CINECA

CUDA: performance tuning SM & Unified Memory Prefetching

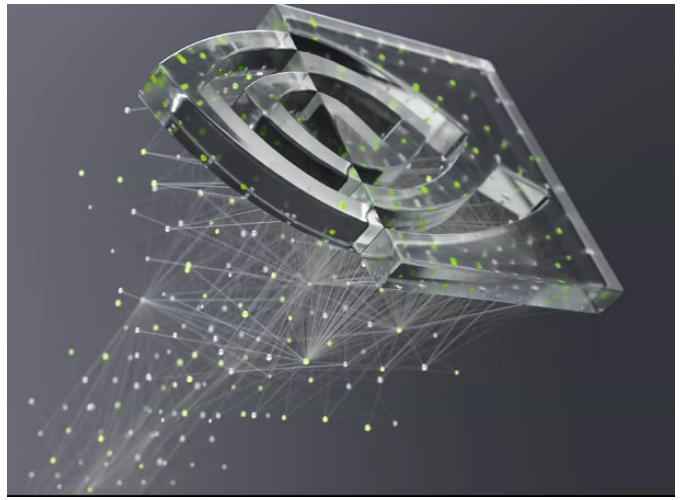
Lara Querciagrossa, Andrew Emerson, Nitin Shukla, Luca Ferraro, Sergio Orlandini

l.querciagrossa@cineca.it

July 12th, 2022

In this lecture…

- Performance tuning using Streaming Multiprocessors
- Unified Memory details
- Asynchronous Memory Prefetching



Performance tuning using Streaming Multiprocessors

Optimization through GPU hardware understanding

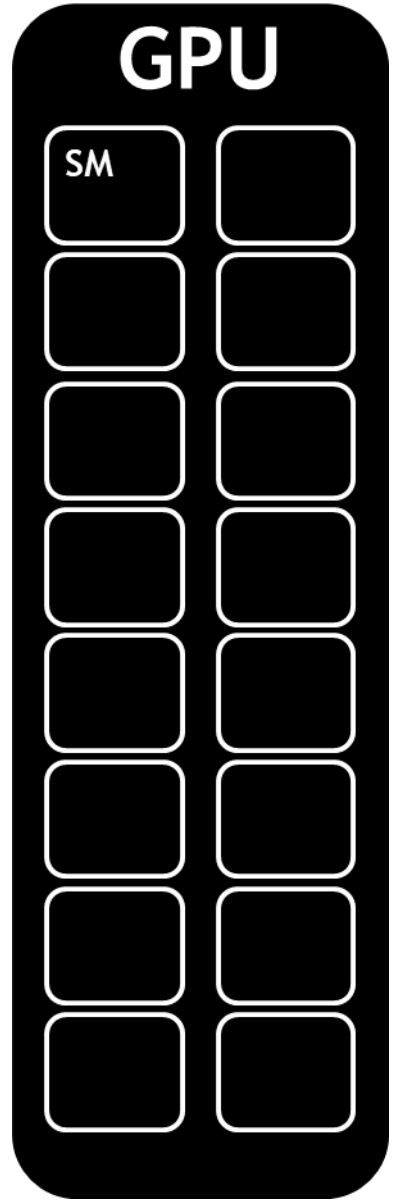
- GPU hardware specific features can promote optimization, in particular processing units called **Streaming Multiprocessors** (**SMs**).
- During kernel execution, **blocks of threads** are given to **SMs** to execute.
- Performance gains can often be had by choosing a grid size that has a **number of blocks that is a multiple of the number of SMs on a given GPU.**
- SMs create, manage, schedule, and execute groupings of 32 threads from within a block called **warps**. Performance gains can also be had by choosing a block size that has a number of threads that is a **multiple of 32.**
- Use a **profiler** (`nsys profile --stats=true`) to generate summary statistics and keep track of performance gain.

nsys profiler output

- `nsys` profile will generate a `qdrep` report file which can be used in a variety of manners. We use the `--stats=true` flag here to indicate we would like summary statistics printed. There is quite a lot of information printed:

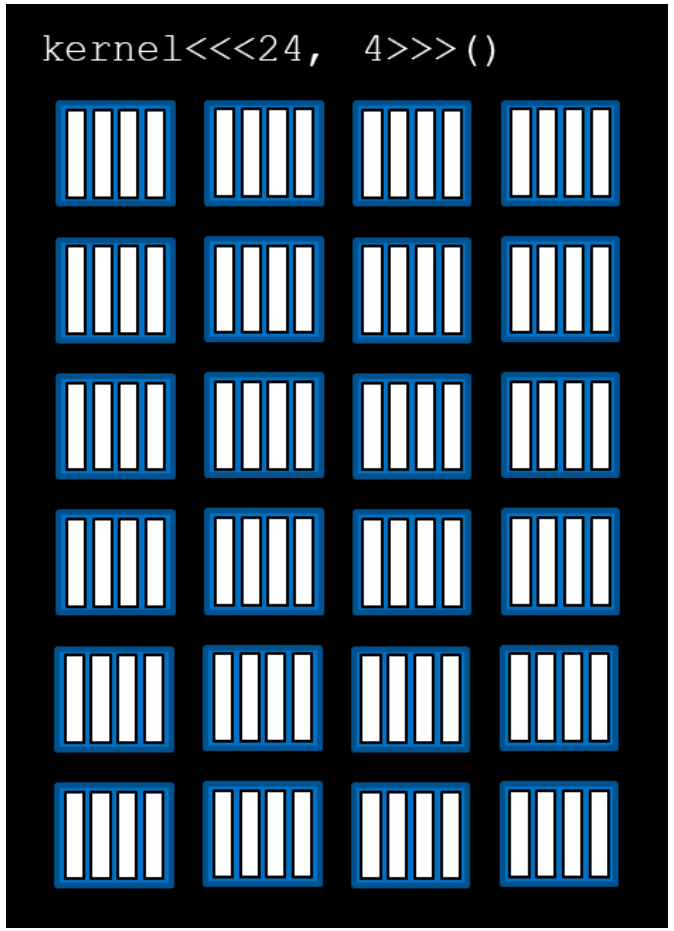
  - Profile configuration details
  - Report file(s) generation details
  - CUDA API Statistics
  - CUDA Kernel Statistics
  - CUDA Memory Operation Statistics (time and size)
  - OS Runtime API Statistics

