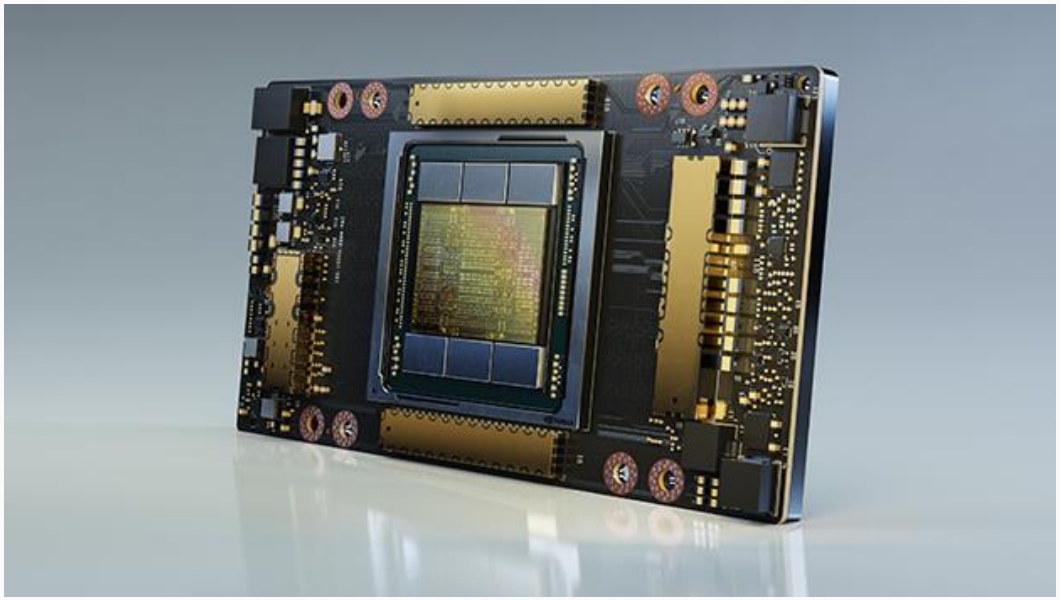
LESIA l'Observatoire de Paris | PSL

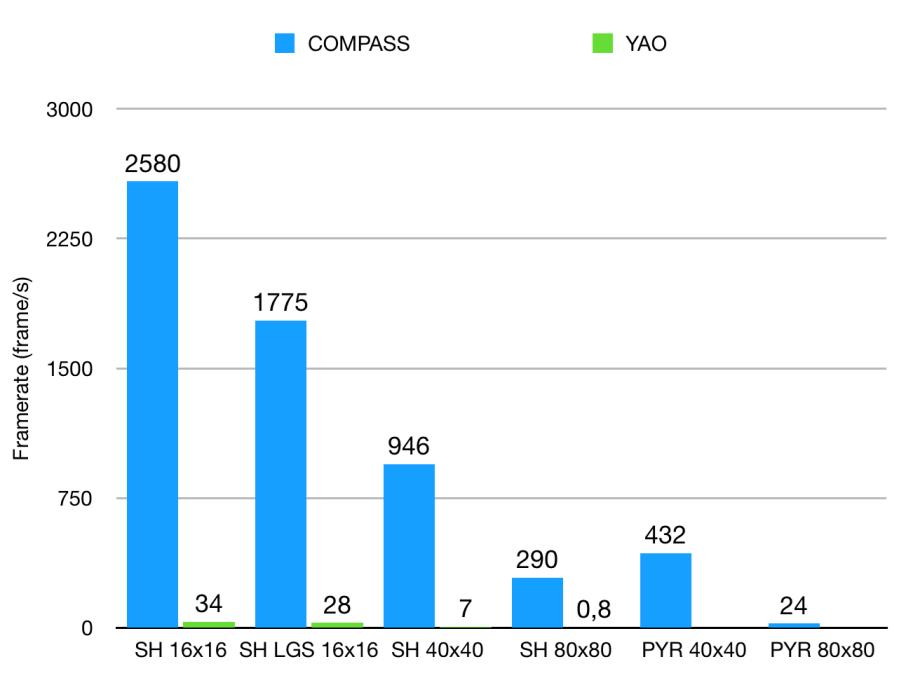


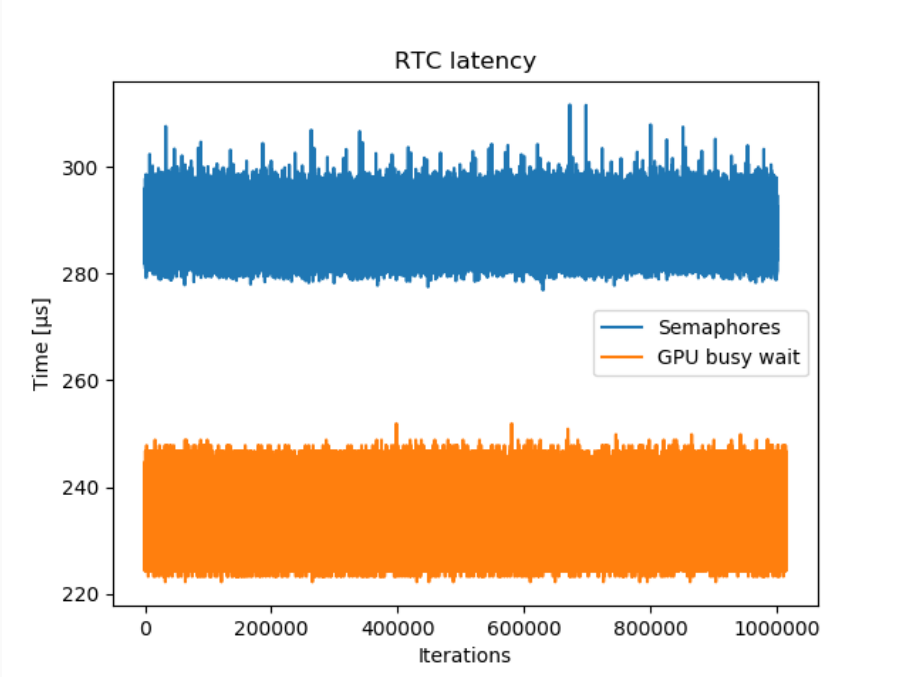
GPU Programming

Dr. Florian Ferreira
florian.ferreira@obspm.fr
ESIEA, January/March 2023

Who AM I ?

- PhD in astronomy & astrophysics, instrumentation speciality
- Research engineer at LESIA for 9 years
  - GPU based numerical simulations for adaptive optics
  - Development of GPU based hard real-time controller for adaptive optics





Module Objectives

- Introduction to GPU programmation
  - GPU hardware architecture
  - General Purpose GPU
  - Basics of NVIDIA CUDA API
  - Best practices and optimisations

Module Organization

- Lectures : 6h (4 x 1.5h)
  - Lecture 1 : Why do we use GPU ?
  - Lecture 2 : CUDA basics
  - Lecture 3 : Optimisations of CUDA code
  - Lecture 4 : GPU, Python and applications
- Directed studies / Practical tutorials : 9h (6 x 1.5h)
- Module evaluation
  - Multiple-choice test
  - Mini-project

PROJECT

MINI-PROJECT, DON'T WORRY

Free project

- Project to be done by group of 2-3 students
- Find (or create) a sequential code of your choice, and try to accelerate it using CUDA
- Your work should contain at least one custom kernel
- Short oral presentation (~5min) of each group during the last directed studies slots
- Presentation guidelines:
  - Short presentation of the base sequential code
  - Main technical points of your CUDA implementation
  - Performance comparison
  - Analyze for further optimisations

References

- *“CUDA by Example”*, Jason Sanders, Edward Kandrot
- “*Professional CUDA C Programming*”, John Cheng, Max Grossman, Ty McKercher
- Cours de Stéphane Vialle “Calcul sur GPU” : http://www.metz.supelec.fr/metz/personnel/vialle/course/PPS-5A-GPGPU/index.htm
- CUDA Toolkit documentation : *https://docs.nvidia.com/cuda/index.html*
- *https://developer.nvidia.com/blog/even-easier-introduction-cuda/*
- https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

BUT FIRST, POLL TIME:

Do you have a laptop equipped with NVIDIA GPU ?

DIRECTED STUDIES

- Exercise 0.1:
  - Make groups of 2-3 students
- Exercise 0.2 (for groups with Nvidia laptop):
  - If your laptop is running on Windows, install WSL
  - Install the CUDA toolkit on your laptop and makes it work !
  - Follows instructions on the CUDA toolkit documentation page
  - Will be great if it could be done before the first slot

WHY DO WE USE GPU ?

(I MEAN, EXCEPT FOR GAMING...)

Overview of the TOP500 (November 2022)

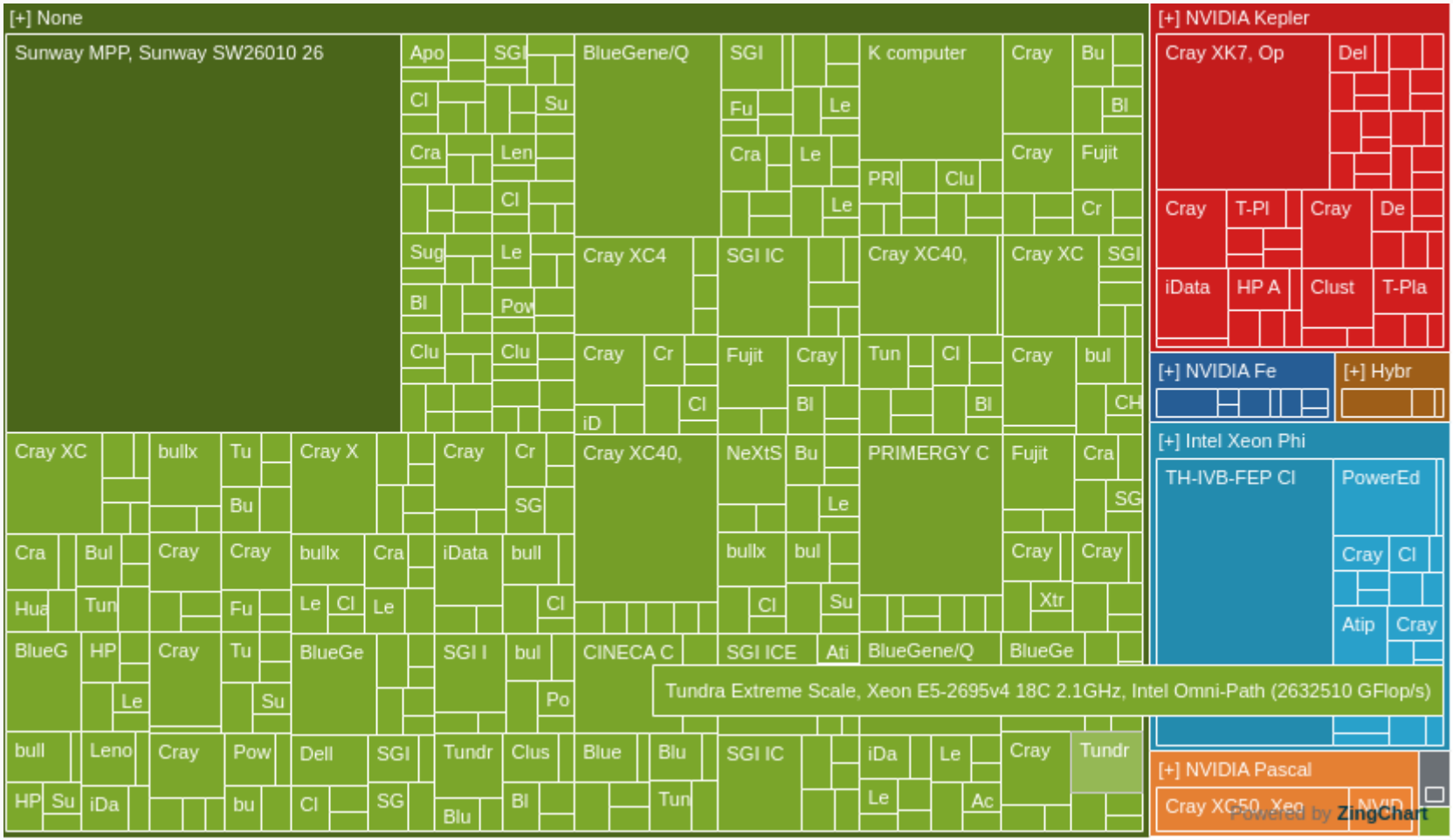
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Source: top500.org

- Top 500 : ranking of the most powerful supercomputers in the world
- In November 2016, a single system equipped with GPU in the top 5
- 6 years later, 4 GPU based systems in the top 5

Overview of the TOP500

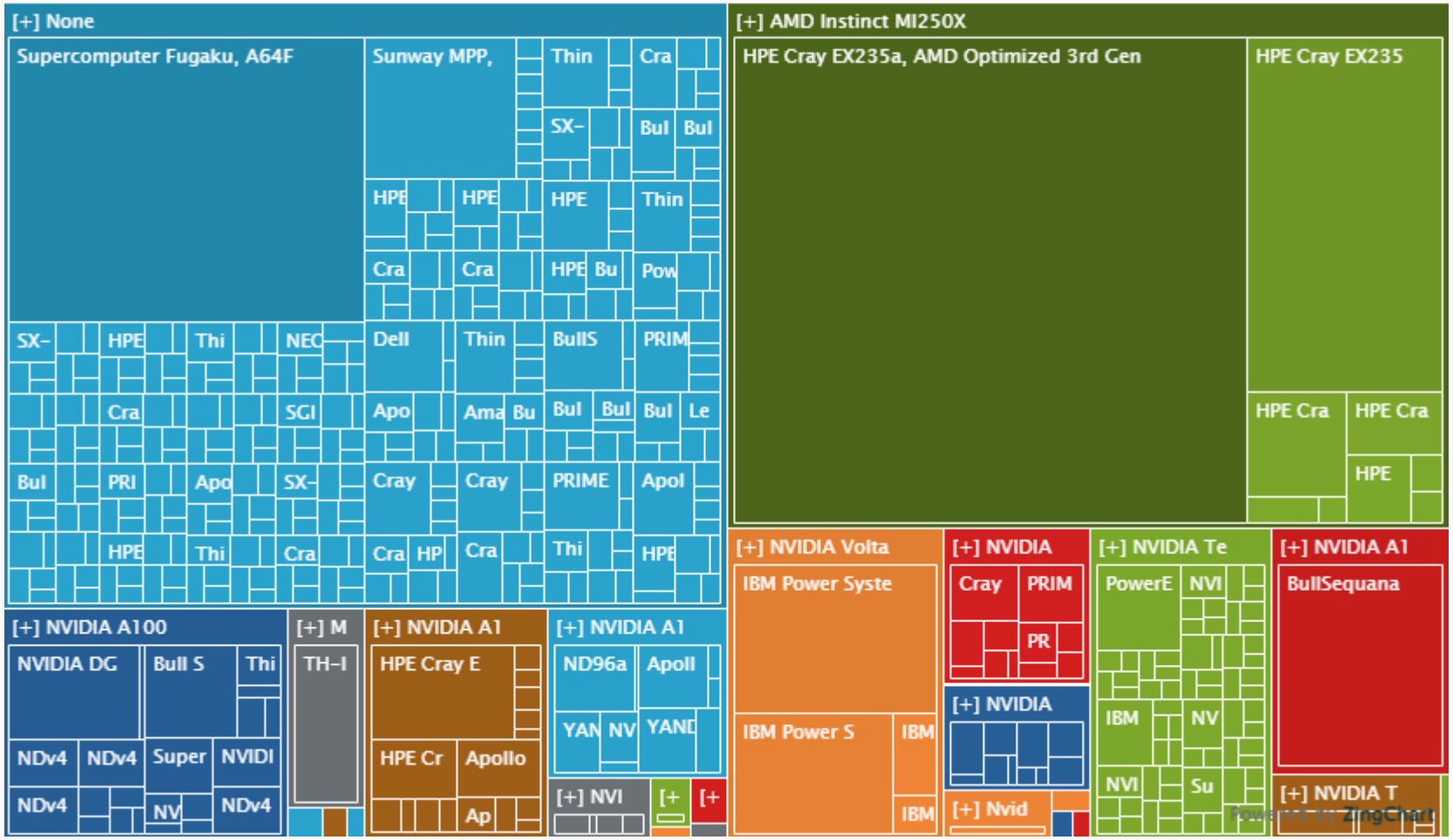
➢ Distribution of accelerator family over the TOP500 (November 2016) :



