Calcul haute performance pour les systèmes embarqués (HPEC)
Lab 4 - GPU devices & CUDA programming

Lab 4 - GPU devices & CUDA programming

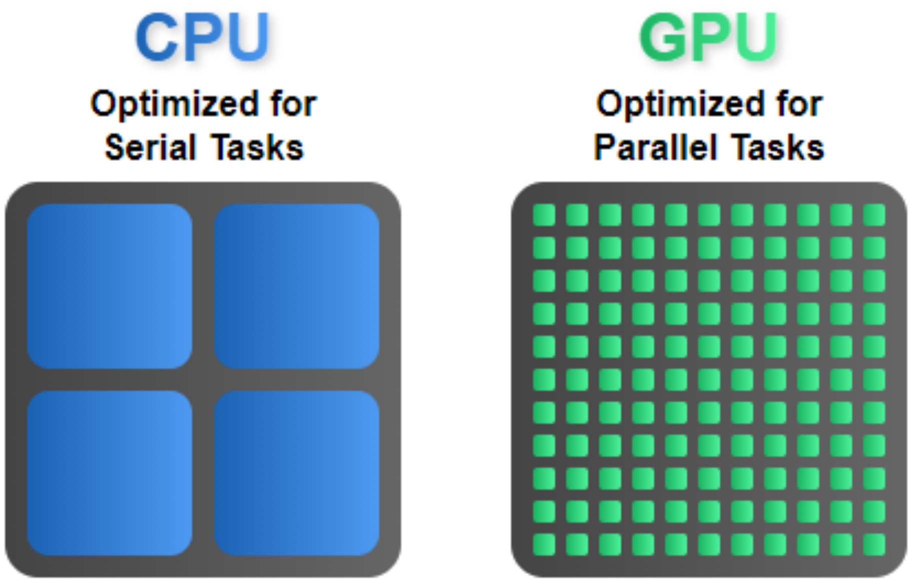
Bertrand LE GAL

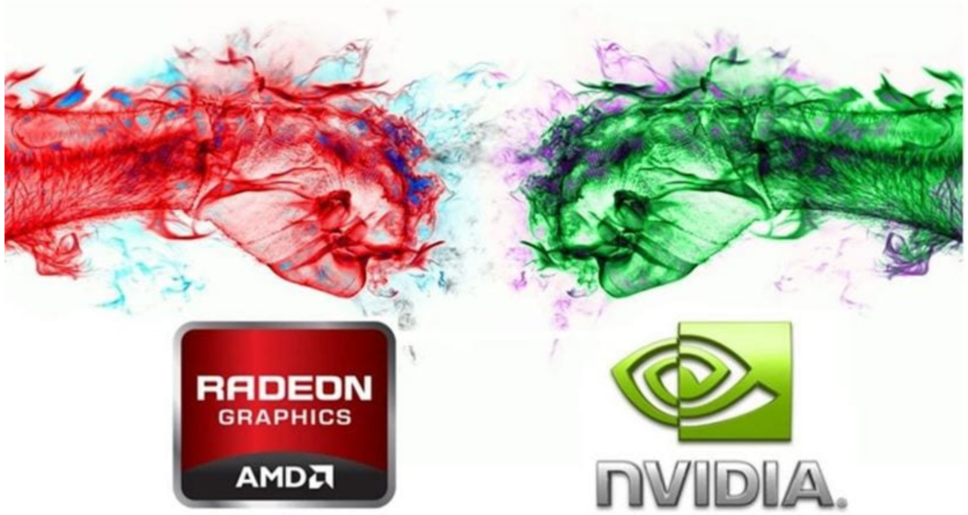
IMS laboratory, CNRS UMR 5218
Digital Circuits and Systems team
Bordeaux-INP, University of Bordeaux
France