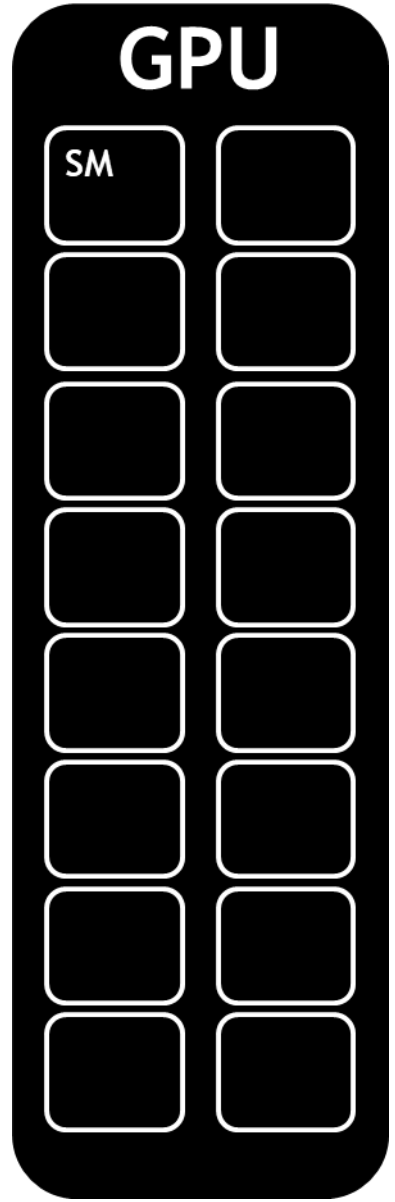
SMs and warps



GPU with
16 SMs

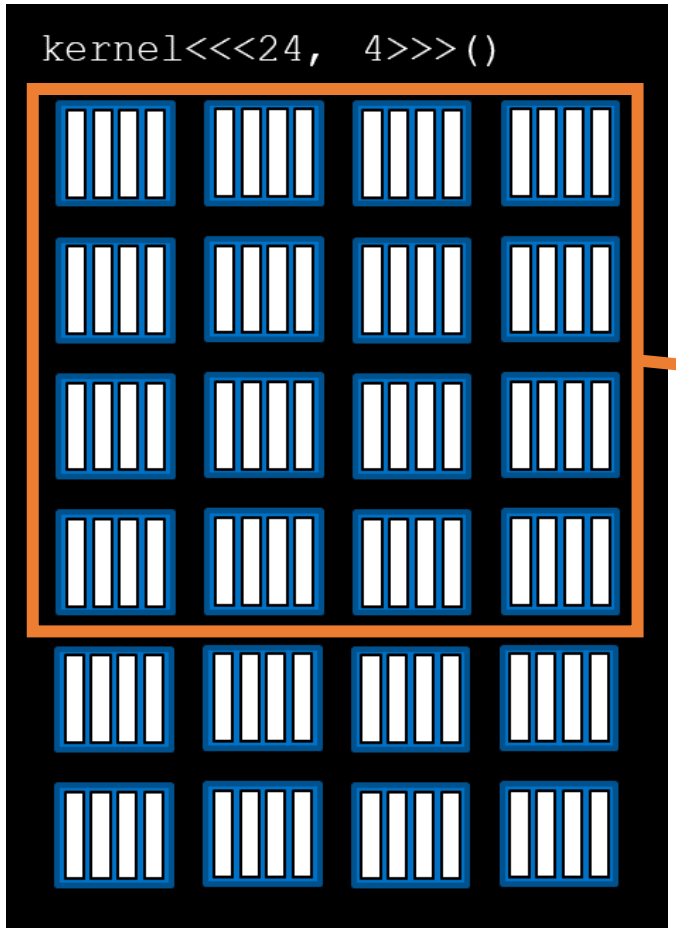
SMs and warps

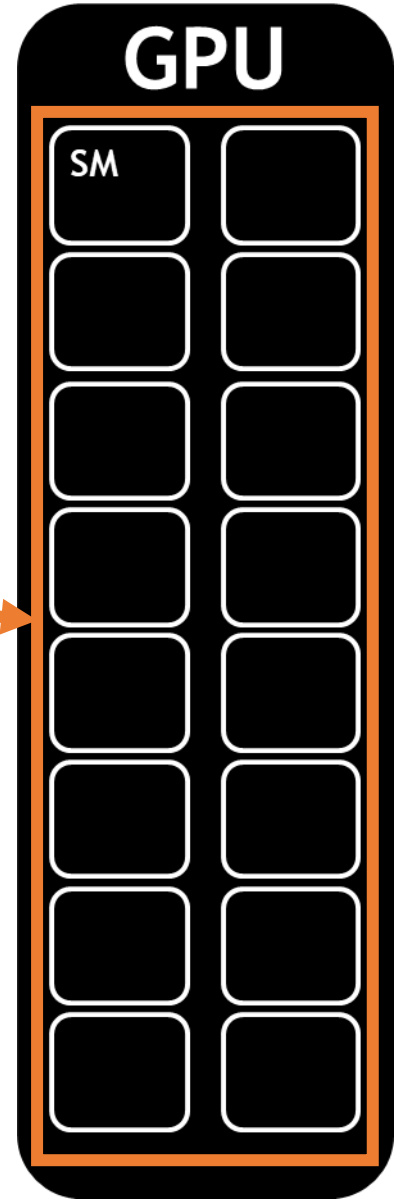




GPU with
16 SMs

SMs and warps

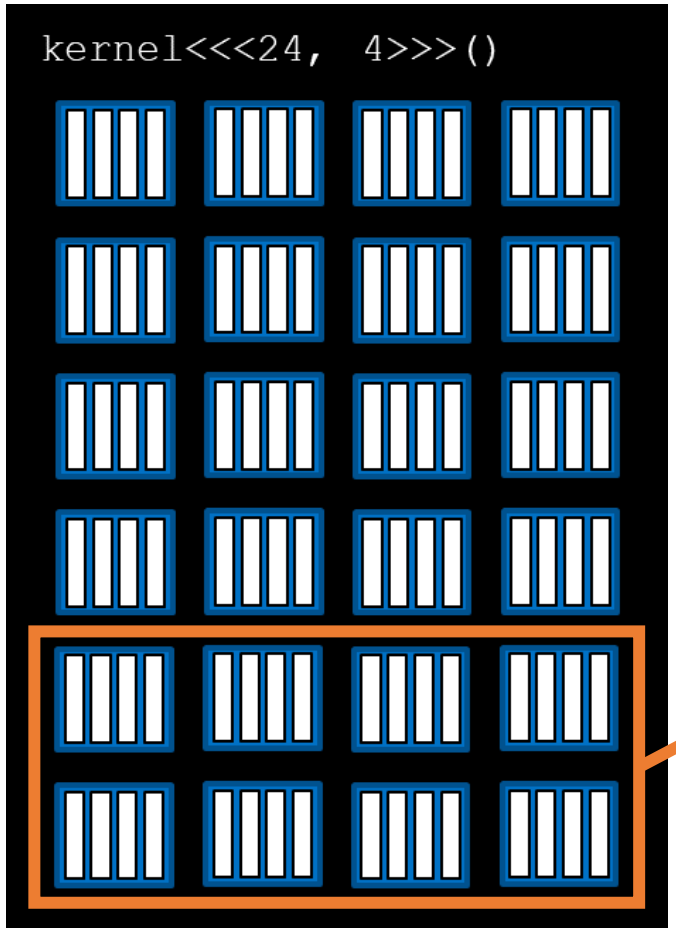


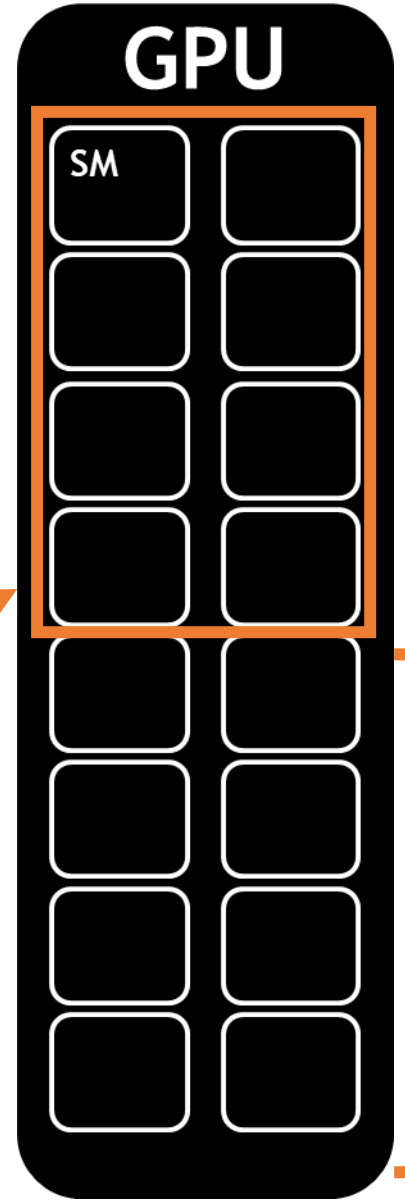


GPU with
16 SMs

SMs and warps

Grid dimensions divisible by the number of SMs on a GPU can promote full SM utilization





GPU with 16 SMs

Unused SMs

Programmatically querying GPU device properties

- **The number of SMs on a GPU** can differ depending on the specific GPU being used, so the number of SMs should **not be hard-coded** into a code bases.
- This information should be acquired **programatically**.

- To obtain the id of the currently active GPU:

```
int deviceId;
cudaGetDevice(&deviceId);
```

- To obtain a C struct which contains many properties about the currently active GPU device, including its number of SMs:

```
cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceId);
```

Exercise: 14_query_device.cu

- This code contains many unassigned variables: it will print gibberish information intended to describe details about the currently active GPU.

