Introduction to GPU Programming

Fabrice Huet

Outline for today (and later)

- Hardware support
  - From multi-core to GPUs
- CUDA
  - Programming and execution model
  - Memory organization
  - Writing kernels
- Numpy and Numba
  - Because C is hard
- Cheat sheet
- Lab sessions

Hardware support

From single core to multi-core

Life was easy

- Moore's law (1965):
  - Every 24 months, the number of transistors on a die can be doubled
- Number of transistors $\simeq$ performance
- So to make your program faster
  - Work hard and optimize it
  - Wait for next generation of cpu

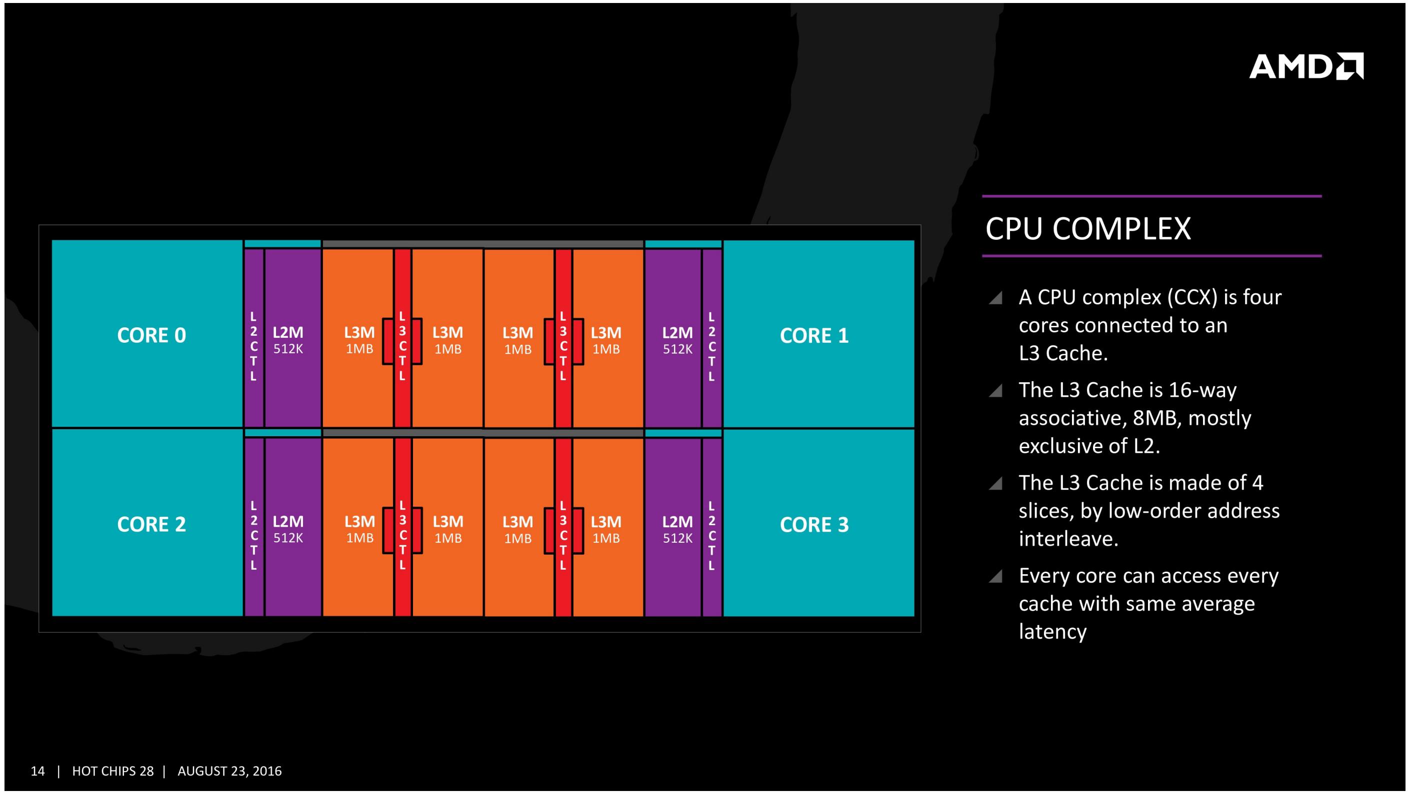
Limits to Moore's law

- Moore's law still holds
  - But no better performance for free
- What went wrong?
  - Physics
- Single core limits
  - Size of transistors
  - Size of die
- Solution :
  - Group many cores on a single die
  - Let's call this multicore