bertrand.legal@enseirb-matmeca.fr

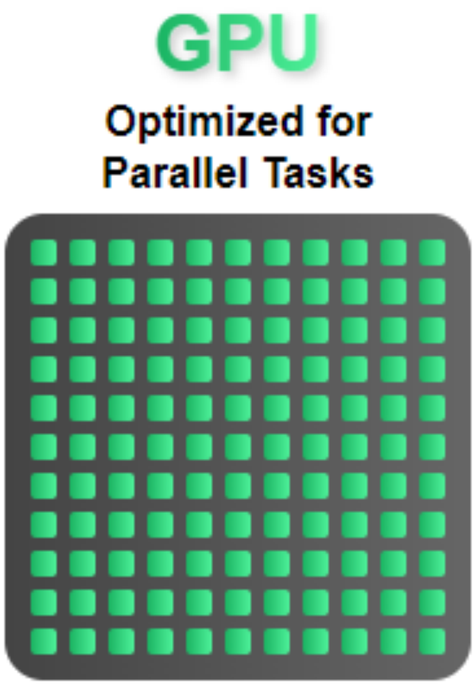
Introduction to GPU devices (1)

- GPU (Graphics Processing Unit)
  - *Processeur graphique en français,*
- Originally developed for graphics processing (OpenGL),
- Diverted from its purpose ~2010,
  - Massively parallel scientific computing,
  - A few tens to thousands of FPU computing cores.
- Currently,
  - digital signal processing, IA, crypto (bitcoin), financial, video processing…
  - Still for video games ?!





Introduction to GPU devices (2)



