Introduction to programming the GPU

Formations HPC, CÉCI
Pieter Heremans, UCLouvain
3 december 2025

Overview

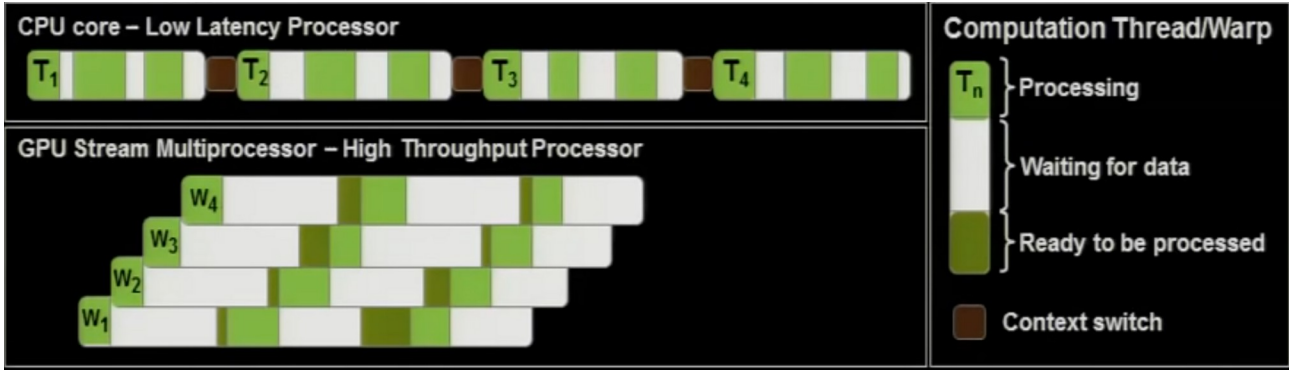
- Difference between CPU and GPU
  - Why and when to use a GPU?
- What is CUDA?
  - When/where can I use cuda?
- Structure of a GPU program
  - Nomenclature
- First example of CUDA programming
- First step in optimisation of a CUDA program
  - Managing memory transfer

CPU and GPU

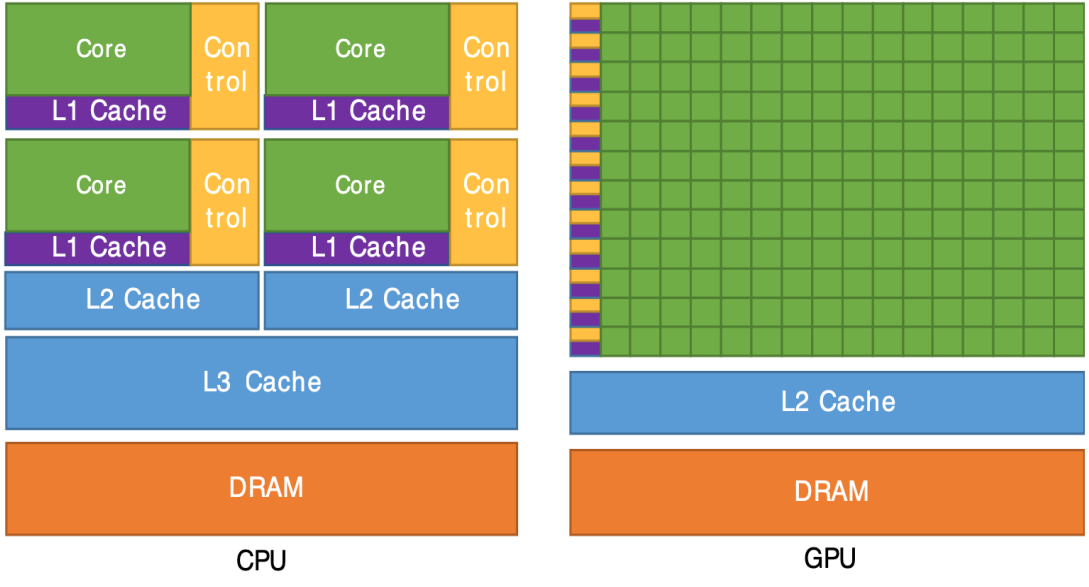
- ‘Graphics’ Processing Unit
  - A bit of history
    - 3d acceleration - mid 1990’s (fixed pipeline)
    - Shaders (2001) and ‘GPGPU’
      - Cast texture memory to scalar types to perform ‘computing shader’
    - Frameworks providing low-level access to GPU hardware : SIMT (cuda/rocm/sycl/etc.) (2007)