Source: top500.org

Overview of the TOP500

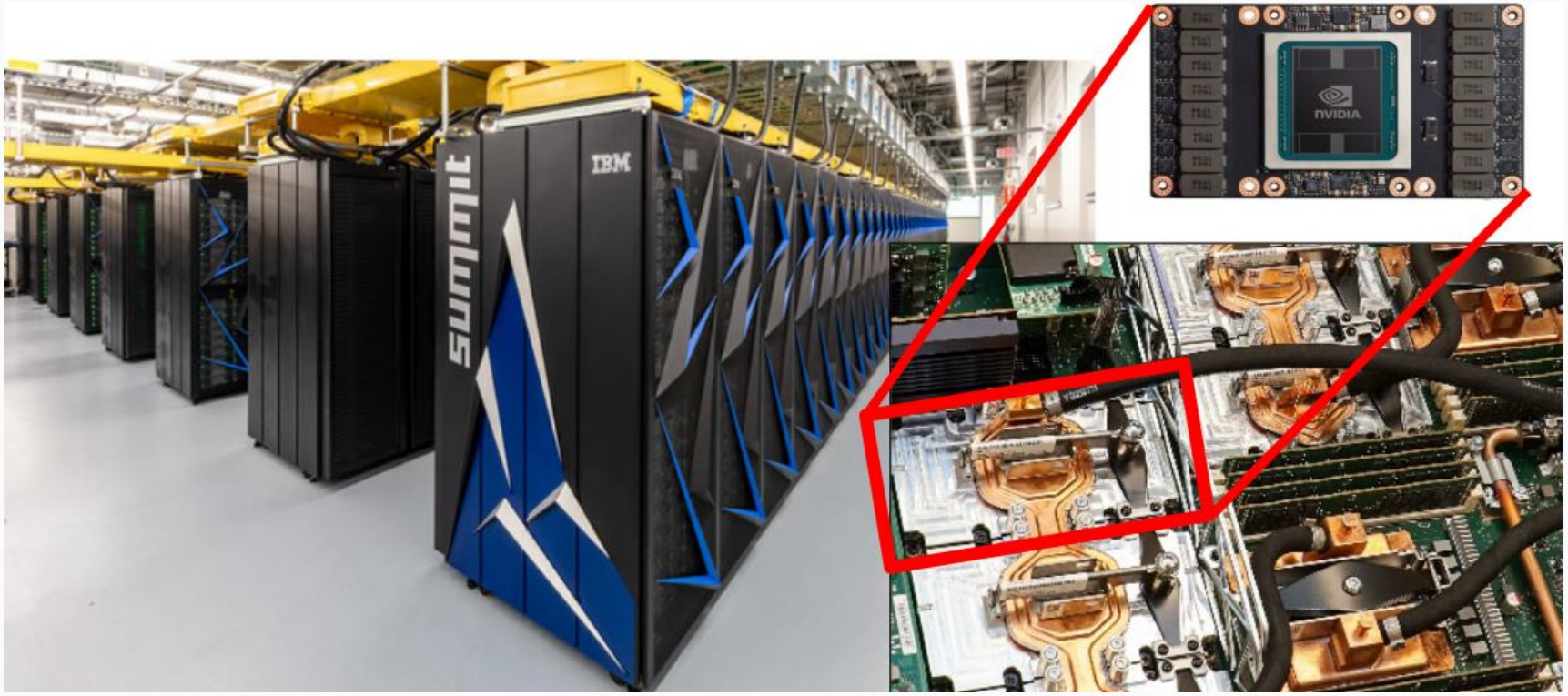
➢ Distribution of accelerator family over the TOP500 (November 2022) :

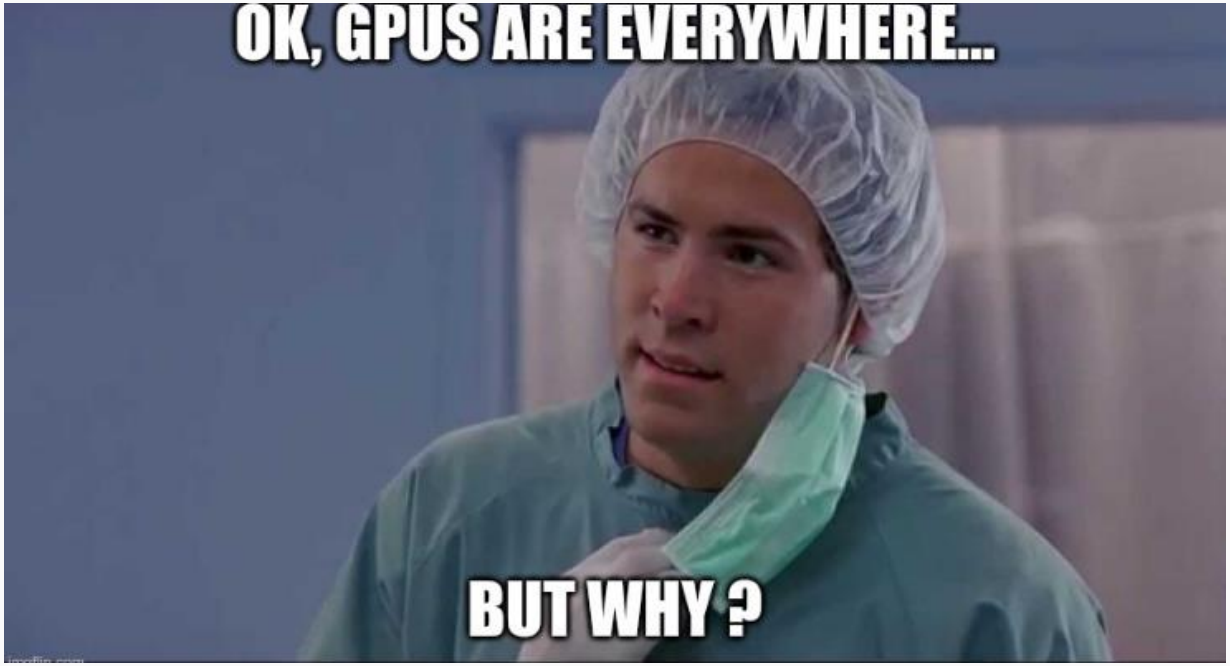


Source: top500.org

SUMMIT (#5 NOVEMBER 2021)

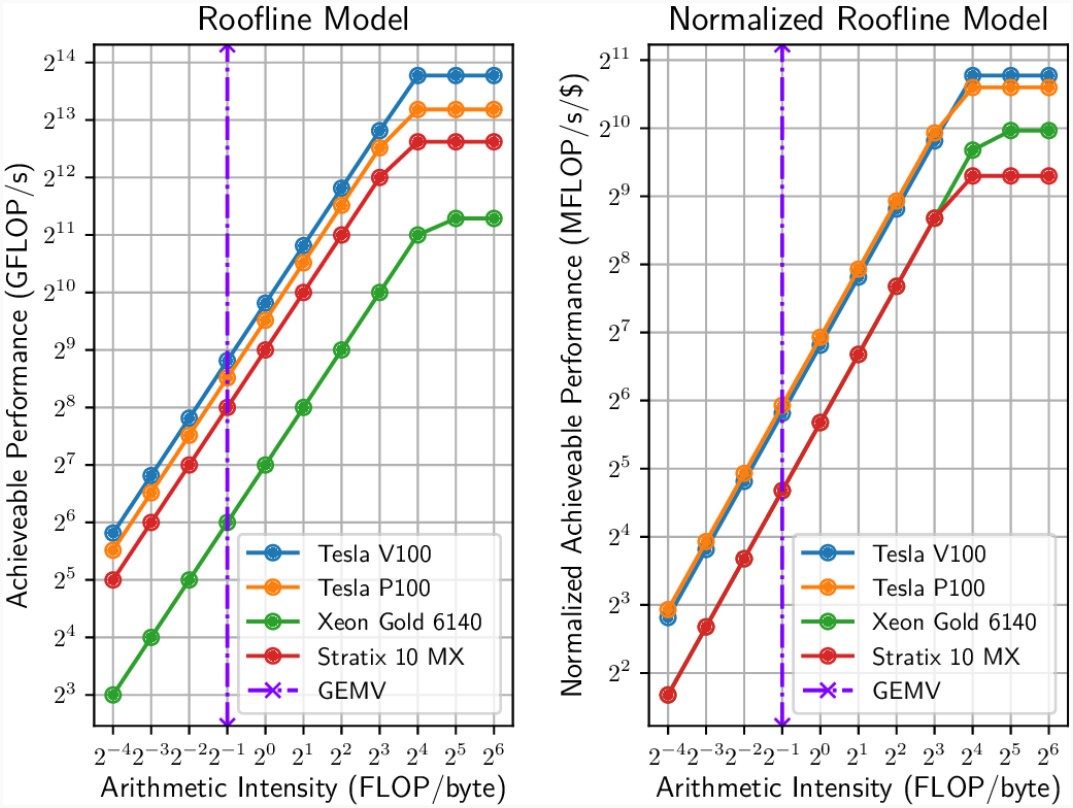
- 143.5 PFlop/s
  - 9,216 IBM Power9 processors
  - 27,684 GPU NVIDIA Volta GV100
  - Mellanox EDR 100G Infiniband Interconnect
  - Peak Power Consumption: 13MW




OK, GPUS ARE EVERYWHERE...
BUT WHY ?

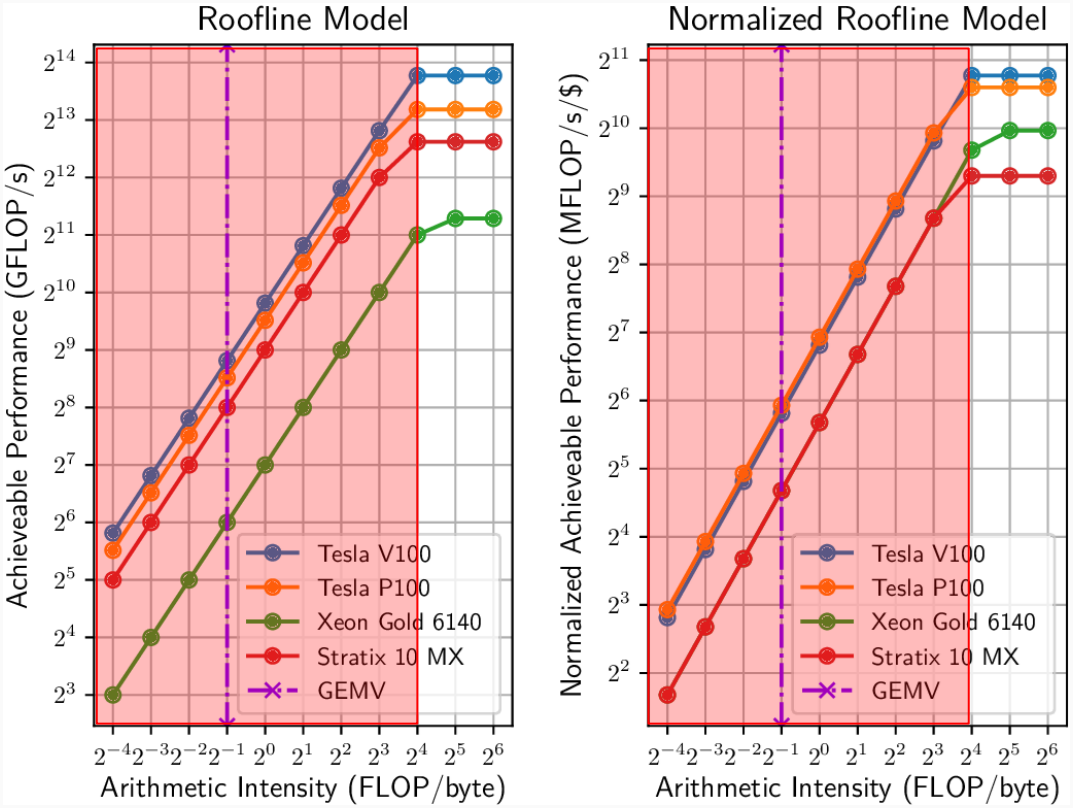
Performance comparison

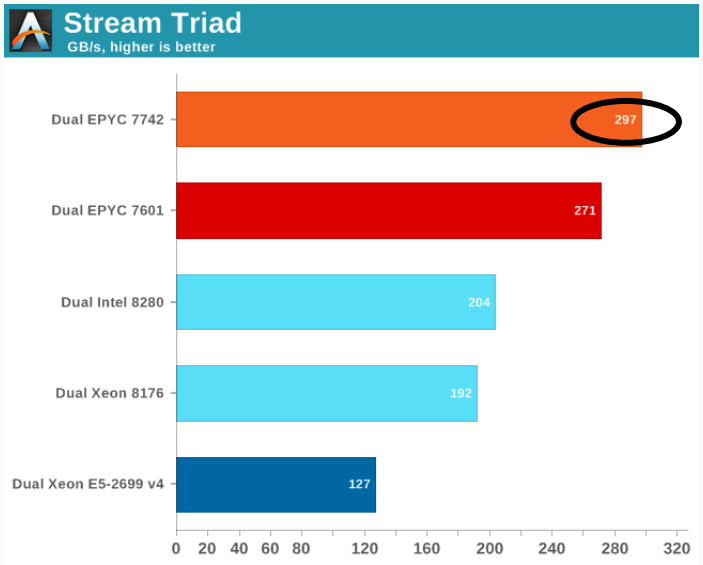
- Roofline model gives an estimate of computing performance of a given complexity
- Normalized roofline model includes the hardware cost



Performance comparison

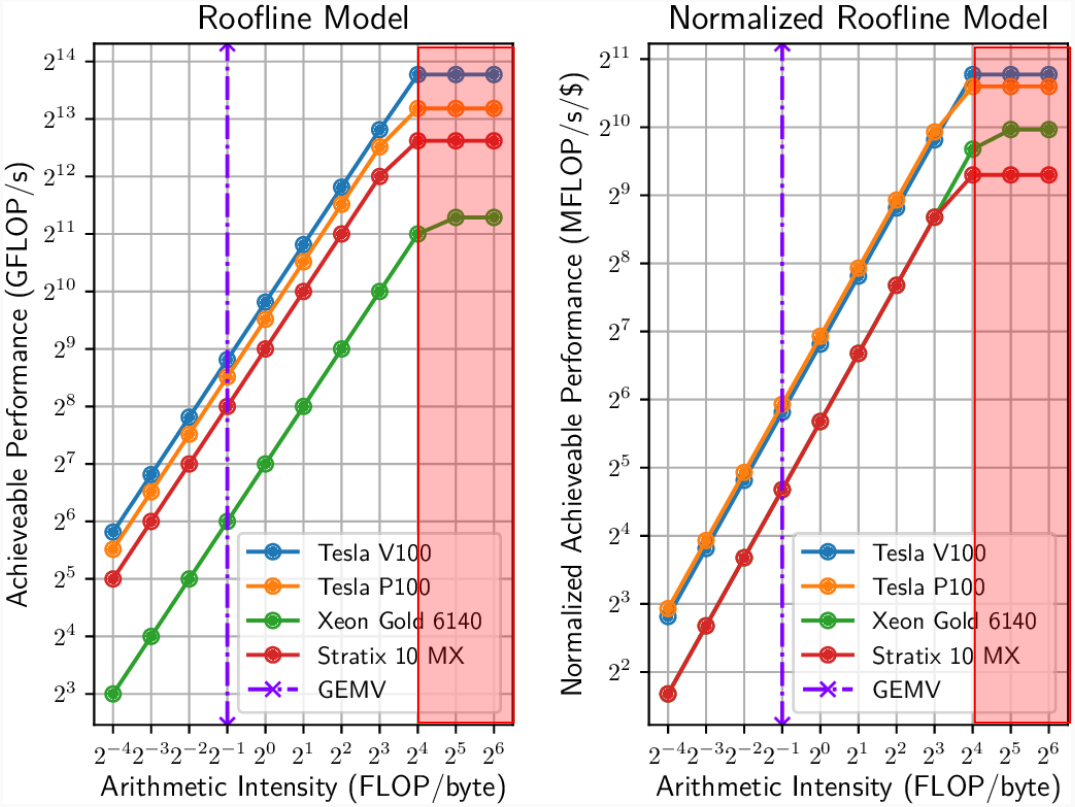
- Performance given by memory bandwidth : “memory bound”
  - NVIDIA A100 : up to 2 TB/s
  - AMD EPYC CPU : up to 300 GB/s





Performance comparison

- Performance given by peak compute performance : “compute bound”
  - NVIDIA A100 FP32 performance : 19.5 TFLOPS, up to 312 TFLOPS with Tensor Cores
  - AMD EPYC CPU FP32 performance : up to 2 TFLOPS

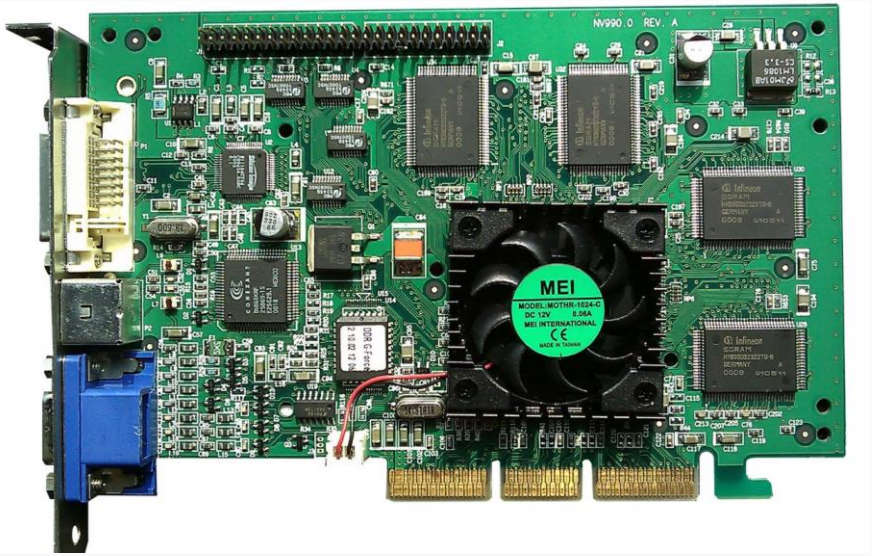


What is a GPU ?