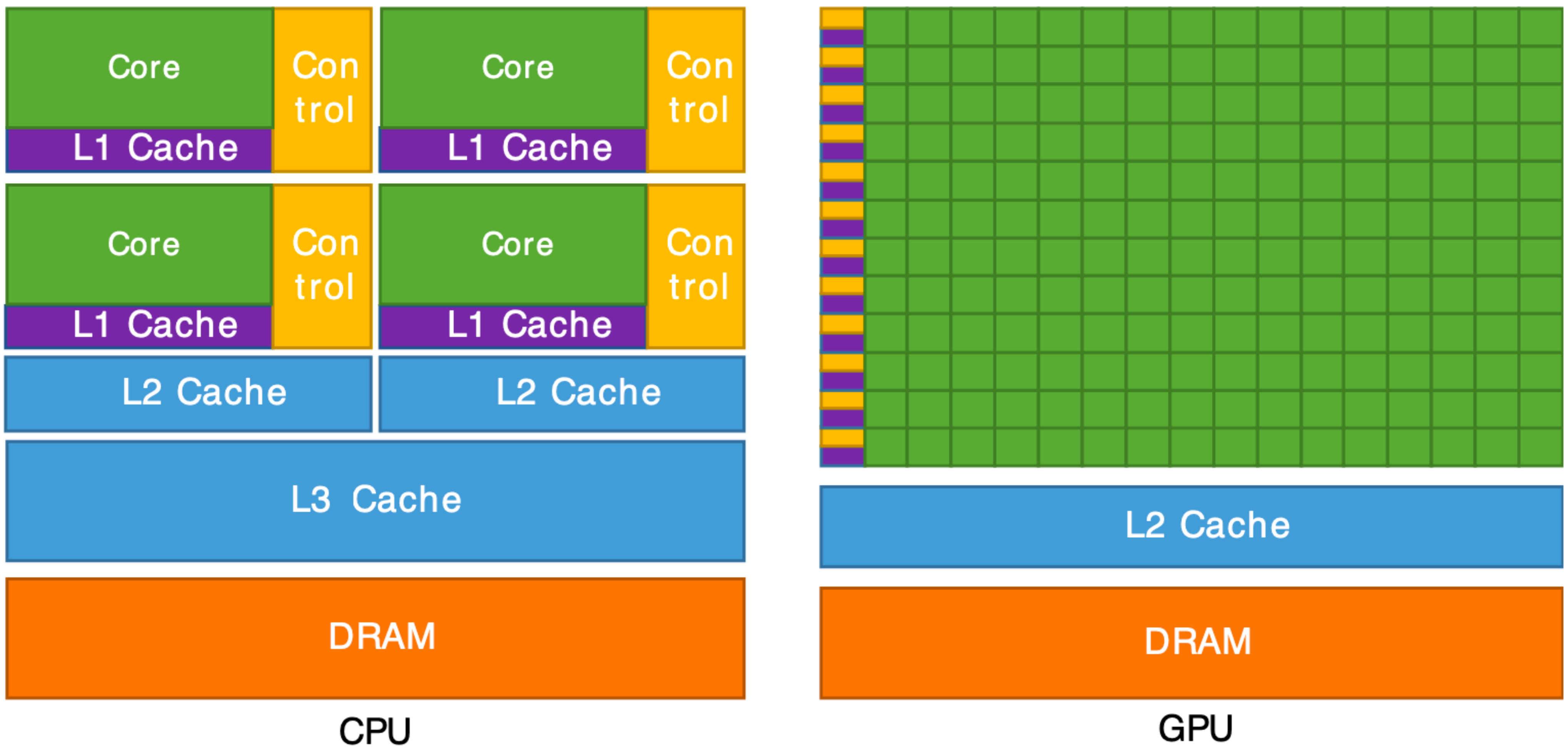
CPU arithmetic resources

- [4 to 8] physical cores
- [8 x float] in SIMD / cycles
- working frequency ~2 to 4 GHz

