

# Heterogenous Computing & Architectures: OpenACC model



Norwegian research infrastructure services

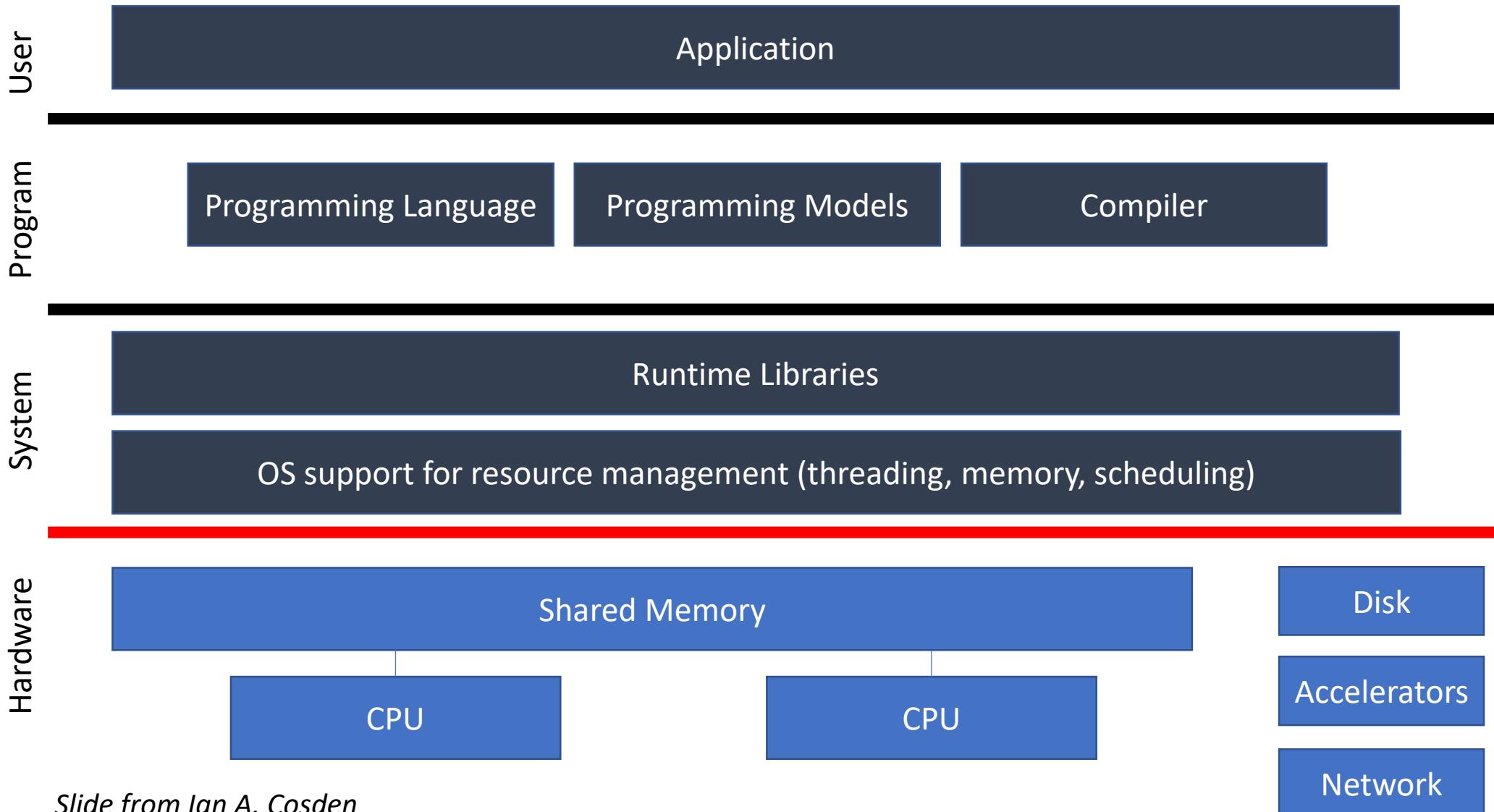
Hicham Agueny, PhD  
Scientific Computing Group  
IT-department, UiB

# Overview

- **Part I**
  - **CPU- & GPU-architectures**
  - **Programming models and supports**
  - **Functionality of OpenACC**
- **Part II**
  - **Synchronous OpenACC**
  - **Combining OpenACC with other models**
  - **Asynchronous OpenACC**
  - **Code-profiling**

# **Part I**

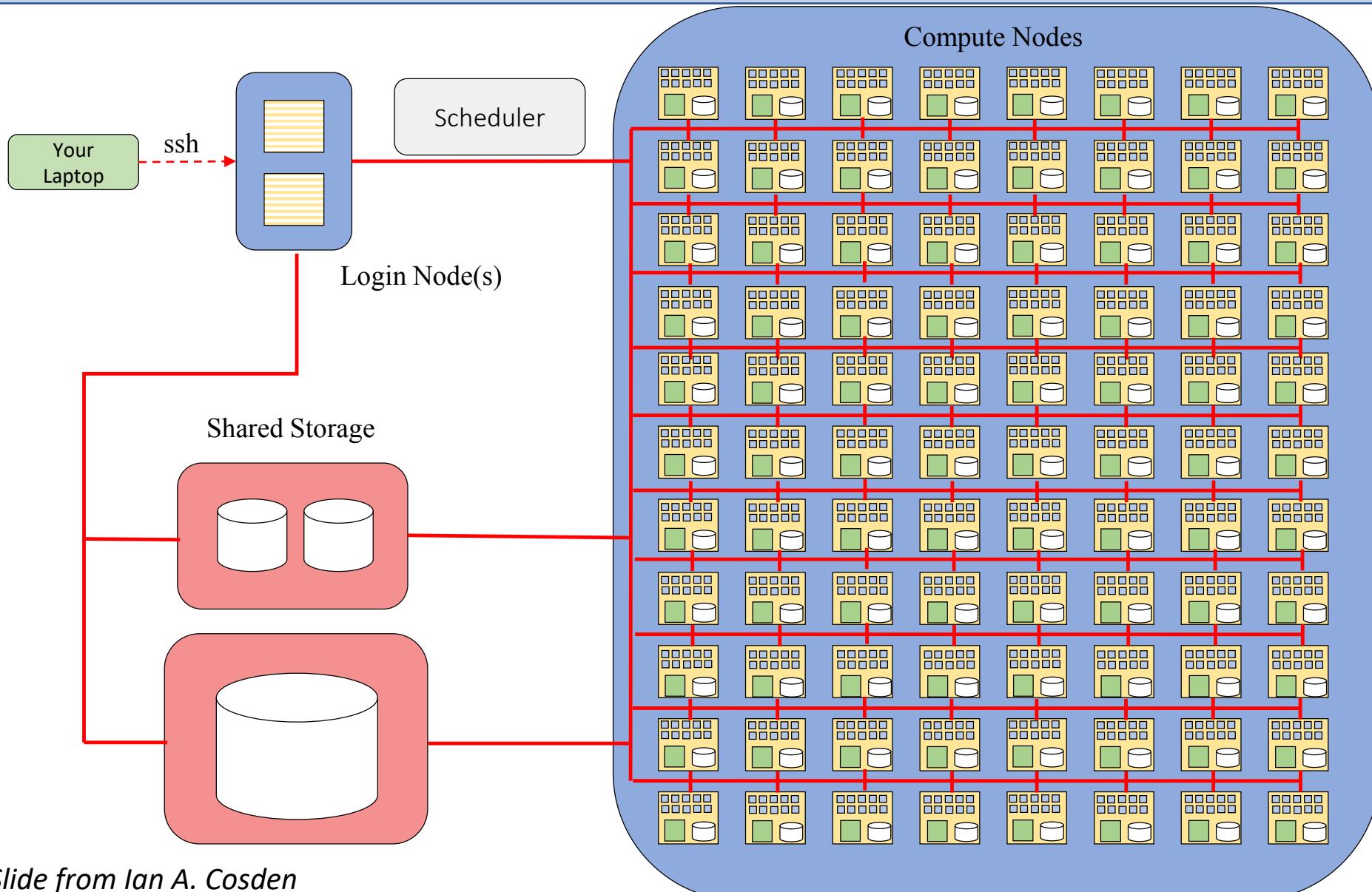
# Hardware-architecture



# **CPU-architecture**

**CPU:** Central Processing Unit

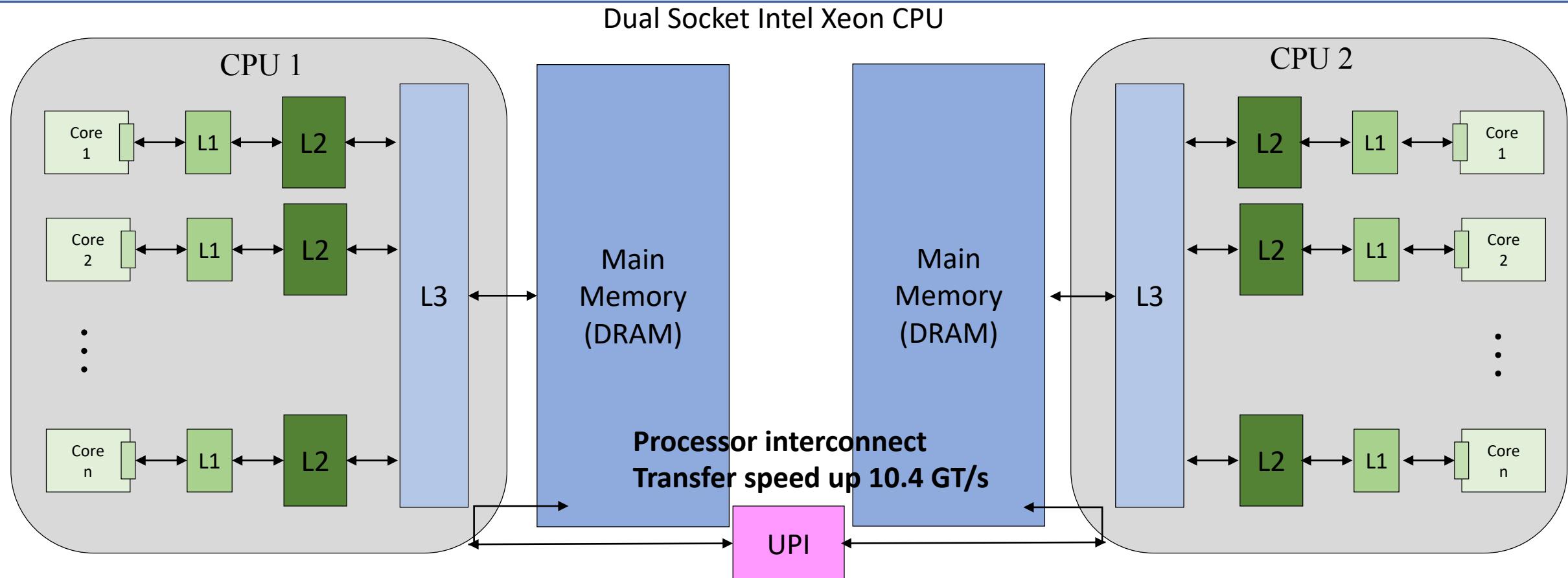
# Architecture of Cluster



Slide from Ian A. Cosden

[https://indico.cern.ch/event/814979/contributions/3401193/attachments/1831477/3105158/comp\\_arch\\_codas\\_2019.pdf](https://indico.cern.ch/event/814979/contributions/3401193/attachments/1831477/3105158/comp_arch_codas_2019.pdf)

# CPU-Architecture: ex. Betzy 128 cores



Access L1  
Memory is  
Faster than L2

	Registers	L1 Cache	L2 Cache	L3 Cache	DRAM	Disk
Speed	1 cycle	~4 cycles	~10 cycles	~30 cycles	~200 cycles	10ms
Size	< KB per core	~32 KB per core	~256 KB per core	~35 MB per socket	~100 GB per socket	TB

# **GPU-architecture**

**GPU:** Graphics Processing Unit

# Architecture of NVIDIA GPU devices



Figure 1. NVIDIA Tesla V100 SXM2 Module with Volta GV100 GPU



Figure 2. Volta GV100 full GPU with 84 SM Units

# Architecture of NVIDIA GPU devices



Figure 1. NVIDIA Tesla V100 SXM2 Module with Volta GV100 GPU

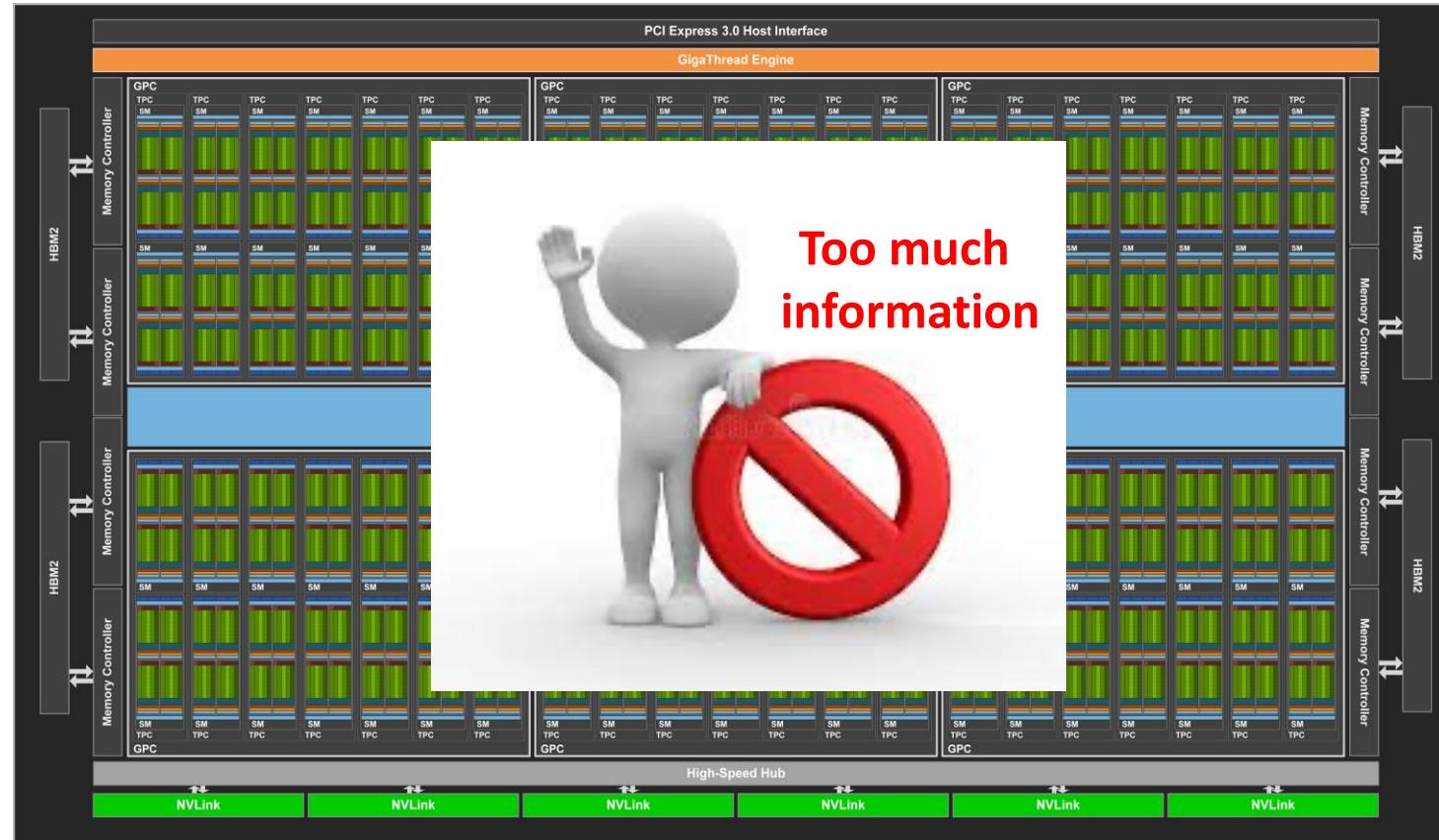
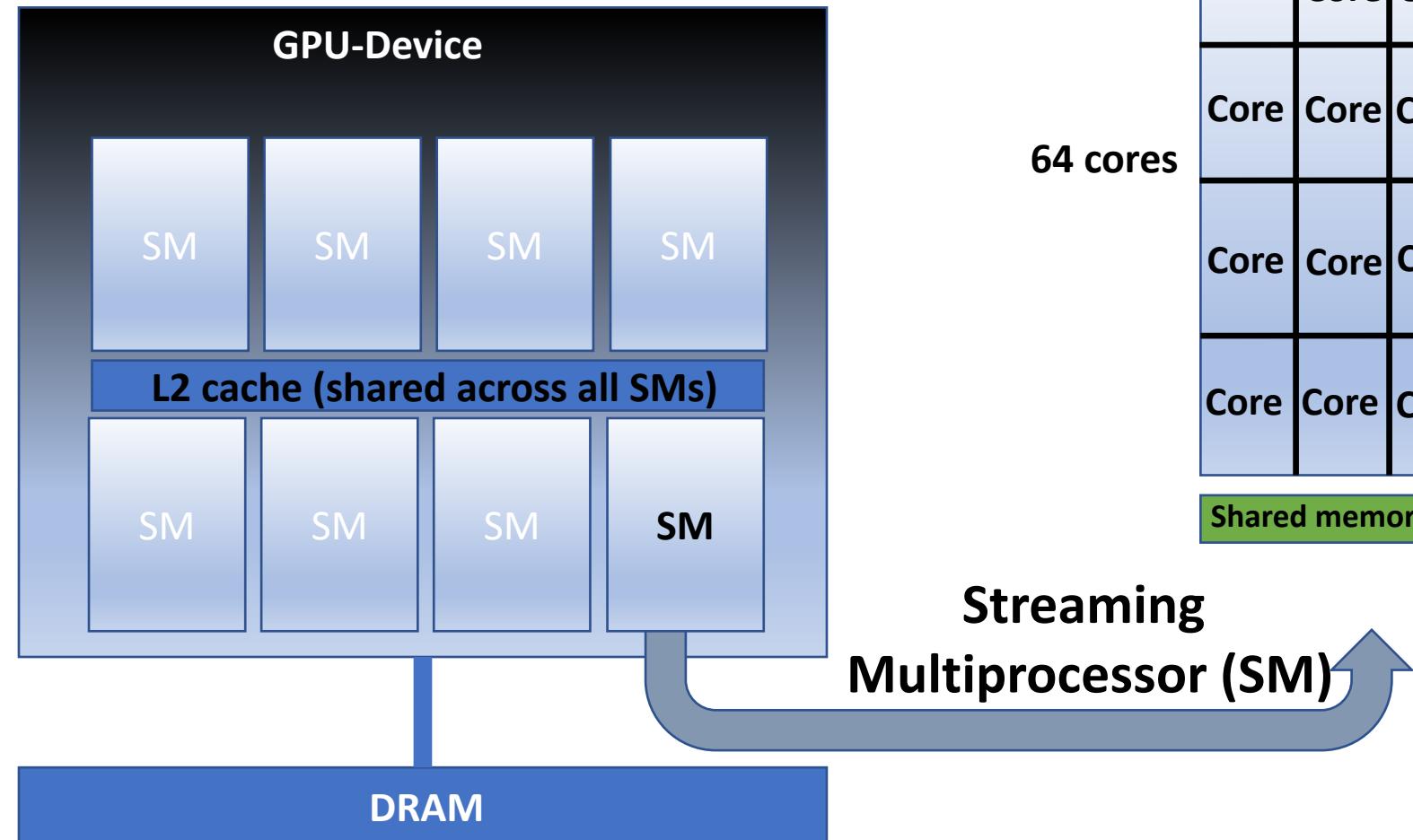