- Modify the code to print the actual values for the **desired device properties** as indicated.

- In order to support your work, and as an introduction to them, use the CUDA Runtime Docs (https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html) to help identify the relevant properties in the device props struct.

Exercise: 15_vector_add_sm.cu

- Refactor the `addVectorsInto` kernel you have been working on in previous exercise so that it launches with a **grid** containing a number of blocks that is a **multiple of the number of SMs on the device**.

- Depending on other specific details in the code you have written, this refactor *may or may not improve*, or significantly change, the performance of your kernel. In the next days, you may want to use *nsys* profiler to quantitatively evaluate performance changes.

Unified Memory details

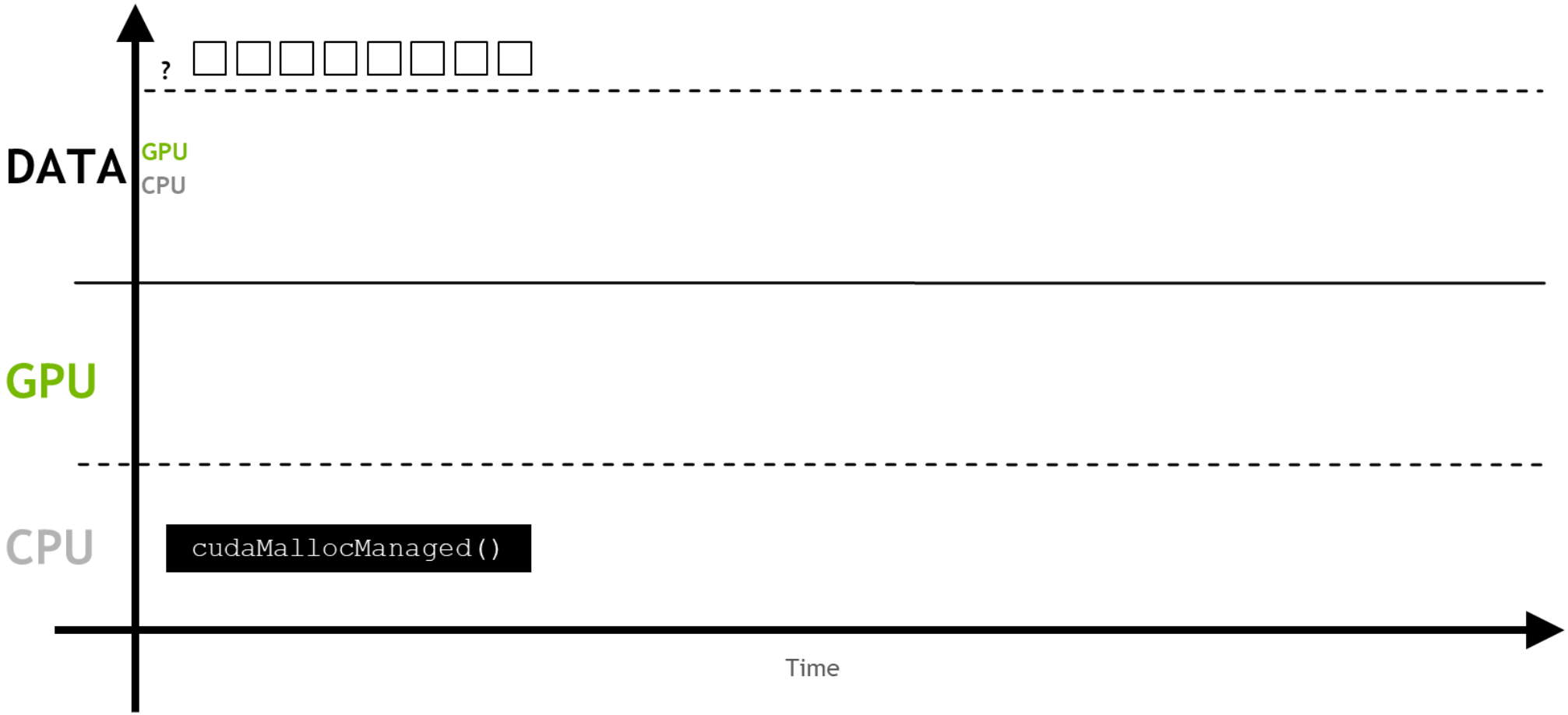
Unified Memory details

- Unified Memory (UM) allows **automatic memory migration** and makes programming **easier**.

- How does `cudaMallocManaged` actually works?