- Wikipedia definition:

“A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device”

- Basically, hardware chip specialized for video rendering (lighting, texturing)
- First “modern GPU” appears in 1999: NVIDIA GeForce 256
- Gaming is the main market for GPU vendors



Birth of General Purpose GPU

- ~2001: new GPU card supports floating point and programmable shaders
- Scientific computing community started to experiment GPU for linear algebra
- Need to reformulate problems as graphics primitives
- 2007: first release of CUDA by NVIDIA
  - Dedicated API for GPGPU on NVIDIA GPU
- GPU vendors then started to propose dedicated architectures

GPU Vendors

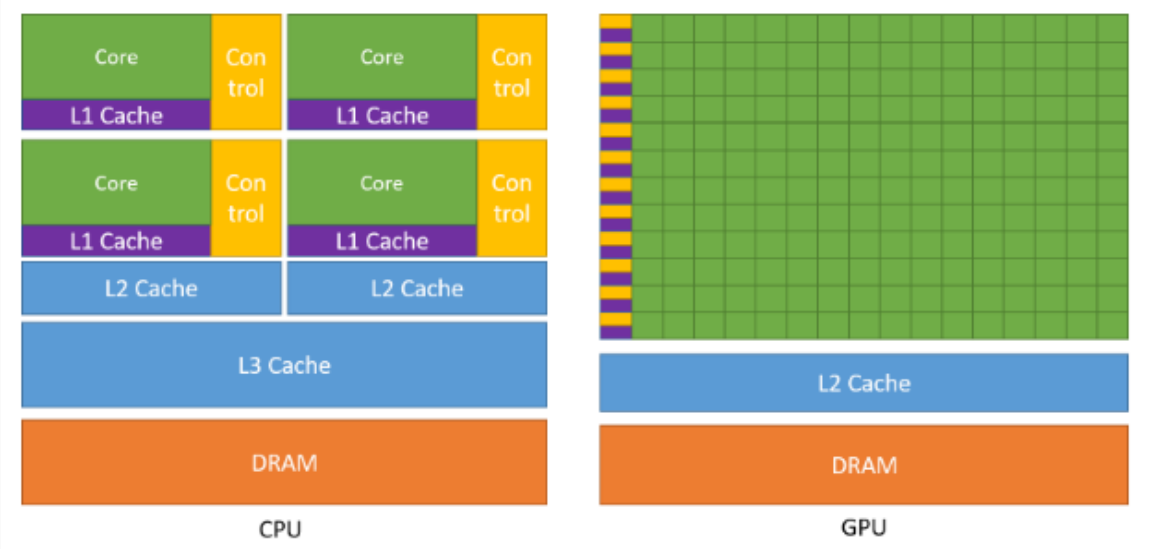
- NVIDIA
  - GeForce Series
  - Tesla Products
  - GPGPU API : CUDA

- AMD
  - Radeon Series
  - Radeon Instinct
  - GPGPU API : HIP

- Intel
  - ARC
  - GPGPU API : oneAPI

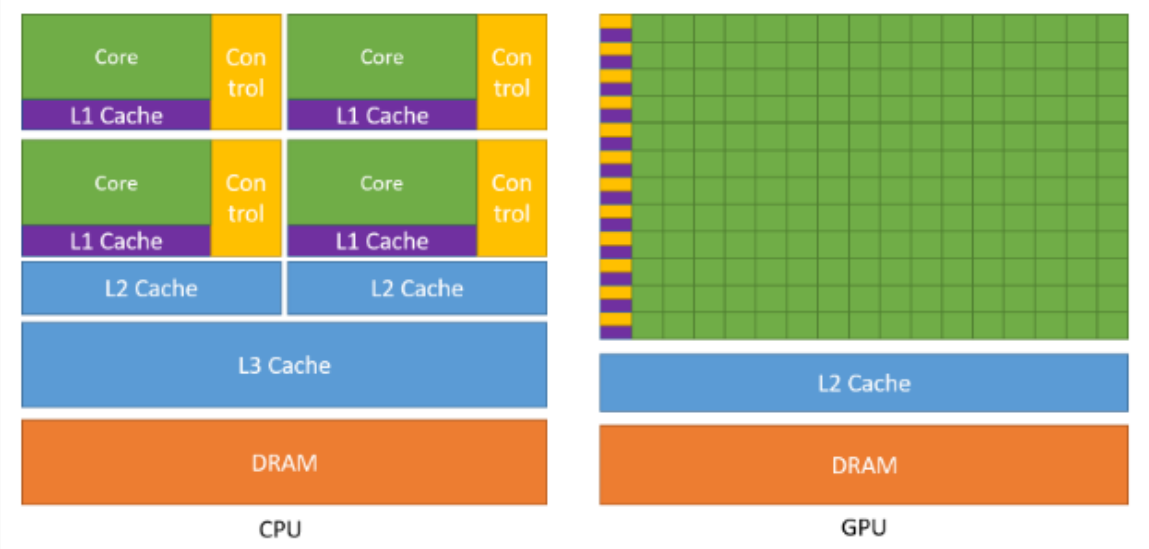
- NVIDIA is leading the professional market for now, notably thanks to the maturity of CUDA
  - Both HiP and oneAPI proposes translation tool from CUDA code

CPU VS GPU ARCHITECTURE



- CPU is designed to handle a wide-range of tasks quickly
  - Low compute density, complex control logic
  - Large caches
  - Optimised for serial operations (few ALU, higher clock speed)
  - Newer CPUs have more parallelism

CPU VS GPU ARCHITECTURE



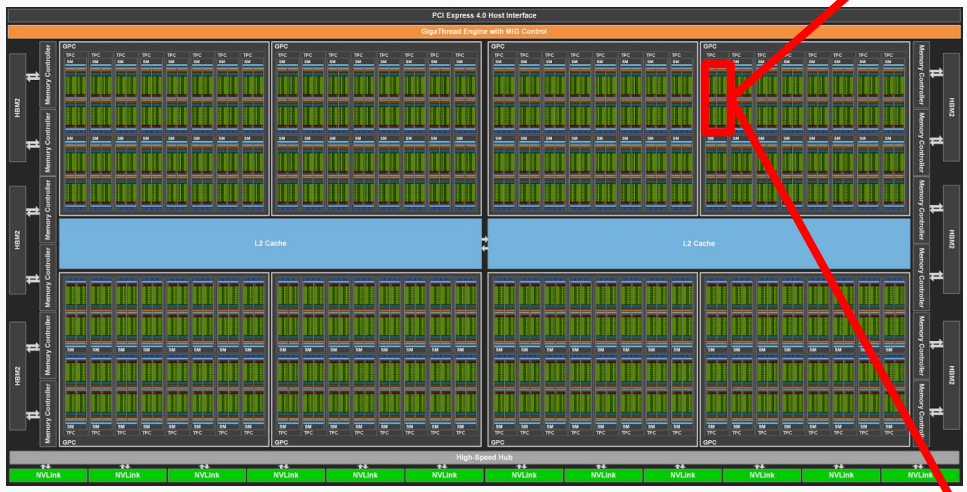
- GPU is designed to quickly render high resolution images and video concurrently:
  - High compute density
  - High computations per memory access
  - Built for parallel operations (many ALUs)
  - High throughput
  - Newer GPUs comes with better control logic (more CPU-like)

CPU VS GPU ARCHITECTURE

AMD EPYC Milan

NVIDIA A100

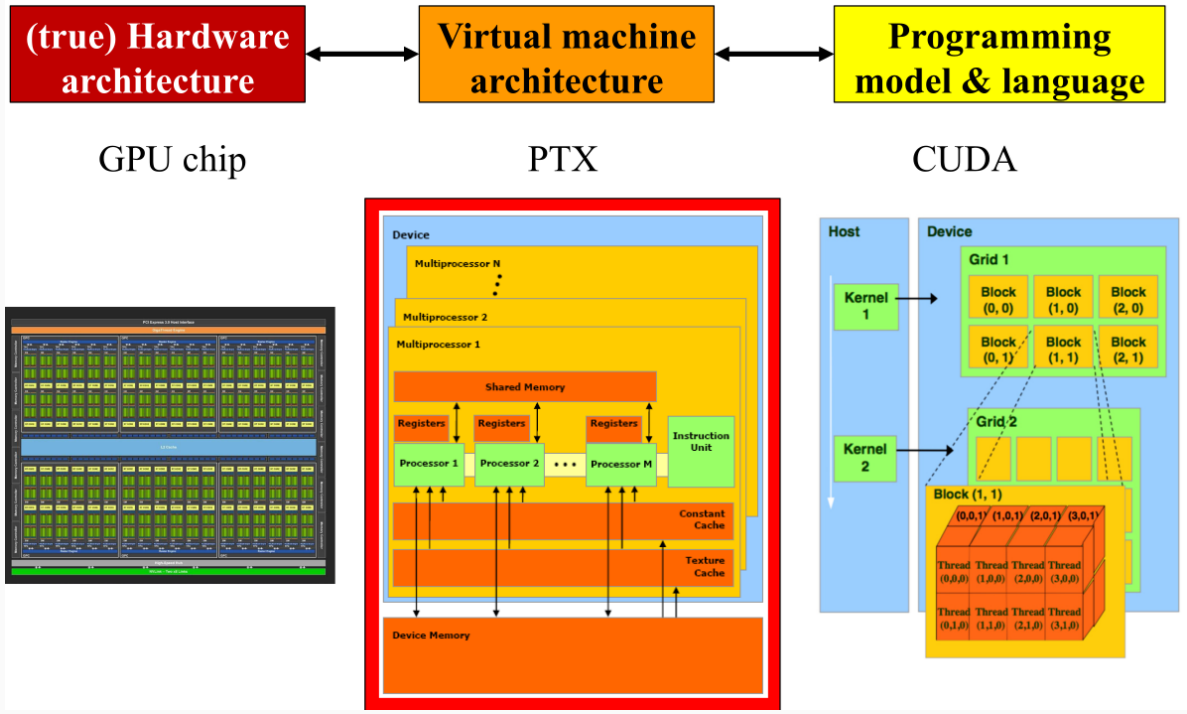
GPU ARCHITECTURE



- GPU is a set of N (128 for A100) Stream Multiprocessors (SM)
- Each SM includes:
  - Hardware threads running warps of 32 threads
  - Fast memory shared between running threads (of the SM)
  - Warp schedulers
  - Many CUDA cores including Tensor Cores

SIMT ARCHITECTURE

- SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps
- Threads in a warp start at the same address, but have their own instructions
  - Full efficiency relies on correct workload distribution and memory accesses across the warps
- SIMT = Single Instruction, Multiple-Thread
  - Thread-level parallel code



SM ARCHITECTURE

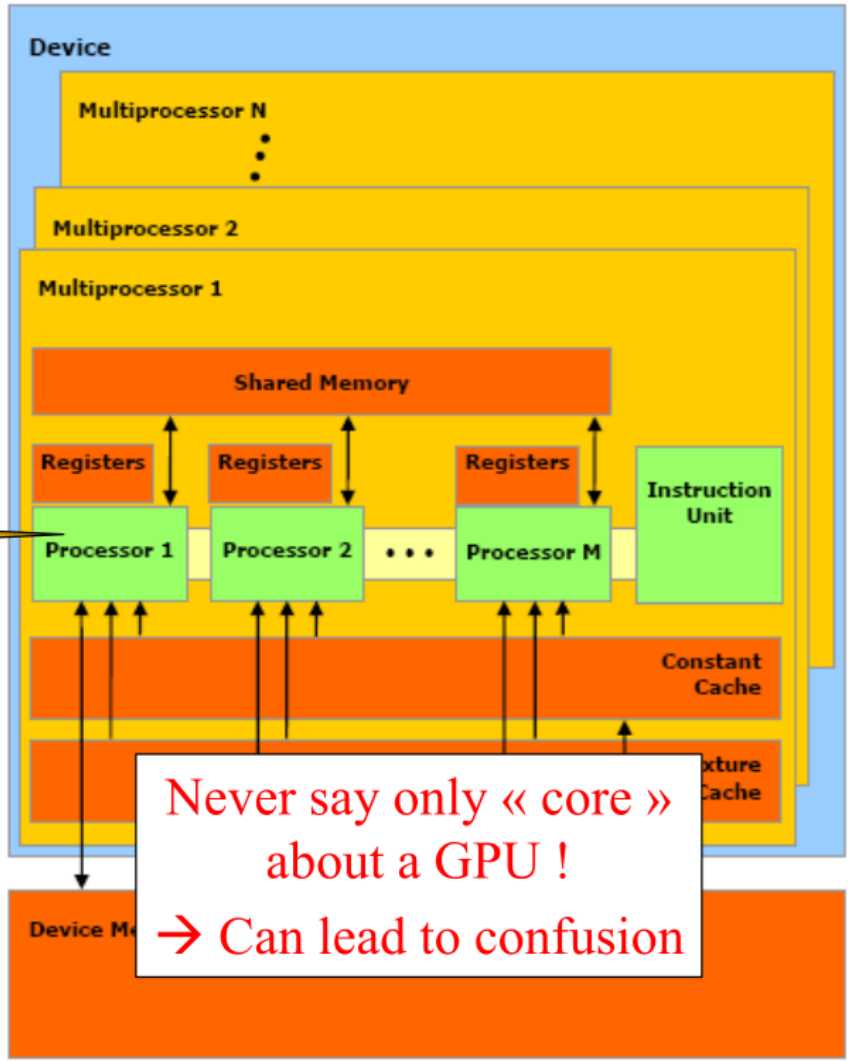
What is a « CUDA core » ?

→ It is a *hardware thread*:
1 ALU + access to SP and DP floating point units + some registers of the SM + access to the shared memory + access to the global memory of the GPU

CUDA core

Comparison to CPU:

- A CUDA core can be compared to one ALU of a CPU SIMD unit (AVX units)
- A CPU core can be compared to one Stream Multiprocessor of a GPU

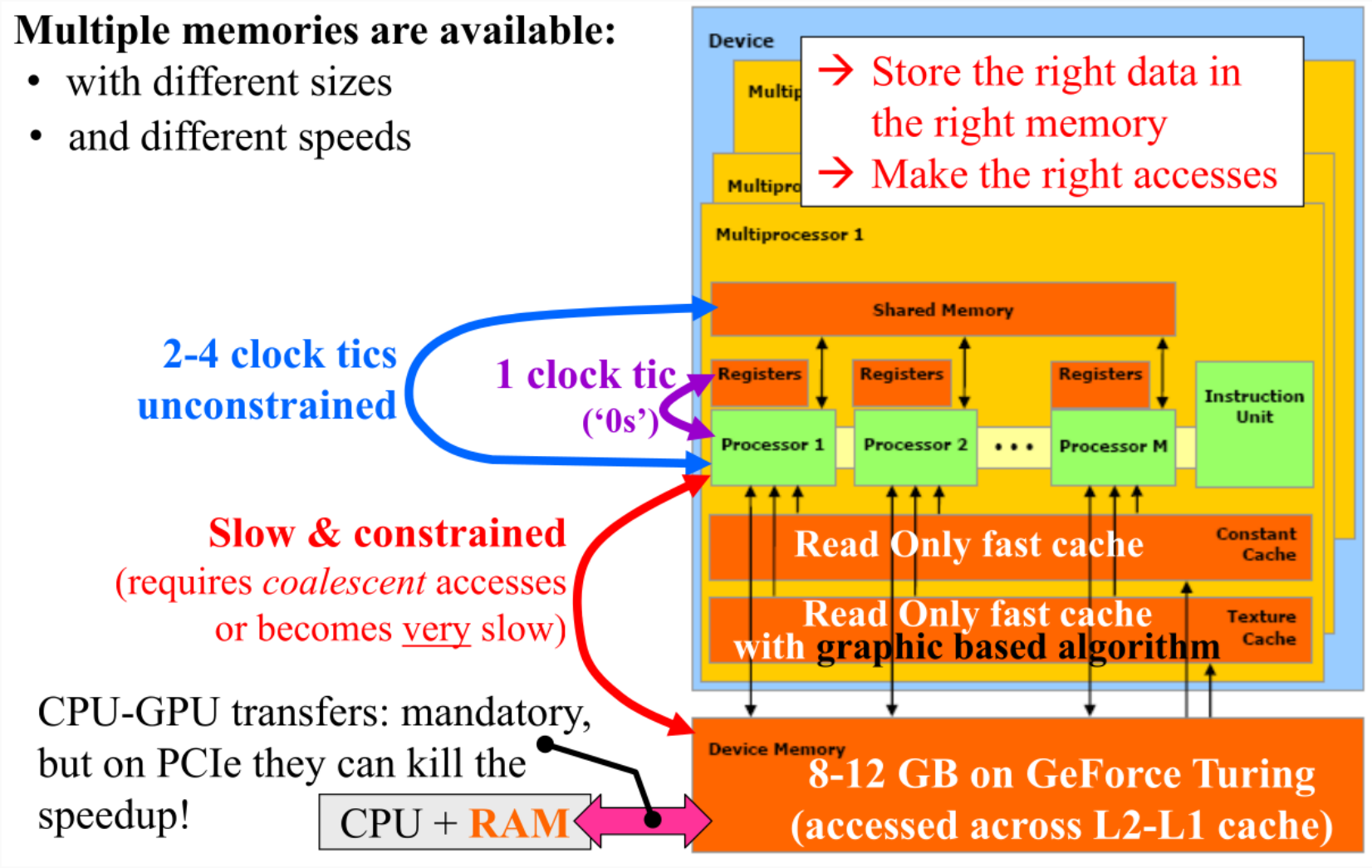


Credit: S. Vialle

GPU MEMORIES

Multiple memories are available:

- with different sizes
- and different speeds



Credit: S. Vialle

GEFORCE VS TESLA

GeForce Ampere RTX SM

Tesla A100 SM

Ray Tracing cores

- Final objective: « real time ray tracing for video »
- Currently: GPU not powerful enough
  → Real Time RT on a subset of rays
  + interpolation with Tensor Cores

SOL MAN from NVIDIA SOL ray tracing demo running on a Turing TU102 GPU with NVIDIA RTX technology in real-time

Video game remains the main market for NVIDIA

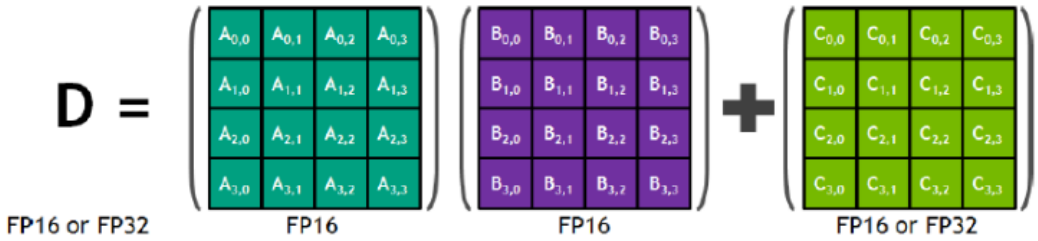
→ GPU architecture evolutions must be useful for the video game market