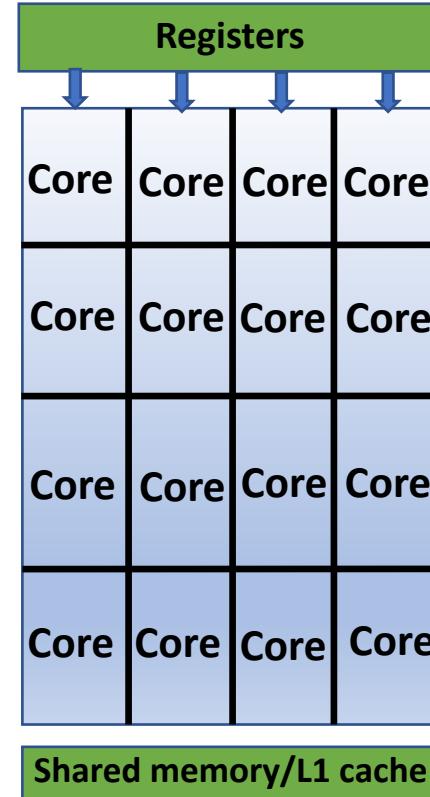
Figure 2. Volta GV100 full GPU with 84 SM Units

# Architecture of NVIDIA-GPU devices (simplified version)

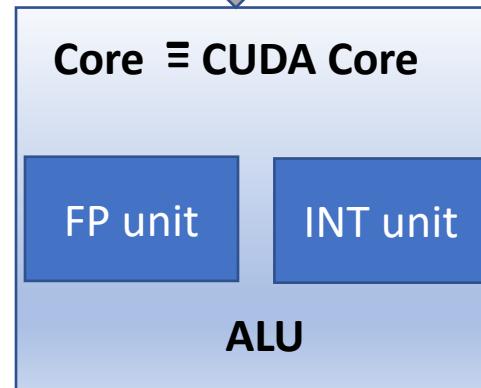


64 cores

Streaming  
Multiprocessor (SM)



Data are stored temporarily. Registers are private to each thread.



Core ≡ CUDA Core

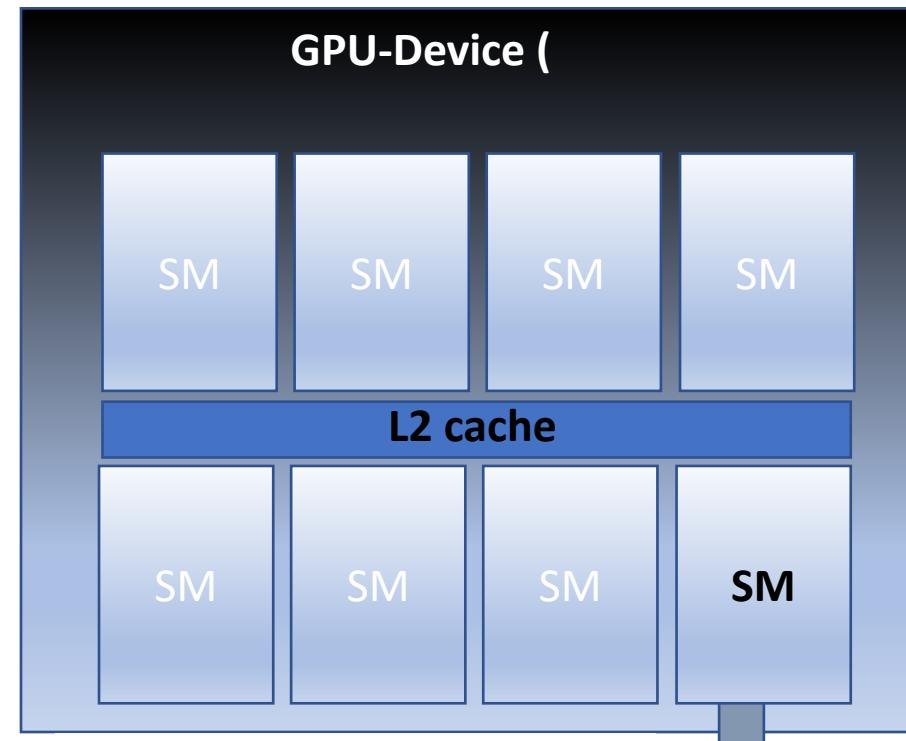
FP unit      INT unit

ALU

All threads share L1 cache.

- Core includes ALU. Various logical operations for computations (FP32, FP64), e.g. addition, multiplication and ...
- ALU executes SIMD instructions. (processing multiple data with the same operation concurrently (in parallel))

# Architecture of NVIDIA-GPU devices (simplified version)

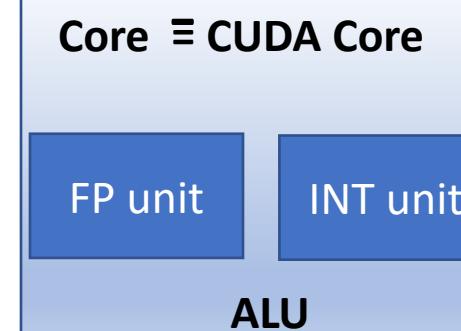
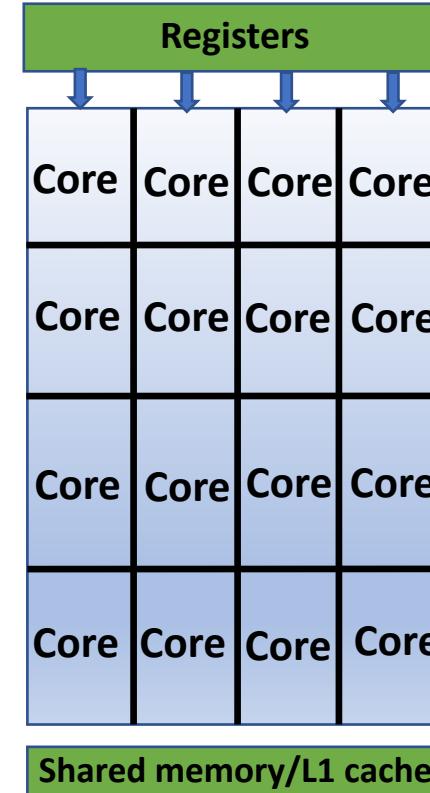


Streaming  
Multiprocessor (SM)

For Tesla A100 GPU: 108 SMs, each SM has:

64 FP32 cores ===== A total of 6912 FP32 CUDA cores

64 INT32 cores ===== A total of 6912 INT32 CUDA cores.



Core ≡ CUDA Core

FP unit      INT unit

ALU



NVIDIA Tesla P100



NVIDIA Tesla V100

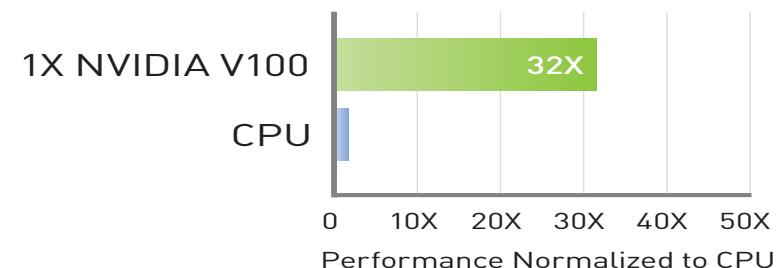


NVIDIA Tesla A100

Architecture	Tesla P100 (Pascal)	Tesla GV100 (Volta)	Tesla A100
SMs	56	84	108
NVIDIA CUDA cores	3584	5376	6912
Tensor cores/GPU	NA	672	432
Peak performance	9.3 TFLOPS	15.7 TFLOPS	39 TFLOPS
Transistors	15.3 billion	21.1 billion	54.2 billion
GPU die size	610 mm <sup>2</sup>	815 mm <sup>2</sup>	826 mm <sup>2</sup>

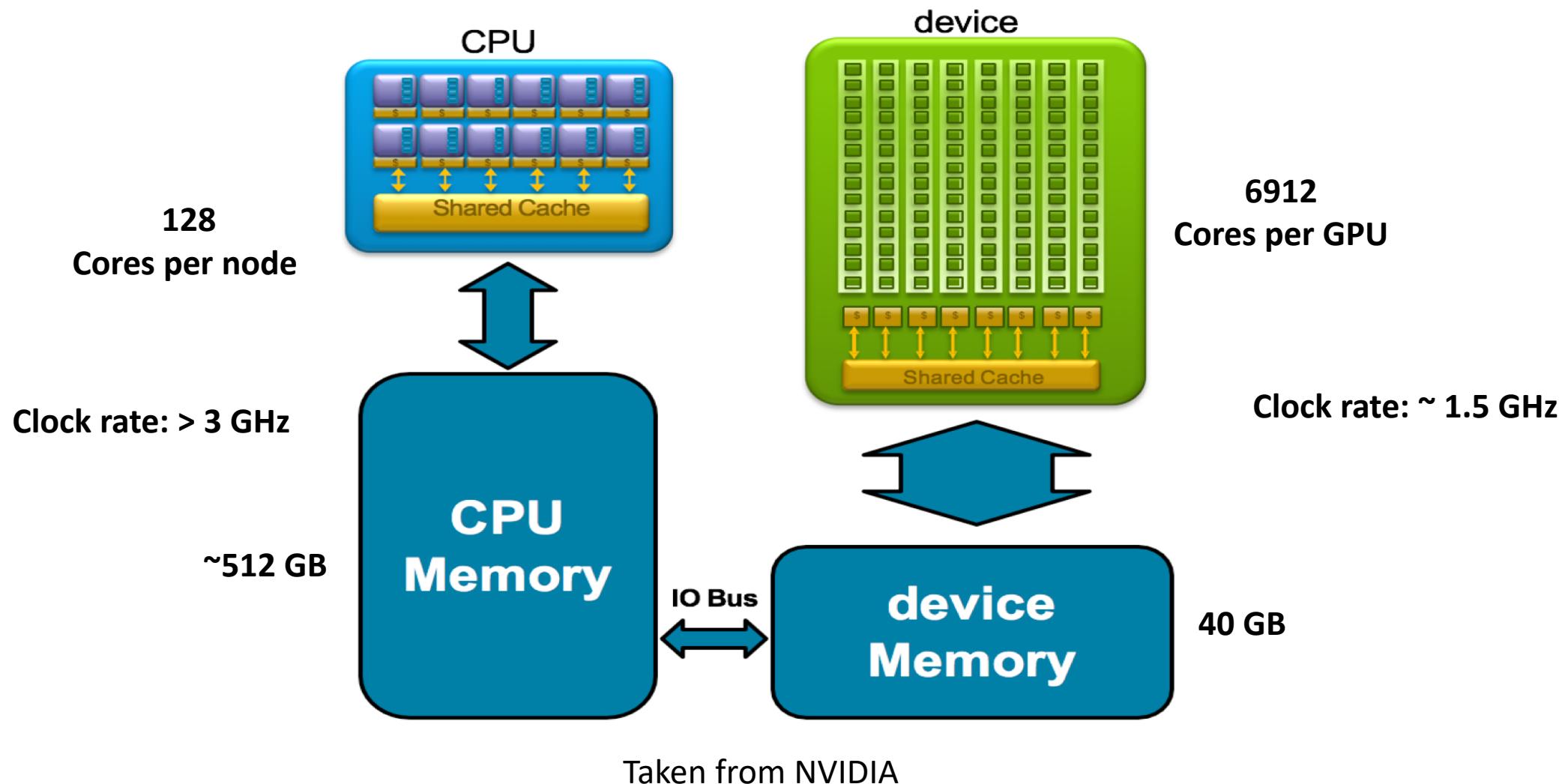


32X Faster Training Throughput  
than a CPU<sup>1</sup>



GPU-Dimension (cm)  
26.7x11x4

# Conclusion: CPU vs GPU



# **Heterogenous programming models**

# Heterogenous programming models

Programming models can be hardware-dependent or hardware-independent

Directive based models:

High-level models

OpenACC

OpenMP

Compiler supports

NVHPC

GCC

Cray (only  
acc .2.0 F)

Clang

Cray

GCC

Icx (Intel)

Hardware

NVIDIA

NVIDIA

NVIDIA

AMD

AMD

AMD

Intel

Low level offload models:

CUDA

Only on NVIDIA

OpenCL

HIP

Only on AMD

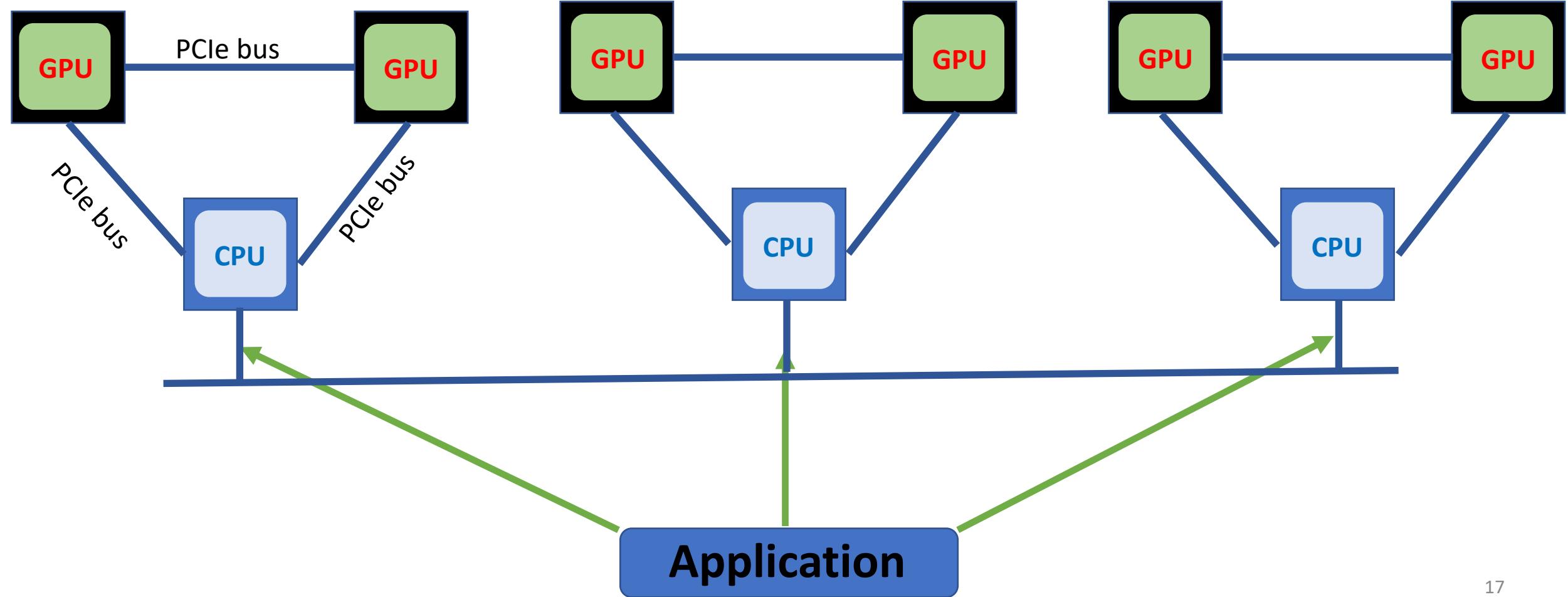
Intel GPU: OpenCL  
can be migrated via  
SYCL API migrating  
from CUDA to  
DPC++. DPC++ tool  
is part of the Intel  
OneAPI Toolkit.

Hybrid multi-GPU programming: Combining MPI/OpenMP threading & OpenACC/OpenMP offloading

# Multi-GPU programming

MPI+GPU programming model

GPU-to-GPU communication



# **Functionality of OpenACC**

## **What is OpenACC?**

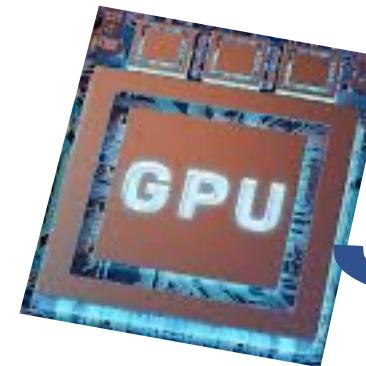
OpenACC is a directive-based approach similar to OpenMP.

It is designed to simplify parallel programming between heterogenous systems CPU/GPU.