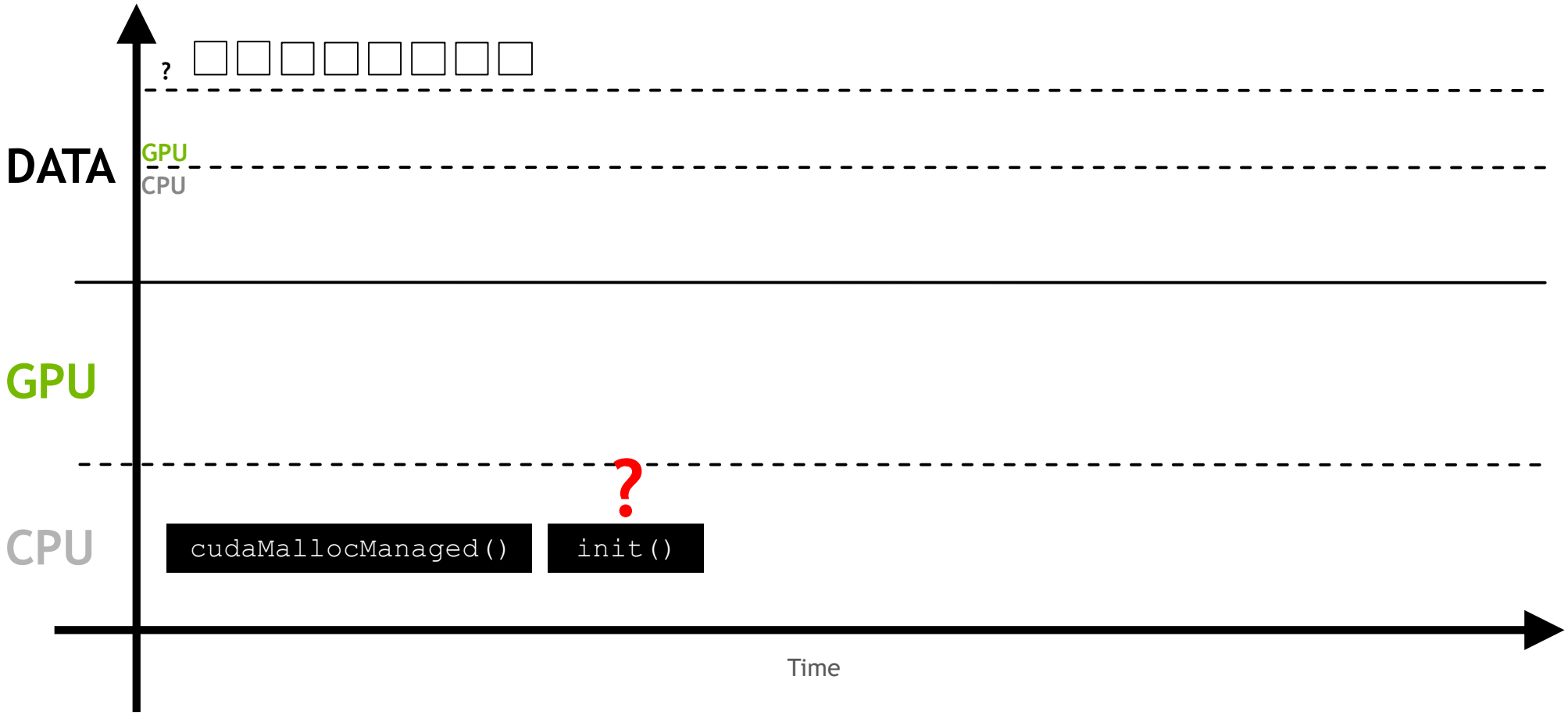
Unified Memory details

When **UM** is allocated, the memory is not resident yet on either the host or the device.



Unified Memory details

When the host or the device attempts to access the memory, a **page fault** will occur.



Focus: what is a page fault?

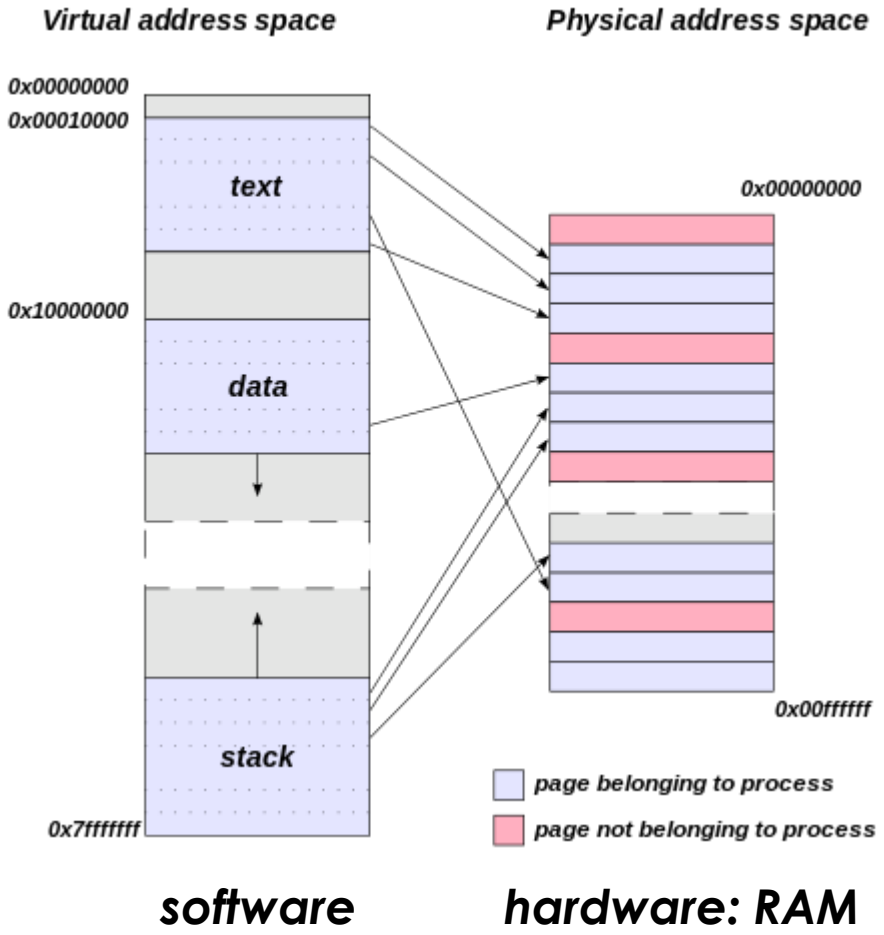
But first: what is a page?

A **page**, **memory page**, or **virtual page,** is a *fixed-length contiguous block of virtual memory*, described by a single entry in the page table.

But first: what is a page?

A **page**, **memory page**, or **virtual page,** is a *fixed-length contiguous block of virtual memory*, described by a single entry in the page table.

The *page table* is a data structure used by a virtual memory system to store the mapping between virtual addresses and physical addresses.

