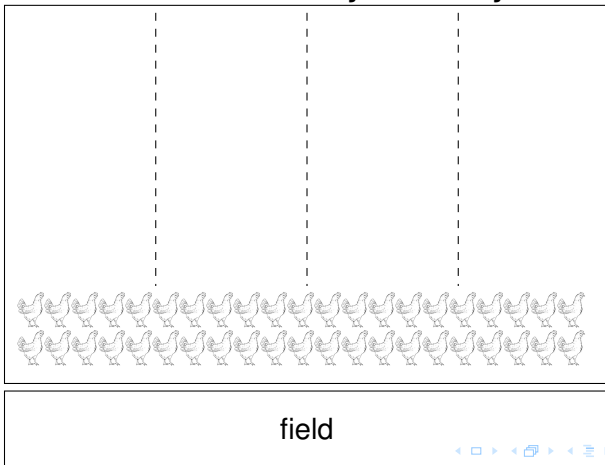
## a bit of history





## a bit of history

"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?" **Seymour Cray**



# CPU VS GPU: demo

<https://www.youtube.com/watch?v=-P28LKWTzrI>

# GPUs in Industry

Many applications have been developed to use GPUs for supercomputing in various fields

- Scientific Computing: CFD, Molecular Dynamics, Genome Sequencing, Mechanical Simulation, Quantum Electrodynamics
- Image Processing: Registration, interpolation, feature detection, recognition, filtering
- Data Analysis: Databases, sorting and searching, data mining

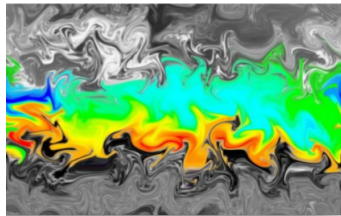
# Major Categories of Algorithm

- 2D/3D filtering operations
- n-body simulations
- Parallel tree operations – searching/sorting
- ...

All suited to GPUs because of data-parallel requirements and uniform kernels

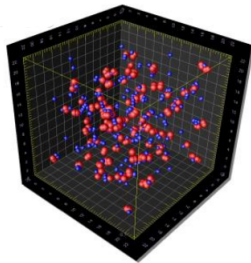
# Computational Fluid Dynamics

- Simulate fluids in a discrete volume over time
- Involves solving the Navier-Stokes partial differential equations iteratively on a grid (can be considered as a filtering operation)
- When parallelized on a GPU using multigrid solvers, 10x speedups have been reported



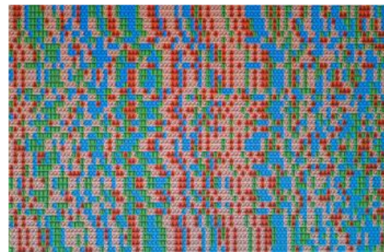
# Molecular Dynamics

- Large set of particles with forces between them – protein behavior, material simulation
- Calculating forces between particles can be done in parallel for each particle
- Accumulation of forces can be implemented as multilevel parallel sums



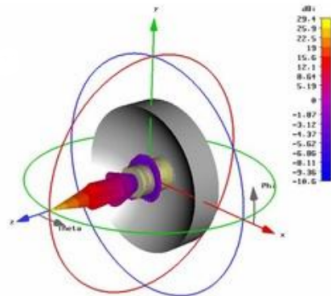
# Genetics

- Large strings of genome sequences must be searched through to organize and identify samples
- GPUs enable multiple parallel queries to the database to perform string matching
- Again, order of magnitude speedups reported



# Electrodynamics

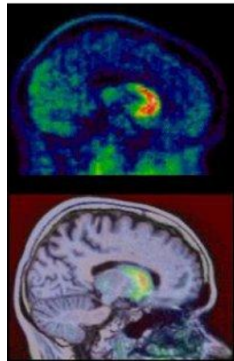
- Simulation of electric fields, Coulomb forces
- Requires iterative solving of partial differential equations
- Cell phone modeling applications have reported 50x speedups using GPUs