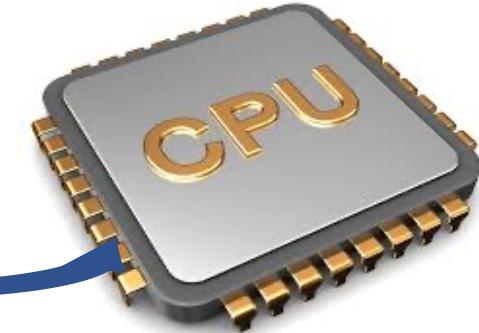
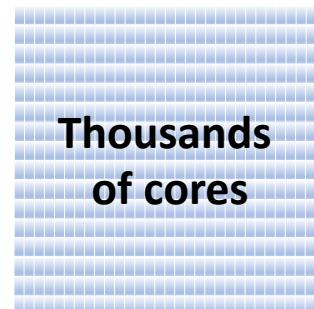
It is API (Application Programming Interface) to communicate between CPU and GPU.

# Parallelism in OpenACC

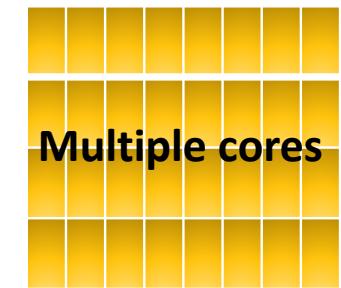
## Heterogenous computing



GPU-Device



CPU-Host



**Programming model**  
To offload a code region  
to a GPU-device

```
do i=1,n  
  do j=1,m  
    A(i,j) = B(i,j) + C(i,j)  
  enddo  
enddo
```

# Parallelism in OpenACC

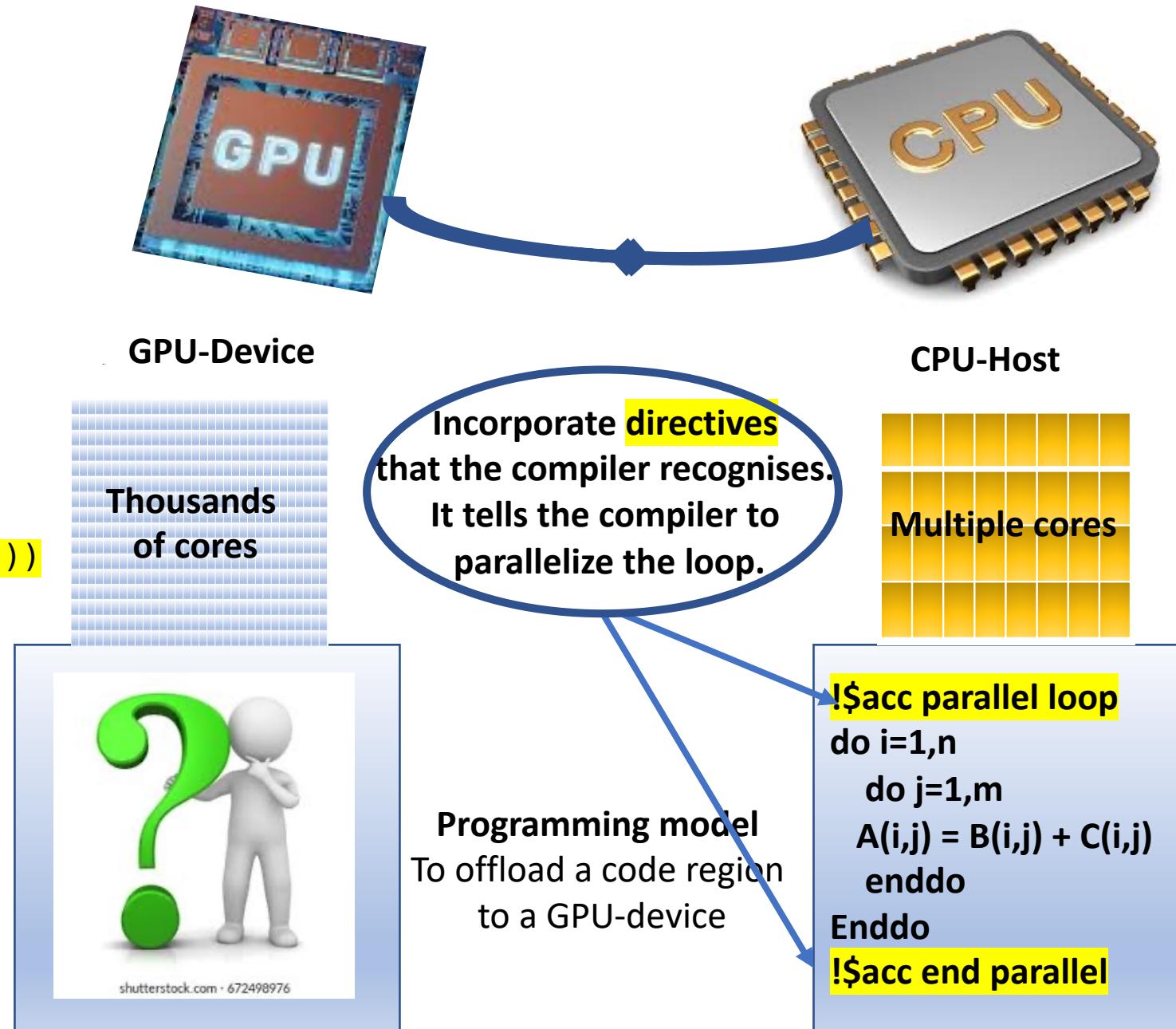
It uses CUDA to exploit parallelism.

## Heterogenous computing

Focus on how parallelism is done.

### Message from the compiler

```
10, Generating Tesla code
11, !$acc loop gang, vector(32), worker(4)
! blockidx%x threadidx%x threadidx%y
12, !$acc loop seq
10, Generating implicit copyin(B(:,:,),C(:,:,))
[if not already present]
Generating implicit copyout(A(:,:,))
[if not already present]
12, Loop is parallelizable
```

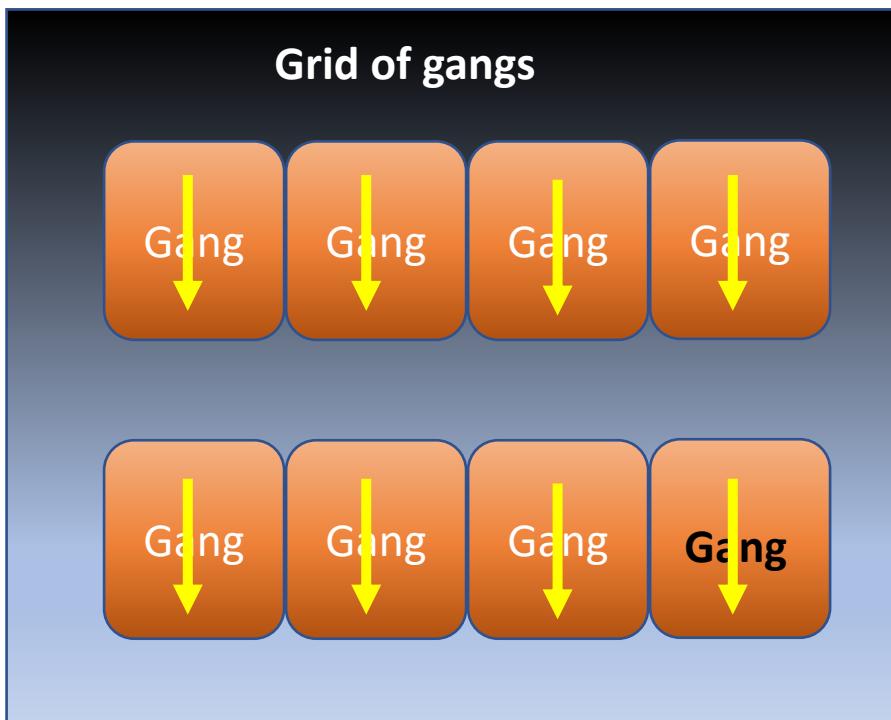


# Parallelism in OpenACC

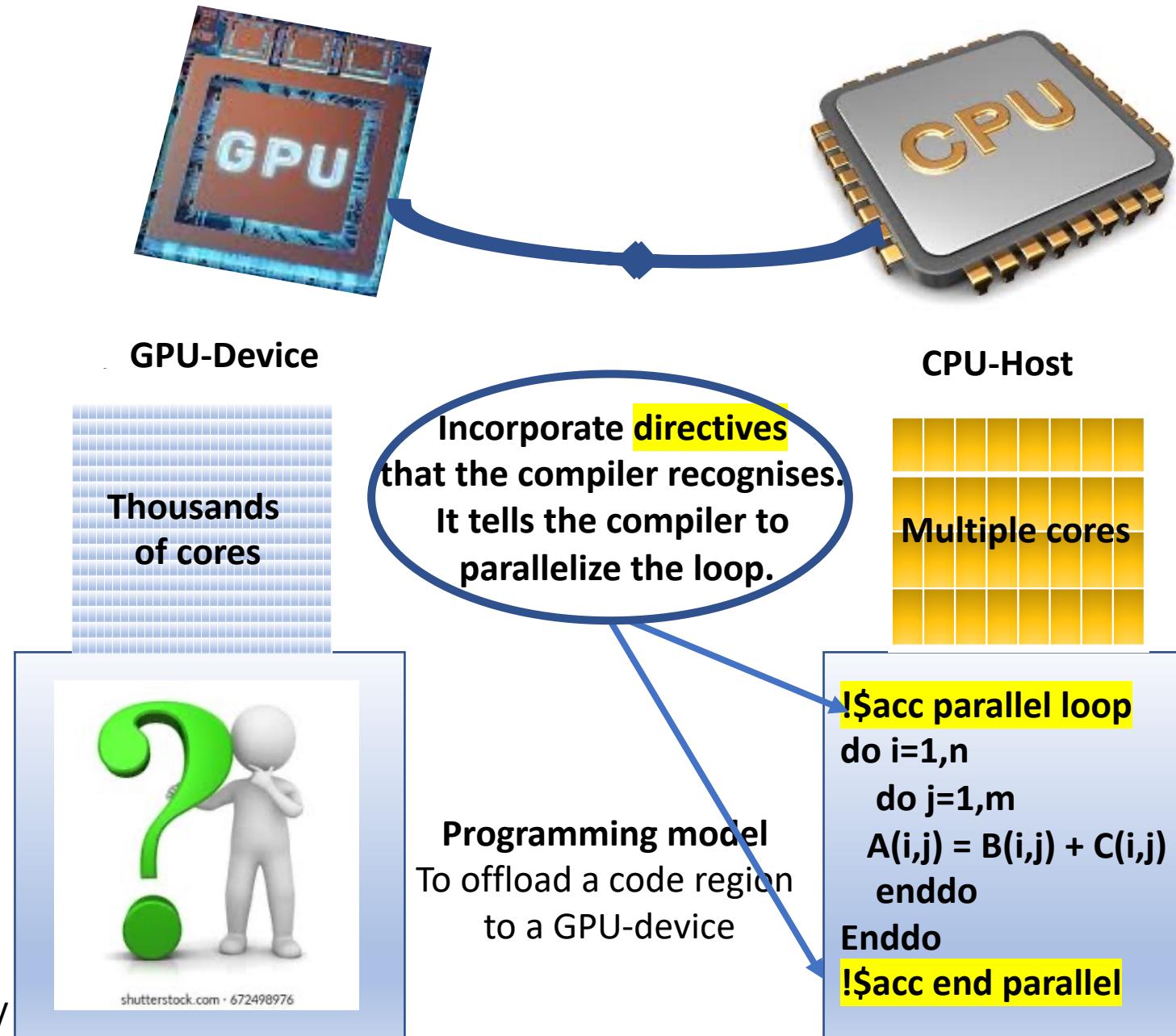
## Heterogenous computing

### Software scheme

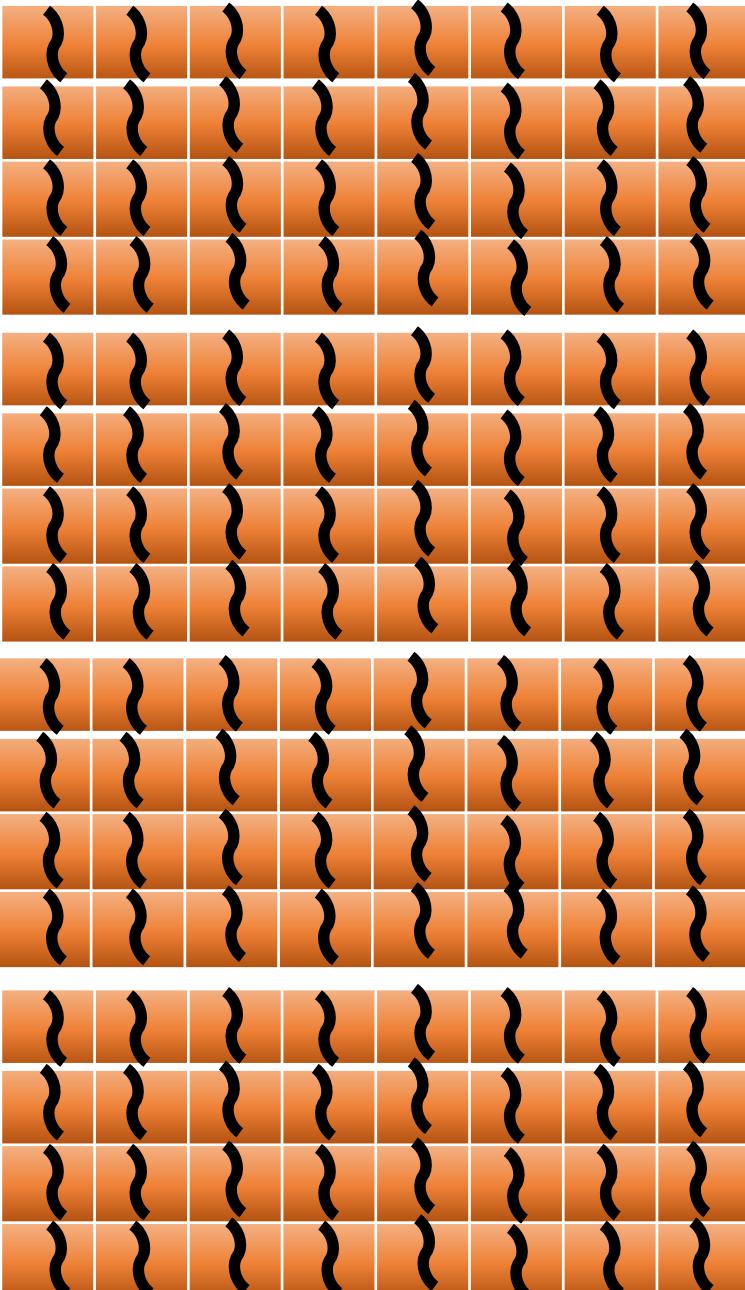
Only the “parallel” construct



**Step 1:** The compiler generates one or more gangs.  
**Step 2:** The offloaded loop gets executed redundantly on each gang (get duplicated).

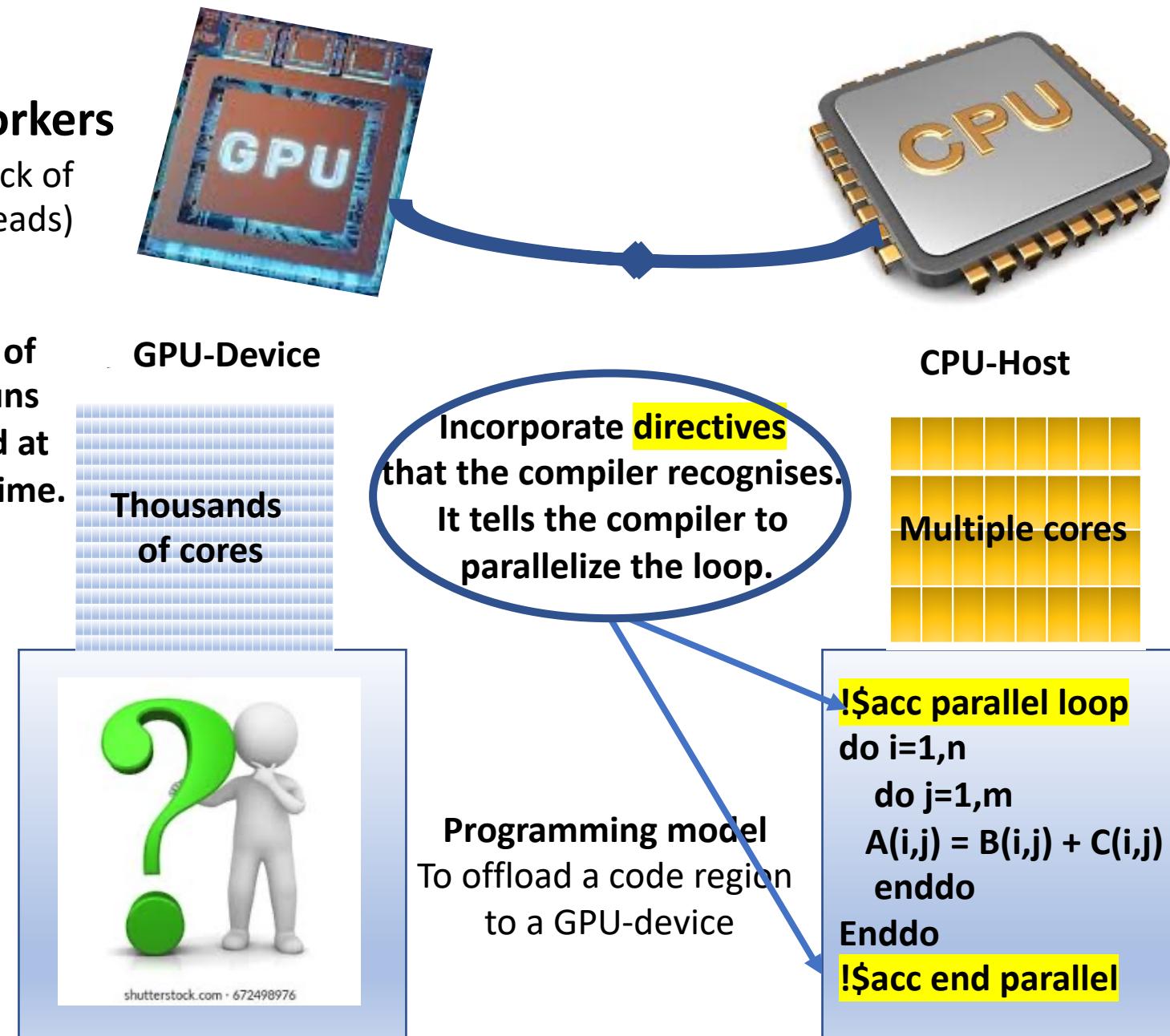


# Parallelism in OpenACC



Each piece of the loop runs by a thread at the same time.