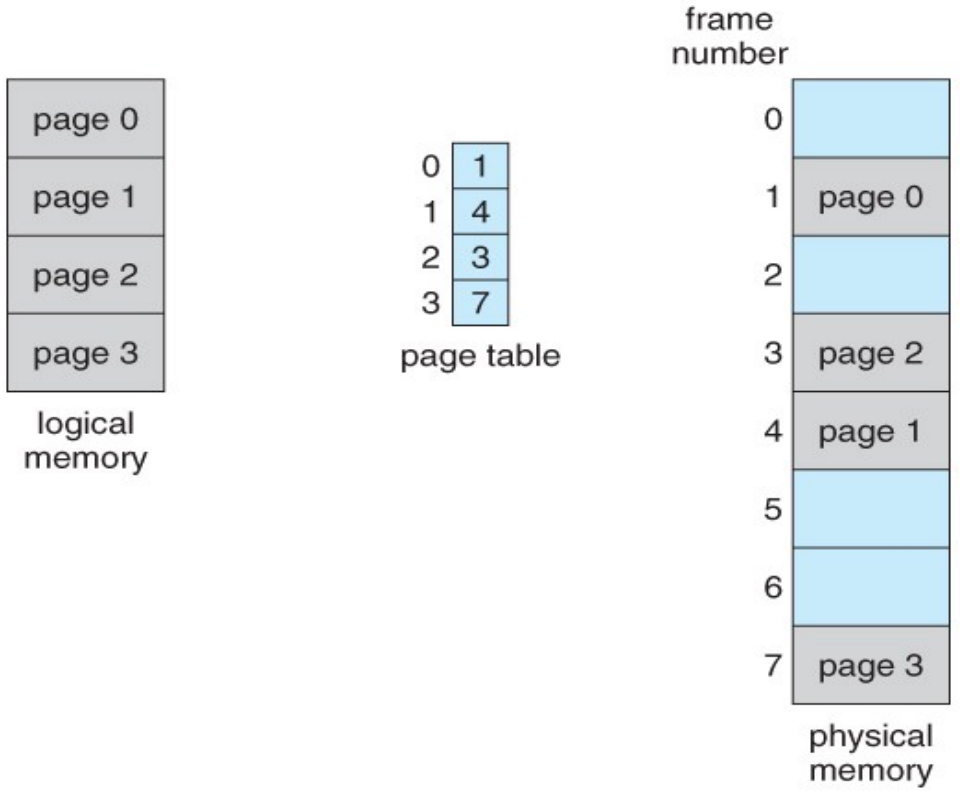


But first: what is a page?

A **page**, **memory page**, or **virtual page,** is a *fixed-length contiguous block of virtual memory*, described by a single entry in the page table.

The *page table* is a data structure used by a virtual memory system to store the mapping between virtual addresses and physical addresses.

It is necessary to the mapping needed to access data in memory.



But first: what is a page?

A **page**, **memory page**, or **virtual page,** is a *fixed-length contiguous block of virtual memory*, described by a single entry in the page table.
It is the *smallest unit of data* for memory management in a virtual memory operating system.

Similarly, a **page frame** is the smallest fixed-length contiguous block of physical memory into which memory pages are mapped by the operating system.

A transfer of pages between main memory and an auxiliary store, such as a hard disk drive, is referred to as **paging or swapping**.

Focus: what is a page fault?

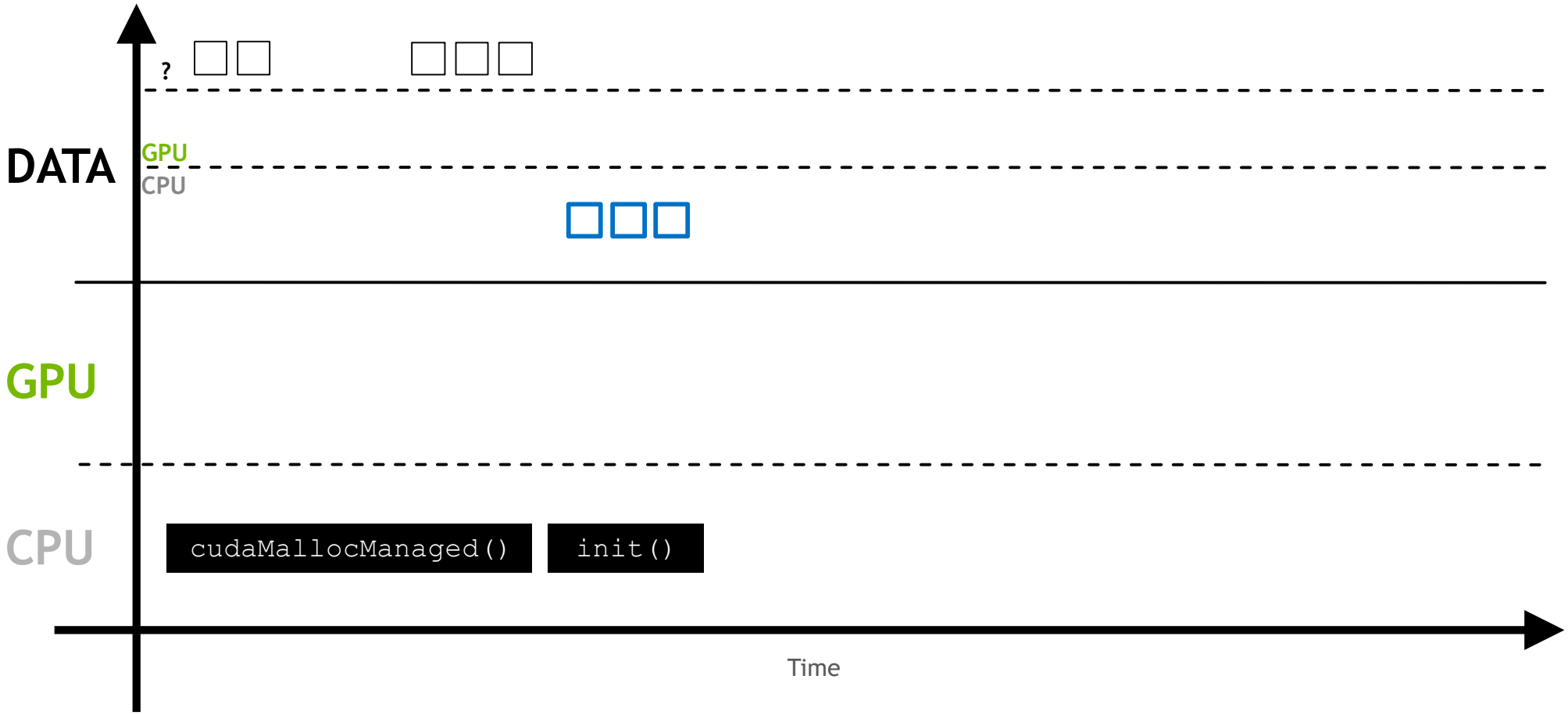
A page fault is an **exception** that the memory management unit (MMU) raises **when a process accesses a memory page without proper preparations** (no valid mapping from virtual to physical address space or page content not in the RAM).

This exception force the operating system to ***find the block in virtual memory and made it accessible***.
Sometimes an invalid page fault can occur (for example when the system tries to access a memory address that does not exist).

Most page faults are usually **valid**: these are common and necessary to increase the amount of memory available to programs in any operating system that utilizes virtual memory.