Credit: S. Vialle

TENSOR CORES

1 TC achieves a flow of product-add on a flow of 4x4 matrixes

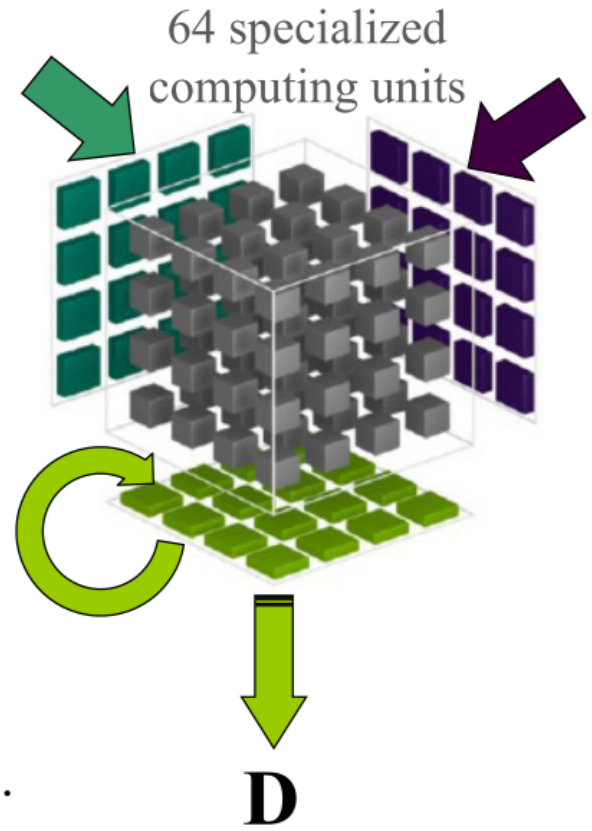
- D = A.B : produces a flow of D output matrixes
- D = A.B + C, with accumulation of A.B product flow into C matrix



Possible mixed-precision:

- input matrixes encoded on 16 bits
- internal computing on 32 bits
- output matrix flow on 16 or 32 bits

Useful for many kinds of applications: image processing, Machine Learning, Linear Algebra…

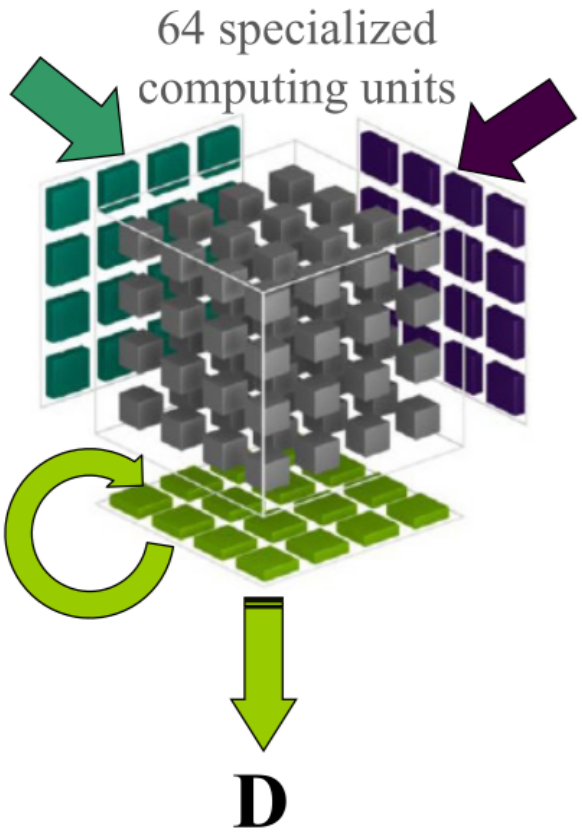


Credit: S. Vialle

TENSOR CORES

1 TC achieves a flow of product-add on a flow of 4x4 matrixes

- D = A.B : produces a flow of D output matrixes
- D = A.B + C, with accumulation of A.B product flow into C matrix



A Tensor core:

- is a hardware implementation of a matrix operator,
- is a very useful operator for modern applications,
- including graphic applications (main GPU market).

→ A math operator who deserves to occupy part of the chip!

Credit: S. Vialle

GPU PERFORMANCE IN A NUTSHELL

- Take care of workload distribution across warps
- Take care of memory accesses
  - Coalescent memory
  - Leverage all available memories (shared, texture, etc…)
- Leverage various types of CUDA cores (Tensor cores notably)
- **Important note: GPU performance is highly application dependent**
  - Relies on fitting between algorithms and hardware
  - CPU-GPU communications are expensive (PCIe bandwidth)

CUDA C BASICS

(ALMOST) EVERYTHING YOU NEED TO KNOW ABOUT CUDA

HELLO WORLD!

```
__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Classical C code with a few novelties:
  - ***__global__ qualifier***: tells to the compiler that the function is meant to run on a GPU device
  - ***Arguments pass to the function call with triple angle brackets <<< >>>:*** runtime parameters for function deployment on the device
- Other qualifiers:
  - __device__: device function to be called from another device function
  - __host__: host function (can be used together with __device__)
- A function which runs on the device is usually called ***a kernel***

A BIT FURTHER

```
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c,
                              dev_c,
                              sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );

    return 0;
```

- We need to allocate memory to do anything useful on a device, such as return values to the host:
  - ***cudaMalloc*** to allocate memory on the device
  - ***cudaFree*** to release it
- We can pass parameters to a kernel as we would with any C function
- We need to get data back from the GPU to the CPU using cudaMemcpy
- Error handling has to be done manually

CUDAMEMCPY

__host__ cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)

Copies data between host and device.

Parameters

`dst`
- Destination memory address

`src`
- Source memory address

`count`
- Size in bytes to copy

`kind`
- Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

Note:

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.
- This function exhibits synchronous behavior for most use cases.
- Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

DEVICE PROPERTIES

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 8 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-16GB"
  CUDA Driver Version / Runtime Version          11.5 / 11.5
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 16160 MBytes (16945512448 bytes)
  (080) Multiprocessors, (064) CUDA Cores/MP:    5120 CUDA Cores
  GPU Max Clock rate:                            1530 MHz (1.53 GHz)
  Memory Clock rate:                             877 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 6 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

- CUDA exposes structure and methods to query cuda capable device properties
- Notably includes:
  - # CUDA cores
  - # SM
  - Memory info
  - *Grid* size (we will see that soon…)

VECTOR ADD EXAMPLE

```
#include "../common/book.h"

#define N   10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
```

Allocate memory device

VECTOR ADD EXAMPLE

```
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
}
```

Copy a and b on the device

Launch kernel on N parallel blocks

Copy the result from the GPU to CPU memory

VECTOR ADD EXAMPLE

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

- Function to be run on each parallel block
- ***blockIdx*** is a built-in variable containing the block index

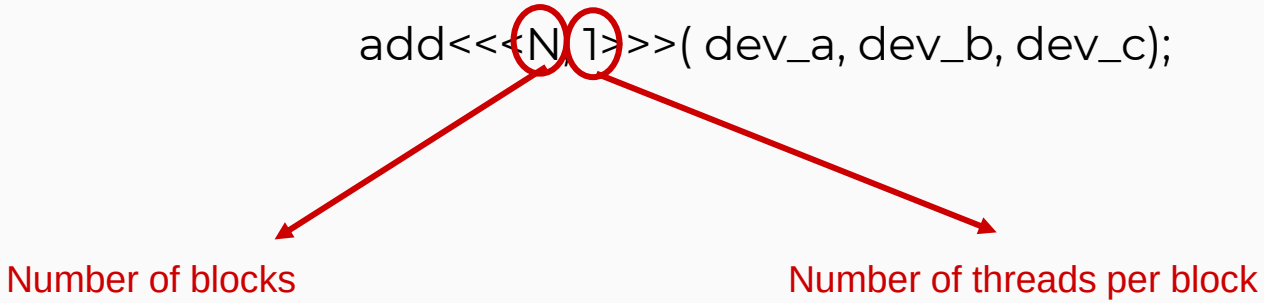
Ok, but…

- Device properties we saw earlier tells us you can launch up to 65,535 blocks

Can you see the issue ?

- What about summing vectors larger than that ?

BLOCKS AND THREADS



- CUDA runtime allows to split the blocks into threads
- Total number of parallel threads is given by N_{blocks} x $N_{threadsPerBlock}$
- As for blocks, thread index can be accessed through built-in variable ***threadIdx***
- Given the device properties, the maximum number of thread per block is 1,024
- Leads to a maximum number of parallel threads of 67,107,840

That's better, but still…
What if the problem to treat is larger than that ?

BLOCKS AND THREADS

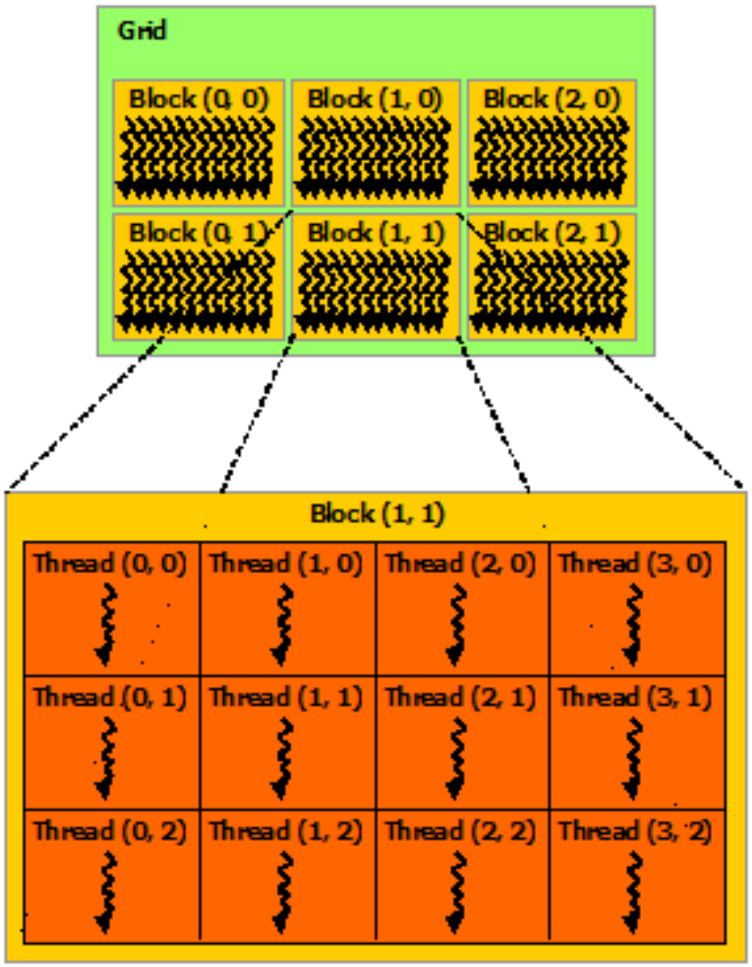
- Best distribution between number of blocks and threads is dependent on the kernel
- A good starting point (by experience):
  - Set the $N_{threads}$ to *maxThreadsPerBlock*
  - Then, calculate the number of blocks:

$$N_{blocks} = \min((N + N_{threads} - 1) / N_{threads}, \text{maxGridSize})$$

- maxGridSize gives the maximum number of blocks
- **Grid !?**

BLOCKS, THREADS AND GRID

- As threads being launched into blocks, the collection of blocks is called a ***grid***
- A grid is a two-dimensional “array” of blocks
- A block is a 3D “array” of threads (x,y,z) (z always equal to 1)
- Specifying N_{blocks} or $N_{threads}$ as scalar values is interpreted as using 1D grid or 1D block
- Using multi-dimensional blocks and grid is never mandatory, it could be just more convenient when treating matrices for example
- In any case,
  x * y * z < maxThreadsPerBlock



BACK TO SUMMING VECTORS…

- Generalisation of the code for vectors of arbitrary size N (N can be large)
- On the host code side, nothing changed except the kernel call :

$$add<<<N_{blocks}, N_{threads}>>>(dev\_a, dev\_b, dev\_c);$$

- Main differences are in the device function

BACK TO SUMMING VECTORS…

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

- First, *tid* refers now to the index of thread inside a block:
  - ***threadIdx.x*** gives the thread index along the X-axis of the block (1D here)
  - ***blockIdx.x*** gives the block index along the X-axis of the grid (1D too)
  - ***blockDim.x*** is the number of threads along the block X-axis

BACK TO SUMMING VECTORS...

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

- Then, we solve the issue of large N with a while loop:
  - As soon as the thread finishes its work, we increment its index by the total number of threads that has been launched ***blockDim.x * gridDim.x***
  - Each thread treats as many data as needed to cover all the vector
  - Threads can have unbalanced amount of work compare to others

IMPORTANT CUDA BASICS LEARNT SO FAR

- CUDA qualifiers: ***__global__, __device__, __host__***
- Notions of grid, blocks and threads
- How to allocate/free memory on the device: ***cudaMalloc / cudaFree***
- How to copy memory from/to the device: ***cudaMemcpy***
- How to call a kernel : ***kernel <<<N_{blocks}, $N_{threads}$>>>***
- Thread identification inside a device function:
  - ***threadIdx.x:*** index of the thread along the X-axis of its block
  - ***blockIdx.x:*** index of the block's thread along the X-axis of the grid
  - ***blockDim.x:*** number of threads along X axis of the block
  - ***gridDim.x:*** number of blocks along the X axis of the grid

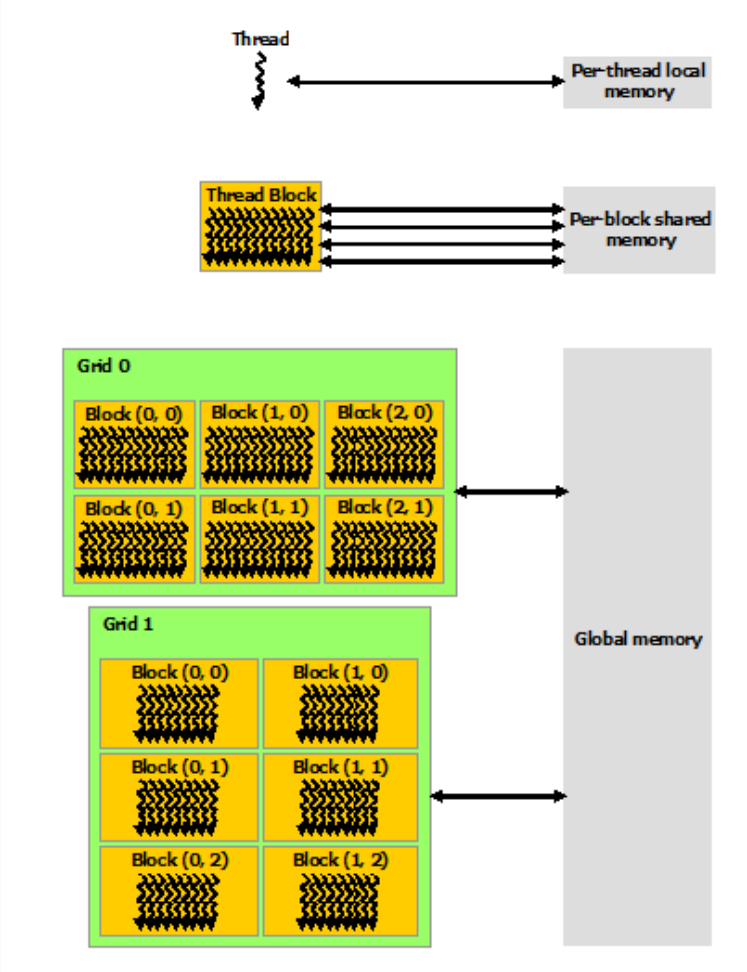
LET'S GO DEEPER...

- For now, all threads was totally independent

What about threads cooperation?

MEMORY HIERARCHY

- Variables declared in a device function are declared for each thread in a ***per-thread local memory***
- Threads of a block can work on a common memory space allocated in shared memory using the qualifier ***__shared__***
- Shared memory is shared only between the threads inside a same block ***(per-block shared memory)***
- All blocks can access the global memory
- Tips: shared memory access is way faster than global memory access



EXAMPLE: DOT PRODUCT

- Consider the dot product of two vectors x and y:

$$x.y = \sum_i x_i \times y_i$$

- **How would you do that using CUDA ?**

ANY GUESS ?

EXAMPLE: DOT PRODUCT

- The easy part: computing $x_i \times y_i$
- Each thread computes its own pair product and stores it in shared memory

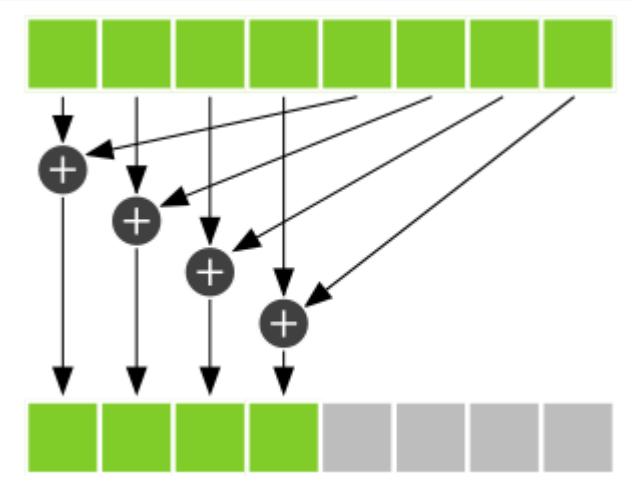
```
const int threadsPerBlock = 256;

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;
```

Static shared memory

Add “int N” here

EXAMPLE: DOT PRODUCT

- The harder part: performing the reduction
- Each thread has stored its xi.yi value in shared memory
- Reduction can then be performed per block only
- Once reductions are done on each block, the final reduction has to be performed by another kernel call, or directly on the CPU for example
- Block reduction can be done using half of the threads iteratively



EXAMPLE: DOT PRODUCT

- Before the reduction, we have to be sure that all threads of the block have finished to compute its products: ***__syncthreads()***
- We start with half of the threads and divide again by 2 at each step
- Finally, the thread 0 of each block write the block reduction result in global memory

```
__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;

    // synchronize threads in this block
    __syncthreads();

    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

EXAMPLE: DOT PRODUCT

- Finally, we finish the job on the host side

```
dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                        dev_partial_c );

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                          blocksPerGrid*sizeof(float),
                          cudaMemcpyDeviceToHost ) );

// finish up on the CPU side
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
```

- It is also possible to use shared memory dyamically
  - *dot<<<blocksPerGrid, threadsPerBlock, **sizeOfShared**>>>(...)*
  - *__shared__ float cache [];*

IMPORTANT CUDA BASICS

- CUDA qualifiers: ***__global__, __device__, __host__***
- Notions of grid, blocks and threads
- How to allocate/free memory on the device: ***cudaMalloc / cudaFree***
- How to copy memory from/to the device: ***cudaMemcpy***
- How to call a kernel : ***kernel <<<N_{blocks}, $N_{threads}$>>>***
- Thread identification inside a device function
- Thread synchronisation mechanism: ***__syncthreads()***
- Use of shared memory: ***__shared__***

IMPORTANT CUDA BASICS

- From this point, you know enough about CUDA to write efficient parallel code on GPU…

CONGRATS !