# Heterogenous computing



# Parallelism in OpenACC (it uses CUDA to exploit the parallelism)

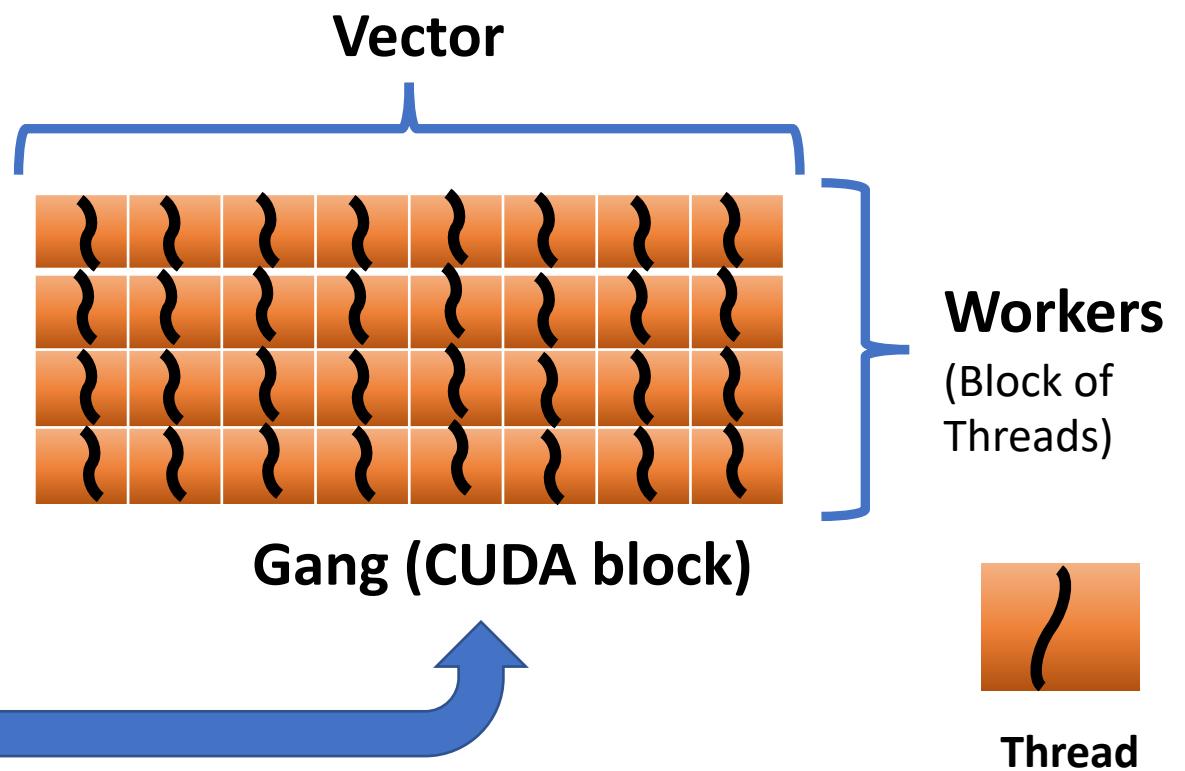
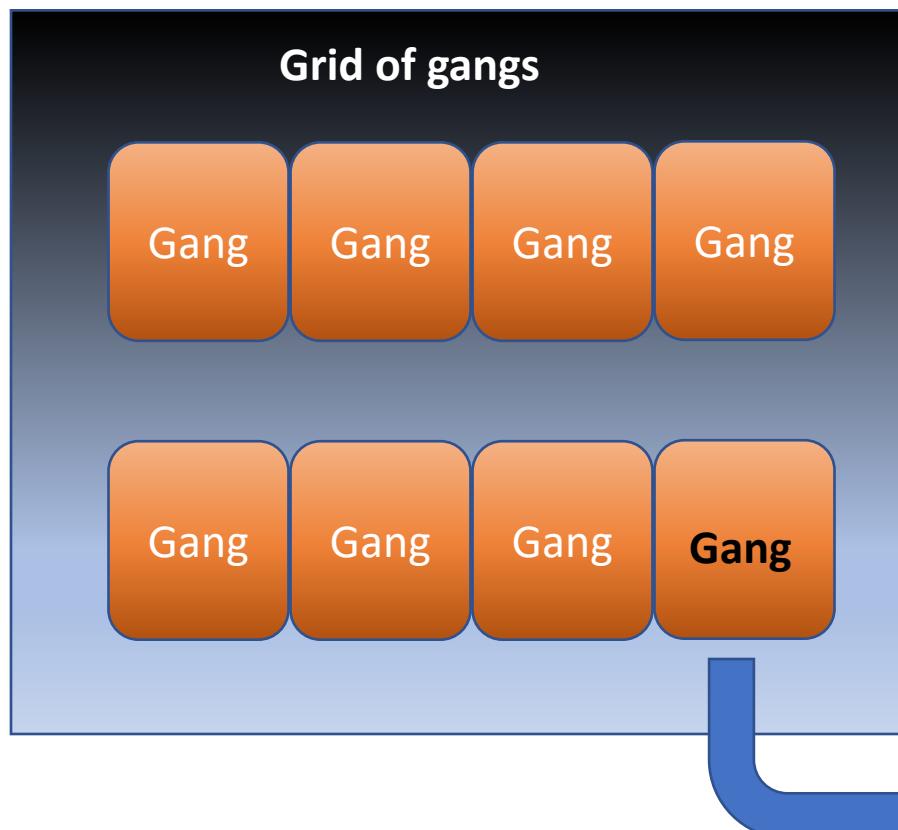
The parallelism in OpenACC is based on three concepts:

- **Gang**: Consists of a set of workers (block of thread)
- **Worker**: threads.
- **Vector**: It tells the OpenACC that it can exploit *vector parallelism*

Each thread has an index: it is used for calculating memory address locations.

vectorization naturally occurs in hardware across a **group of threads** referred to as a **warp (32 threads)** at the **same time**.

## Software scheme



Thread

Mapping a matrix into a device

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Each thread executes a different piece  
Of data item (data parallelism).

Each CUDA block has 1024 threads  
(32 warps. A warp=collection of 32 threads  
are executed simultaneously by a SM)

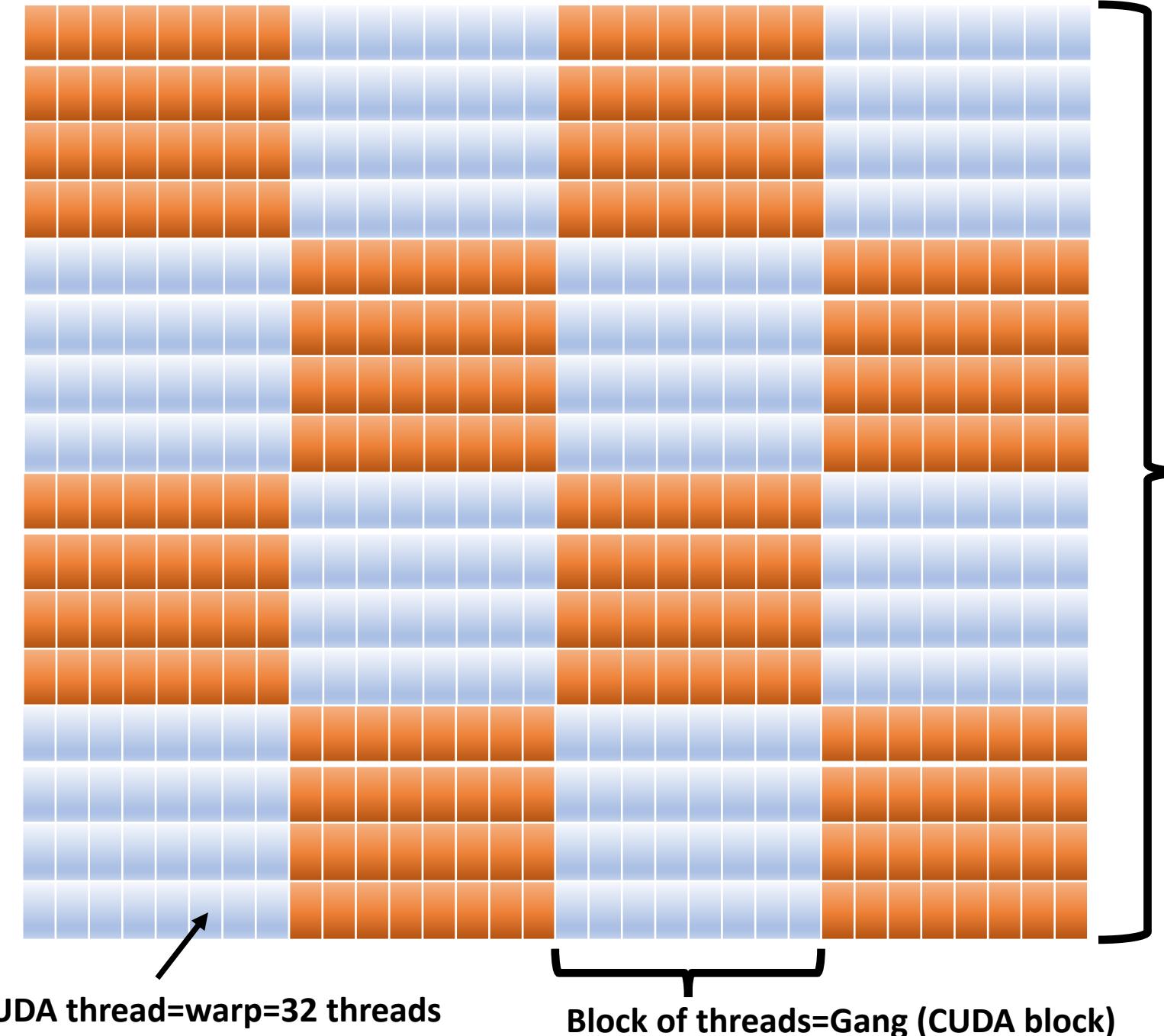
Execution on hardware

CUDA thread

CUDA block

CUDA grid

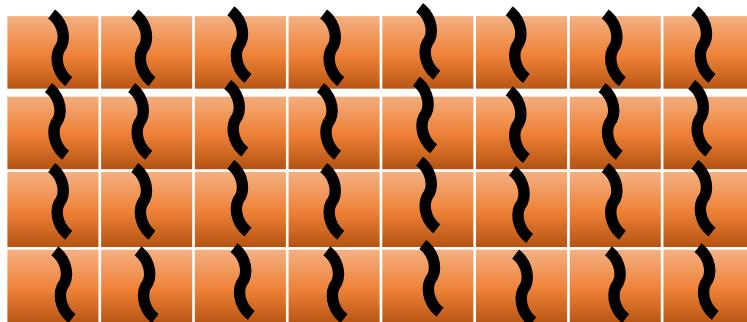
Grid of threads



# Execution on GPU

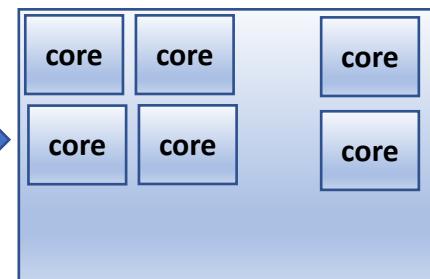
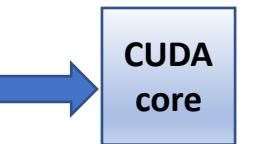
## Software scheme

## Hardware scheme



CUDA thread block

**is executed on**

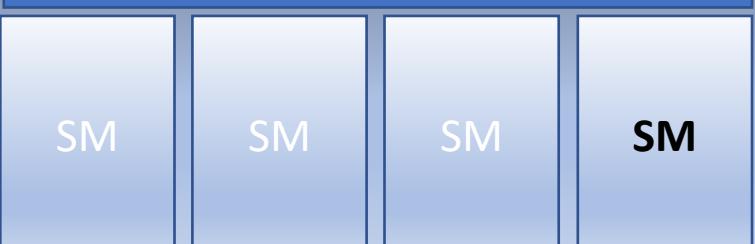
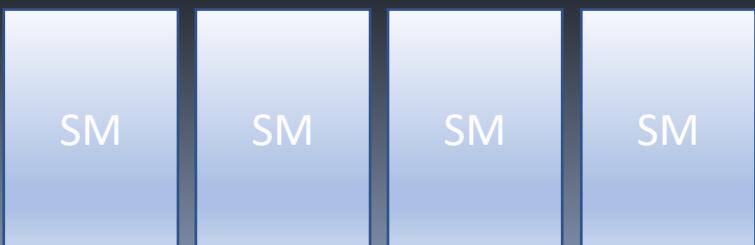


Streaming  
Multiprocessor  
(SM)

Grid of gangs  
(CUDA kernel grid)



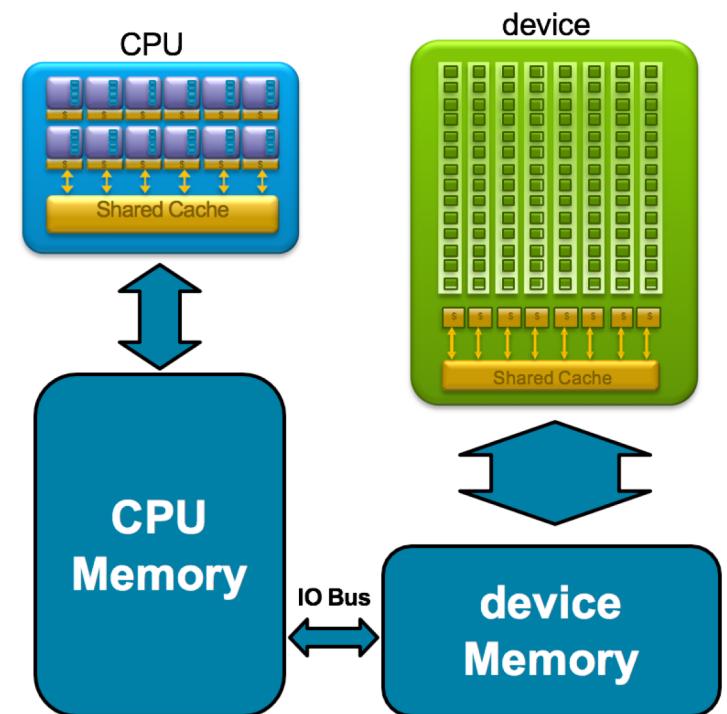
GPU-Device



# Conclusion

The parallelism in OpenACC is based on introducing three basic steps:

- Step 1: The compiler generates one or more gangs.
- Step 2: The offloaded loop gets executed redundantly on each gang (get duplicated).
- Step 3: A block of thread will execute different pieces of the offloaded loop.
- ***vector parallelism***: to perform multiple data-parallel operations at the same time.
- vectorization naturally occurs in hardware across a group of threads referred to as a warp at the same time.
- Data transfer between CPU and GPU is time consuming.
- The performance of GPU can be slower than CPU in some cases.
- Bottleneck in GPU-programming: Portability and hardware limitation.
- Converting codes affects the performance.
- Knowledge of GPU-architecture is important for GPU-programming.



Taken from NVIDIA

# **Part II**

# Overview

- **Synchronous OpenACC**
  - Various directives and clauses:
    - Parallel loop vs kernels
    - gang, worker, vector, num\_workers(), vector\_length()
    - Kernels, Reduction, collapse, routine clauses
    - Data locality (structured and unstructured data)
    - Update device/self
    - Atomic operations
    - Host\_data use\_device()
    - Deviceptr()
- **Asynchronous OpenACC**
  - Async, wait
  - Offloading to multiple GPUs
- **Code-profiling (Gprof, Gcov & Nsight Systems)**

# Synchronous OpenACC

# Parallelism in OpenACC: parallel loop vs kernels

OpenACC uses two different approaches (directives) for exposing parallelism:

**Parallel loop** construct and **Kernels** construct.

The compiler performs the parallelism of a specified loop and maps it into a GPU-device.

**Fortran:**

```
!$ acc parallel loop
```

```
!$ acc kernels
```

**Ex.**

**C/C++:**

```
#pragma acc parallel loop
```

```
#pragma acc kernels
```

**Differences: parallel loop vs kernels:**

**Parallel loop construct:** The programmer has more control on the parallelism (e.g. adding more clauses).

```
!$ acc parallel loop  
do j=1,Nx  
  do i=1,Ny  
    A(i,j) = B(i,j) + C(i,j)  
  enddo  
enddo  
!$ acc end parallel
```

```
!$ acc kernels  
do j=1,Nx  
  do i=1,Ny  
    A(i,j) = B(i,j) + C(i,j)  
  enddo  
enddo  
!$ acc end kernels
```

**Kernels construct:** The compiler has more flexibility (generation of efficient parallel codes) and responsibility (safety in parallelising loops).

**Portability**

**Ex.** combining loops into a single parallel kernel is included in **kernels** construct.

# Message from the OpenACC compiler

## Parallel loop construct

9, Generating Tesla code

```
10, !$acc loop gang ! blockidx%x  
11, !$acc loop vector(128) ! threadidx%x
```

4 workers (warps) each  
with 32 threads

9, Generating implicit `copyin(b(:,:,),c(:,:,))` [if not already present]

Generating implicit `copyout(a(:,:,))` [if not already present]

11, Loop is parallelizable

```
9  !$ acc parallel loop  
10 do j=1,Nx  
11   do i=1,Ny  
      A(i,j) = B(i,j) + C(i,j)  
      enddo  
    enddo  
  !$ acc end parallel
```

```
9  !$ acc kernels  
10 do j=1,Nx  
11   do i=1,Ny  
      A(i,j) = B(i,j) + C(i,j)  
      enddo  
    enddo  
  !$ acc end kernels
```

## Kernels construct

9, Generating implicit `copyin(b(:,:,),c(:,:,))` [if not already present]  
Generating implicit `copyout(a(:,:,))` [if not already present]

10, Loop is parallelizable

11, Loop is parallelizable

Generating Tesla code

```
10, !$acc loop gang, vector(128) collapse(2) !blockidx%x threadidx%x  
      collapsed-innermost
```

```
11, ! blockidx%x threadidx%x auto-collapsed
```

# Another way of specifying the parallel loop

- **Gang**: Executes loops in parallel across *num\_gangs* gangs.
- **Worker**: Executes loops in parallel across *num\_workers* workers of a single gang.
- **Vector**: Executes loops in SIMD or vector mode, with a maximum *vector\_length*.

```
!$ acc parallel loop
do j=1,Nx
    do i=1,Ny
        A(i,j) = B(i,j) + C(i,j)
    enddo
enddo
!$ acc end parallel
```

```
!$ acc parallel loop gang worker vector
do j=1,Nx
    do i=1,Ny
        A(i,j) = B(i,j) + C(i,j)
    enddo
enddo
!$ acc end parallel
```

- **num\_workers( *N* )**  
Controls how many workers are created in each gang.