Multi-core CPUs

- Many core on the same die
  - Share some caches
  - Have a faster (direct) access to some memory region
    - Still a shared memory system
    - NUMA : Non Uniform Memory Access
- Cores can be simple or complex
  - If complex cores, MIMD : each core can execute arbitrary code
  - If simple cores, SIMD : all cores must execute the same code

AMD Ryzen 1 & 2



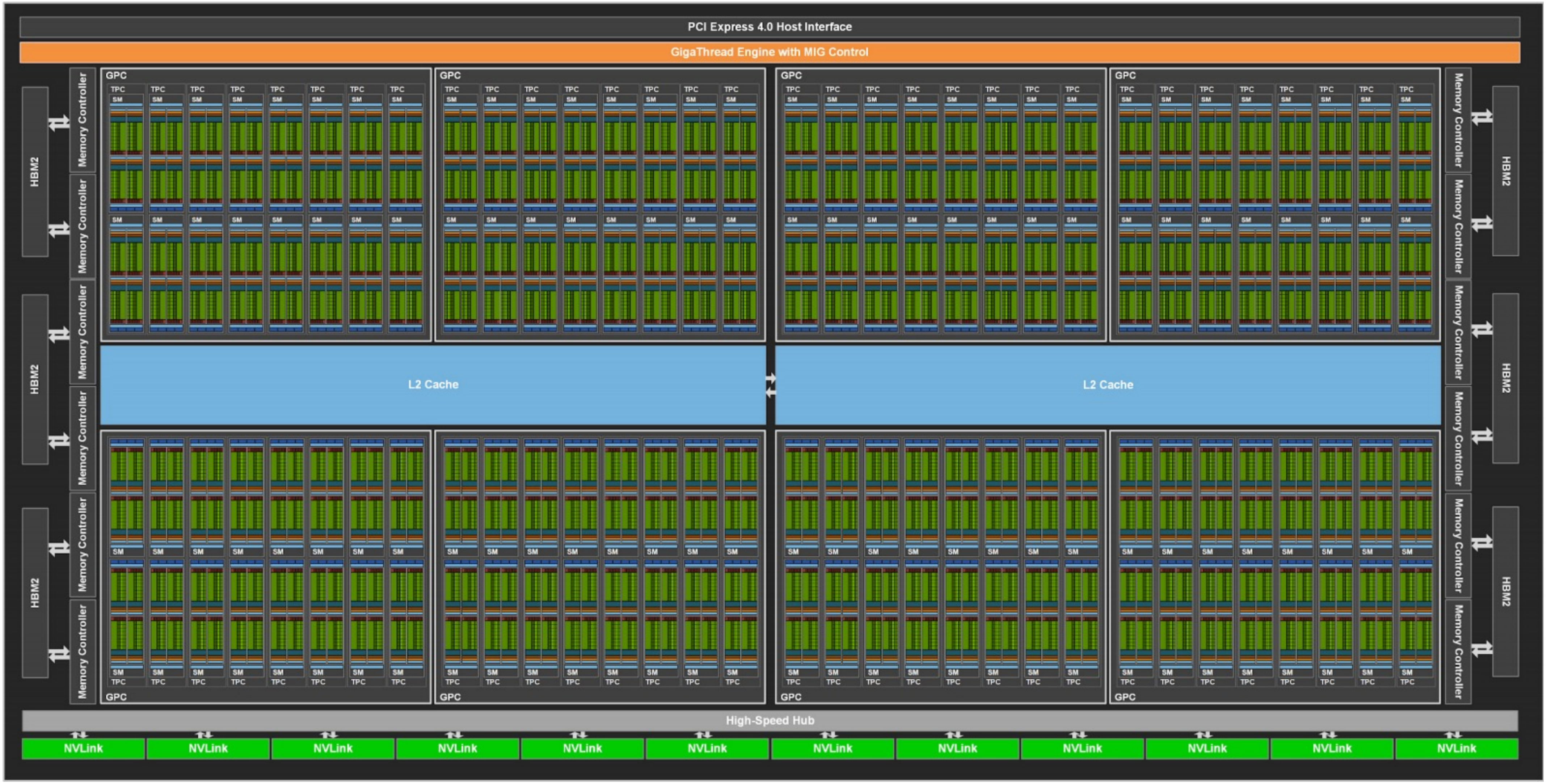
- Cores are grouped by 4 in a CCX
- Two CCX are grouped a CCD (Core Chiplet Die)
- Very modular architecture, increase CCD to have more cores
- Overall a MIMD
- 3.8 billions transistors per CCD