CPU and GPU

- High Throughput, but high Latency
  - Speed: number of operation per second
  - Latency: delay in the first operation

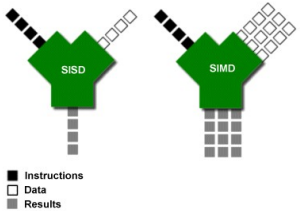


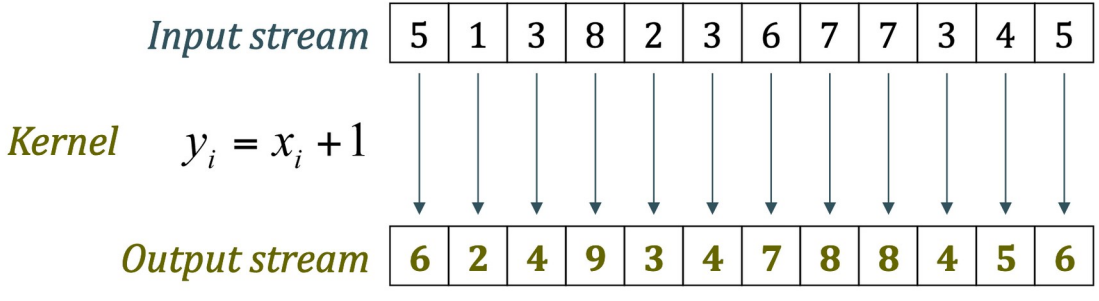
CPU and GPU


Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
L2 Cache
L2 Cache
L3 Cache
DRAM
CPU
L2 Cache
DRAM
GPU

Stream computing

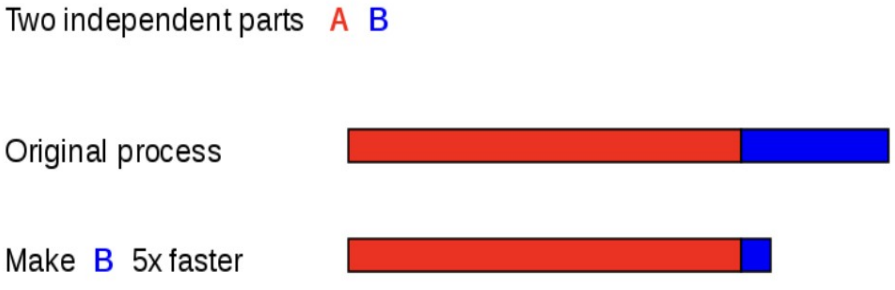
- ‘single instruction, multiple threads’ SIMT
  - cfr. SIMD
    - multiple data : vector instructions on CPU
  - Threads, with synchronisation/atomic operations and local registers, shared memory