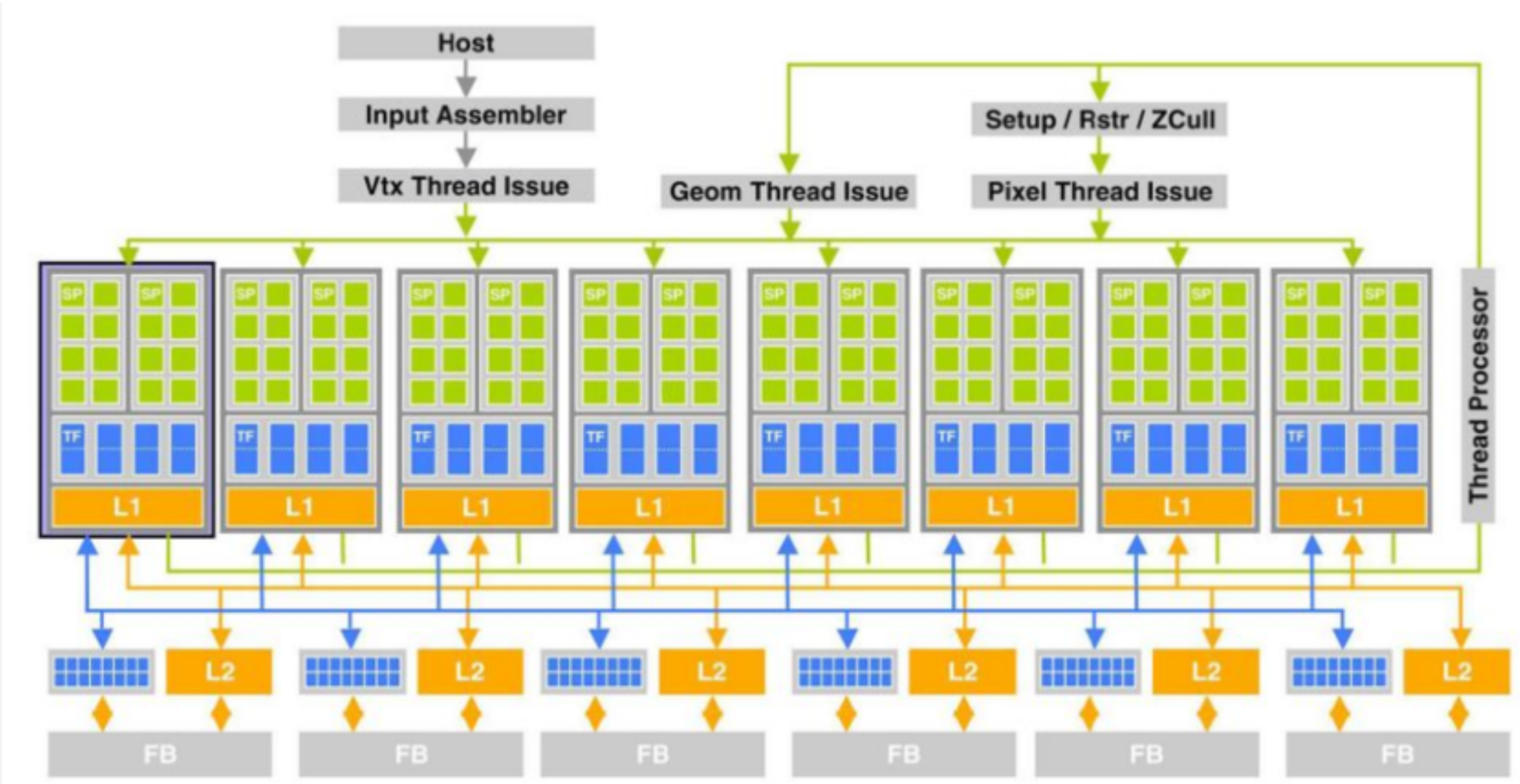
CPU arithmetic resources

- [4 to 20] stream processors
- up to 4608 float / cycles
- working frequency is ~1 GHz

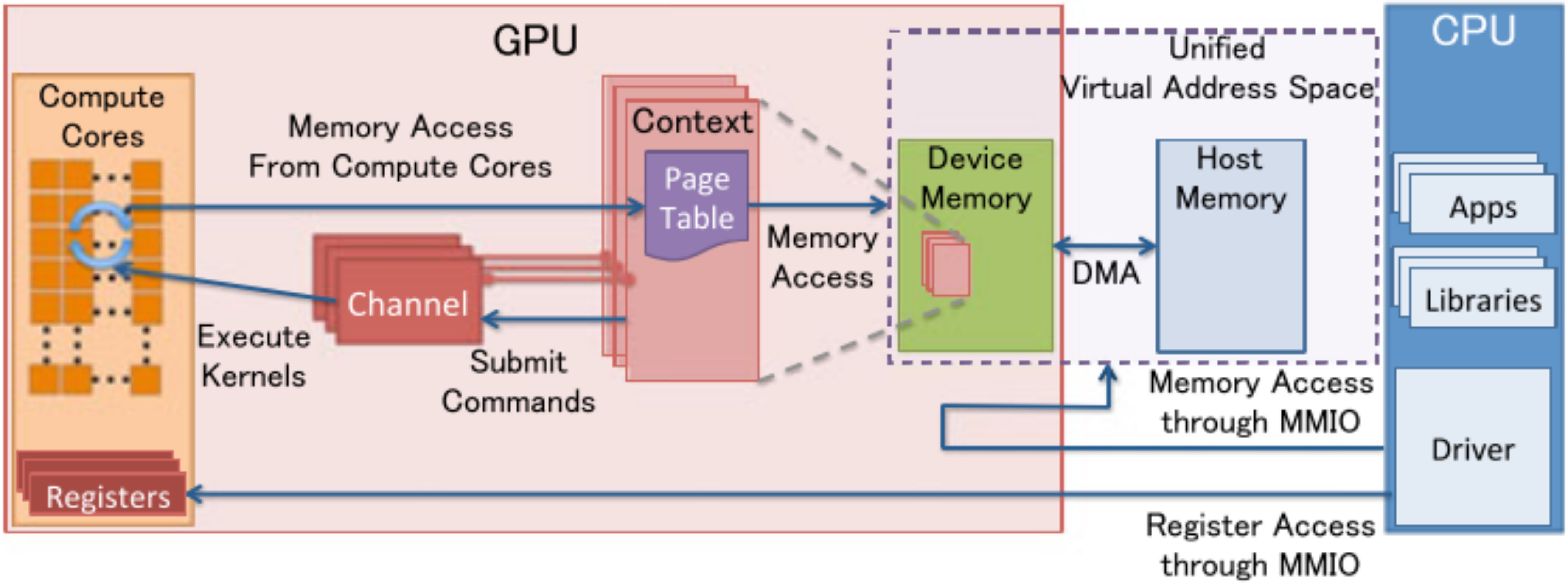
Internal architecture of a GPU (simple)