# Image Processing

- Medical Imaging was the early adopter
  - Registration of massive 3D voxel images
  - Both the cost function for deformable registration and interpolation of results are filtering operations
- Generic feature detection, recognition, object extraction are all filters
- For object recognition, one can search a database of objects in parallel
- Running these algorithms off the CPU can allow real-time interaction



# Data Analysis

- Huge databases for web services require instant results for many simultaneous users
- Insufficient room in main memory, disk is too slow and doesn't allow parallel reads
- GPUs can split up the data and perform fast searches, keeping their section in memory



# Artificial Intelligence and Big Data

- Some GPUs are efficient for matrix multiplication and convolution (tensor cores)
- Bandwidth-optimised, ability to process a large amount of data
- Machine Learning models can be trained 215 times faster.



# CPU vs. GPU

- Latency vs. Throuput
- Task parallelism vs. Data parallelism
- Multi-threaded vs. SIMD
- Tens of threads vs. Tens of thousands of threads

# Latency & Throughput

- Latency is the delay between the time an operation is initiated, and the moment when its effects become detectable
  - A car has a lower latency than a bus (faster)
- Throughput is the amount of work done over time
  - A bus has a higher throughput than a car (more people at a time)

# Latency & Throughput

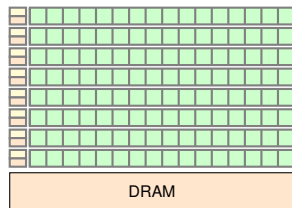
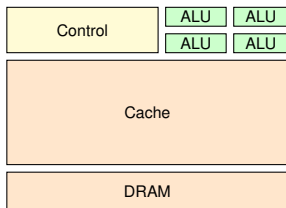
- CPUs must **minimize latency** (neglecting throughput 😞)
    - keyboard input
  - CPUs maximize out-of-cache operations (pre-fetch, out-of-order execution, ...)
- CPUs need large caches

# Latency & Throughput

- CPUs must **minimize latency** (neglecting throughput 😞)
  - keyboard input
- CPUs maximize out-of-cache operations (pre-fetch, out-of-order execution, ...)
- CPUs need large caches
- GPUs are **high performance, high throughput processors**
  - They do not need large caches
  - More transistors can be dedicated to computation

# Chipset

- More transistors are dedicated to data processing instead of cache management
- Chip of the same size but with more ALUs, so more threads for computation





# Thread management on GPUs

- **How manage**

- Synchronisation between as many threads
- Scheduling, context switching
- Programmation

# Thread management on GPUs

- **How manage**

- Synchronisation between as many threads
- Scheduling, context switching
- Programmation

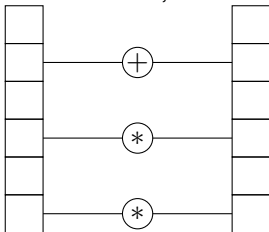
- threads on GPUs are:

- Independent (no synchronisation)
- SIMD (reduced scheduling cost)
- Programming by block of threads

# CPU Parallelism vs. GPU Parallelism

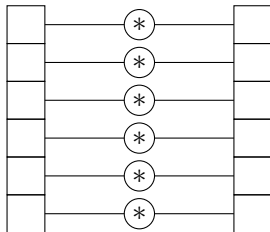
## CPU: Task Parallelism

- Simultaneous execution of several functions on different cores and on different data, or not



## GPU: Data Parallelism

- Simultaneous execution of the same function by several cores on different data




# Stream processing

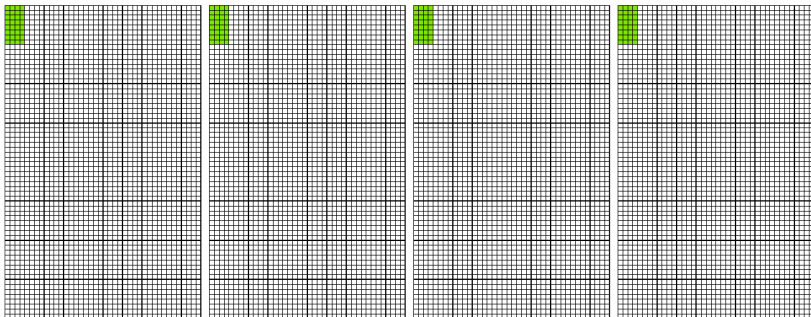
- The fundamental unit of a GPU is the **stream processor (SP)**
  - Large amount of data ("*stream*")
  - Execute the same operation ("**kernel**" or "shader") on all the data
- we have also SM (**stream multi-processor**)

# Architecture of a converged node on Jean Zay

The number of compute cores on machines with GPUs is much greater than on a classical machine.