- **vector\_length( *N* )**  
Controls the vector length on each worker.

```
!$ acc parallel loop gang worker num_workers(32) vector_length(32)
do j=1,Nx
    do i=1,Ny
        A(i,j) = B(i,j) + C(i,j)
    enddo
enddo
!$ acc end parallel
```

Fig taken from NVIDIA  
Speed-up Varying Number of Workers

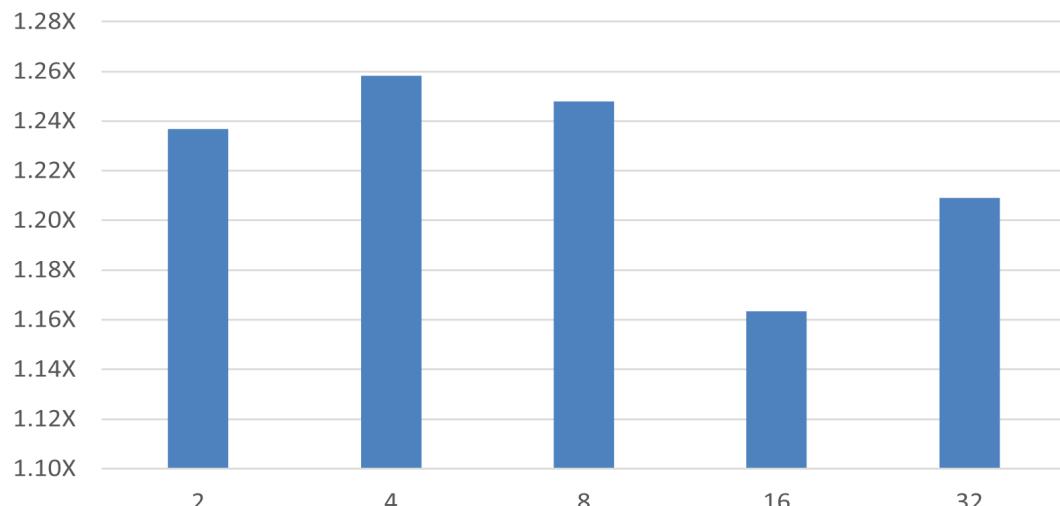


Figure 5.3: Speed-up from varying number of workers for a vector length of 32.

# Reduction clause

Reduction(Operator:Val)

Operator: max,min,+,\*... The syntax is valid for C/C++ and fortran.

Val can be a variable or array.

The **reduction** clause implies copying data back and forth between a CPU-host and a GPU-device.

In C/C++: `#pragma acc parallel loop reduction(operator:variable)`

In Fortran: `!$acc parallel loop reduction(operator:variable)`

```
!$acc parallel loop reduction(max:max_err)
  do j=1,ny
    do i=1,nx
      max_err = max(abs(fnew(i,j) - fold(i,j)),max_err)
    enddo
  enddo
 !$acc end parallel
```

```
!$acc parallel loop reduction(max:max_err)
  do j=1,ny
    !$acc loop reduction(max:max_err)
      do i=1,nx
        max_err = max(abs(fnew(i,j) - fold(i,j)),max_err)
      enddo
    enddo
  !$acc end parallel
```

# Reduction clause

Reduction(Operator:Val)

Operator: max,min,+,\*... The syntax is valid for C/C++ and fortran.

Val can be a variable or array.

The **reduction** clause implies copying data back and forth between a CPU-host and a GPU-device.

In C/C++: `#pragma acc parallel loop reduction(operator:variable)`

In Fortran: `!$acc parallel loop reduction(operator:variable)`

**max**

```
!$acc parallel loop reduction(max:max_err)
do j=1,ny
  do i=1,nx
    max_err = max(abs(fnew(i,j) - fold(i,j)),max_err)
  enddo
enddo
 !$acc end parallel loop
```

**sum**

```
total=0.
!$acc parallel loop reduction(+:total)
do i=1,nx
  total = total + data(i)
enddo
 !$acc end parallel loop
```

# Collapse clause

**Collapse(N):** transforms N tightly nested loops to a single loop (to improve the performance).

N corresponds to the number of nested loops to be collapsed to a single loop.

The total number of counts or elements involved in the loops remains the same.

In C/C++: **#pragma acc parallel loop collapse(N)**

In Fortran: **!\$acc parallel loop collapse(N)**

```
!$acc parallel loop collapse(2)
do j=1,ny
do i=1,nx
.....
enddo
enddo
!$acc end parallel
```

**!\$acc parallel loop collapse(2)**

```
do j=1,ny
do i=1,nx
!$acc loop collapse(3)
do k1=1,nz1
do j1=1,ny1
do i1=1,nx1
.....
```

```
enddo
enddo
enddo
enddo
enddo
!$acc end parallel
```

# Combining reduction and collapse clauses

In C/C++: `#pragma acc parallel loop reduction(operator:variable) collapse(N)`

In Fortran: `!$acc parallel loop reduction(operator:variable) collapse(N)`

## Parallel loop clause

```
!$acc parallel loop reduction(max:max_err) collapse(2)
```

```
do j=1,ny  
  do i=1,nx
```

```
    max_err = max(dabs(fnew(i,j) - fold(i,j)),max_err)
```

```
  enddo  
enddo
```

```
!$acc end parallel
```

## Kernels clause

~~```
!$acc kernels reduction(max:max_err) collapse(2)
```~~~~```
do j=1,ny  
  do i=1,nx
```~~~~```
    max_err = max(dabs(fnew(i,j) - fold(i,j)),max_err)
```~~~~```
  enddo  
enddo
```~~~~```
!$acc end kernels
```~~

Note: The **reduction** and **collapse** clauses are not allowed in the **kernels** clause

### Error messages:

NVFORTTRAN-S-0533-Clause 'REDUCTION' not allowed in ACC KERNELS

NVFORTTRAN-S-0533-Clause 'COLLAPSE' not allowed in ACC KERNELS

# Routine directive

The **routine** clause is used for calling functions or subroutines from a region specified by an OpenACC directive.

The **routine** clause must be added to all functions or subroutines underlying a called ones from a parallel loop.

The clause is referred to as **routine seq (i.e. sequential routine)** as it is called by each iteration within a parallel loop.

The clause must be located **below** the corresponding function or subroutine.

In C/C++: `#pragma acc routine seq`

In Fortran: `!$acc routine seq`

```
!$acc parallel loop reduction(max:max_err)
do j=1,ny
  do i=1,nx
    .....
    call host_fnew(fnew,fold)
    max_err = max(dabs(fnew(i,j) - fold(i,j)),max_err)
  enddo
enddo
 !$acc end parallel
```

`subroutine host_fnew(fnew,fold)`

`!$acc routine seq`

`implicit none`

.....

.....

`end subroutine`

# Data management (data locality): Data directive

Data movement between a **CPU-host** and a **GPU-device**.

The **data** directive permits to control and **optimize** memory placement between a host and a device.

The **data** directive permits sharing data between multiple **parallel blocks of loops** within a **data region**.

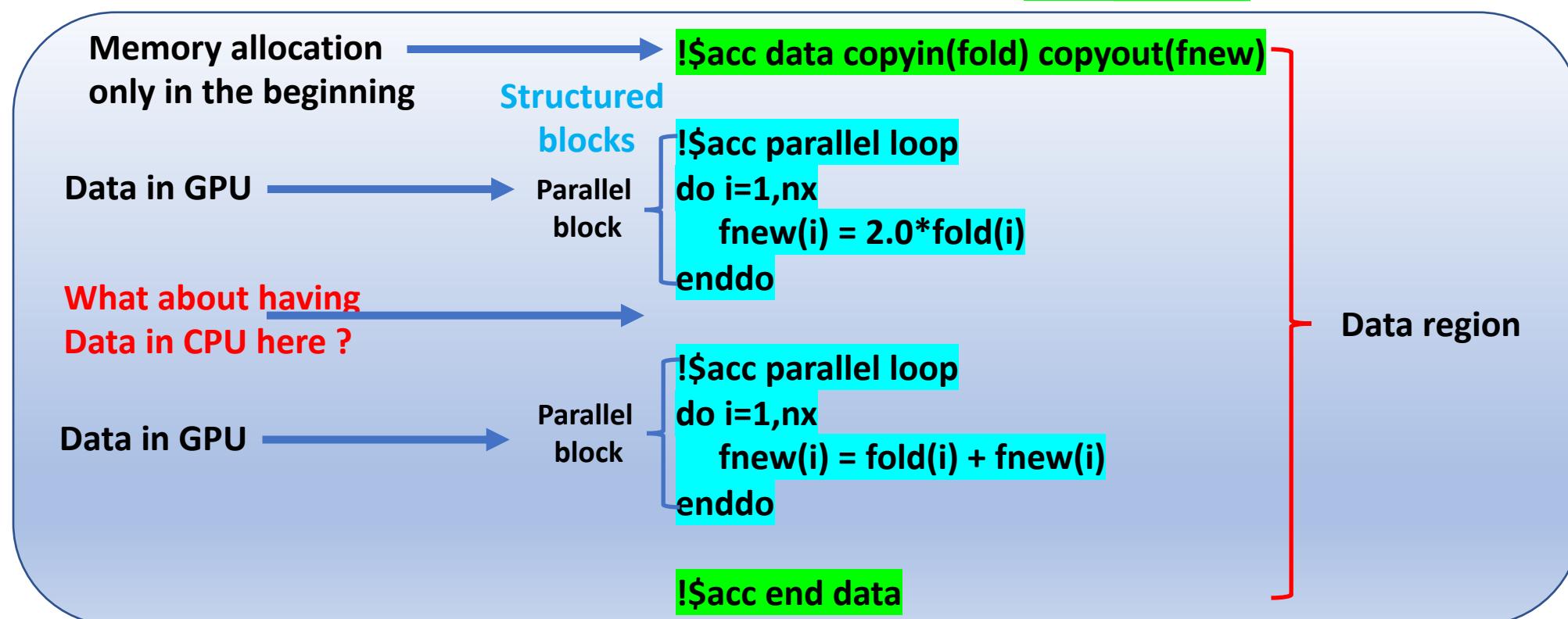
Two types of data: **structured** and **unstructured** data.

**Structured data:** Data remain in a device during the dynamics within a data region. **Offloading ONLY in the beginning.**

In C/C++: `#pragma acc data copy(array[0:n])`

In Fortran: `!$acc data copy(array)`

`!$acc end data`



# Data management: Update directive

The **update** directive permits to synchronize the change of data between CPU-host and GPU-device.  
Two classes of the **update** directives can be distinguished.

The **update device** directive permits to update the change of data in a device by **copying them from a CPU-host to a GPU-device**.  
The **update self** directive permits to update data in a host by **copying them from a GPU-device to a CPU-host**.

In C/C++: `#pragma acc update device(array[0:N])`

`#pragma acc update self(array[0:N])`

In Fortran: `!$acc update device (array)`

`!$acc update self(array)`

`!$acc data copyin(fold) copyout(fnew)`

`subroutine host_fnew(fnew)`

Compute the array **fnew** from GPU-device

`!$acc update self(fnew) ! GPU → CPU`

`!$acc parallel loop`

`do i=1,nx`

`do j=1,ny`

`fnew(i,j) = 3*fold(i,j)`

`enddo`

`enddo`

`end subroutine`

Now **fnew** can be used from the host  
ex. Print **fnew** in a file or call a subroutine

`call host_fnew(fnew)`

`!$acc update device(fnew) !CPU → GPU`  
`!$acc end data`

# Data management: Data clauses

## Data clauses:

**copy(array)** -- allocates space on a device and copy data to a device and back to a host at the end of a data region.

**copyin(array)** -- allocates space on a device and copy data to a device at the beginning of a data region.

**copyout(array)** -- allocates space on a device and copy data back to the host at the end of a data region.

**create(array)** -- allocates space in a device, no copy to or from the device/host.

**present(array)** – tells the compiler that no data movement is required.

**delete(array)** –removes data from a device. It is only used with **exit data clause**.

**Note:** No data movement will take place if the data are already present in a device.

# Data management: Enter Data directive

**Unstructured data:** The use of structured data is not always possible. Ex. When allocating and deallocating arrays within a data region.

In C/C++: `#pragma acc enter data copyin(array[0:n]) create(u[0:n])`

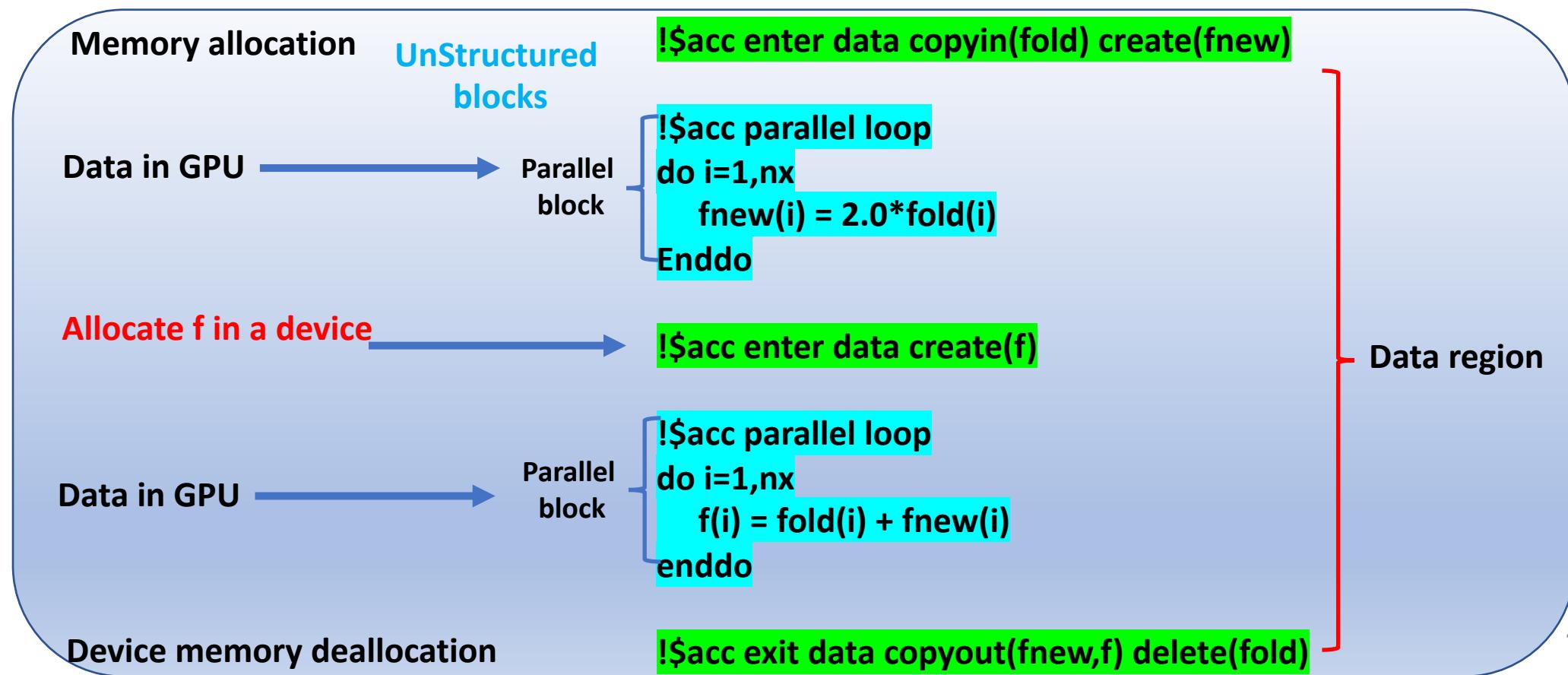
.....

`#pragma acc exit data copyout(u[0:n]) delete(array)`

In Fortran: `!$acc enter data copyin(array) create(u)`

.....

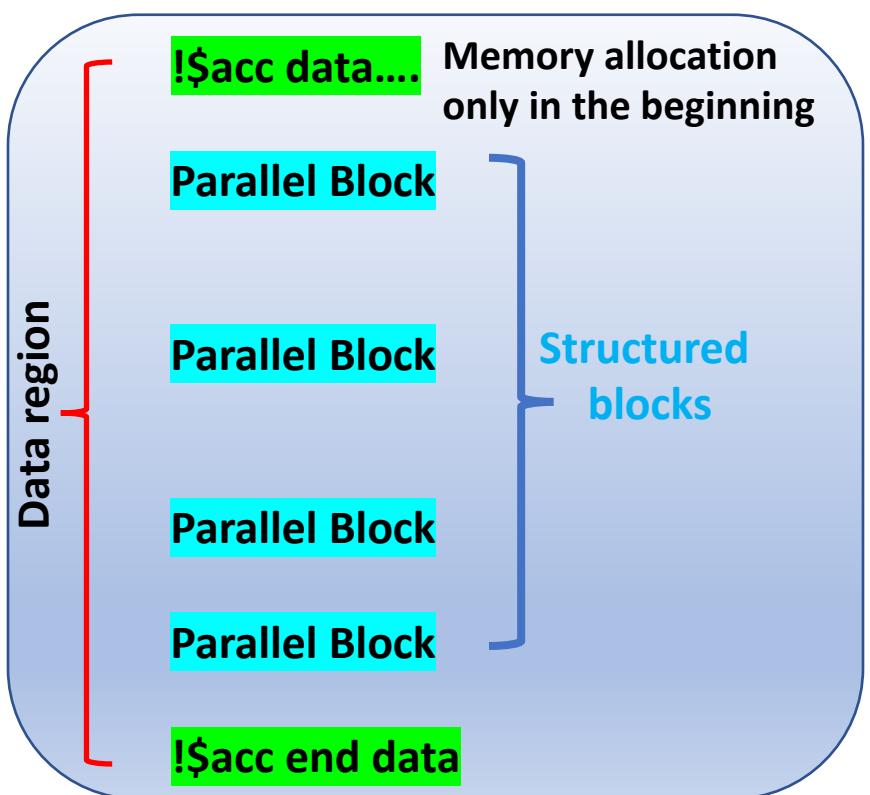
`!$acc exit data copyout(u) delete(array)`



# Data management: Structured data vs UnStructured data

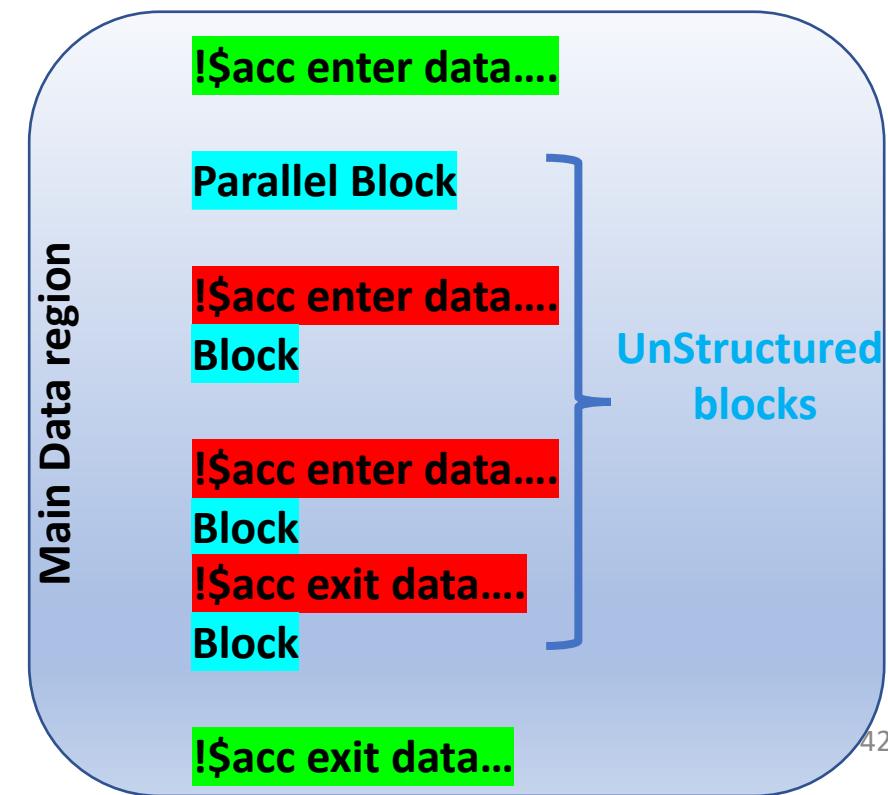
## Structured data

- The concept of a data region is well defined: it starts with **!\$acc data** and ends with **!\$acc end data**.
- The device memory allocation is specified only in the beginning of a data region.



## UnStructured data

- Create multiple enter/exit data directives within the main data region .
- Ability of multiple memory allocation/deallocation within the main data region



# Atomic operations

## Data dependency

When multiple threads access the same variable simultaneously, causing incorrectness of the obtained results (**race conditions**).

Ex. When one loop modifies a variable and a second loop reads the same variable in parallel: **Data dependencies**.

This is not the case in sequential programs.

Ex. Computing a sum. The **reduction directive** ensures the correctness.

The use of the **atomic directive** protects against race conditions.

In C/C++: `#pragma acc atomic update`

In Fortran: `!$acc atomic update`