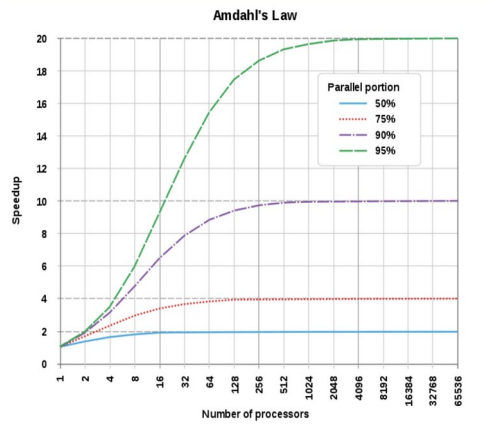




CPU and GPU

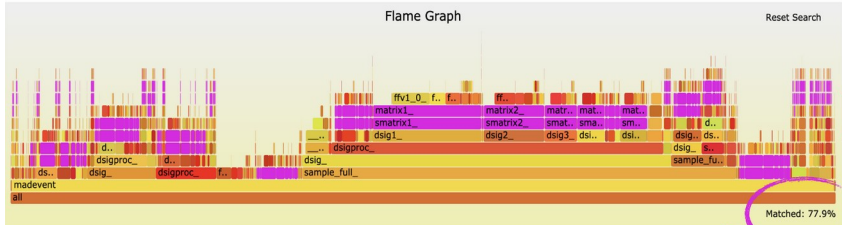
- A CPU has 8 cores a GPU 2056 cores
  - *Should my code should be 200 times faster?*
- which part of your code can use parallelism ?





Speedup in practice

- Comparing speed of code between cpu and gpu are not really fair
  - Cost of the GPU/CPU
  - A “normal” speedup is around 5-20
  - Huge speed-up typically means non-optimized reference
- GPU clock is slower than CPU clock
  - GPU ~ MHz
  - CPU ~ GHz

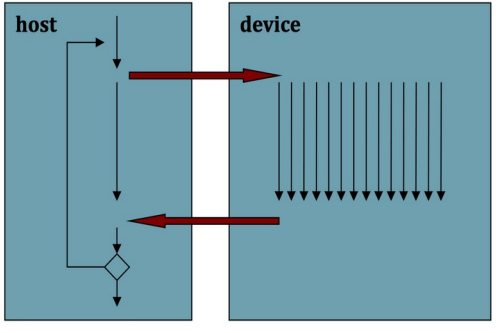


CUDA

- Hardware vendors propose their low-level framework to program the GPU :
  - CUDA
    - Released in 2007
    - targets NVidia GPUs
  - HIP (compatibility layer for AMD/ROCm)
  - Oneapi (intel)
  - ...

CUDA Programming model

- A GPU is controlled by a CPU:
  - All programs start on the CPU
  - Data are prepared on the CPU and moved to the GPU
  - GPU is crunching data
  - Move data back to CPU
  - End the program



Kernel execution is asynchronous

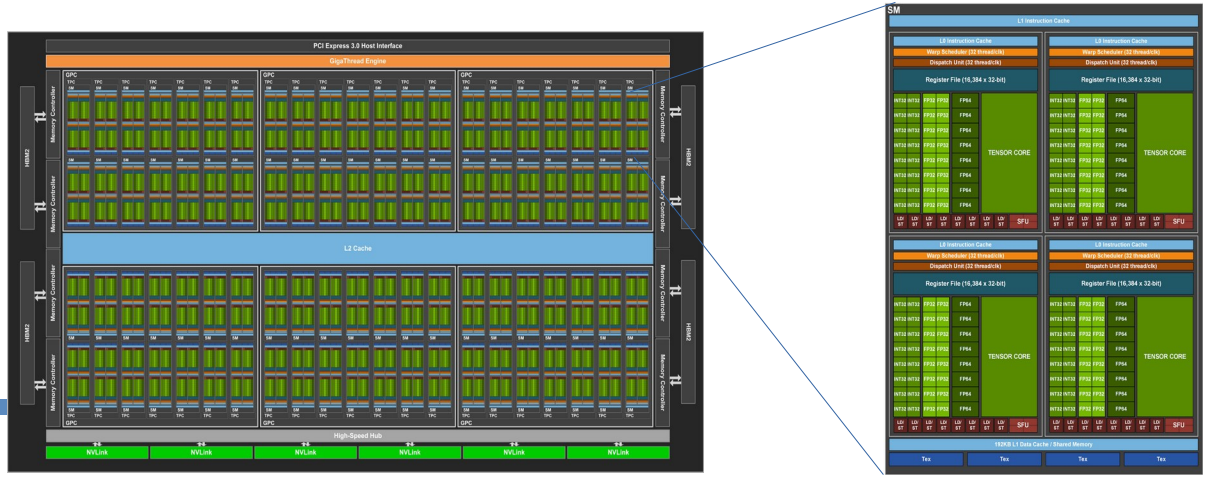
Asynchronous memory transfers also available