- Just let me tell you it exists 2 others kinds of device memory which can be useful to optimise your code:
  - __constant__ memory : fast & read-only
  - Texture memory: fast, useful to treat problems with neighbours
- Those memories will not be included in this course by lack of time
- If you are interested, check the CUDA toolkit documentation

CUDA OPTIMISATIONS

That's the tough one... stay strong !

THE REAL WORK

- As we saw, it's quite easy to write efficient CUDA code
- Typical development of CUDA code can be summarised as (from my experience):
  - **10 % of the work gives you 60 to 80 % of the GPU performance**
  - **90 % of the work consists in optimisations to get the full performance**
- Optimisations are application dependent
- CUDA provides tools to analyze the performance
- This lecture will give you a few general tips to optimise performance using some more advanced CUDA concepts

HOW TO OPTIMISE ?

- **Measure performance !**
  - Optimising a code to get the maximum theoretical performance out of the GPU is hard, or even impossible
  - Optimisation should always be done with respect to time requirements (note that this requirement could be “best time possible” in R&D for example)
- **Analyze the code**
  - Analyze the algorithms to detect possible bottlenecks
  - Analyze the GPU throughput (using dedicated tools)
- **From previous points informations, modify the code and iterates again**

FEW GENERAL TIPS FOR OPTIMISED CODE

- If possible, use NVIDIA standard libraries instead of custom kernels
  - An army of engineers have probably already done the job for you !
- Use (and abuse) CUDA tools to analyze your code
  - Gives you precious informations very easily
- If you have to write custom kernels:
  - Think your algorithm in terms of maximizing the parallelism
  - Take care of memory access: is it useful to use on-chip memory ? coalescent access to the global memory ?
  - Test it !
- Avoid as much as possible CPU-GPU memory transfer
  - PCIe bandwidth is a performance killer
- Think about concurrency (we will see soon what concurrency is)

CUDA LIBRARIES (NOT EXHAUSTIVE)

- ***cuBLAS:*** BLAS implementation in CUDA (linear algebra)
- ***cuFFT :*** Fast Fourier Transform
- ***CUB:*** collection of sorting and reduction methods
- ***cuRAND:*** RNG
- ***cuSPARSE:*** Sparse linear algebra
- ***cuSOLVER:*** decompositions and linear system solutions

- Official documentation : https://docs.nvidia.com/cuda/index.html

MEASURING PERFORMANCE

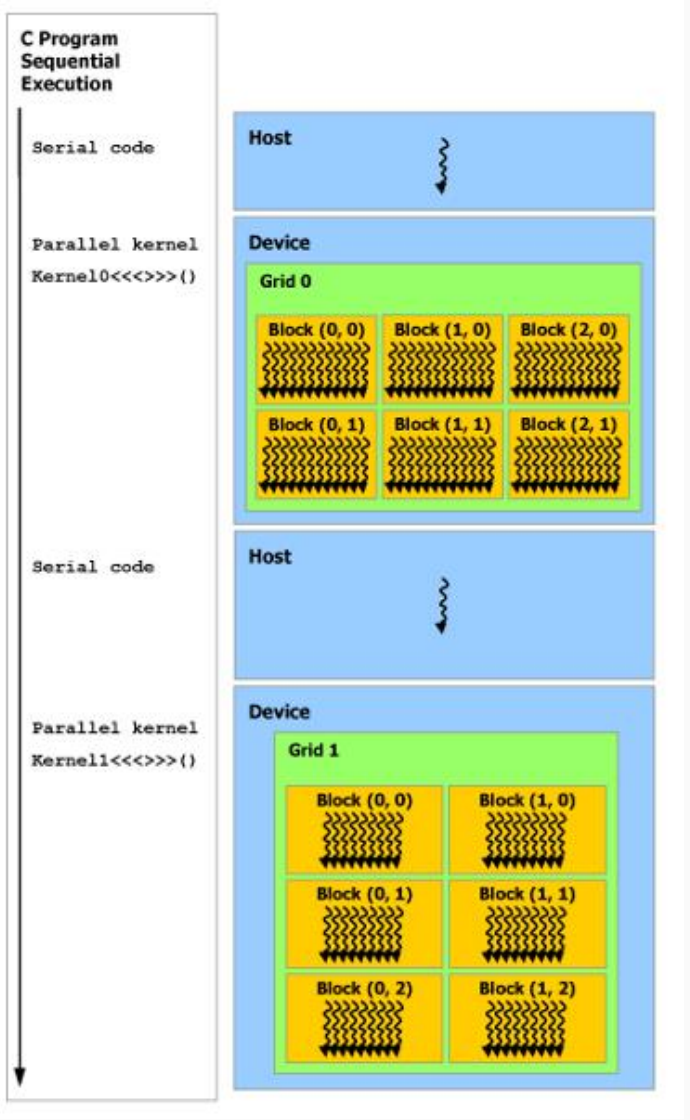
- How can we measure the execution time of a kernel ?

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

t1 = myCPUTimer();
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);

t2 = myCPUTimer();

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

- Problem:
  - This code will give you an execution time around 5 µs even for big workload !
  - Why ? → **Asynchronous computation !**

HETEROGENEOUS COMPUTATION

- CPU and GPU are two physical different and independent devices
  - They have their own compute capabilities and memory spaces
  - Kernel launches are asynchronous with respect to the host
- Allow simultaneous computations on the device and on the host
- Explicit CPU-GPU synchronisations exists:
  - ***cudaDeviceSynchronize()***: wait for all jobs on the GPU to be done



MEASURING PERFORMANCE

- Back to the question: how can we measure the execution time of a kernel ?

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

t1 = myCPUTimer();
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaDeviceSynchronize();
t2 = myCPUTimer();

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

- Problem (again):
  - cudaDeviceSynchronize() completely stall the GPU pipeline
- A more efficient approach is provided by the ***CUDA Event API***

MEASURING PERFORMANCE WITH CUDA EVENTS

- CUDA Events make use of the concept of ***CUDA streams***
- A CUDA stream is simply a sequence of operations that are performed in order on the device
- The default stream is the *NULL stream*, or *stream 0*, or *legacy stream*
- Other streams can be declared explicitly (cf. concurrency later)
- A CUDA event can be put on a stream to be recorded and trigger other operations
- CUDA event also embeds timestamping allowing timing measurement between 2 events

MEASURING PERFORMANCE WITH CUDA EVENTS

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Declare and create two events

Record an event before kernel launch

Record an event after kernel launch

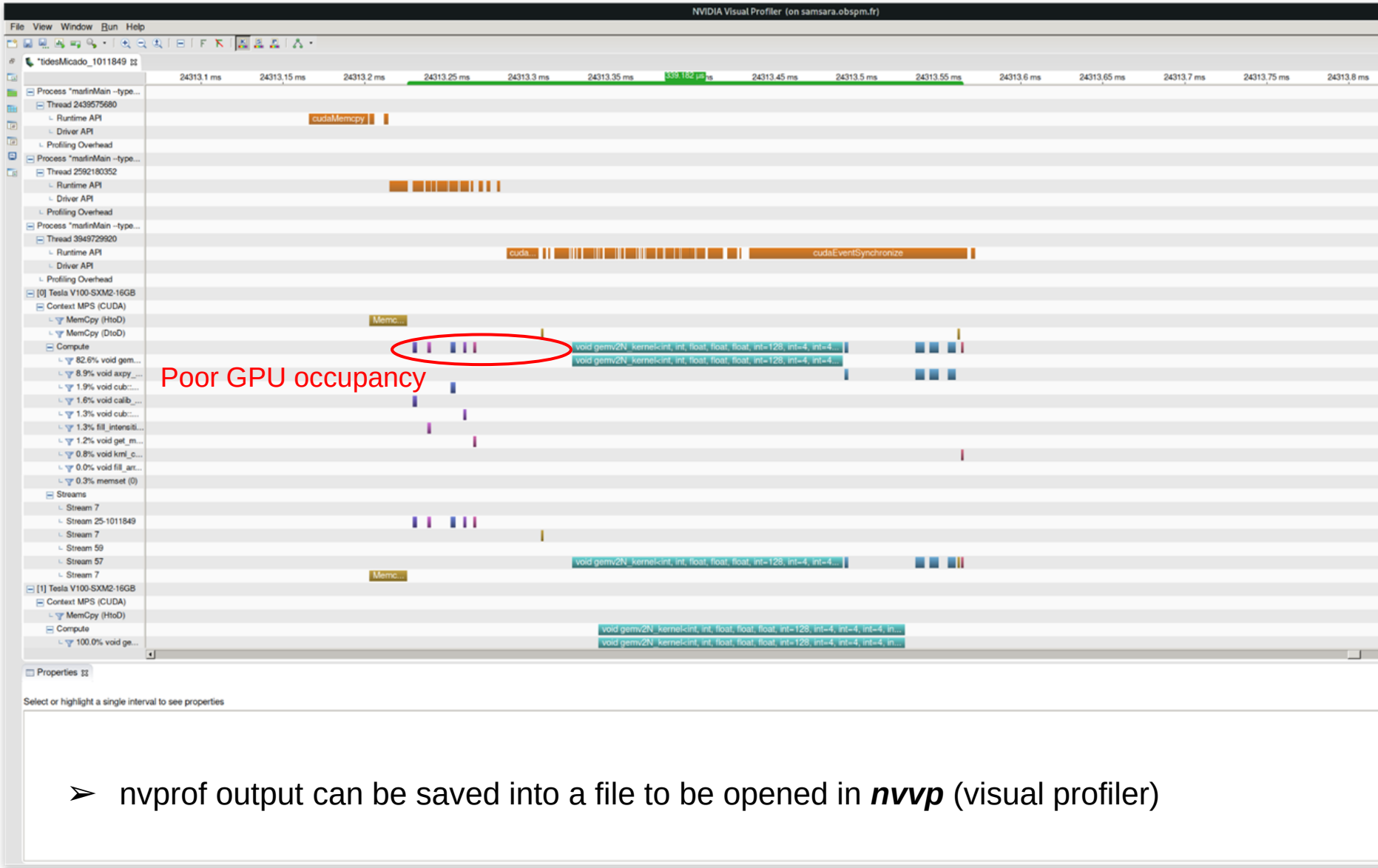
Wait for last event to be recorded (i.e. wait for kernel to finish)
Get time elapsed between the two events

MEASURING PERFORMANCE WITH PROFILER

- CUDA provides a profiling tool : ***nvprof***
- Provides infos on kernel execution, memory transfers, CUDA API calls, etc…
- At the price of some overheads

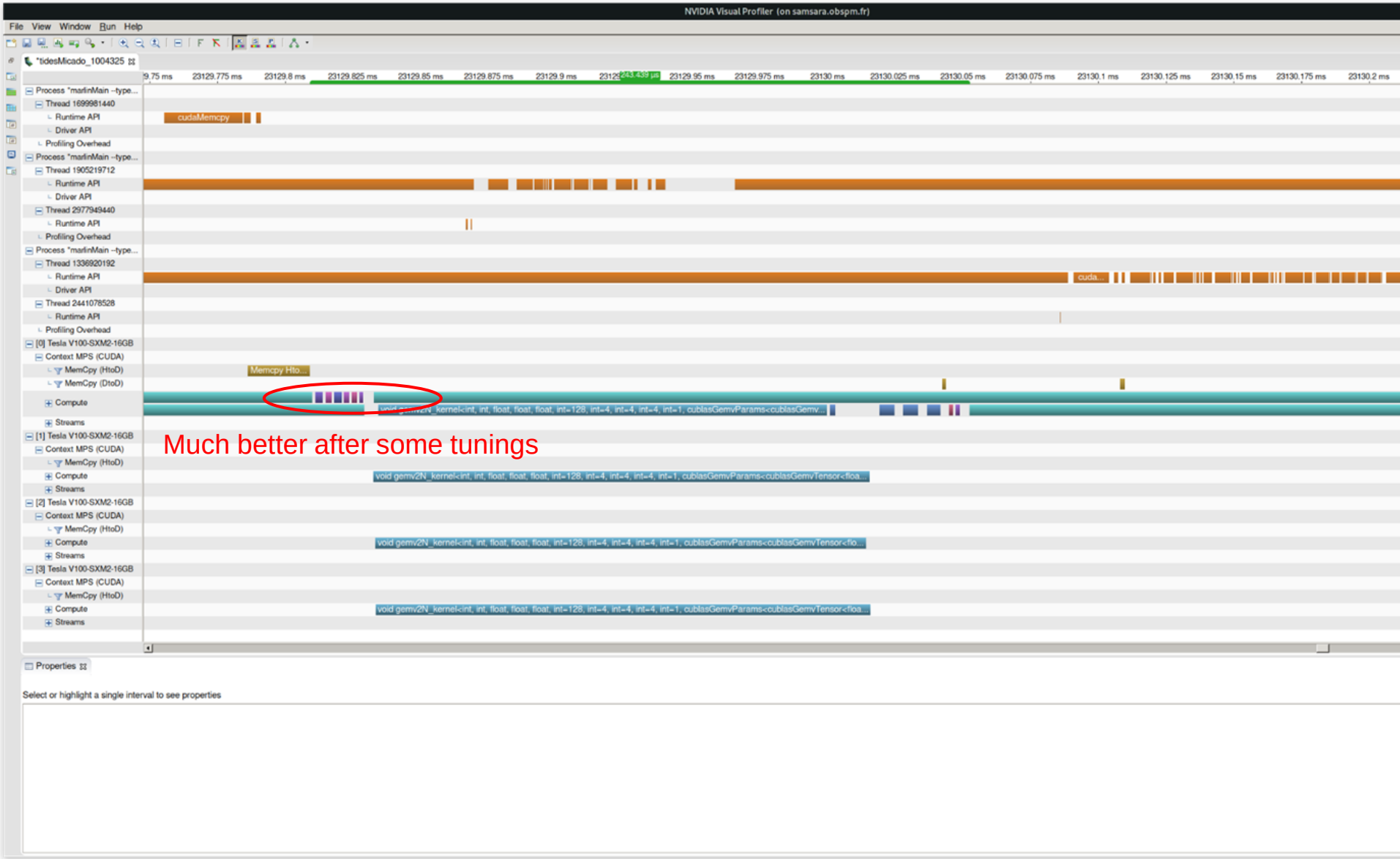
```
$ nvprof ./sumArraysOnGPU-timer
./sumArraysOnGPU-timer Starting...
Using Device 0: Tesla M2070
==17770== NVPROF is profiling process 17770, command: ./sumArraysOnGPU-timer
Vector size 16777216
sumArraysOnGPU <<<16384, 1024>>>  Time elapsed 0.003266 sec
Arrays match.
==17770== Profiling application: ./sumArraysOnGPU-timer
==17770== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 70.35%  52.667ms         3  17.556ms  17.415ms  17.800ms  [CUDA memcpy HtoD]
 25.77%  19.291ms         1  19.291ms  19.291ms  19.291ms  [CUDA memcpy DtoH]
  3.88%  2.9024ms         1  2.9024ms  2.9024ms  2.9024ms  sumArraysOnGPU
(float*, float*, int)
```

MEASURING PERFORMANCE WITH VISUAL PROFILER



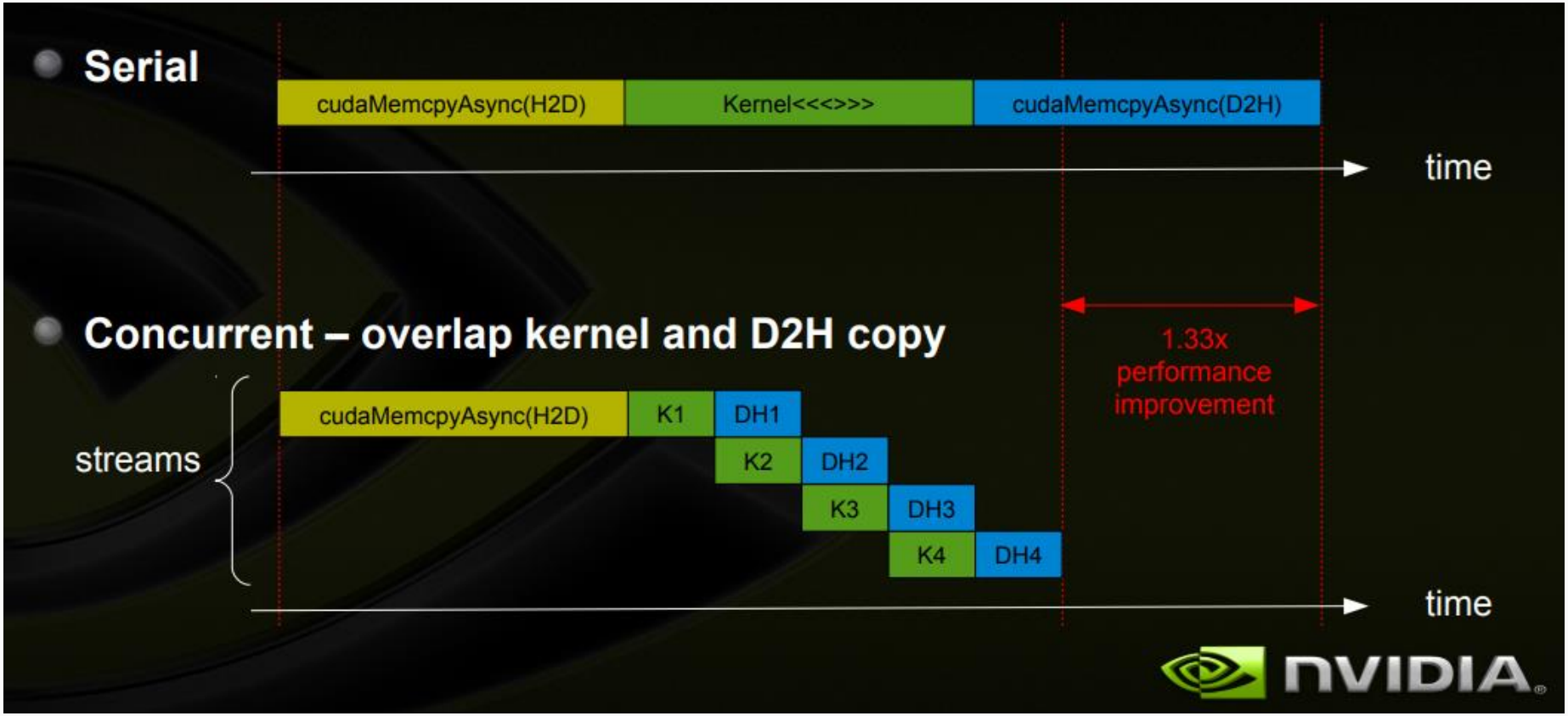
➢ nvprof output can be saved into a file to be opened in ***nvvp*** (visual profiler)