```
Sum=0.  
 !$acc parallel loop reduction(+:sum)  
   do i=1,n  
     sum = sum + v(i)  Data dependency  
   enddo  
 !$acc end parallel loop
```

```
 !$acc parallel loop copyin(v) copy(u)  
   do i=1,n  
     !$acc atomic update  
       u(i) = u(i) + v(i)  Data dependency  
     enddo  
   !$acc end parallel loop
```

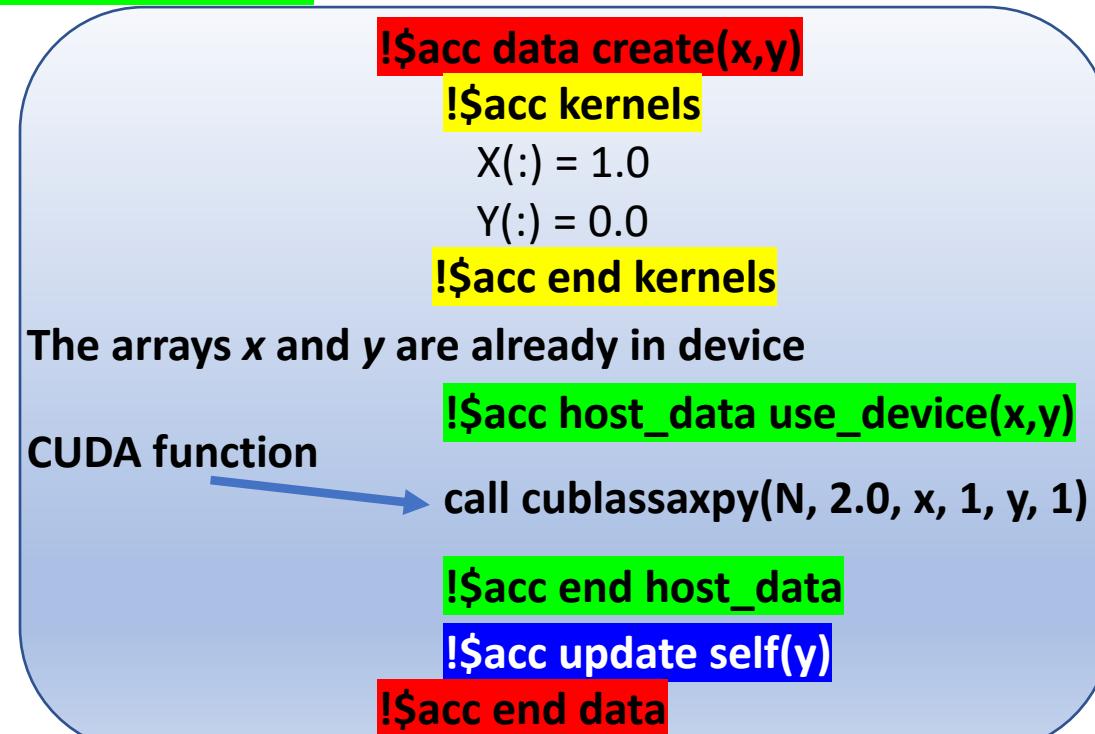
**Combining OpenACC with other programming models  
(e.g.CUDA, OpenCL or accelerated libraries e.g. FFT)**

# Host\_data directive

- The **host\_data** directive enables to pass arrays already created in a device to e.g. a CUDA function or to an accelerated library.
- The directive makes the address of the device data available on the host, so that the data can be used by a CUDA function.
- The compiler generates a code to enable copying the arrays created in a device.
- When using the **host\_data** directive, a list of the device arrays should be provided using the **use\_device** clause.

In C/C++: `#pragma acc host_data use_device(list of arrays)`

In Fortran: `!$acc host_data use_device(list of arrays)`



*The example is taken from*

<https://developer.nvidia.com/blog/3-versatile-openacc-interoperability-techniques/>

# Deviceptr clause

- The **deviceptr()** clause is a reverse concept of the **host\_data** directive.
- The clause informs the compiler that arrays, which have been allocated using e.g. CUDA, are already present in a device.
- The clause can be used inside a data region with a **parallel** or **kernels** constructs.
- This clause is generally used when OpenACC is combined with another programming model.

In C/C++: **#pragma acc kernels deviceptr(list of arrays)**

In Fortran: **!\$acc kernels deviceptr(list of arrays)**

```
!$acc kernels deviceptr(x,y)
y(:) = y(:) + b*x(:)
 !$acc end kernels
```

# Acc\_map\_data routine

The routine `acc_map_data` permits mapping host data to device data.

The routine can be used when sharing memory between CUDA and OpenACC.

The use of the routine requires specifying the header:

In C: `#include <openacc.h>`

In Fortran: `use openacc`

In C/C++

`acc_map_data( host, dev , size)`

In Fortran

subroutine `acc_map_data( host, dev )`

or

subroutine `acc_map_data( host, dev , size)`

Where *host* is a host array (host memory).  
*dev* is a CUDA fortran device array (device memory).  
*size* specifies the length of the array in bytes (integer).

# Acc\_unmap\_data routine

The `acc_unmap_data` routine unmaps the device data from the specified host data.

In C/C++

`acc_unmap_data( host)`

In Fortran

subroutine `acc_unmap_data( host)`

# **Asynchronous OpenACC**

# Asynchronous features

- Data transfer to a GPU might be time consuming on systems with distinct memories.
- To reduce the computation time, **different parallel regions can be ran asynchronously.**
- The asynchronous process can be done via the following clauses:

**async(argument)** clause:

- It allows to combine multiple tasks (e.g. offloading to GPU and computation).
- While a GPU-device is executing a task, the CPU-host may do computation.
- This is only possible if a block of loops to be treated asynchronously **are independent.**
- The clause can be added to **parallel**, **kernels** and **update** directives.
- Computation and data transfer can be done concurrently.
- argument**: integer number that defines the queue number of a specific task.

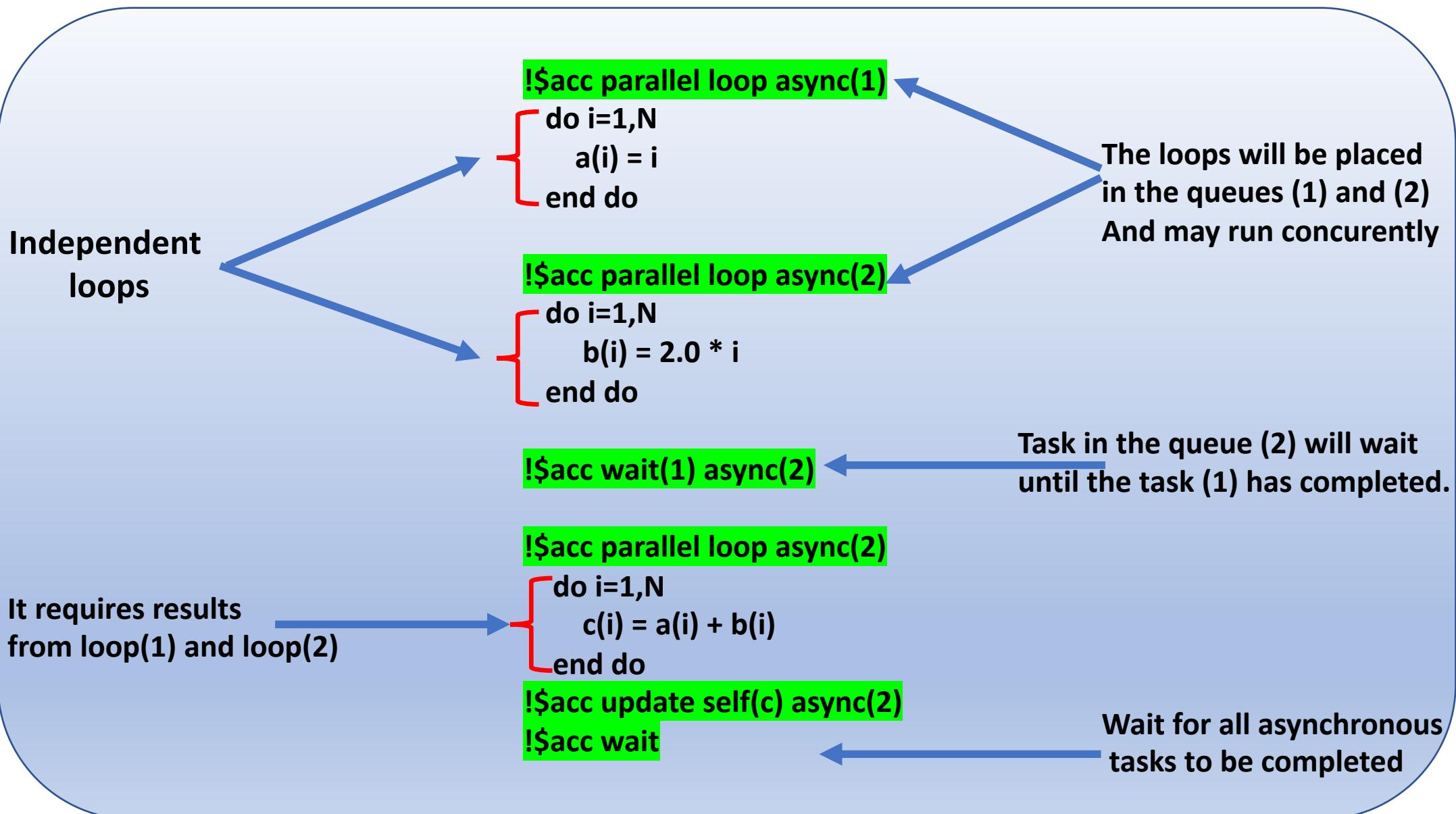
**wait(argument)** clause: No execution will take place until all previous asynchronous tasks are complete.

In C/C++: **#pragma acc parallel loop async**

In Fortran: **!\$ acc parallel loop async**

```
!$acc data copyin(fold)
copyout(fnew)
 !$acc parallel loop
do i=1,nx
    fnew(i) = 2.0*fold(i)
enddo
WAIT
 !$acc parallel loop
do i=1,nx
    fnew(i) = fold(i) + fnew(i)
enddo
 !$acc end data
```

# Asynchronous features



# Offloading to Multiple GPU-devices

Copy of data on multiple GPU-devices.

Specifying a device for each unstructured **enter data** directive using the function **acc\_set\_device\_num()** .

## Ex. Allocate arrays on multiple devices

### Fortran

```
do gpu=0,1  
call acc_set_device_num(gpu,acc_device_nvidia)  
!$acc enter data create(array)  
enddo
```

### C

```
for(int gpu=0; gpu<2; gpu ++)  
{  
    acc_set_device_num(gpu,acc_device_nvidia);  
    #pragma acc enter data create(array[:N])  
}
```

## Other relevant routines

**acc\_get\_device\_type**: returns the device type to be used when executing a parallel or kernels region.

**acc\_get\_num\_devices**: returns the number of devices available for the specified device type.

**acc\_get\_num\_devices**: returns the number of accelerator devices of the given type attached to the host.

**acc\_set\_device\_num**: to set the device number and the device type to be used.

# **Compilation process**

# Compilation process

Load NVIDIA HPC environment.

Find a module:

```
$ module avail NVHPC
```

```
$ load module
```

## Fortran

```
nvfortran -fast -acc -Minfo=accel -o executable myCode.f90
```

or

```
nvfortran -gpu=cc60 -Minfo=accel -o executable myCode.f90
```

## C

```
nvc -fast -acc -Minfo=accel -o executable myCode.c
```

For C++, compile with **nvc++**

- Flags **-acc** and **-gpu=<target>** enables OpenACC directives.
- The option **<target>** reflects the name of the GPU device.
- The latter is set to be **<cc60>** for the device name Tesla **P100** and **<cc70>** for the tesla **V100** device and **<cc80>** for **A100 GPU**. This information can be viewed by running the command \textbf{pgaccelinfo}.
- The flag option **-Minfo** enables the compiler to print out the feedback messages on optimizations and transformations.

# Exercice

Ex. Solving the laplace equation

1-Run serial code with gprof

2-Introduce parallel loops

3-Perform profiling with Nsight systems: %kernels (computation) vs %memory (data transfer)

The majority of the computing time is spent in copying data between the host and device.

4-Introduce data locality

5-Perform profiling again: %kernels (computation) vs %memory (data transfer)

Comment on how the data transfer occurs and the time it takes between each iterations.

# **Code profiling**

## Code profiling:

**Gprof & Gcov:** serial codes

**Nsight Systems:** NVIDIA GPU-based codes

### What is code profiling ?

- It is an advanced optimization technique.
- It helps to manage codes/programs.
- Improving the performance of codes/programs.
- Identify regions in which the majority of time is spent.
- It is a dynamical tool (based on gathering statistical data during the running procedure of a code/program).
- There are various source code profiling software.

# Gprof (function-by-function analysis)

Gprof is a **GNU** binary tool for performing code profiling.

**What do we learn from this tool ?**

**Gprof** provides time information used by functions:

- Computing time used in each function.
- How often a function is called by other functions.
- How often a function called other functions.

**The gathered statistical data by the Gprof tool allows us to determine which regions in the code the optimization efforts should be done.**

# Implementation of Gprof

**1-Compilation:** include the flag **-pg** in the compilation syntax.

**Ex. (Fortran 90 code)** `gfortran -pg -o executable MyCode.f90`

The option “**-pg**” enables the profiling to be performed while compiling.

**2-Execution:** `./executable`

It generates a profile data file “**gmon.out**”. The file contains statistical information about the running time of the code.

**3-Running Gprof:** this step allows to interpret the “**gmon.out**” file

`gprof executable gmon.out > analysis.out`

The results of **Gprof** are thus stored in the **analysis.out** file and can be viewed by any text editor.

# Example of profiling with Gprof

Flat profile:

| % cumulative | self    | self    | total |        |        |                |
|--------------|---------|---------|-------|--------|--------|----------------|
| time         | seconds | seconds | calls | s/call | s/call | name           |
| 60.04        | 13.49   | 13.49   | 7     | 1.93   | 2.32   | deriv_         |
| 12.19        | 16.23   | 2.74    | 21    | 0.13   | 0.13   | aura_          |
| 11.57        | 18.83   | 2.60    | 1     | 2.60   | 7.24   | rot_           |
| 5.92         | 20.16   | 1.33    | 1     | 1.33   | 1.33   | calc_th_       |
| 5.03         | 21.29   | 1.13    | 1     | 1.13   | 19.96  | calc_pv_       |
| 1.91         | 21.72   | 0.43    | 1     | 0.43   | 0.43   | input_levels_  |
| 1.87         | 22.14   | 0.42    | 3     | 0.14   | 0.14   | input_data_    |
| 1.42         | 22.46   | 0.32    | 3     | 0.11   | 0.11   | output_write_  |
| 0.04         | 22.47   | 0.01    | 1     | 0.01   | 0.01   | output_open_   |
| 0.00         | 22.47   | 0.00    | 14    | 0.00   | 0.00   | getind_        |
| 0.00         | 22.47   | 0.00    | 2     | 0.00   | 0.00   | input_close_   |
| 0.00         | 22.47   | 0.00    | 2     | 0.00   | 0.00   | input_open_    |
| 0.00         | 22.47   | 0.00    | 1     | 0.00   | 22.47  | MAIN_          |
| 0.00         | 22.47   | 0.00    | 1     | 0.00   | 0.00   | input_getvars_ |
| 0.00         | 22.47   | 0.00    | 1     | 0.00   | 0.00   | input_grid_    |
| 0.00         | 22.47   | 0.00    | 1     | 0.00   | 0.00   | output_close_  |

**%** the percentage of the total running time of the  
**time** program used by this function.

**cumulative** a running sum of the number of seconds accounted  
**seconds** for by this function and those listed above it.

**self** the number of seconds accounted for by this  
**seconds** function alone. This is the major sort for this  
listing.

**calls** the number of times this function was invoked, if  
this function is profiled, else blank.

**self** the average number of milliseconds spent in this  
**ms/cal** function per call, if this function is profiled,  
else blank.

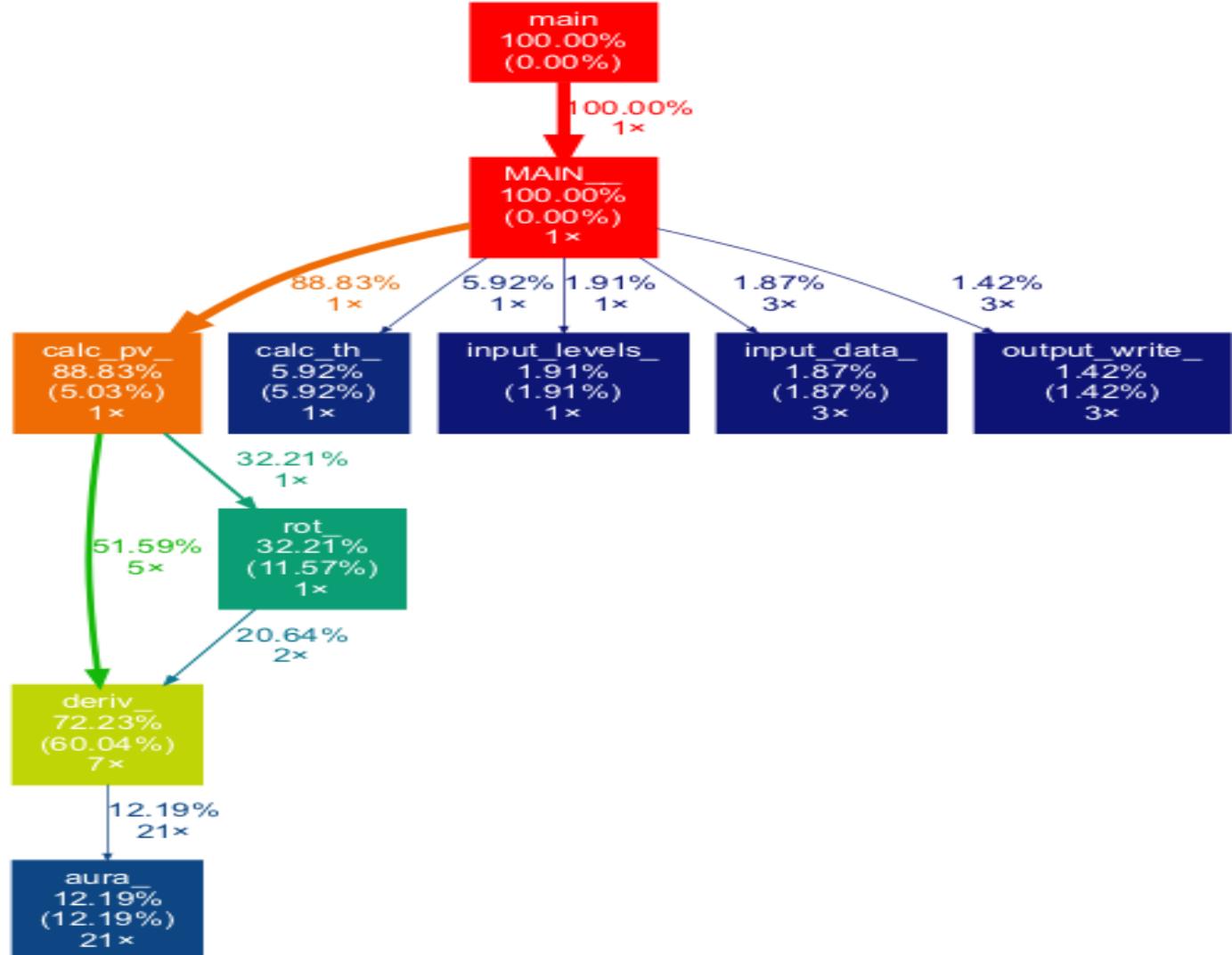
**total** the average number of milliseconds spent in this  
**ms/call** function and its descendants per call, if this  
function is profiled, else blank.

**name** the name of the function.

# Visualization using Gprof2dot

Converting data to graph: gprof2dot analysis.out | dot -Tpng -o output.png

Installation: brew install gprof2dot



# Gcov profiling (line-by-line analysis)

**Gcov** is a test coverage tool, it defines the % of lines of a code get executed

The **Gcov** tool tells about (**line-by-line**):

- How often each line of a code gets executed.
- What lines of code are actually executed.

**1-Compilation:** `gfortran -fprofile-arcs -ftest-coverage subroutine.f90`

The option “**-fprofile-arcs**” enables the profiling to be performed while compiling.

**2-After executing the code; use:** `gcov subroutine.f90`

It generates a logfile “**subroutine.f90.gcov**”. The file contains statistical information about line-by-line execution.

Combining **Gprof** with **Gcov** helps to analyse the code’s performance in a more detailed way. **Gprof** provides time information analysis of **function-by-function**, while **Gcov** aids to tune the analysis by performing **line-by-line** investigation.

# Example of Gcov profiling

gcov subroutine.f90 creates **subroutine.f90.gcov**

The **gcno** and **gcda** files are also generated. These files are required for  
The visualization using **lcov**

-It can be linked to **Lcov**, which is a Graphical tool  
for GCC's coverage testing tool (gcov)

-**Installation:** brew install lcov

-To make lcov generate html reports, use the following commands:

**lcov --directory . --zerocounters**

**lcov --directory . --capture --output-file app.info**

**genhtml app.info**

## LCOV - code coverage report

Current view: [top level](#) - Lcov\_Analysis

Test: [app.info](#)

Date: 2021-08-24 19:36:59

|            | Hit | Total | Coverage |
|------------|-----|-------|----------|
| Lines:     | 25  | 52    | 48.1 %   |
| Functions: | 1   | 1     | 100.0 %  |

| Filename                 | Line Coverage | Functions |
|--------------------------|---------------|-----------|
| <a href="#">fft0.f90</a> | 48.1 %        | 100.0 %   |

Generated by: [LCOV version 1.15](#)