CUDA Programming model

- The CPU is called the **“host”**
- The GPU is called the **“device”**
  - Viewed as a co-processor
- Function executed on GPU are called **kernel**
  - Executed in parallel (on different data)
- Both the host/device have their own memory
  - Memory management is handled by the host
  - Automatic management is possible
  - Physically shared memory also exists (APU)

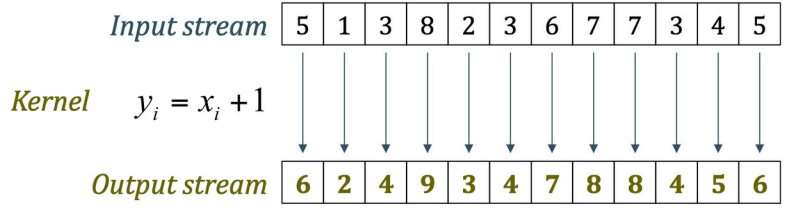
Multi-processor/block/thread

- Main components
  - Streaming-Multiprocessor
  - Memory



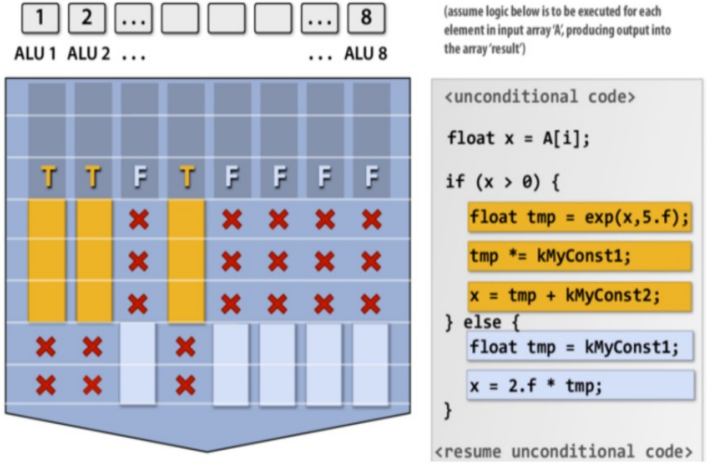
Block

- Thread are grouped by block
  - Collaboration between threads
    - synchronization, atomic-memory operations
    - shared memory
- Up to 2048 threads per block
- Blocks are fully independent
  - Can be executed in any order
  - Can be executed on different GPU



Execution model

- block are organised in ‘warp’ of 32 threads
- Those 32 threads are working in lockstep
  - Run the same command at the ‘same’ time
  - ‘If’ statement slows down the code



SAXPY

- In a demonstration, we'll have a look at developing a 'saxpy' kernel
- Kernel
  - Calculates a * x[i] + y[i]

```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>

void saxpy_cpu(float* vecX, float* vecY, float alpha, int n)
{
  for (int i=0; i < n; i++){
    vecY[i] = alpha * vecX[i] + vecY[i];
  }
}

int main(){

  int N = 1<<20; // 2^20 = 1,048,576
  float* x;
  float* y;
  x = (float *) malloc(N*sizeof(float));
  y = (float *) malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  saxpy_cpu(x, y, 2.f, N);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, std::abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);


}
```