MEASURING PERFORMANCE WITH VISUAL PROFILER



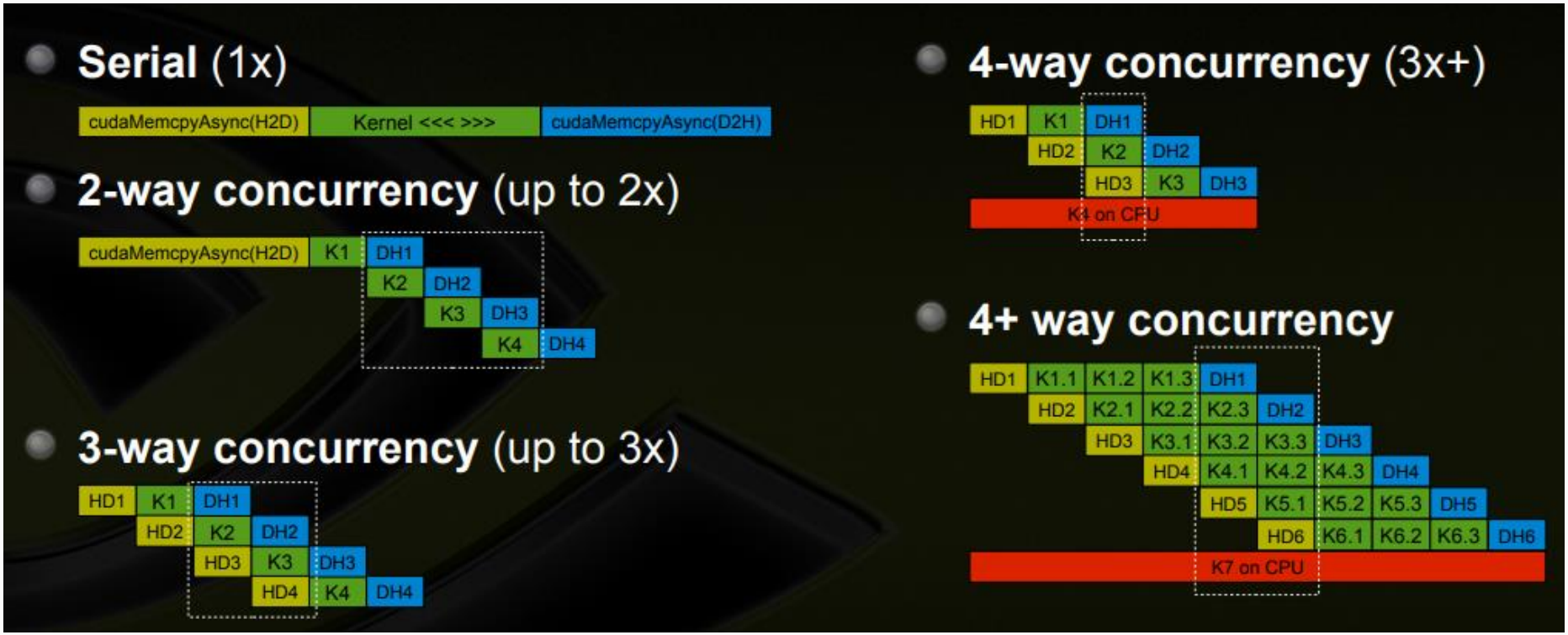
CONCURRENCY AND STREAMS

- Concurrency is the ability to perform multiple CUDA operations simultaneously
- This ability relies on the use of CUDA streams:
  - The default stream is synchronous w.r.t. host and device
  - Non-default stream are asynchronous

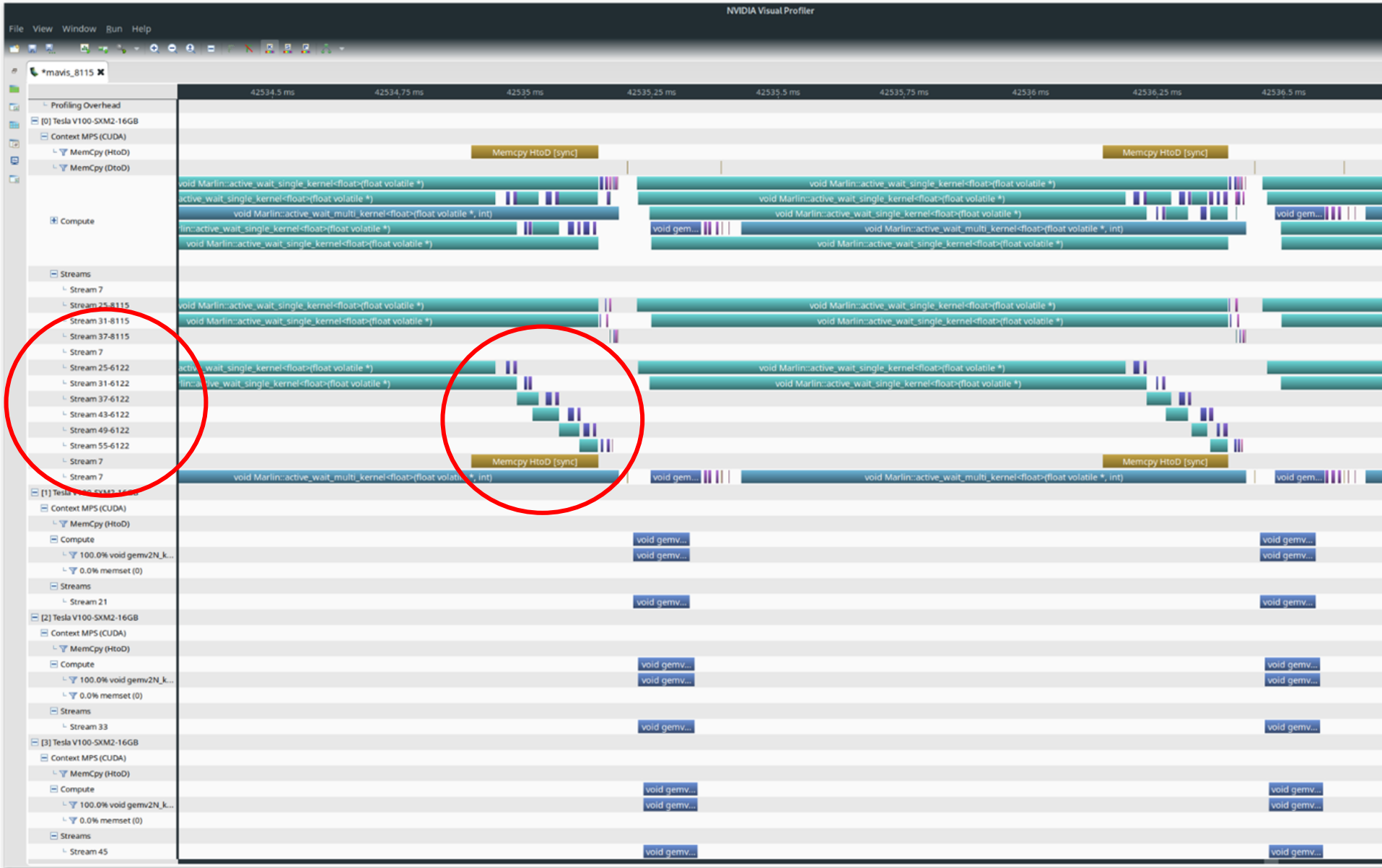


CONCURRENCY AND STREAMS

- The level of concurrency can vary:
  - Concurrency between memcpy and kernel (depends on the number of copy engines)
  - Concurrency between device and host



CONCURRENCY AND STREAMS



CONCURRENCY AND STREAMS

- CUDA operations must be on non-default streams to get concurrency
- Memcpy must be called with the asynchronous version ***cudaMemcpyAsync***
  - Host to device and device to host memcpy are asynchronous only if the source/target host memory is ***pinned***
  - Pinned host memory is page locked and allows faster transfer to the device
- Pinned memory allocation : ***cudaHostAlloc()***
- Stream creation : ***cudaStreamCreate(&stream);***
- Launching a kernel on a stream :
  kernel <<<blocks, threads, sharedMem, stream>>>();
- Synchronisation between streams using events : ***cudaStreamWaitEvent();***

CONCURRENCY AND STREAMS: MULTI-GPU EXAMPLE

➢ Goal: distributing kernel computation over multiple GPU

1. Create 1 stream and 1 event per device

```
cudaEvent_t ev[this->P2Pdevices.size()];
cudaStream_t st[this->P2Pdevices.size()];
for (auto dev_id : this->P2Pdevices) {
  cudaSetDevice(dev_id);
  cudaEventCreate(&ev[dev_id]);
  this->events.push_back(ev[dev_id]);
  cudaStreamCreate(&st[dev_id]);
  this->streams.push_back(st[dev_id]);
}
```

2. Distribute data over the devices

```
for (auto dev_id : this->P2Pdevices) {
  cudaSetDevice(dev_id);
  cudaMemcpyAsync(d_centroids_ngpu[dev_id]->get_data(),
                  centroids->get_data() + cc,
                  d_centroids_ngpu[dev_id]->get_nb_elements() * sizeof(T),
                  cudaMemcpyDeviceToDevice,
                  this->streams[dev_id]);
  cc += d_centroids_ngpu[dev_id]->get_nb_elements();
 }
}
```

Launching memcpy from each GPU to make them concurrent

CONCURRENCY AND STREAMS: MULTI-GPU EXAMPLE

➢ Goal: distributing kernel computation over multiple GPU

3. Launch computations

```
for (auto dev_id : this->P2Pdevices) {
    cudaSetDevice(dev_id);
    kernel<<<nBlocks, nThreads, 0, streams[dev_id]>>>(d_centroids_ngpu[dev_id]->get_data());
    cudaEventRecord(events[dev_id], streams[dev_id]);
}
```

Launching kernel on a stream

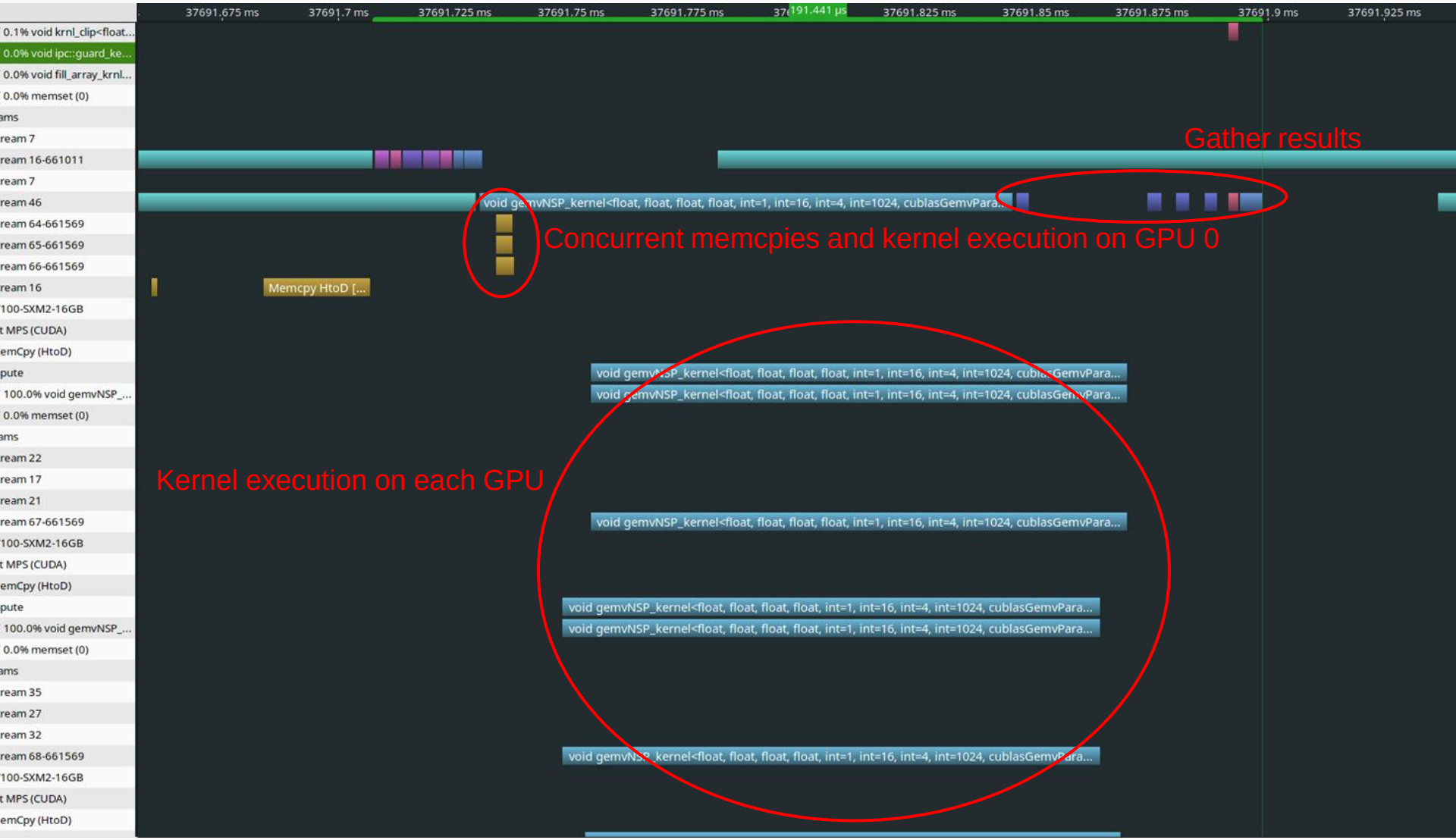
Recording an event on the same stream to notify end of computation

4. Gather the results

Wait for the GPU dev_id to finish its computation before gathering

```
cudaSetDevice(0);
for (auto dev_id : this->P2Pdevices) {
    cudaStreamWaitEvent(streams[0], events[dev_id], 0);
    gatherKernel<<<nBlocks, nThreads, 0, streams[0]>>>(d_centroids_ngpu[dev_id]->get_data(), d_centroids);
}
```

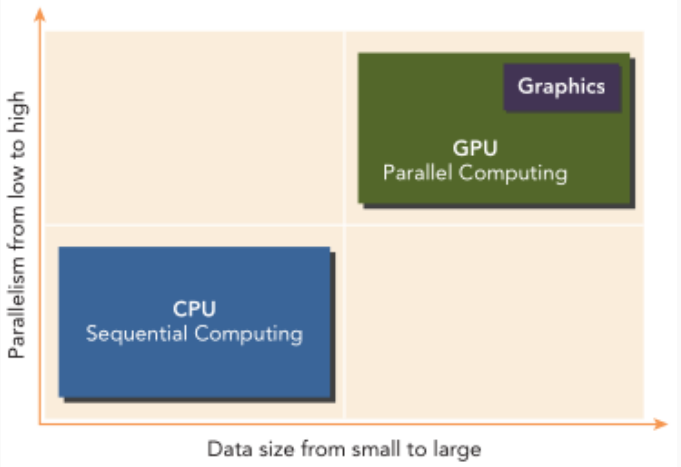
CONCURRENCY AND STREAMS: MULTI-GPU EXAMPLE

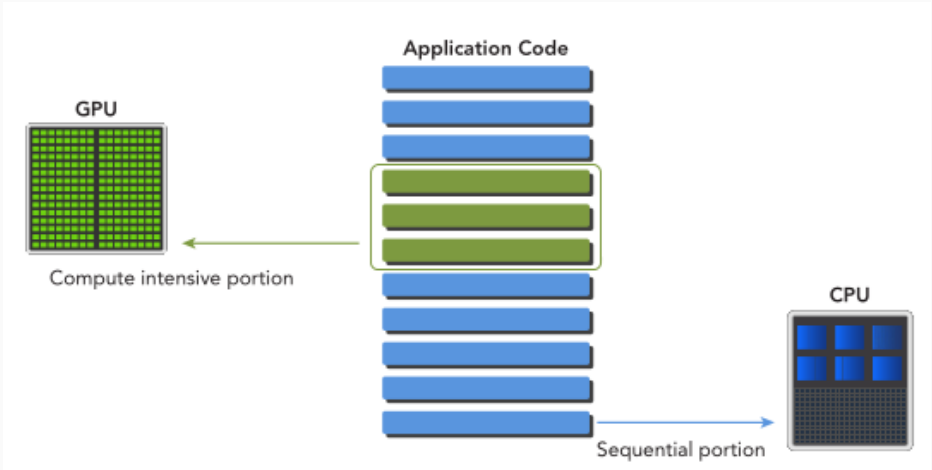


HETEROGENEOUS COMPUTING

- GPU is efficient for massively parallel code thanks to its architecture
- Porting a code to GPU is not always the solution for speed-up:
  - GPU are more efficient with high workload
  - Algorithms running on the GPU should be able to leverage its architecture
- Heterogeneous computing can bring the best of the two worlds:
  - Amdahl's law gives the maximum speedup S considering the fraction P of the total execution time that can be parallelized on N processors:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$





WRAP-UP

- This lecture was the most intense of the module…

STILL HERE ?