Internal architecture of a GPU (extended)



Using a GPU device in a embedded system



NVIDIA, AMD … GPU devices

- are slave devices that need an host controller (CPU)
- host controller load instruction sequences (program)
- memory are exchanged host <=> GPU (time penalties)

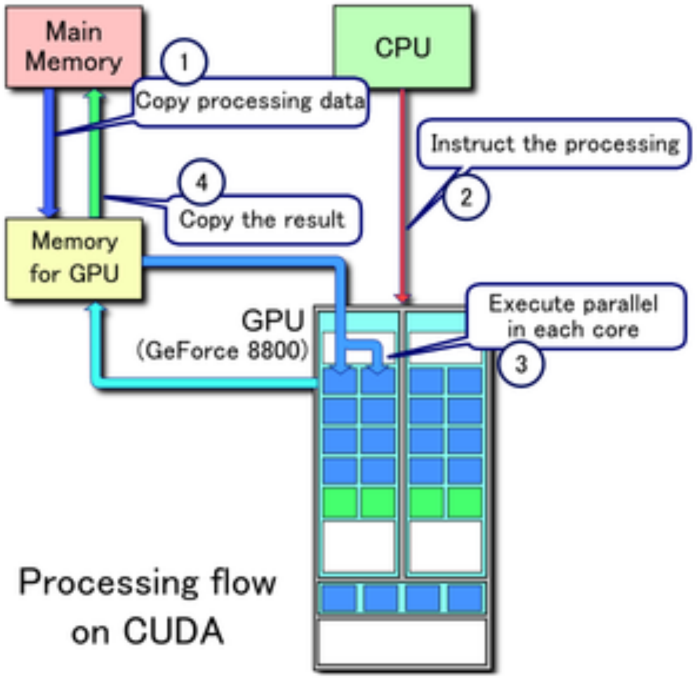
Programming GPU devices

- GPUs could be programmed using different APIs / frameworks
  - **CUDA** (multi-OS & NVIDIA),
  - **OpenCL** (multi-OS & device),
  - **Metal** (MacOS & multi-device),
  - **Vulkan** (multi-OS & device),
  - and so on…
- Programming language depends on device and OS
  - Programming paradigm is still the same,
  - Device initialization, memory copy, and so on … depends on the framework.

GPU programming is not CPU programming !!!

CUDA framework for GPU programming

- CUDA library
  - Compute Unified Device Architecture
  - Works on Windows, Linux, MacOS
- Developed by NVIDIA
  - Version 1.0 (2007)
  - Version 11 (2022)
- A set of optimized libraries exists
  - cuBLAS
    CUDA Basic Linear Algebra Subroutines
  - cuFFT
    CUDA Fast Fourier Transform
  - And so on…
- Algorithms should be optimized.



Example of GPU programming (not working)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__
void mykernel(float *A1, float *A2, float *R)
{
   int p = threadIdx.x;
   R[p] = A1[p] + A2[p];
}

int main()
{
   float A1[] = {  1,  2,  3,  4,  5,  6,  7,  8,  9 };
   float A2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
   float R[9];

   mykernel<<<1 ,9>>>(A1, A2, R);

   for (int i = 0; i < 9; i++) {
      printf("%f\n", R[i]);
   }
}
```

https://fr.wikipedia.org/wiki/Compute_Unified_Device_Architecture

Example of GPU programming (working)