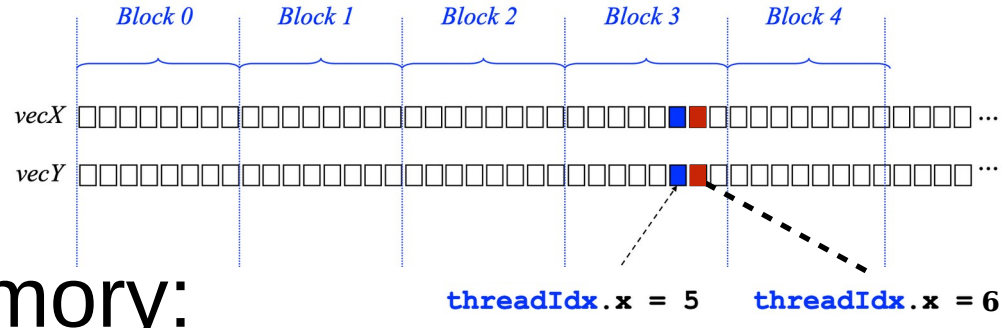
1. Initialize GPU
2. Initialize variable on the host (cpu)
3. Allocate memory on the device (gpu)
4. Move data from host to device
5. Execute kernel on device
6. Move back results
7. Clean up (deallocation)

kernel

- No loop anymore !!
  - Each thread will take care of one (scalar) data
  - Need to compute which element each thread has to handle.
  - Various variable defined for that
    - blockIdx.x (.y / .z if 2D and 3D): id of the current block
    - blockDim.x: number thread in Block (for that dimension)
    - threadIdx.x: id of the current thread inside the block

Kernel call

- How do you call a kernel?
- saxpy<<<numblock, blocksize>>>(d_x, d_y, a, n)
- blocksize: number of thread in a block
  - Should be multiple of 32 (due to wrap)
  - Maximum of 2048
  - depends of the GPU capabilities
- Number of blocks (grid):
  - a block is executed on one SM



- ‘Coalesced’ memory:
  - Efficient access pattern where threads in a warp read from, or write to, consecutive memory locations

- Module load CUDA
- nvcc -arch=sm_70 saxpy.cu -o saxpy
  - You can have additional flags for C++ par of the code (library linking, -O3,…)
  - Arch allows to have a minimum target gpu
  - No dedicated flag for additional GPU optimisation
  - GPU does support multiple file source code
    - But seriously limit optimisation
    - Cuda11 starts supports for that but still limited.