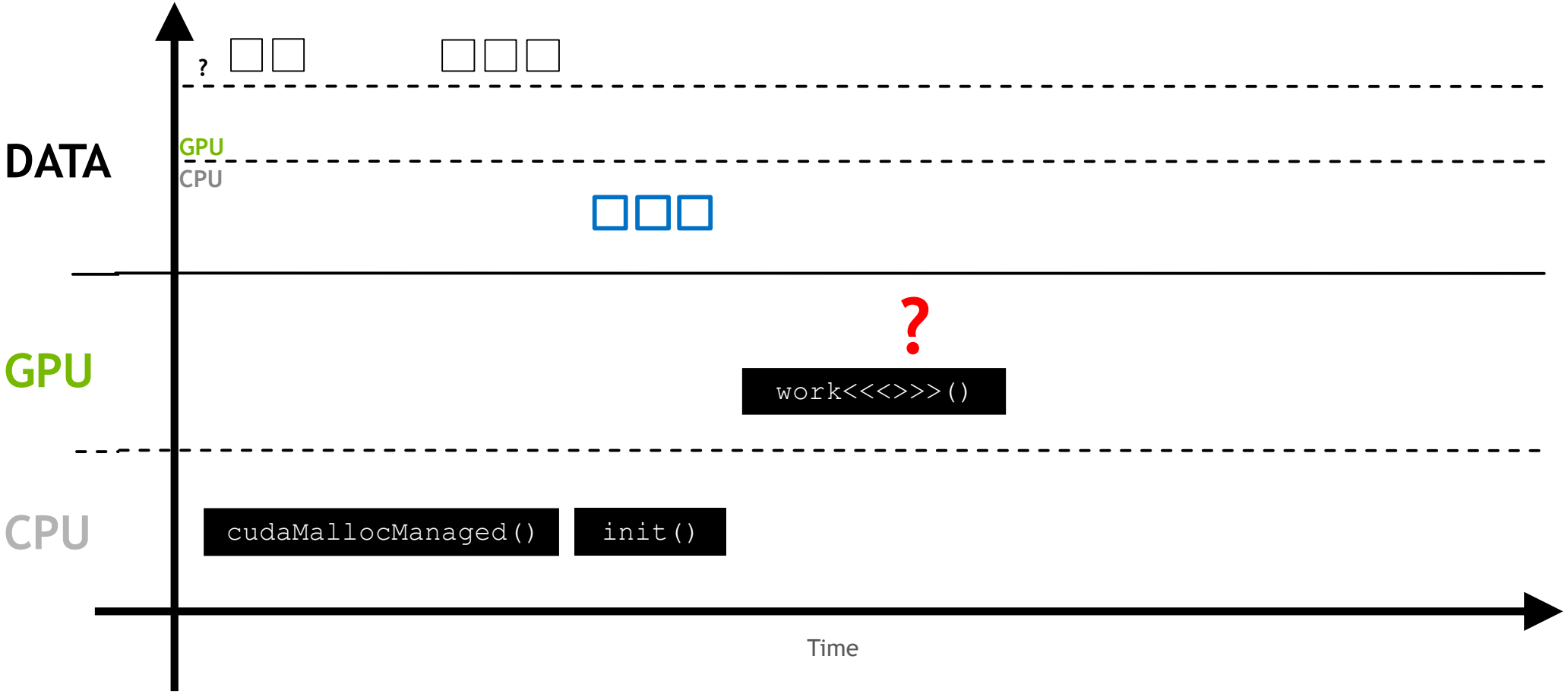
Unified Memory details

This page fault triggers the migration of the required memory.



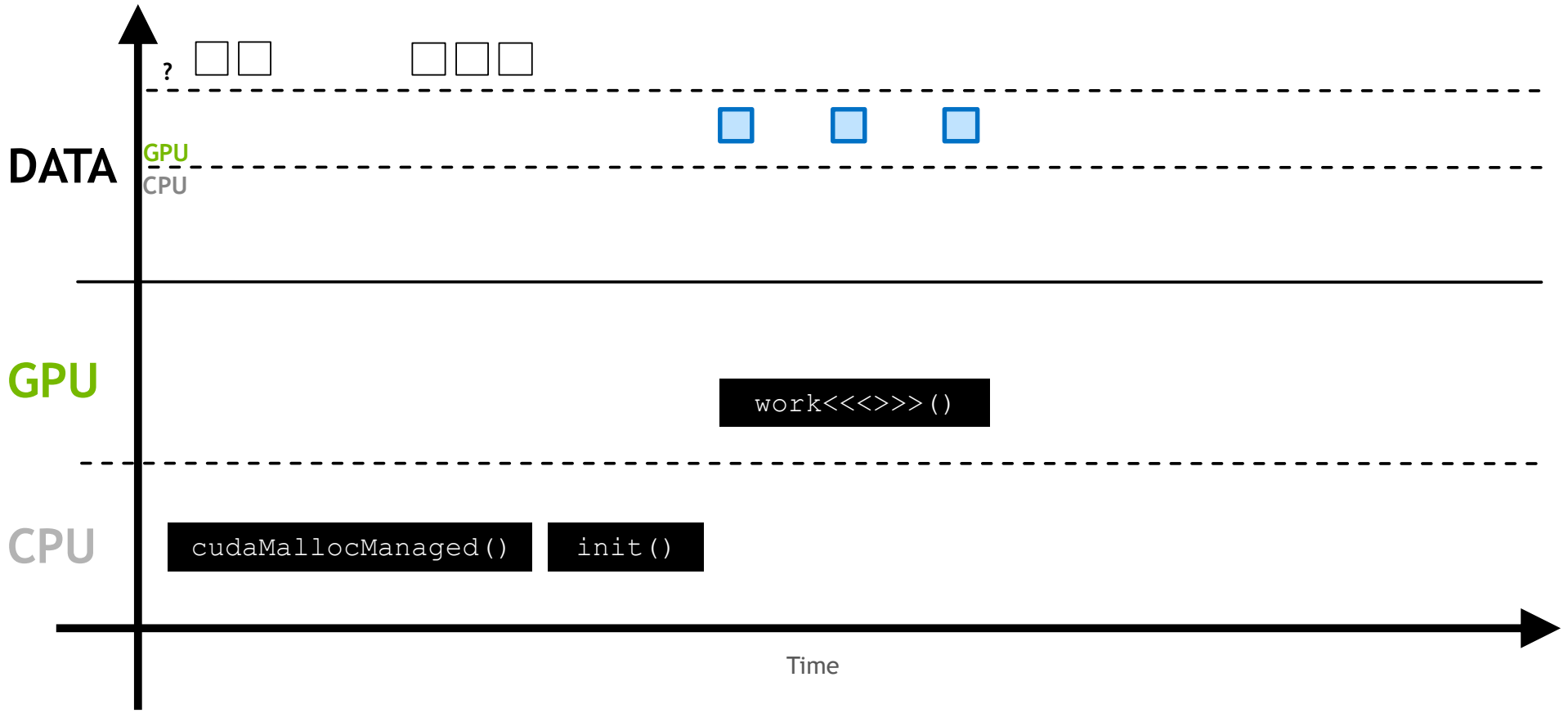
Unified Memory details

This process repeats every time the memory is requested somewhere in the system where it is not resident.



Unified Memory details

Here the page fault triggers the migration of the requested memory to the GPU.



Exercise: 16_page_faults

In this exercise we will explore how UM data migration behaves under the hood.
The code contains two functions that fills a vector (of 33 million elements). One, `hostFunction`, does the work in the CPU, while `deviceKernel` works on the GPU. Currently, no function call is done.

Let's see what happen in different scenarios. We will use text output of `nsys profile --stats=true`. Answer these three questions for each scenario.

1. Is there a ***CUDA Memory Operation Statistics*** section in the output?
2. If yes, does it indicate *host to device (**HtoD**)* or *device to host (**DtoH**)* migrations?
3. If there are migrations, how **many *operations*** there were? (many operations = on-demand page faulting)

Exercise: 16_page_faults

The scenarios to analyze are the following.