```
29      0 :     write(0,"fft: n is not a power of 2")
30      0 :     stop
31      :   end if
32      :
33      0 :   allocate(x(0:n-1),y(0:n-1))
34      :
35      0 :   x(0:n-1) = real(xy(1:n))
36      0 :   y(0:n-1) = imag(xy(1:n))
37      :
38      :   ! do the bit reversal
39      0 :   i2 = n/2
40      0 :   j = 0
41 268439548 :   do i=0,n-2
42 268388472 :     if (i < j) then
43      0 :       tx = x(i)
44      0 :       ty = y(i)
45      0 :       x(i) = x(j)
46      0 :       y(i) = y(j)
47      0 :       x(j) = tx
48      0 :       y(j) = ty
49      :   end if
50 138090688 :   k = i2
51      0 :   do while (k <= j)
52      0 :     j = j-k
53      0 :     k = k/2
54      :   end do
55 268439548 :   j = j+k
56      :   end do
57      :
58      :   ! compute the FFT
59      0 :   c1 = -1d0
60      0 :   c2 = 0d0
61      0 :   l2 = 1
62 720939 :   do l=0,m-1
63 1310770 :     l1 = l2
64 1310770 :     l2 = l2*2
65 1310770 :     u1 = 1d0
66 1310770 :     u2 = 0d0
67 1310770 :     do j=0,l1-1
68 1744463814 :       do i=j,n-1,l2
69 1477466112 :         i1 = i + l1
70 1477466112 :         t1 = u1 * x(i1) - u2 * y(i1)
71 1477466112 :         t2 = u1 * y(i1) + u2 * x(i1)
72 1477466112 :         x(i1) = x(i) - t1
73 1477466112 :         y(i1) = y(i) - t2
74 1477466112 :         x(i) = x(i) + t1
75 1477466112 :         y(i) = y(i) + t2
76      :     end do
77      0 :     z = u1 * c1 - u2 * c2
78      0 :     u2 = u1 * c2 + u2 * c1
79      0 :     u1 = z
```

# NVIDIA Nsight Systems

NVIDIA Nsight Systems tool provides timeline information about GPU and CPU activities.

The tool allows to identify issues, such that: *Taken from <https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/>*

- GPU starvation
- Unnecessary GPU synchronization
- Expensive computing time during the host-device data transfers.

The tool thus allows to optimize the performance of CUDA, OpenACC and OpenMP applications.

**How to measure the computing time during the host-device data transfer without modifying the source code?**

It can be done using the command-line **nvprof**

Compile

**For fortran:** nvfortran -fast -acc -Minfo=accel -g -o **executable** Mycode.f90

**For C:** compile with **nvcc**

Run

**nsys profile -t cuda,openacc -f true -o output ./executable**

**nvprof ./executable**

# Output of profiling with nvprof

Running **nvprof** generates an output file containing timeline information about:

- GPU activities
- API calls
- OpenACC

Here is an example of profiling a code based on solving the laplace equation:

## GPU activities

Profiling result:

| Type                   | Time(%) | Time     | Calls | Avg      | Min      | Max      | Name                          |
|------------------------|---------|----------|-------|----------|----------|----------|-------------------------------|
| <b>GPU activities:</b> | 48.82%  | 746.66ms | 243   | 3.0727ms | 3.0634ms | 3.0791ms | <b>laplace_acc_55_gpu</b>     |
|                        | 43.84%  | 670.48ms | 243   | 2.7592ms | 2.7041ms | 2.8123ms | <b>laplace_acc_42_gpu</b>     |
|                        | 2.84%   | 43.437ms | 34    | 1.2776ms | 1.8240us | 1.3648ms | [CUDA memcpy HtoD]            |
|                        | 2.71%   | 41.398ms | 276   | 149.99us | 1.3120us | 1.3360ms | [CUDA memcpy DtoH]            |
|                        | 1.76%   | 26.985ms | 243   | 111.05us | 109.95us | 117.54us | <b>laplace_acc_55_gpu_red</b> |
|                        | 0.02%   | 292.80us | 243   | 1.2040us | 1.1520us | 1.5680us | [CUDA memset]                 |

|            | Type   | Time(%)  | Time | Calls    | Avg      | Min      | Max                       | Name |
|------------|--------|----------|------|----------|----------|----------|---------------------------|------|
| API calls: | 82.09% | 1.44880s | 974  | 1.4875ms | 1.6330us | 3.1892ms | cuStreamSynchronize       |      |
|            | 13.48% | 237.88ms | 1    | 237.88ms | 237.88ms | 237.88ms | cuDevicePrimaryCtxRetain  |      |
|            | 2.31%  | 40.836ms | 339  | 120.46us | 1.2510us | 1.3429ms | cuEventSynchronize        |      |
|            | 1.47%  | 26.002ms | 1    | 26.002ms | 26.002ms | 26.002ms | cuMemHostAlloc            |      |
|            | 0.20%  | 3.5720ms | 729  | 4.8990us | 3.3450us | 32.912us | cuLaunchKernel            |      |
|            | 0.14%  | 2.4206ms | 7    | 345.80us | 3.2670us | 1.4096ms | cuMemAlloc                |      |
|            | 0.10%  | 1.7416ms | 276  | 6.3100us | 3.2380us | 24.165us | cuMemcpyDtoHAsync         |      |
|            | 0.06%  | 974.17us | 1    | 974.17us | 974.17us | 974.17us | cuMemAllocHost            |      |
|            | 0.05%  | 899.70us | 341  | 2.6380us | 1.5080us | 11.193us | cuEventRecord             |      |
|            | 0.05%  | 884.09us | 243  | 3.6380us | 2.3710us | 28.147us | cuMemsetD32Async          |      |
|            | 0.03%  | 537.81us | 34   | 15.817us | 12.358us | 27.082us | cuMemcpyHtoDAsync         |      |
|            | 0.02%  | 314.05us | 1    | 314.05us | 314.05us | 314.05us | cuModuleLoadDataEx        |      |
|            | 0.00%  | 16.502us | 1    | 16.502us | 16.502us | 16.502us | cuStreamCreate            |      |
|            | 0.00%  | 11.589us | 3    | 3.8630us | 1.9130us | 6.5810us | cuPointerGetAttributes    |      |
|            | 0.00%  | 11.249us | 4    | 2.8120us | 506ns    | 4.9420us | cuEventCreate             |      |
|            | 0.00%  | 9.1630us | 1    | 9.1630us | 9.1630us | 9.1630us | cuDeviceGetPCIBusId       |      |
|            | 0.00%  | 3.4260us | 3    | 1.1420us | 280ns    | 2.5620us | cuModuleGetFunction       |      |
|            | 0.00%  | 3.3130us | 2    | 1.6560us | 181ns    | 3.1320us | cuDeviceGet               |      |
|            | 0.00%  | 3.1270us | 3    | 1.0420us | 281ns    | 2.0310us | cuCtxSetCurrent           |      |
|            | 0.00%  | 1.9530us | 5    | 390ns    | 133ns    | 818ns    | cuDeviceGetAttribute      |      |
|            | 0.00%  | 1.6750us | 3    | 558ns    | 203ns    | 1.2180us | cuDeviceGetCount          |      |
|            | 0.00%  | 1.6540us | 1    | 1.6540us | 1.6540us | 1.6540us | cuCtxGetCurrent           |      |
|            | 0.00%  | 416ns    | 1    | 416ns    | 416ns    | 416ns    | cuDeviceComputeCapability |      |
|            | 0.00%  | 176ns    | 1    | 176ns    | 176ns    | 176ns    | cuDriverGetVersion        |      |

# OpenACC (excl) information

Profiling result:

| Type                   | Time(%) | Time     | Calls | Avg      | Min      | Max      | Name                                                           |
|------------------------|---------|----------|-------|----------|----------|----------|----------------------------------------------------------------|
| <b>OpenACC (excl):</b> | 43.44%  | 775.36ms | 486   | 1.5954ms | 5.8650us | 3.1901ms | acc_wait@laplace_acc.f90:55                                    |
|                        | 37.62%  | 671.56ms | 243   | 2.7636ms | 2.7097ms | 2.8170ms | acc_wait@laplace_acc.f90:42                                    |
|                        | 11.30%  | 201.67ms | 1     | 201.67ms | 201.67ms | 201.67ms | acc_exit_data@laplace_acc.f90:40                               |
|                        | 6.47%   | 115.53ms | 1     | 115.53ms | 115.53ms | 115.53ms | acc_enter_data@laplace_acc.f90:40                              |
|                        | 0.36%   | 6.3747ms | 1     | 6.3747ms | 6.3747ms | 6.3747ms | acc_wait@laplace_acc.f90:66                                    |
|                        | 0.11%   | 2.0415ms | 243   | 8.4010us | 5.8850us | 22.482us | acc_enqueue_download@laplace_acc.f90:62                        |
|                        | 0.11%   | 1.9219ms | 243   | 7.9090us | 5.1370us | 9.9690us | acc_wait@laplace_acc.f90:62                                    |
|                        | 0.09%   | 1.6290ms | 243   | 6.7030us | 5.0930us | 38.698us | acc_enqueue_launch@laplace_acc.f90:42 (laplace_acc_42_gpu)     |
|                        | 0.09%   | 1.5643ms | 243   | 6.4370us | 4.8360us | 28.268us | acc_enqueue_launch@laplace_acc.f90:55 (laplace_acc_55_gpu)     |
|                        | 0.07%   | 1.2504ms | 243   | 5.1450us | 4.3380us | 14.431us | acc_enqueue_launch@laplace_acc.f90:55 (laplace_acc_55_gpu_red) |
|                        | 0.06%   | 1.0981ms | 243   | 4.5180us | 3.0720us | 31.541us | acc_enqueue_upload@laplace_acc.f90:55                          |
|                        | 0.05%   | 977.64us | 243   | 4.0230us | 3.1710us | 21.249us | acc_enter_data@laplace_acc.f90:55                              |
|                        | 0.05%   | 933.82us | 243   | 3.8420us | 3.0370us | 28.420us | acc_exit_data@laplace_acc.f90:55                               |
|                        | 0.04%   | 756.50us | 33    | 22.924us | 6.0210us | 33.184us | acc_enqueue_download@laplace_acc.f90:66                        |
|                        | 0.04%   | 752.64us | 34    | 22.136us | 14.306us | 39.348us | acc_enqueue_upload@laplace_acc.f90:40                          |
|                        | 0.04%   | 627.04us | 243   | 2.5800us | 2.2000us | 7.7770us | acc_compute_construct@laplace_acc.f90:55                       |
|                        | 0.03%   | 512.35us | 243   | 2.1080us | 1.7520us | 19.561us | acc_compute_construct@laplace_acc.f90:42                       |
|                        | 0.02%   | 340.94us | 1     | 340.94us | 340.94us | 340.94us | acc_device_init@laplace_acc.f90:40                             |
|                        | 0.00%   | 8.8790us | 1     | 8.8790us | 8.8790us | 8.8790us | acc_wait@laplace_acc.f90:40                                    |
|                        | 0.00%   | 0ns      | 243   | 0ns      | 0ns      | 0ns      | acc_delete@laplace_acc.f90:62                                  |

# Guide on the use of NVIDIA Nsight Systems

## On local systems

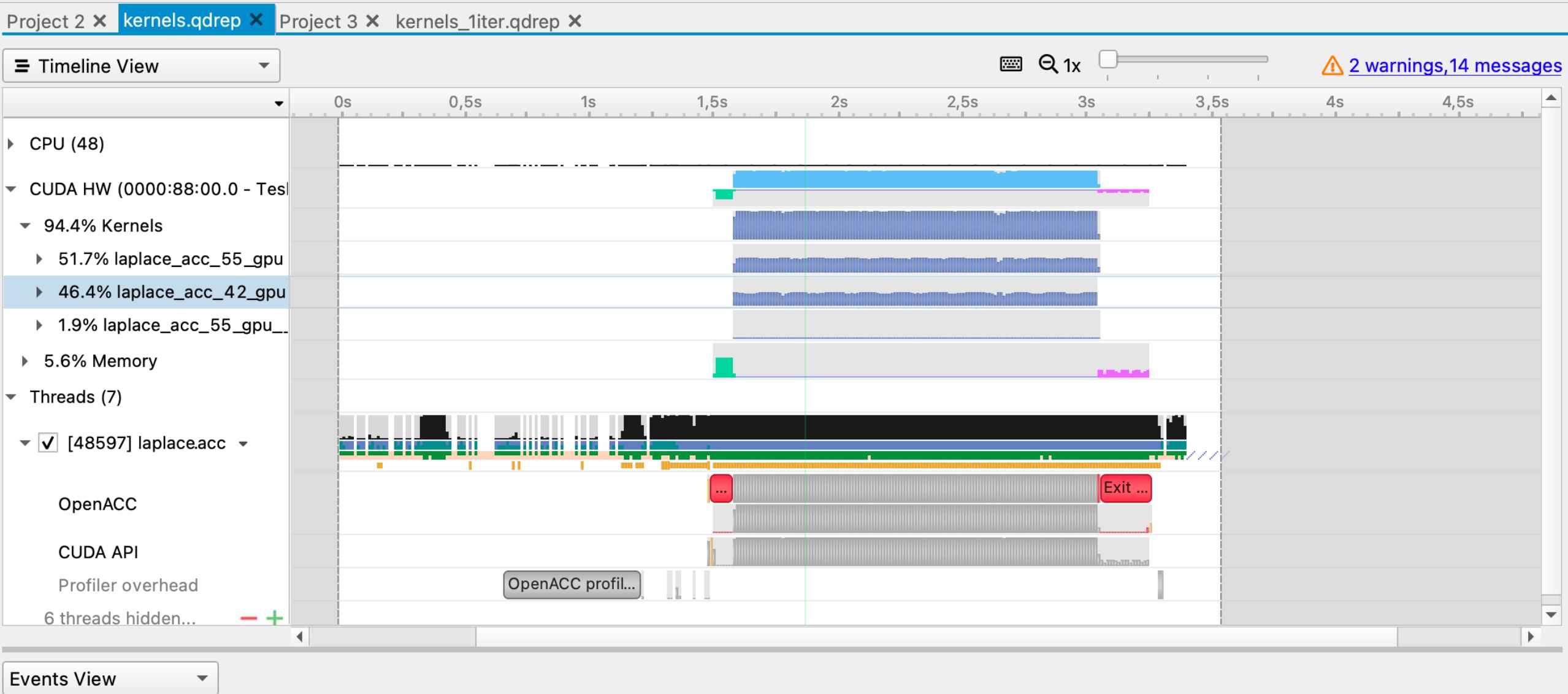
- 1- Installation guide <https://docs.nvidia.com/nsight-systems/InstallationGuide/index.html>
- 2- Download and install from here <https://developer.nvidia.com/nsight-systems>
- 3- Launch GUI directly: run **nsight-sys** executable from the host directory of installation  
Or directly from the host desktop where the icon of **NVIDIA Nsight Systems** is located.



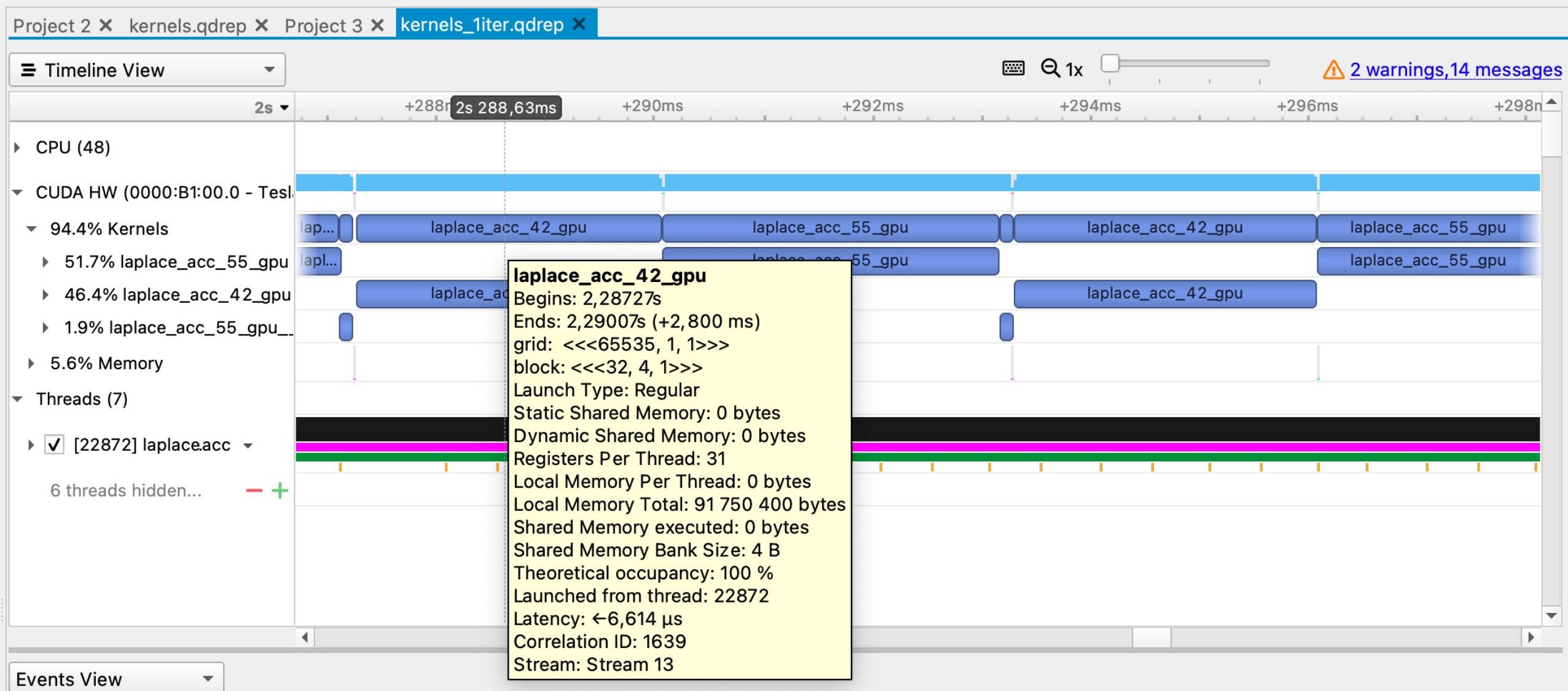
## Launching GUI From Saga:

- 1- Login to Saga with ssh -X
- 2- Load ml load CUDA/11.4.1 X11/20210802-GCCcore-11.2.0
- 3- Run **nsight-sys** or **nsys-ui**

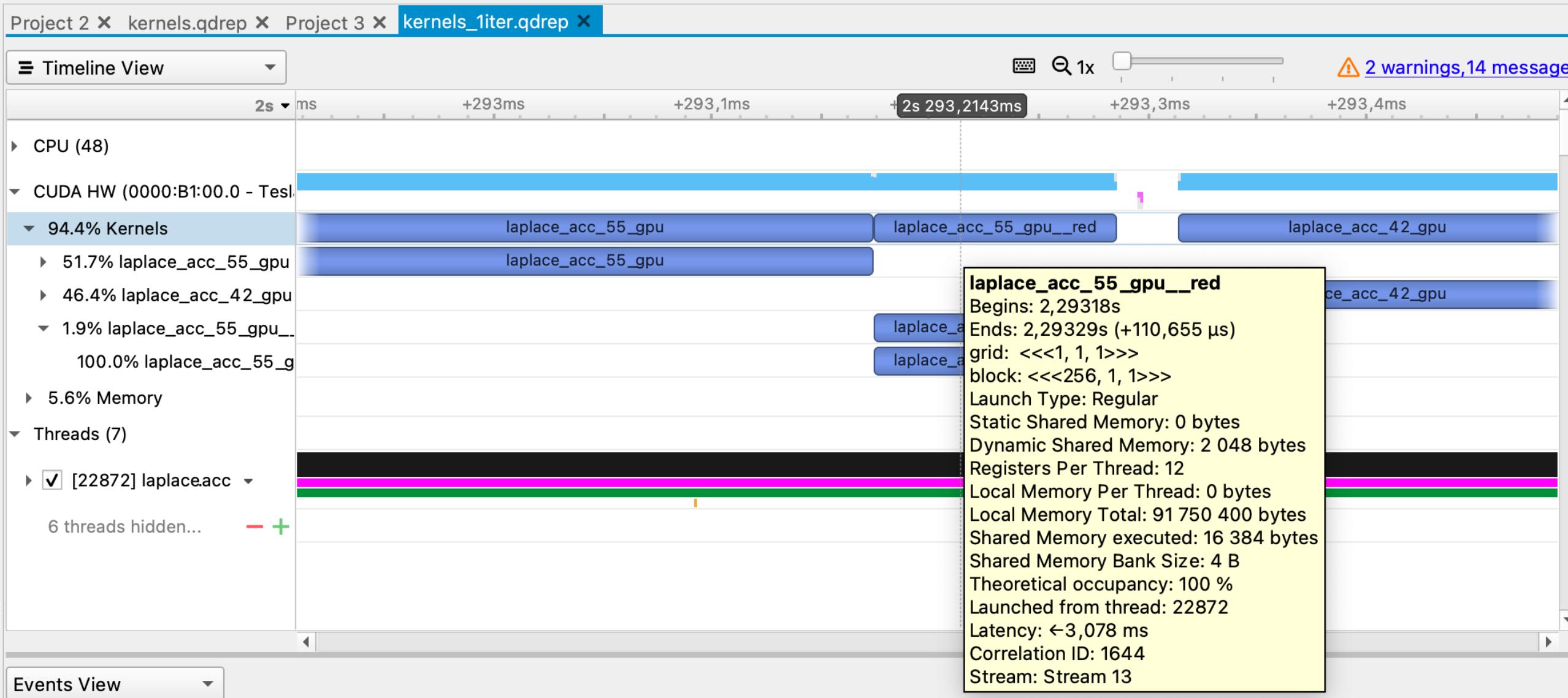
# NVIDIA Nsight Systems



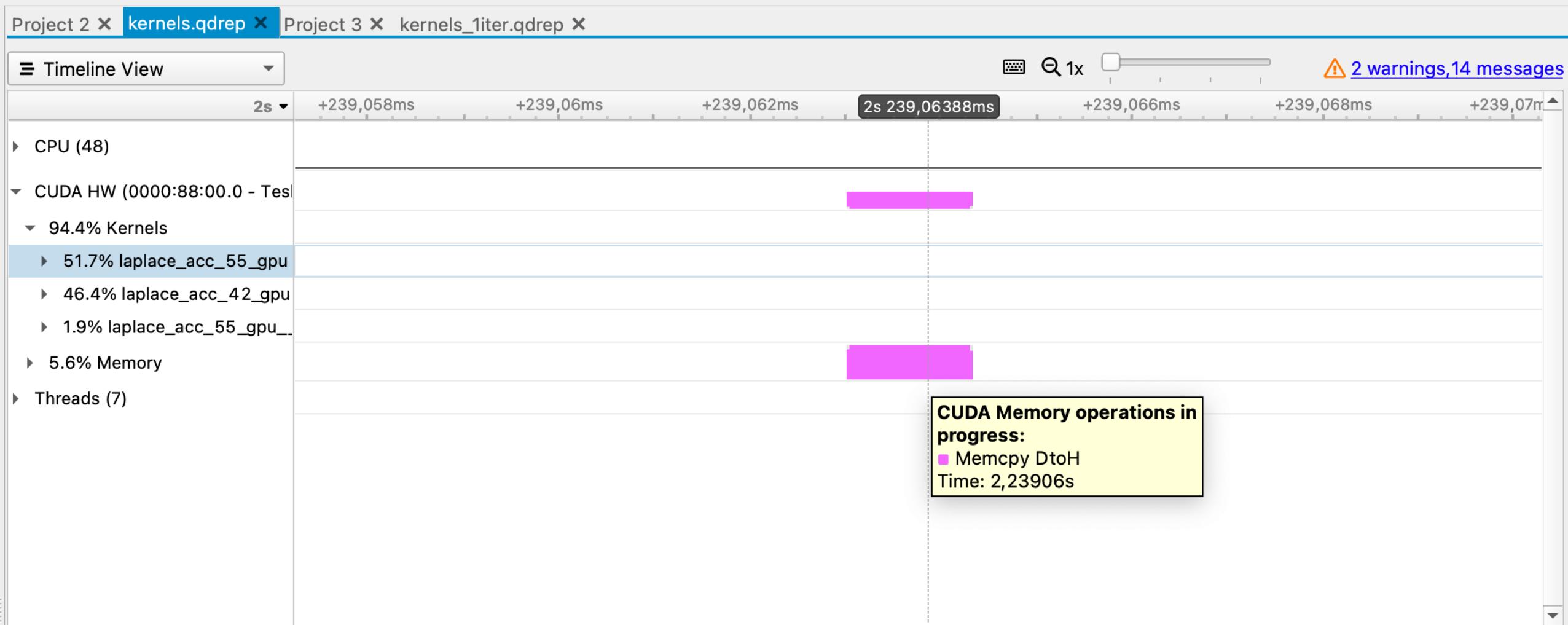
# NVIDIA Nsight Systems



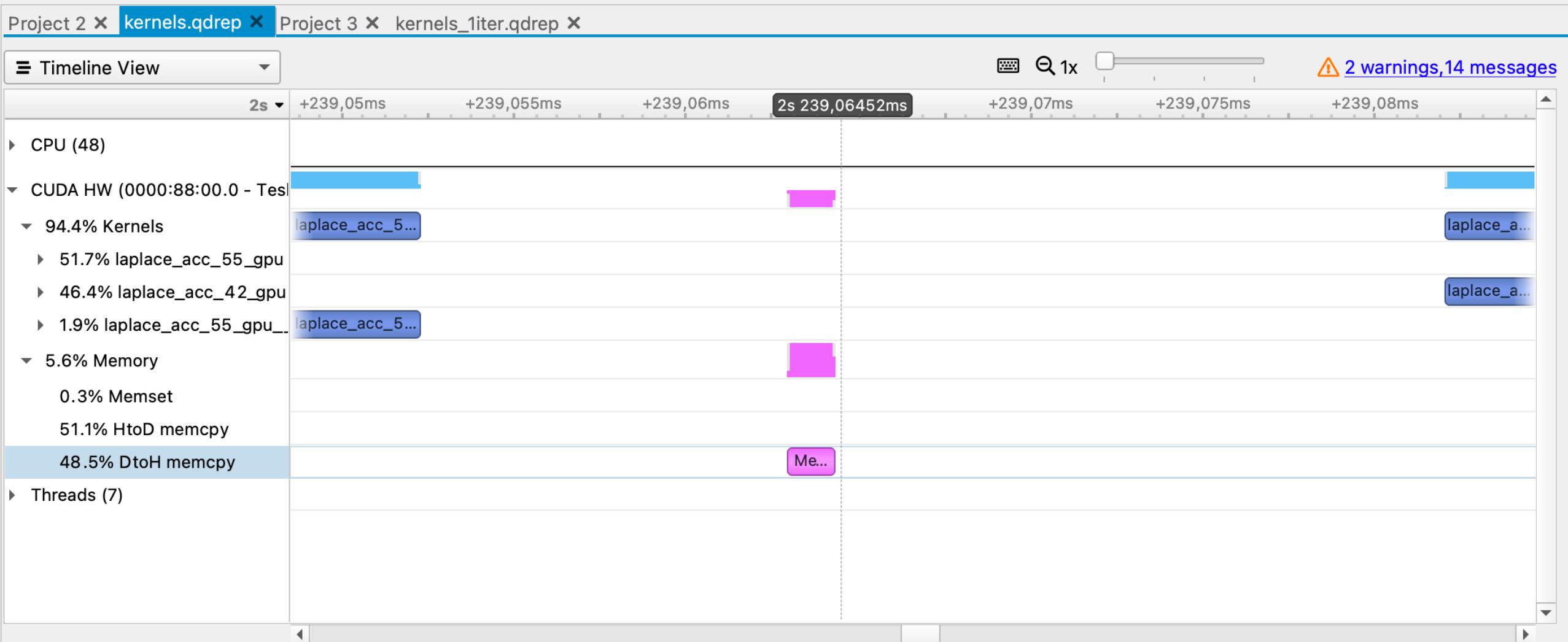
# NVIDIA Nsight Systems



# NVIDIA Nsight Systems



# NVIDIA Nsight Systems



# References

- OpenACC programming guide: [https://www.openacc.org/sites/default/files/inline-files/OpenACC\\_Programming\\_Guide\\_0\\_0.pdf](https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0_0.pdf)
- <https://docs.nvidia.com/hpc-sdk/pgi-compilers/20.4/x86/openacc-gs/index.htm>
- Parallel Programming with OpenACC-Morgan Kaufmann (2016)
- <https://developer.nvidia.com/blog/3-versatile-openacc-interoperability-techniques/>
- Profiling
- [https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html\\_node/gprof\\_1.html](https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_1.html)
- <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>
- <https://developer.nvidia.com/nsight-systems>
- <https://docs.nvidia.com/cuda/profiler-users-guide/index.html>