- Optimising GPU code is the harder part of GPU computing

- But,
  - Optimisation must be done only if it is useful
  - The process can be very long, so as soon as your code fits the requirements, ends it !
  - Following simple best practices while coding in CUDA is often enough to get good performance
  - ***THE*** guide : https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

GPU, PYTHON & APPLICATIONS

Let's end this course smoothly

PYTHON

- TIOBE index (December 2021): award given to the programming language that has had the highest increase in ratings

Dec 2021	Dec 2020	Change	Programming Language	Ratings	Change
1	3	↑	Python	12.90%	+0.69%
2	1	↓	C	11.80%	-4.69%
3	2	↓	Java	10.12%	-2.41%
4	4		C++	7.73%	+0.82%
5	**5**		**C#**	**6.40%**	**+2.21%**

Source : https://www.tiobe.com/tiobe-index/

- PYPL index (December 2021): popularity index based on how often language tutorials are searched on Google

Worldwide, Dec 2021 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	30.21 %	-0.5 %
2		Java	17.82 %	+1.3 %
3		JavaScript	9.16 %	+0.6 %
4		C#	7.53 %	+1.0 %
5		C/C++	6.82 %	+0.6 %

Source : https://pypl.github.io/PYPL.html

Why python ?

- Easy to learn
  - Interpreted language
  - Intuitive syntax
- Easy to extend
  - Package Installer pip
  - Package and environment manager (Anaconda)
- Flexible
  - Imperative
  - Object-Oriented
- Libraries
  - Matplotlib
  - Numpy
  - Literally tons of others…
- Large community
  - Easy to find support on the net

CUDA PYTHON

- Naturally, NVIDIA is pushing CUDA to Python: CUDA Python is released with CUDA Toolkit 11.4
- Provides python wrappers for CUDA driver and runtime APIs
- Not intended for direct use
- Provides common interfaces for third-party libraries based on CUDA
- Most known libraries:
  - CuPy: numpy and scipy for GPU
  - Numba: JIT (Just In Time) compiler

```python
from cuda import cuda, nvrtc
import numpy as np

saxpy = """\
extern "C" __global__
void saxpy(float a, float *x, float *y, float *out,
size_t n)
{
 size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid < n) {
   out[tid] = a * x[tid] + y[tid];
 }
}
"""

# Create program
err, prog = nvrtc.nvrtcCreateProgram(str.encode(saxpy),
b"saxpy.cu", 0, [], [])

# Compile program
opts = [b"--fmad=false", b"--gpu-architecture=compute_75"]
err, = nvrtc.nvrtcCompileProgram(prog, 2, opts)

# Get PTX from compilation
err, ptxSize = nvrtc.nvrtcGetPTXSize(prog)
ptx = b" " * ptxSize
err, = nvrtc.nvrtcGetPTX(prog, ptx)

# Initialize CUDA Driver API
err, = cuda.cuInit(0)

# Retrieve handle for device 0
err, cuDevice = cuda.cuDeviceGet(0)

# Create context
err, context = cuda.cuCtxCreate(0, cuDevice)

# Load PTX as module data and retrieve function
ptx = np.char.array(ptx)

# Note: Incompatible --gpu-architecture would be detected
here
err, module = cuda.cuModuleLoadData(ptx.ctypes.data)
err, kernel = cuda.cuModuleGetFunction(module, b"saxpy")
```

```python
NUM_THREADS = 512  # Threads per block
NUM_BLOCKS = 32768  # Blocks per grid

a = np.array([2.0], dtype=np.float32)
n = np.array(NUM_THREADS * NUM_BLOCKS, dtype=np.uint32)
bufferSize = n * a.itemsize

hX = np.random.rand(n).astype(dtype=np.float32)
hY = np.random.rand(n).astype(dtype=np.float32)
hOut = np.zeros(n).astype(dtype=np.float32)

err, dXclass = cuda.cuMemAlloc(bufferSize)
err, dYclass = cuda.cuMemAlloc(bufferSize)
err, dOutclass = cuda.cuMemAlloc(bufferSize)

err, stream = cuda.cuStreamCreate(0)

err, = cuda.cuMemcpyHtoDAsync(
   dXclass, hX.ctypes.data, bufferSize, stream
)
err, = cuda.cuMemcpyHtoDAsync(
   dYclass, hY.ctypes.data, bufferSize, stream
)

# The following code example is not intuitive
# Subject to change in a future release
dX = np.array([int(dXclass)], dtype=np.uint64)
dY = np.array([int(dYclass)], dtype=np.uint64)
dOut = np.array([int(dOutclass)], dtype=np.uint64)

args = [a, dX, dY, dOut, n]
args = np.array([arg.ctypes.data for arg in args],
dtype=np.uint64)

err, = cuda.cuLaunchKernel(
   kernel,
   NUM_BLOCKS,  # grid x dim
   1,  # grid y dim
   1,  # grid z dim
   NUM_THREADS,  # block x dim
   1,  # block y dim
   1,  # block z dim
   0,  # dynamic shared memory
   stream,  # stream
   args.ctypes.data,  # kernel arguments
   0,  # extra (ignore)
)
```

CUPY

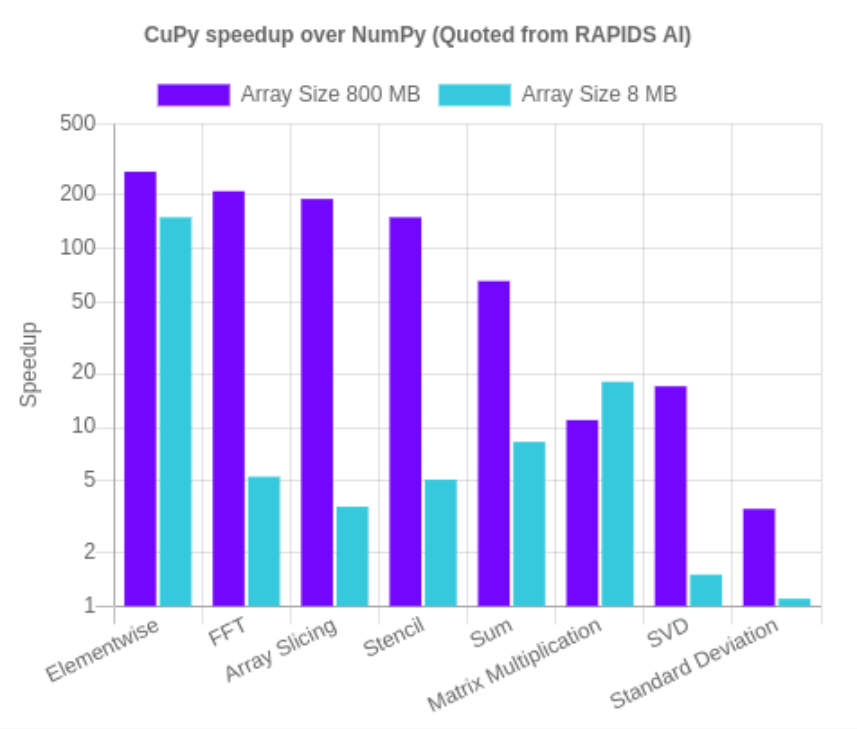
- API highly compatible with numpy & scipy

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([  3.,  12.], dtype=float32)
```

- Custom CUDA kernel

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''', 'my_add')
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y))  # grid, block and arguments
>>> y
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

- Great speed-up



NUMBA

- JIT compiler for Python
  - Python is slow
  - Code is compiled “just-in-time” for execution →speed up !
- API based on decorators
- Effective on functions which manipulates numpy arrays or loops

```python
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

NUMBA

- Supports NVIDIA GPU through CUDA and dedicated decorators
- Allow writing and calling kernel in a pythonic way

```
@cuda.jit
def increment_by_one(an_array):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < an_array.size:  # Check array boundaries
        an_array[pos] += 1
```

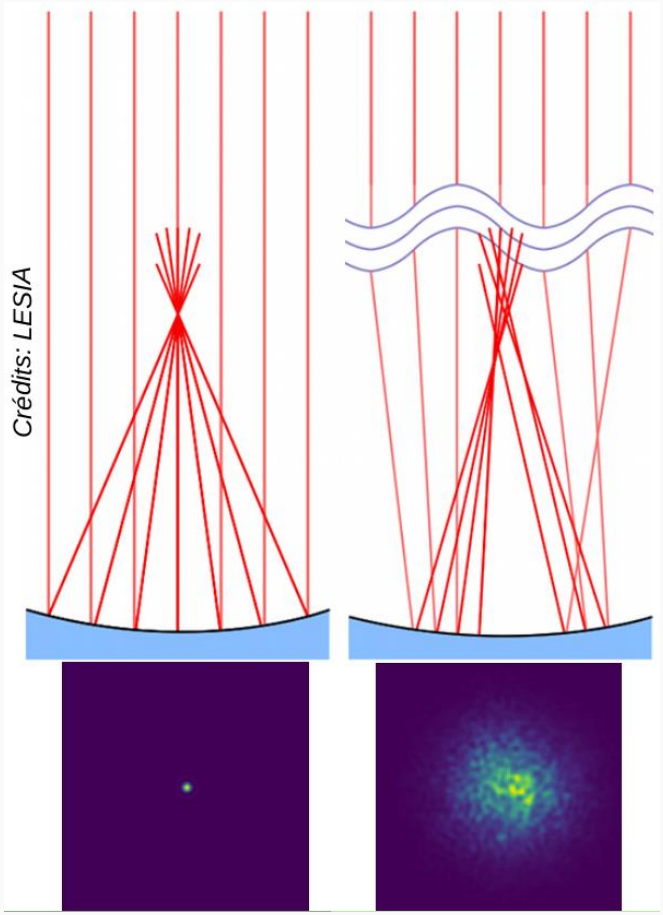
```
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) // threadsperblock
increment_by_one[blockspergrid, threadsperblock](an_array)
```

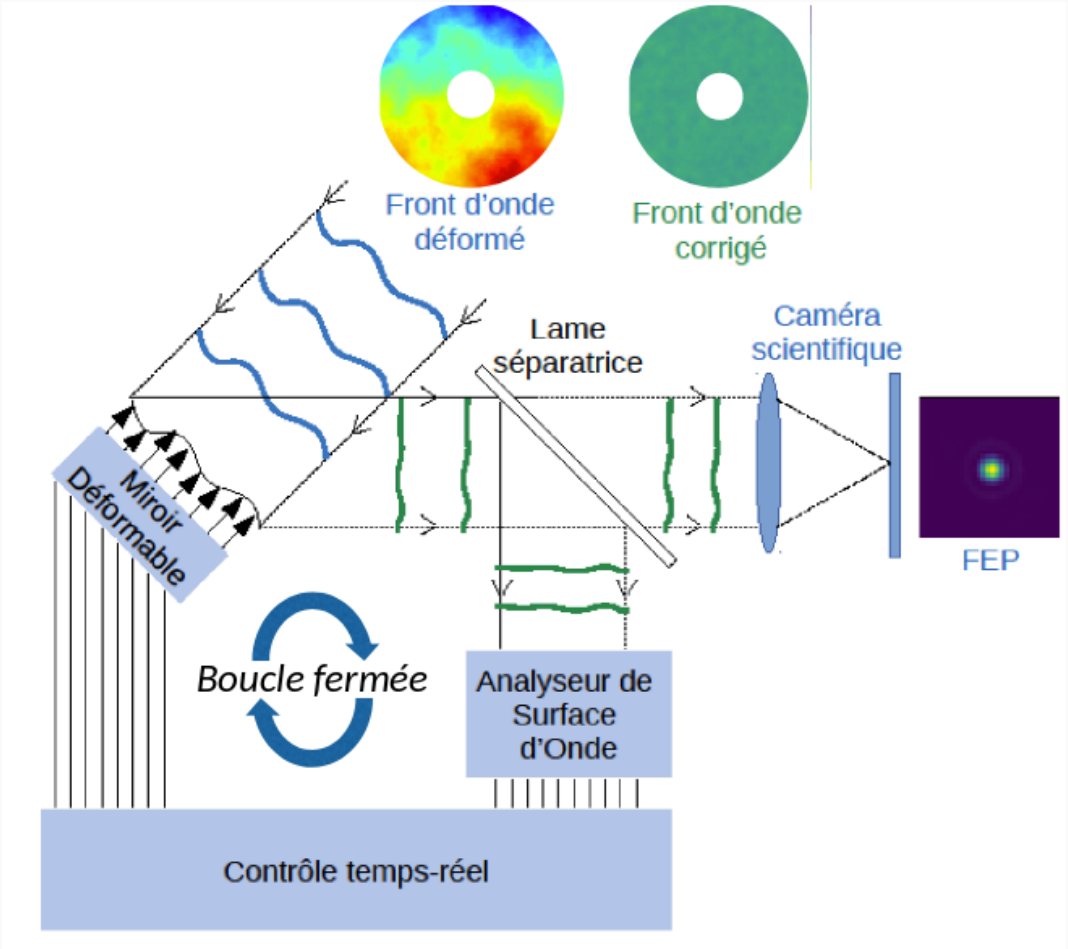
CHOOSE THE RIGHT TOOL FOR YOUR TASK

- The choice of using CUDA directly in C, or through a Python library is, again, dependent on your application
  - Python is easier to handle and to debug…
  - …but can add some performance overhead (first compilation, etc…)
  - All the CUDA features are not available through Python
- Another possibility is to develop CUDA related code in C, and to build the wrappers to Python:
  - Pybind11 is a great library to interface C and Python code
  - Cython

SOME REAL-LIFE APPLICATIONS OF GPU PROGRAMMING

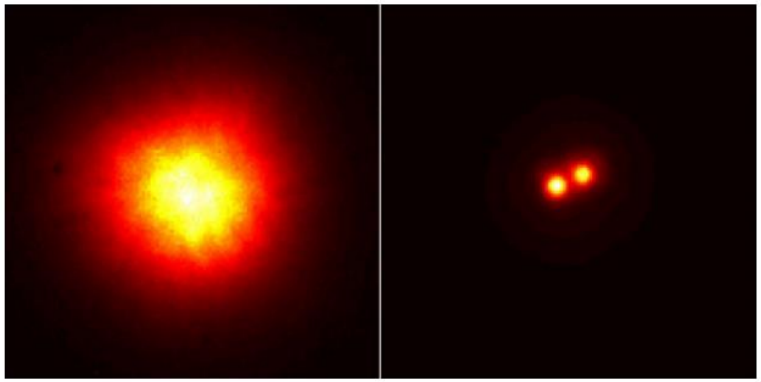
- Scientific computing:
  - A *(not so)* random example: adaptive optics for astronomy
  - Goal: compensating atmospheric turbulence impact on observations