1. Only `hostFunction` is called.
2. Only `deviceKernel` is called.
3. Access UM first from the host and then from the GPU.
4. Access UM first from the device and then from the CPU.

Exercise: 17_init_vector_kernel

In this exercise you will refactor the routine that initializes vectors to be added to be a CUDA parallel kernel, instead of CPU function.
Profile your solution with `nsys profile --stats=true`.
Try to answer the following questions before collecting data from the profiler.

1. How will the UM memory migration behavior change?

2. Will the runtime of `addVectorsInto` change?

Asynchronous Memory Prefetching

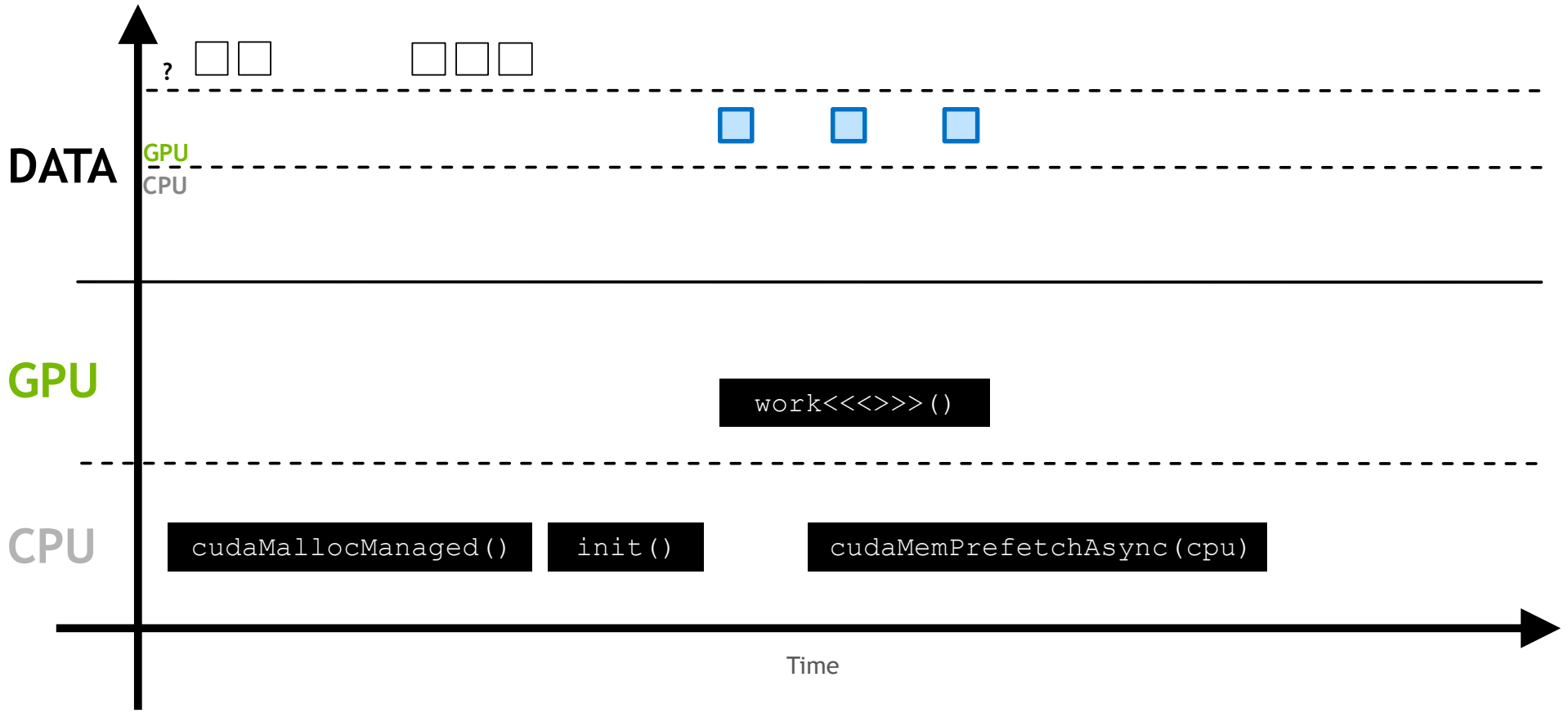
Unified Memory: prefetching

- Using UM automatic memory migration is useful especially when:
  - working with data that **exhibits sparse access patterns,** for example when it is impossible to know which data will be required to be worked on until the application actually runs,
  - when data might be **accessed by multiple GPU devices** in systems with multiple GPUs.

- On the other hand, there are some cases in which the **overhead** of page faulting and on demand data migration are quite high and would be **better avoided**:
  - when *data needs are known prior* to runtime,
  - when *large contiguous blocks of memory* are required.