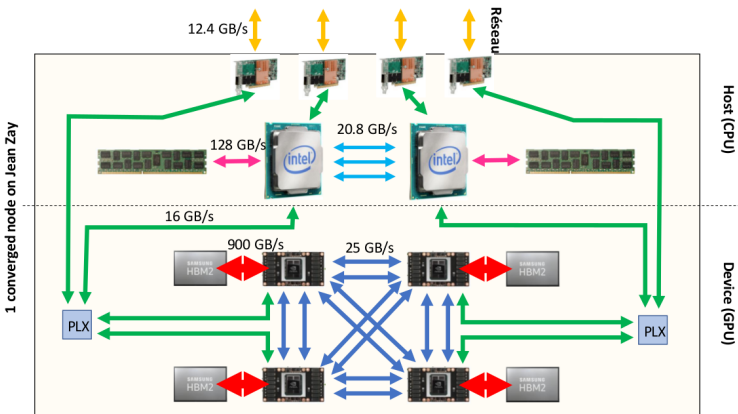
IDRIS' Jean-Zay has 2 partitions:

- Non-accelerated :  $2 \times 20 = 40$  cores 
- Accelerated with 4 Nvidia V100 =  $32 \times 80 \times 4 = 10240$  cores

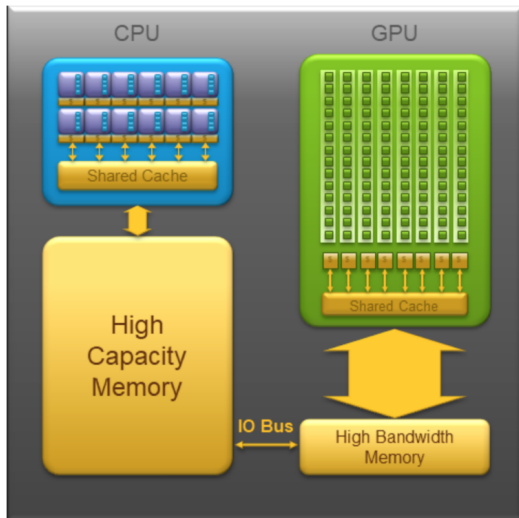


# Architecture of a converged node on Jean Zay

Architecture of a converged node on Jean Zay (40 cores, 192 GB of memory, 4 GPU V100)



# General Architecture of a converged node



# Programming GPUs

Programming effort and technical expertise

## Libraries

- cuBLAS
- cuSPARSE
- cuRAND
- AmgX
- MAGMA

## Directives

- OpenACC
- OpenMP 5.1
- OpenMP 5.2

## Programming languages

- CUDA
- OpenCL
- HIP



# Programming GPUs

Programming effort and technical expertise

## Libraries

- cuBLAS
- cuSPARSE
- cuRAND
- AmgX
- MAGMA

- + Minimum change in the code
- + Free performance
- Limited by the available libraries and their functionalities

## Directives

- OpenACC
- OpenMP 5.1
- OpenMP 5.2

- + Easy to use
- + Portable
- Performance
- (user has less control)

## Programming languages

- CUDA
- OpenCL
- HIP

- Harder to use
- Need to rewrite application
- Less portable
- ++ Fine control of performance & flexibility

# Directives

## OpenAcc

- <http://www.openacc.org/>
- Cray, NVidia, PGI, CAPS
- First standard 1.0 (11/2011)
- Latest standard 3.0 (11/2019)
- Main compilers:
  - PGI
  - Cray (for Cray hardware)
  - GCC (since 5.7)

## OpenMP target

- <http://www.openmp.org/>
- First standard OpenMP 4.5 (11/2015)
- Latest standard ~~OpenMP 5.1 (11/2020)~~ OpenMP 5.2 (11/2021)
- Main compilers:
  - Cray (for Cray hardware)
  - GCC (since 7)
  - CLANG
  - IBM XL
  - PGI