ADAPTIVE OPTICS: FEW EXAMPLES

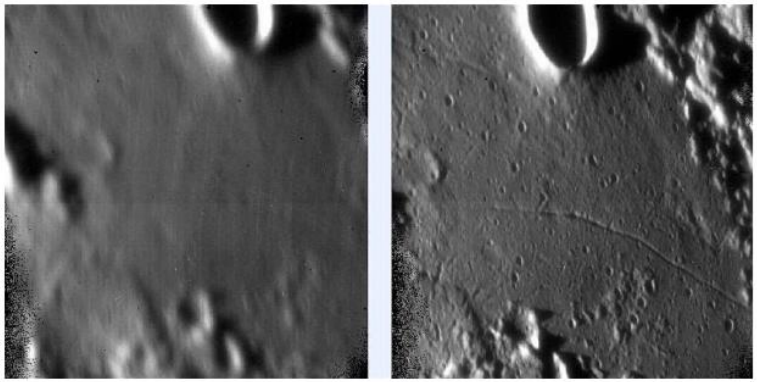
Étoile double : HIC 59206



Crédit : ESO

Sans OA Avec OA

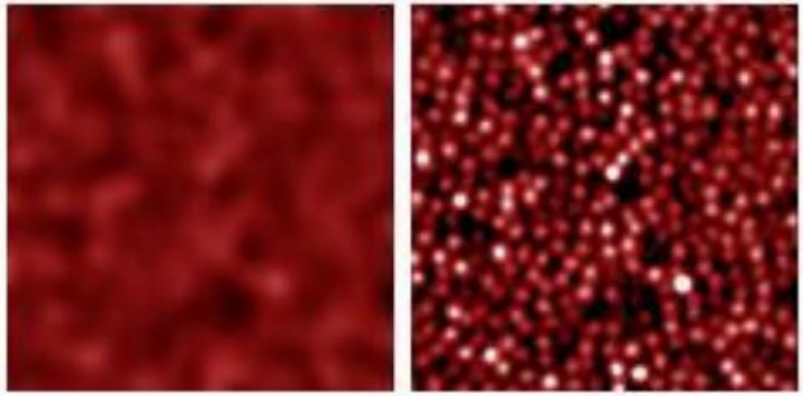
Lune



Crédit : ESO

Sans OA Avec OA

Cellules photo-réceptrices de la rétine



Crédit : INOVEO

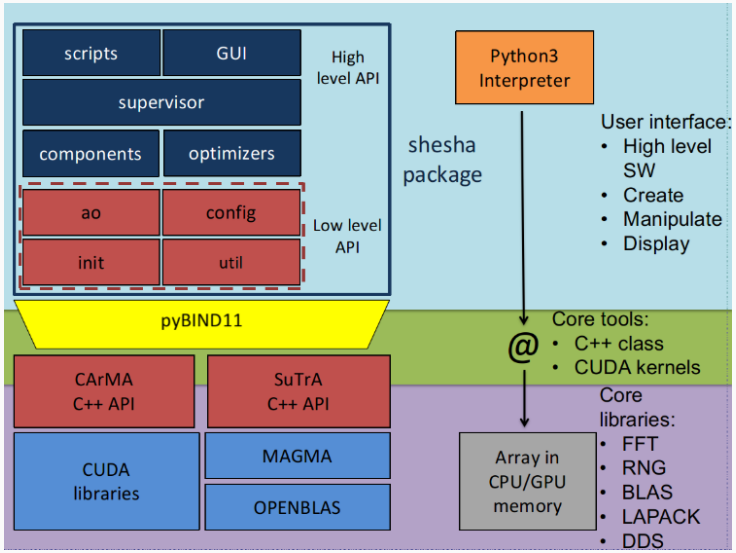
Sans OA Avec OA

The Extremely Large Telescope

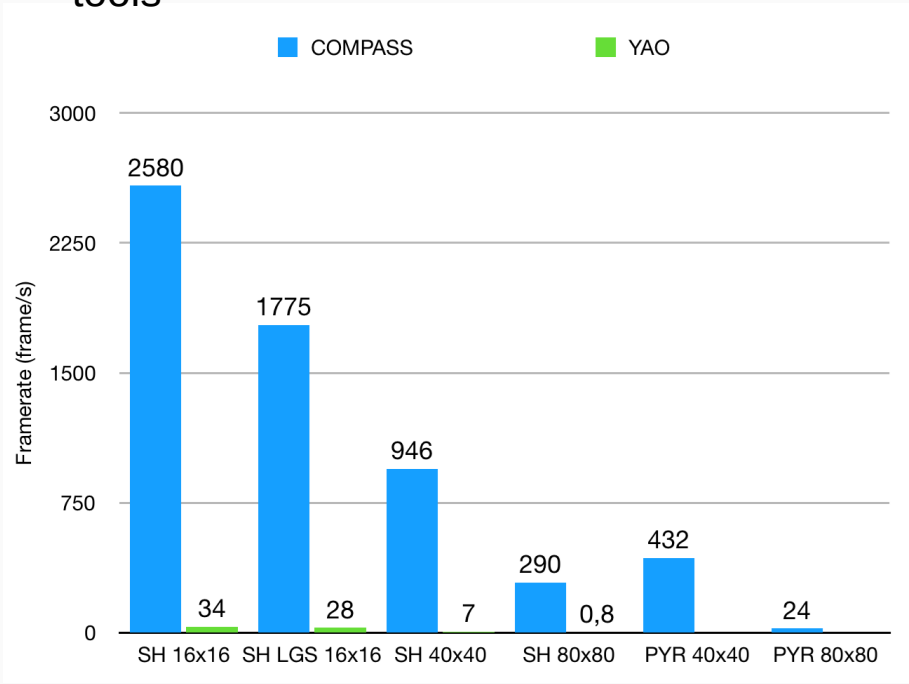
- Primary mirror of 39m diameter composed of ~800 segments
- 100m height
- 2,800 tons structure rotating at 360°
- First light in 2027
- Budget > 1G€

END-TO-END SIMULATION USING GPU

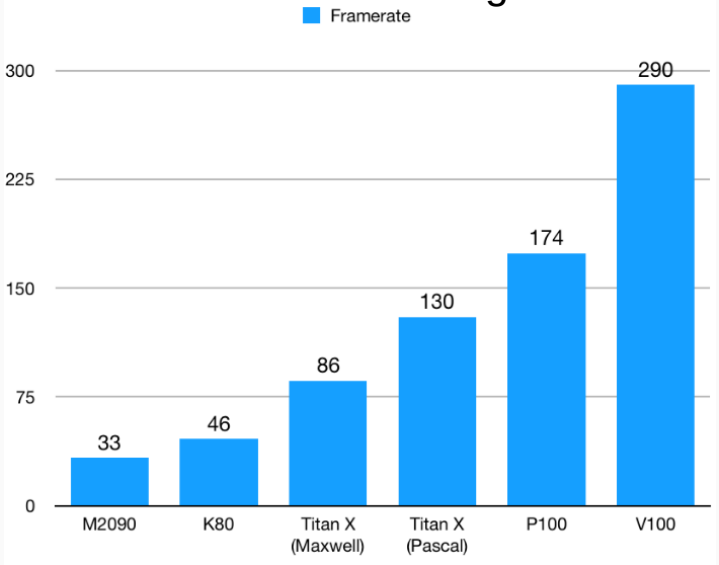
- Leverage best of 2 worlds:
  - User-friendly Python API
  - Core CUDA C computations



- Huge speed-up compare to other tools

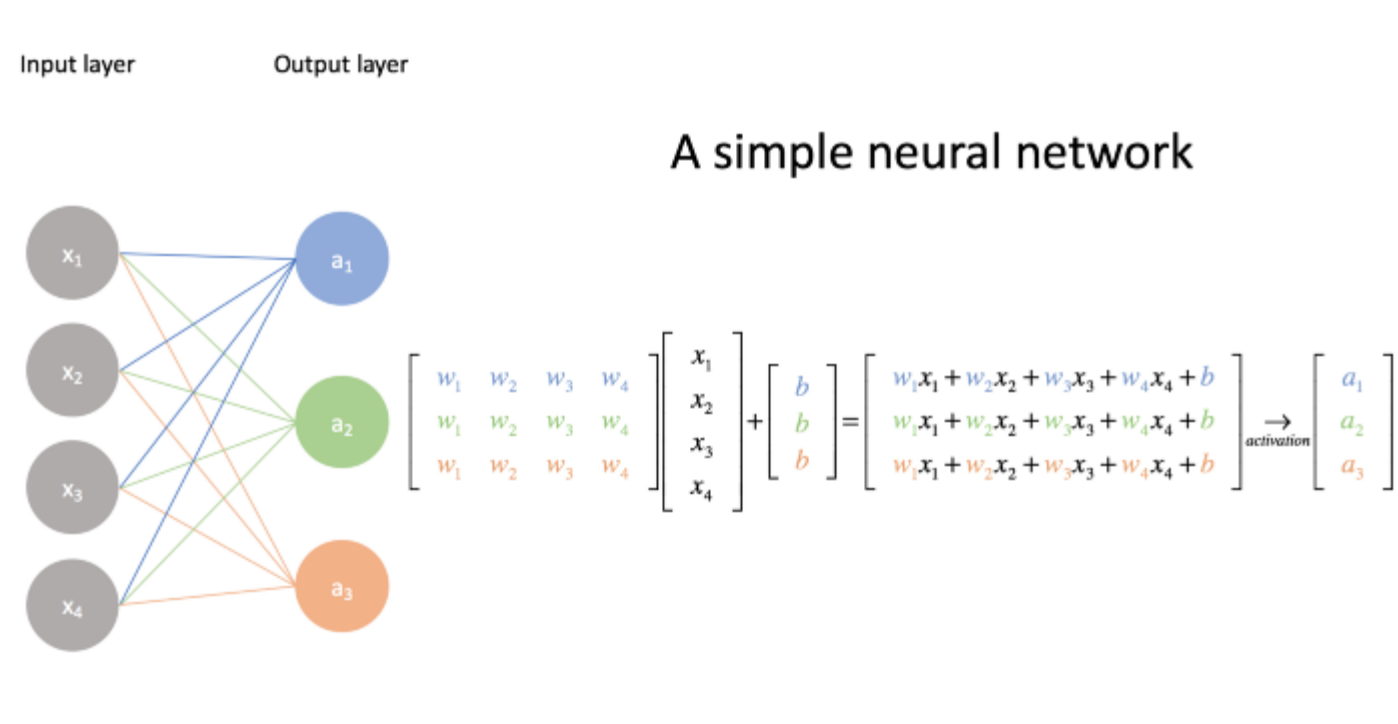


- Scale with new GPU generations



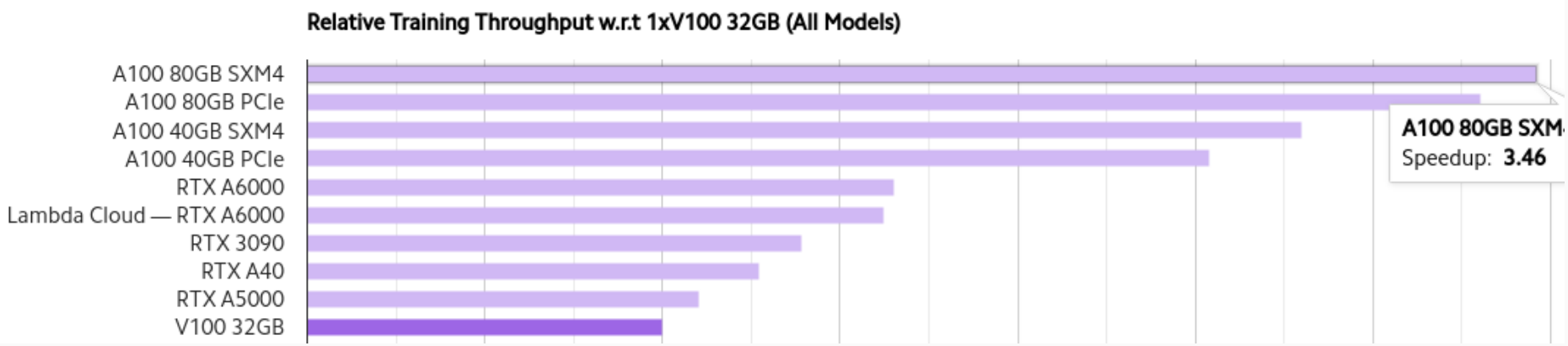
SOME REAL-LIFE APPLICATIONS OF GPU PROGRAMMING

- Artificial Intelligence:
  - GPU are particularly well suited for accelerating AI
  - Neural Network relies mostly on matrix-matrix multiplication
  - Leverage Tensor Cores



Artificial Intelligence

- Also very performant to accelerate AI models training:
  - Training is a massively parallel task
  - Requires huge datasets → Leverage GPU high memory bandwidth !



- AI frameworks come with native GPU support
  - PyTorch
  - TensorFlow

SOME REAL-LIFE APPLICATIONS OF GPU PROGRAMMING

- Embedded systems:
  - Generally associated to real-time constraints
  - Applications in avionics, automotive, robotics, etc…

BUT WAIT… HOW DO WE PUT A GPU IN SUCH SYSTEM ?

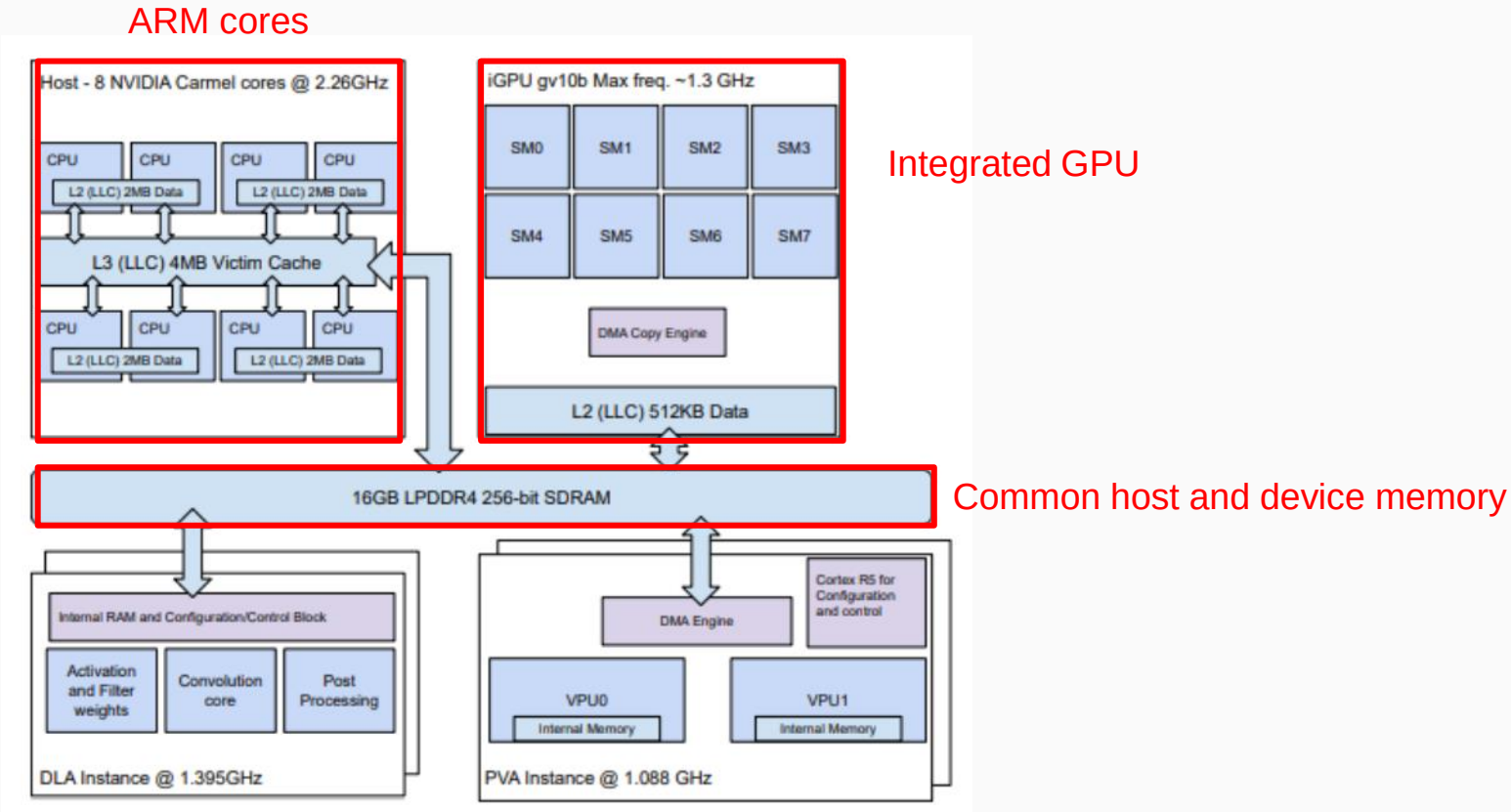
EMBEDDED SYSTEMS

- Example: NVIDIA Jetson AGX Xavier
  - Volta GPU
  - ARM CPU cores
  - Power consumption around 30 W
- CUDA Toolkit compatible:
  - Only a few advanced features are missing
  - Have to cope with memory architecture shared between CPU & GPU

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-Bit LPDDR4x | 137GB/s
Storage	32GB eMMC 5.1
DL Accelerator	(2x) NVDLA Engines
Vision Accelerator	7-way VLIW Vision Processor
Encoder/Decoder	(2x) 4Kp60 | HEVC/(2x) 4Kp60 | 12-Bit Support
Size	105 mm x 105 mm x 65 mm
Deployment	Module (Jetson AGX Xavier)

EMBEDDED SYSTEMS

➢ NVIDIA Jetson AGX Xavier architecture



➢ Some impacts on memory management with CUDA :
https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

OTHER PLAYERS

- NVIDIA is not the only GPU vendor:
  - So far, we focused on CUDA because it is the most mature
- Each vendor comes with its own toolkit:
  - AMD: ROCm
  - Intel: oneAPI
- Fortunately, they are very similar to CUDA
- AMD also develops HIP: Heterogeneous-Computing Interface for Portability
  - C++ runtime API to develop portable code
  - Same source code can be run on AMD GPU, NVIDIA GPU or CPU
  - Syntax close to CUDA

OTHER PLAYERS

- HIP sample code

```
template <typename T>
__global__ void
vector_square(T *C_d, const T *A_d, size_t N)
{
    size_t offset = (blockIdx.x * blockDim.x + threadIdx.x);
    size_t stride = blockDim.x * gridDim.x;
    for (size_t i=offset; i<N; i+=stride) {
        C_d[i] = A_d[i] * A_d[i];
    }
}
```

```
hipMalloc(&A_d, Nbytes));
hipMalloc(&C_d, Nbytes));
hipMemcpy(A_d, A_h, Nbytes, hipMemcpyHostToDevice);
const unsigned blocks = 512;
const unsigned threadsPerBlock = 256;
hipLaunchKernel(vector_square,   /* compute kernel*/
                dim3(blocks), dim3(threadsPerBlock), 0/*dynamic shared*/, 0/*stream*/,     /*
launch config*/
                C_d, A_d, N);  /* arguments to the compute kernel */
hipMemcpy(C_h, C_d, Nbytes, hipMemcpyDeviceToHost);
```

- Equivalent CUDA libraries available:
  - HIPblas, HIPfft, HIPcub, etc…
- Comes with tools to automatically translate CUDA code into HIP code
- HIP code is then compiled using NVIDIA or AMD compiler depending on the targeted device

Conclusions

Very quickly

What to retain ?

(ON TOP OF A FEW TECHNICAL DETAILS...)

- GPU computing is useful for massively parallel application
  - Thanks to its architecture
  - Many cores
  - High memory bandwidth
- GPU programming is made quite easy using dedicated API such CUDA
- Writing CUDA code is easy
- Optimising CUDA code is hard
- Leverage as much as possible existing libraries
- Other vendors are currently pushing to catch up: do not forget them

PROJECT

MINI-PROJECT, DON'T WORRY

Free project

- Project to be done by group of 2 students
- Find (or create) a sequential code of your choice, and try to accelerate it using CUDA
- Your work should contain at least one custom kernel
- Short oral presentation (~5min) of each group during the last directed studies slots
- Presentation guidelines:
  - Short presentation of the base sequential code
  - Main technical points of your CUDA implementation
  - Performance comparison
  - Analyze for further optimisations

DIRECTED STUDIES

- Exercise 0.1:
  - If you do not have a laptop powered by Nvidia, find a partner who have one !
  - Groups of 2 students is great, 3 max (if no other solutions)
- Exercise 0.2:
  - Install the CUDA toolkit on your laptop and makes it work !
  - Follows instructions on the CUDA toolkit documentation page
  - Will be great if it could be done before the first slot