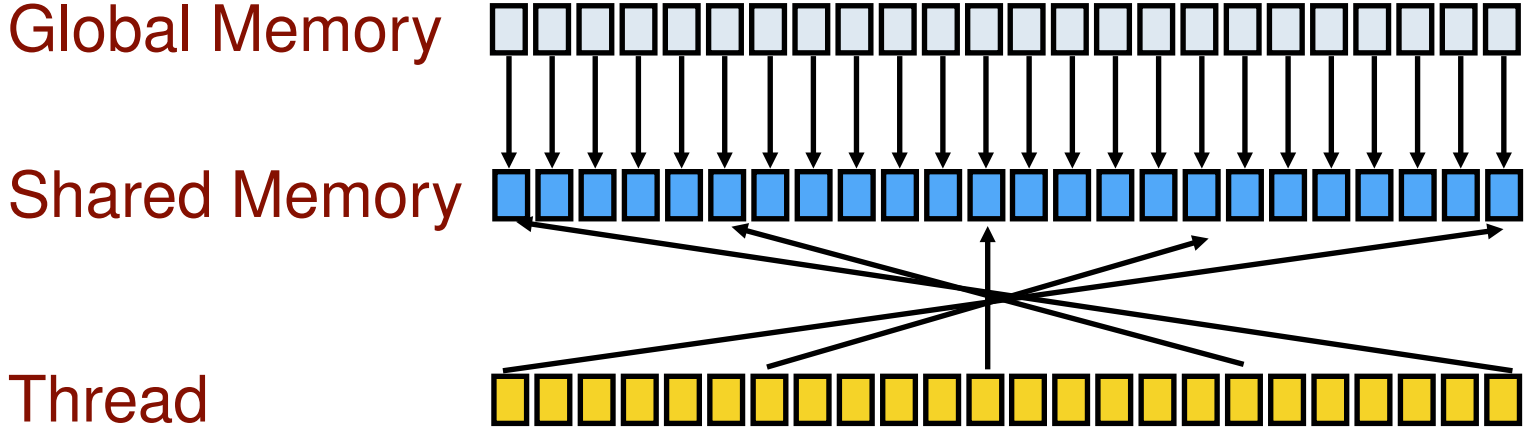
Memory

- You have to manage memory:
  - Plenty of types of memory on the GPU
- In each Stream-Multiprocessor :
  - Register File
    - Fastest memory
    - Thread specific
    - Very limited amount
    - Overflow goes to L1 data cache
  - Shared memory
    - Limited amount (shared with L1 data cache)
    - Block wide memory (could need synchronisation!)
    - __shared__

Memory

- Plenty of type of memory on the GPU
  - Outside the SM
    - Global memory
    - High bandwidth (900Gb/s) but High latency
    - High number of thread needed to hide this latency
    - Default memory for cpu/gpu pointer
- ! Efficiency in memory transfer = maximise the usage of the data transfer, need to use all data in a block before needing a new block transfer

- Coalesced access not possible?
  - Use shared memory as a cache

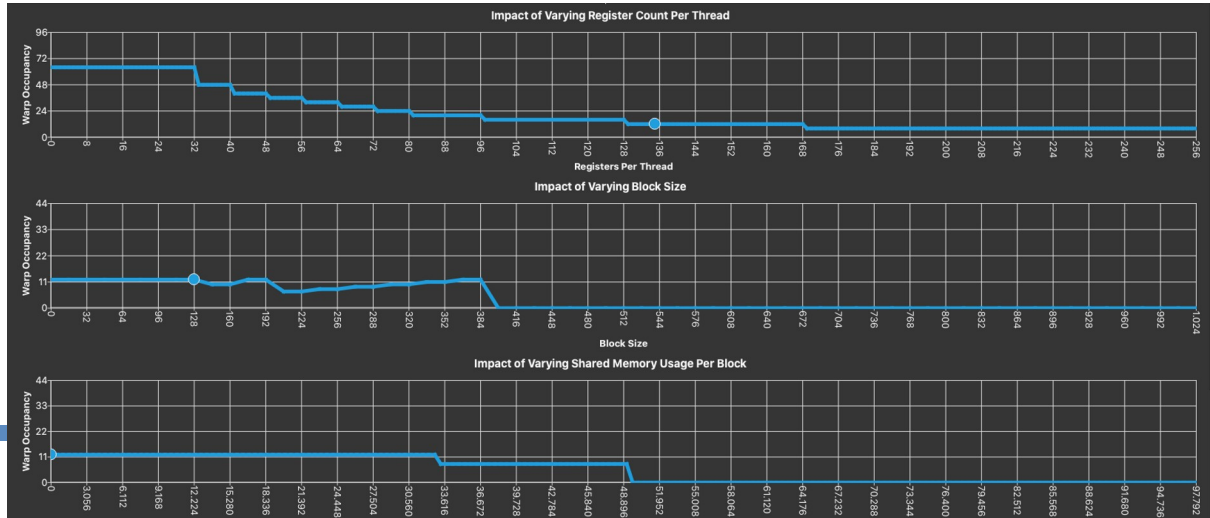


Memory

- Constant memory
  - Will be put in cache (same speed as shared memory)
- Texture memory
  - specific to graphics
- ‘Unified’ Memory
  - Automagically accessible both on host and device
  - cudaMallocManaged(&&x, size)
  - Rarely advantageous for speed

Occupancy

- Occupancy is limited
  - Each SM has limited ressources
    - Maximum number of wrap (64)
    - Maximum number of block (32)
    - Register usage (256Kb)
    - Shared memory usage (64Kb)