# Sample OpenMP and OpenACC Codes

```
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma omp target
#pragma omp teams distribute
for (i=0; i<N; ++i) {
#pragma omp parallel for simd
    for (j=0; j<N; ++j) {
        x[j+N*i] *= 2.0;
    }
}
```

```
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma acc parallel
#pragma acc gang worker
for (i=0; i<N; ++i) {
#pragma acc vector
    for (j=0; j<N; ++j) {
        x[j+N*i] *= 2.0;
    }
}
```

directives

# CUDA code example (for information)

## kernel

```
--global__ void add (int *a, int *b, int *c){  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

## main

```
#define N 512  
int main (void){  
    int *a, *b, *c;  
    int *gpu_a, *gpu_b, *gpu_c;  
    int size = N * sizeof(int);  
    // memory allocation on GPU  
    cudaMalloc((void **)&gpu_a, size);  
    cudaMalloc((void **)&gpu_b, size);  
    cudaMalloc((void **)&gpu_c, size);  
    // allocation on CPU  
    a=(int*) malloc (size);  
    b=(int*) malloc (size);  
    c=(int*) malloc (size);
```

# CUDA code example (for information)

## main (cont.)

```
// Initialization of a and b
...
// Data transfert to GPU
cudaMemcpy (gpu_a,a,size ,cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,size ,cudaMemcpyHostToDevice);

// call the kernel on N threads
add <<< N, 1 >>> (gpu_a, gpu_b, gpu_c);

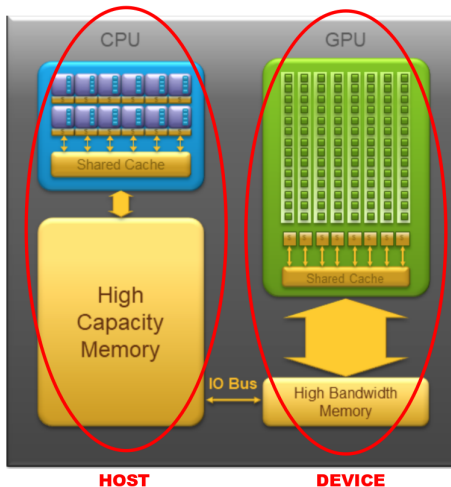
// Data transfert from GPU
cudaMemcpy(c, gpu_c, size , cudaMemcpyHostToDevice);