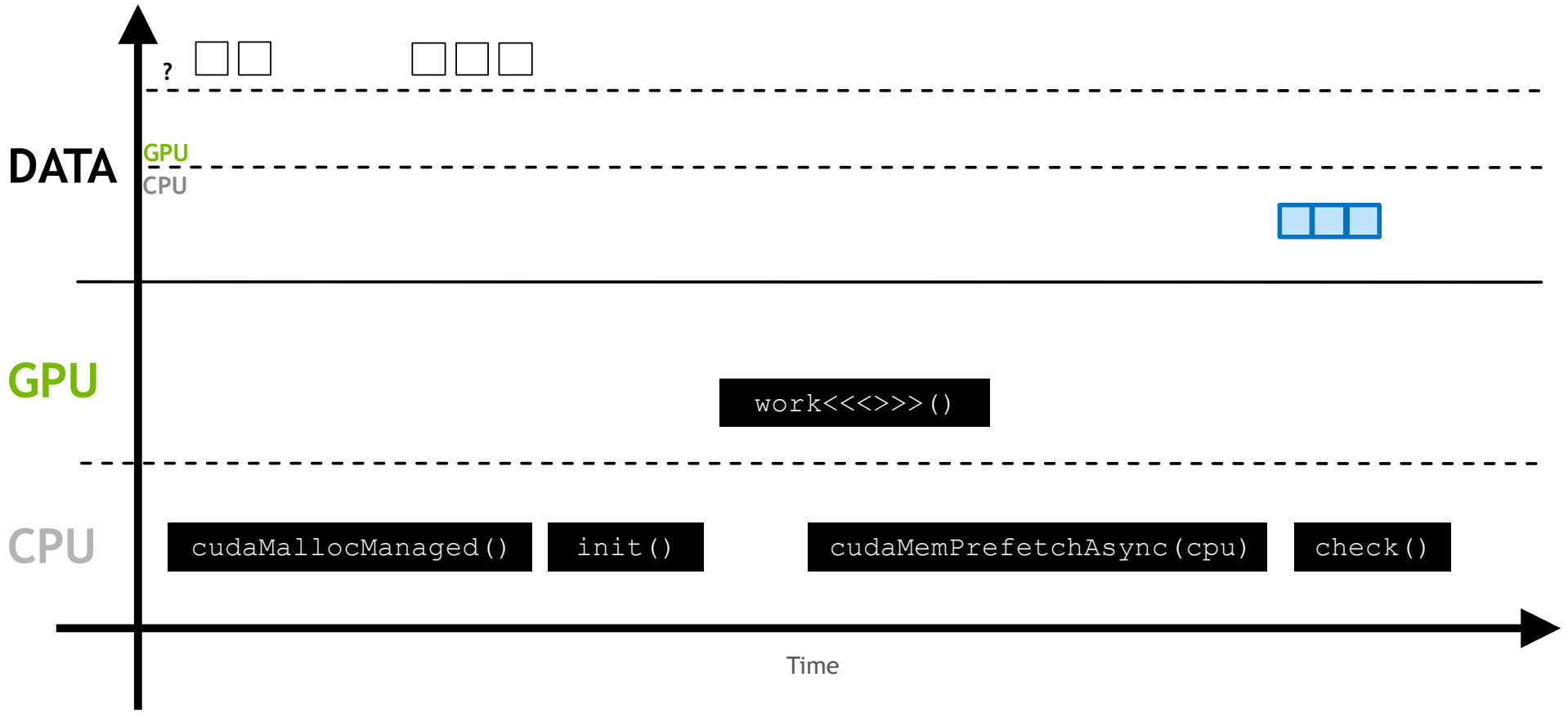
Unified Memory: prefetching

If it is known that the memory will be accessed somewhere it is not resident, **asynchronous prefetching** can be used.



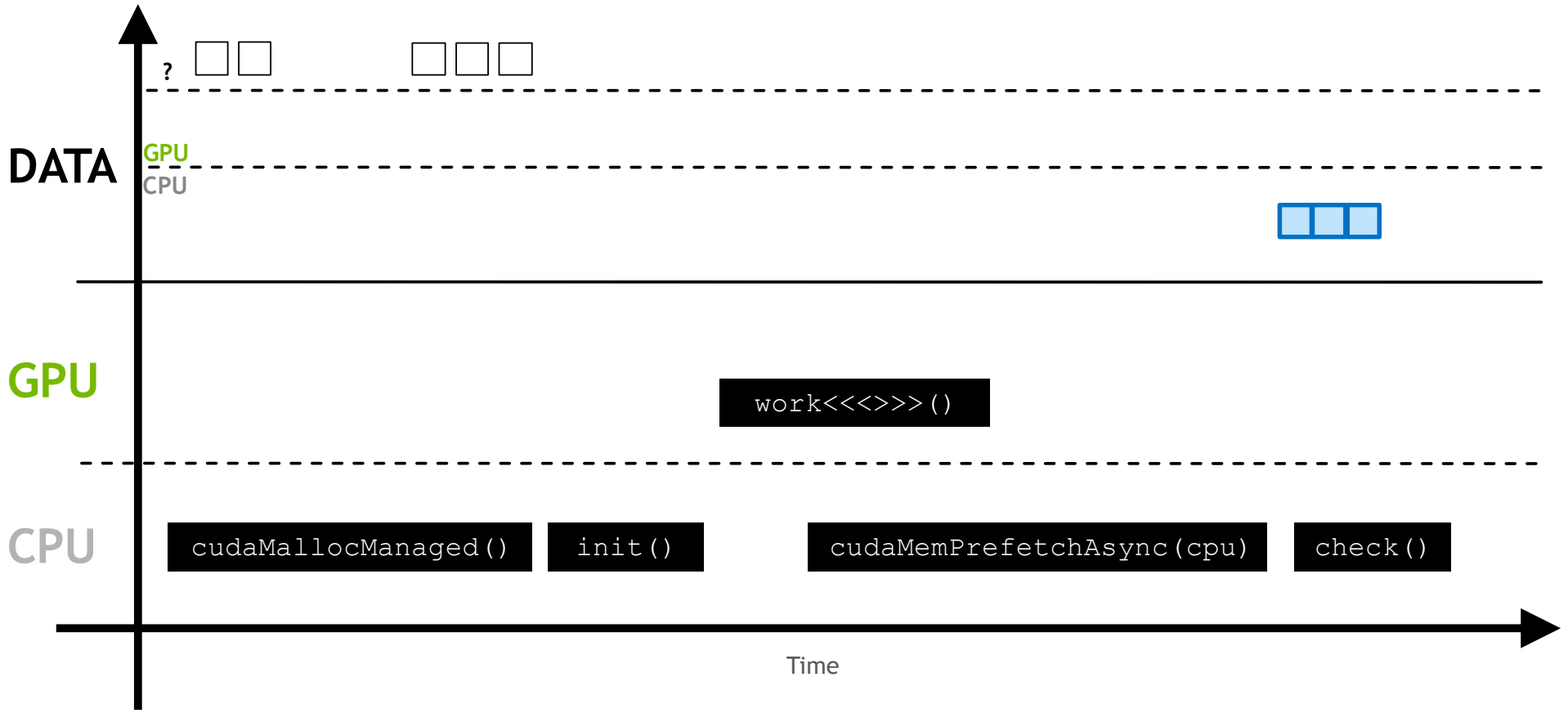
Unified Memory: prefetching

This moves the memory in larger batches, and prevents page faulting.



Unified Memory: prefetching

This moves the memory in larger batches, and prevents page faulting.



Asynchronous Memory Prefetching: sum up

- Asynchronous memory prefetching allows to **asynchronously migrate unified memory to any CPU or GPU device in the system**, explicitly, but in the background, before they are needed by application code itself.

- Increase of GPU kernels and CPU function performance thanks to **reduced page fault and on-demand data migration overhead.**

- **Data** tends to be migrated **in larger chunks,** so fewer trips are needed.

- Can be used when data access needs are known before runtime, and when data access patterns are not sparse.

Asynchronous Memory Prefetching: how to

CUDA uses `cudaMemPrefetchAsync` function to asynchronously prefetching managed memory to either a GPU device or the CPU easily.

```
int deviceId;
cudaGetDevice(&deviceId);

cudaMemPrefetchAsync(pointerToSomeUMData, size, deviceId);

cudaMemPrefetchAsync(pointerToSomeUMData, size, cudaCpuDeviceId);
```

prefetch to GPU

prefetch to host

built-in variable

Exercise: 18_vector_add_prefetch.cu

Now, our vector add application initializes vectors and sum 2 of them with CUDA kernels.
To further understand memory migration and page-faulting, try these four scenarios. Make hypotheses on UM behavior/page-faulting and initialization kernel runtime.

1. What happens when you **prefetch one of the initialized vectors** to the device?
2. What happens when you **prefetch two of the initialized vectors** to the device?
3. What happens when you **prefetch all three of the initialized vectors** to the device?
4. What happens when you prefetch all three initialized vectors to the device and **prefetch also back to the CPU** before correctness check?

References

References

- Previous editions of this school at CINECA
- Oakridge National Laboratory's "Introduction to CUDA C++": https://www.olcf.ornl.gov/calendar/introduction-to-cuda-c/
- NVIDIA DL Institute Online Course: **main source of exercises**
- www.computerhope.com/jargon/p/pagefaul.htm
- blogs.nvidia.com
- Wikipedia

THANK YOU!

Lara Querciagrossa
l.querciagrossa@cineca.it