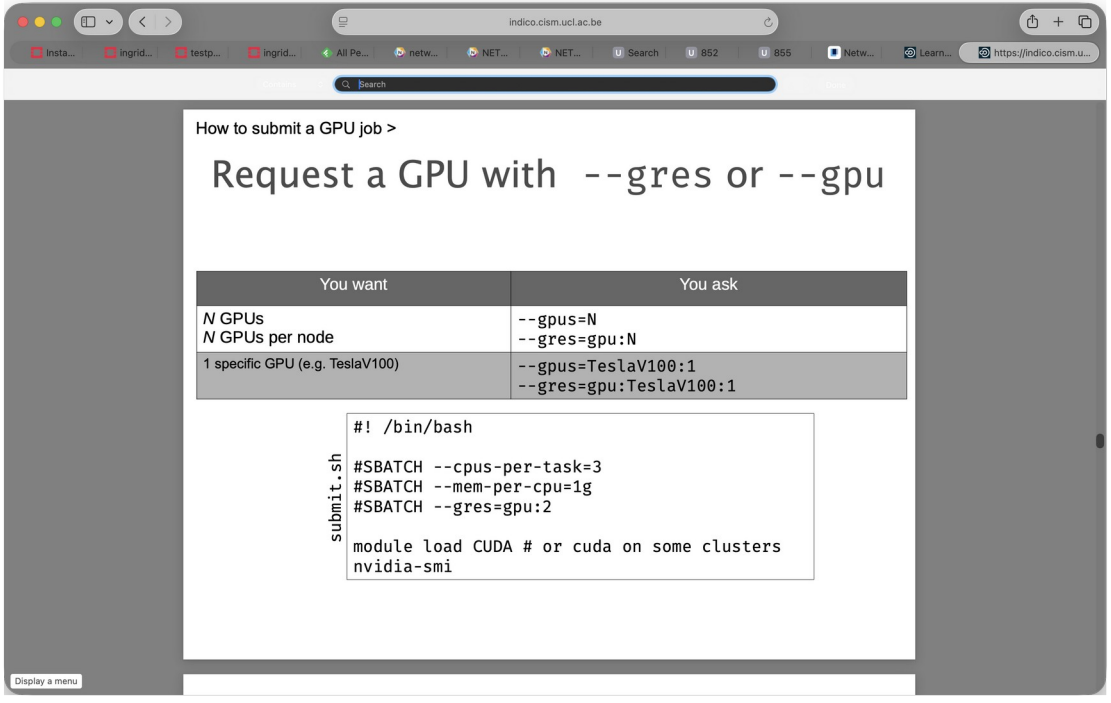
GPU at CÉCI

- Dragon2
  - 2 nodes (2xV100)
- Hercules
  - 1 node (8xA40)
- Lyra
  - 40 nodes (1xRTX6000)
- Lucia (TIER-1 project)
  - 50 nodes (4xA100)
-

GPU at CÉCI

- @CÉCI : Slurm
  - Check resources available
  - sinfo --format="%N %.6D %P %G"
- run interactively
  - srun --gpus=1 --pty bash
- Check module on the machine
  - module av
  - module spider CUDA
- Check that you have access to the GPU
  - nvidia-smi

GPU at CÉCI


indico.cism.ucl.ac.be
How to submit a GPU job >
Request a GPU with --gres or --gpu
You want
You ask
N GPUs
N GPUs per node
--gpus=N
--gres=gpu:N
1 specific GPU (e.g. TeslaV100)
--gpus=TeslaV100:1
--gres=gpu:TeslaV100:1
submit.sh
#! /bin/bash
#SBATCH --cpus-per-task=3
#SBATCH --mem-per-cpu=1g
#SBATCH --gres=gpu:2
module load CUDA # or cuda on some clusters
nvidia-smi
Display a menu