// Free memory (both CPU and GPU)
free(a); free(b); free(c);
cudaFree(gpu_a);
cudaFree(gpu_b);
cudaFree(gpu_c);
return 0;
}
```

# Advantages of Directive Based Parallelism

- Incremental parallel programming
  - Find hotspot, parallelize, check correctness, repeat
- Single source code for sequential and parallel programs
  - Use compiler flag to enable or disable
  - No major overwrite of the serial code
- Works for both CPU and GPU
- Low learning curve, familiar C/C++/Fortran program environment
  - Do not need to worry about lower level hardware details
- Simple programming model than lower level programming models
- Portable implementation:
  - Different architectures, different compilers handle the hardware differences

# Host $\Leftrightarrow$ Device Model

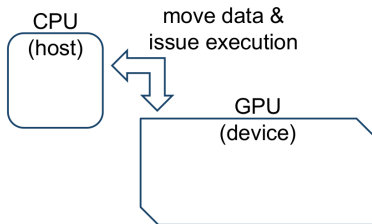


# Device Execution Model

- Device: An implementation-defined logical execution unit
- Can have a single host and one or more target devices (accelerators)
- Host and Device have separate data environment (except with managed memory or unified shared memory)
- The execution model is host-centric
  - Host creates/destroys data environment on the device(s)
  - Host maps data to the device data environment.
  - Host then offloads accelerator regions to the device for execution
  - Host updates the data between the host and the device.
  - Host destroys data environment on device.

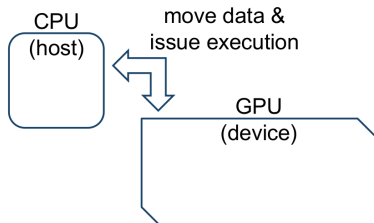


# Device Execution Model (OpenMP)

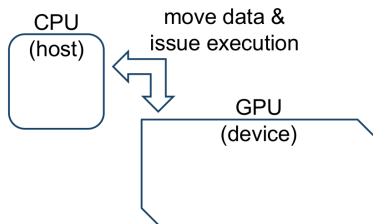


# Device Execution Model (OpenMP)

- The `target` construct offloads the enclosed code to the accelerator: single thread on a device (GPU)

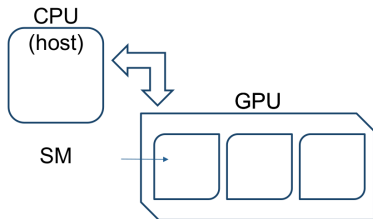


# Device Execution Model (OpenMP)



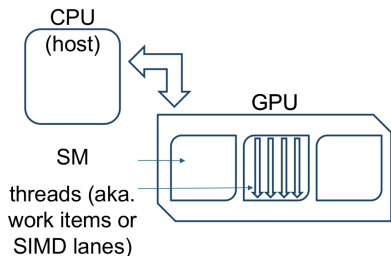
- The `target` construct offloads the enclosed code to the accelerator: single thread on a device (GPU)
- The `map` clause manages the data movements

# Device Execution Model (OpenMP)



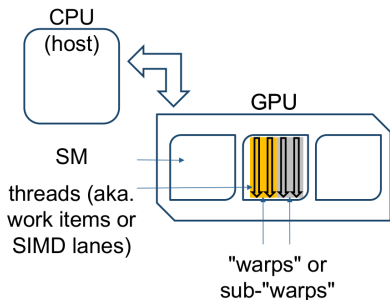
- The `target` construct offloads the enclosed code to the accelerator: single thread on a device (GPU)
- The `map` clause manages the data movements
- The `teams` construct creates a league of teams: one thread each, concurrent (not parallel) execution (on SMs - Stream Multiprocessors)

# Device Execution Model (OpenMP)



- The `target` construct offloads the enclosed code to the accelerator: single thread on a device (GPU)
- The `map` clause manages the data movements
- The `teams` construct creates a league of teams: one thread each, concurrent (not parallel) execution (on SMs - Stream Multiprocessors)
- The `parallel` construct creates a new team of threads: parallel execution (by sub-"warps")

# Device Execution Model (OpenMP)



- The `target` construct offloads the enclosed code to the accelerator: single thread on a device (GPU)
- The `map` clause manages the data movements
- The `teams` construct creates a league of teams: one thread each, concurrent (not parallel) execution (on SMs - Stream Multiprocessors)
- The `parallel` construct creates a new team of threads: parallel execution (by sub-"warps")
- The `simd` construct indicates SIMD execution is allowed: SIMD execution (among sub-"warps")

# GPU Programming with the OpenMP API

## OpenMP Webinar

### Intro to GPU Programming with the OpenMP API

[https://www.openmp.org/events/  
intro-to-gpu-programming-with-the-openmp-api/](https://www.openmp.org/events/intro-to-gpu-programming-with-the-openmp-api/)

# CUDA Interoperability

- OpenMP is a high-level language, sometimes low level optimizations will be necessary for best performance
- CUDA Kernels or Accelerated libraries for example
- The `use_device_ptr map` type allows OpenMP device arrays to be passed to CUDA or accelerated libraries
- The `is_device_ptr map` clause allows CUDA arrays to be used within OpenMP target regions



## Example of use\_device\_ptr

```
#pragma omp target data map(alloc:x[0:n]) map(from:y[0:n]) ←  
{  
    #pragma omp target teams distribute parallel for  
    for( i = 0; i < n; i++)  
    {  
        x[i] = 1.0f;  
        y[i] = 0.0f;  
    }  
  
    #pragma omp target data use_device_ptr(x,y) ←  
    {  
        cublasSaxpy(n, 2.0, x, 1, y, 1);  
    }  
}
```

Manage data  
movement using  
map clauses

Expose the device  
arrays to CUBLAS