```
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void mykernel(float *A1, float *A2, float *R)
{
    int p = threadIdx.x;
    R[p] = A1[p] + A2[p];
}

int main()
{
    float A1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    float A2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
    float R[9];
    int taille_mem = sizeof(float) * 9;

    float *a1_device, *a2_device, *r_device;
    cudaMalloc((void**) &a1_device, taille_mem);
    cudaMalloc((void**) &a2_device, taille_mem);
    cudaMalloc((void**) & r_device, taille_mem);

    cudaMemcpy(a1_device, A1, taille_mem, cudaMemcpyHostToDevice);
    cudaMemcpy(a2_device, A2, taille_mem, cudaMemcpyHostToDevice);

    mykernel<<<1, 9>>>(a1_device, a2_device, r_device);

    cudaMemcpy(R, r_device, taille_mem, cudaMemcpyDeviceToHost);

    for(int i = 0; i < 9; i++) { printf("%f\n", R[i]); }
}
```

Example of GPU programming (2 dimensions)

```
//
// Produit de 2 matrices carrées (Width x Width)
//
__global__ void Kernel(float *Md, float *Nd, float *Pd, int Width) {

  // Calculate the column index of the Pd element, denote by x
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  // Calculate the row index of the Pd element, denote by y
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  float Pvalue = 0;
  // each thread computes one element of the output matrix Pd.
  for (int k = 0; k < Width; ++k) {
    Pvalue += Md[y*Width + k] * Nd[k*Width + x];
  }

  // write back to the global memory
  Pd[y*Width + x] = Pvalue;
}

//
// Host code
//

dim3 dimBlock(32, 32);
dim3 dimGrid(Width/32, Width/32);
Kernel<<<dimGrid, dimBlock>>>( Md, Nd, Pd, Width);
```

From CUDA to (Apple) Metal

```
#include <metal_stdlib>
using namespace metal;

kernel void add_arrays(device const float* inA,
                       device const float* inB,
                       device float* result,
                       uint index [[thread_position_in_grid]])
{
    result[index] = inA[index] + inB[index];
}


// The HOST source code

MTL::Device  *_mDevice          = MTL::CreateSystemDefaultDevice();
MTL::Library *lib               = _mDevice->newlib();
NS::String::string str          = NS::String::string("add_arrays", NS::ASCIIStringEncoding);
MTL::ComputePipelineState* mFX = _mDevice->newComputePipelineState(addFunction, &error);
MTL::CommandQueue* cmdQueue     = _mDevice->newCommandQueue();

MTL::Buffer* buffA = _mDevice->newBuffer(nElements * sizeof(float), MTL::ResourceStorageModeShared);
MTL::Buffer* buffB = _mDevice->newBuffer(nElements * sizeof(float), MTL::ResourceStorageModeShared);
MTL::Buffer* buffC = _mDevice->newBuffer(nElements * sizeof(float), MTL::ResourceStorageModeShared);

MTL::cmdBuffer *cmdBuffer          = cmdQueue->cmdBuffer();
MTL::ComputeCommandEncoder *cmds = cmdBuffer->computeCommandEncoder();

cmds->setComputePipelineState(mFX);
cmds->setBuffer(buffA, 0, 0);
cmds->setBuffer(buffB, 0, 1);
cmds->setBuffer(buffC, 0, 2);

MTL::Size gridSize            = MTL::Size::Make(nElements, 1, 1);
NS::UInteger threadGroupSize = mFX->maxTotalThreadsPerThreadgroup();
MTL::Size threadgroupSize     = MTL::Size::Make(threadGroupSize, 1, 1);

cmds->dispatchThreads(gridSize, threadgroupSize);
cmds->endEncoding();
cmdBuffer->commit();

cmdBuffer->waitUntilCompleted();
```

From CUDA to OpenCL

Conclusion on GPU acceleration

- **GPU** devices are powerful but power hungry !
- **MIMT** programming model
  - Code or algorithm rewriting (kernels)
  - Dedicated memory structure
- Achievable performances depends on
  - Depends on kernel complexity
  - Data transfers & control penalties
  - Depends on GPU device
- Interesting ! Efficient ?