~~GP~~GPU

- General Purpose Graphics Processing Unit (GPGPU)
  - Based on graphical processors
  - Very good at floating point computation (and Integer now)
  - Massively parallel architecture
- Hybrid execution model
  - MIMD & SIMD
  - A GPU can execute multiple programs (aka kernels).
  - Each kernel is a SIMD code executed by many different cores
- Hierarchical architecture (NVIDIA)
  - GPU Processing Clusters
    - Streaming Multiprocessors
      - CUDA Cores

- NVIDIA Ampere (GA100)
  - 128 Streaming Multiprocessors
  - 8192 Cuda Cores
  - 512 Tensor Cores
  - 54.2 billions transistors

https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf



https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[redacted]

CUDA

Introduction

- Compute Unified Device Architecture
- Created by NVIDIA in 2007
- Encompass
  - Hardware architecture
  - Programming model
  - Execution Model
  - API
- Main idea
  - Execute part of a program on a GPU

Hardware architecture

- A standard machine called *host*
- One or many GPUs called *device*
  - Each with their own memory space (NUMA)
  - Support for large number of threads
- GPUs are co-processor
  - Main code is executed on a
  - Some part are sent for execution on GPUs
- Reminder :
  - GPU NVIDIA
    - Stream processors
      - Cuda cores

Cuda

Programming and execution model

Programming model

- Data parallelism
- You have to write the code executed by hardware threads
  - *Kernel* in NVIDIA's terminology
- You don't handle thread creation
- Threads are executed as SIMD
  - Same code for all threads
- Each thread has an id
  - Useful for having specific behavior
  - Very useful for data parallelism
- It's like a PRAM!

Thread blocks

- Threads are grouped into blocks
- Threads from a block
  - Are executed on the same Stream Processos
  - Can share variables
  - Can be synchronized (barrier)
- Number of threads per block
  - Hardware limit
  - 512 or 1024

Thread blocks

- A block has a structure 1D, 2D or 3D
  - Logical organization
  - Decided at runtime
- A block has dimensions
  - blockDim.x, blockDim.y, blockDim.z
- Each thread has coordinates (id) <u>inside</u> the block
  - threadIdx.x, threadIdx.y, threadIdx.z
- Why ?
  - Match threads IDs to problem at hand
  - Examples
    - Flat array : 1D
    - Image processing : 2D
    - Volume computation : 3D
- Limits :
  - Each dimension has a specific limit
  - Example : 1024 threads in a block which max dims (1024,1024,64)

Thread blocks

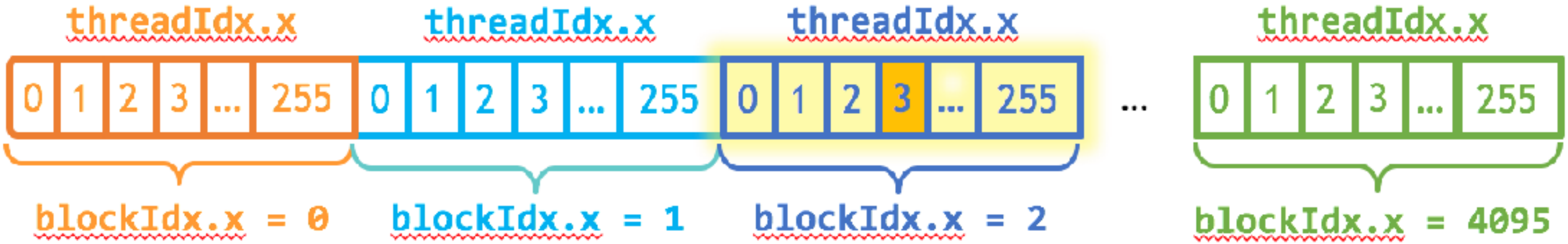
0	1	2	3	4	5	6	...	255

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

Grid

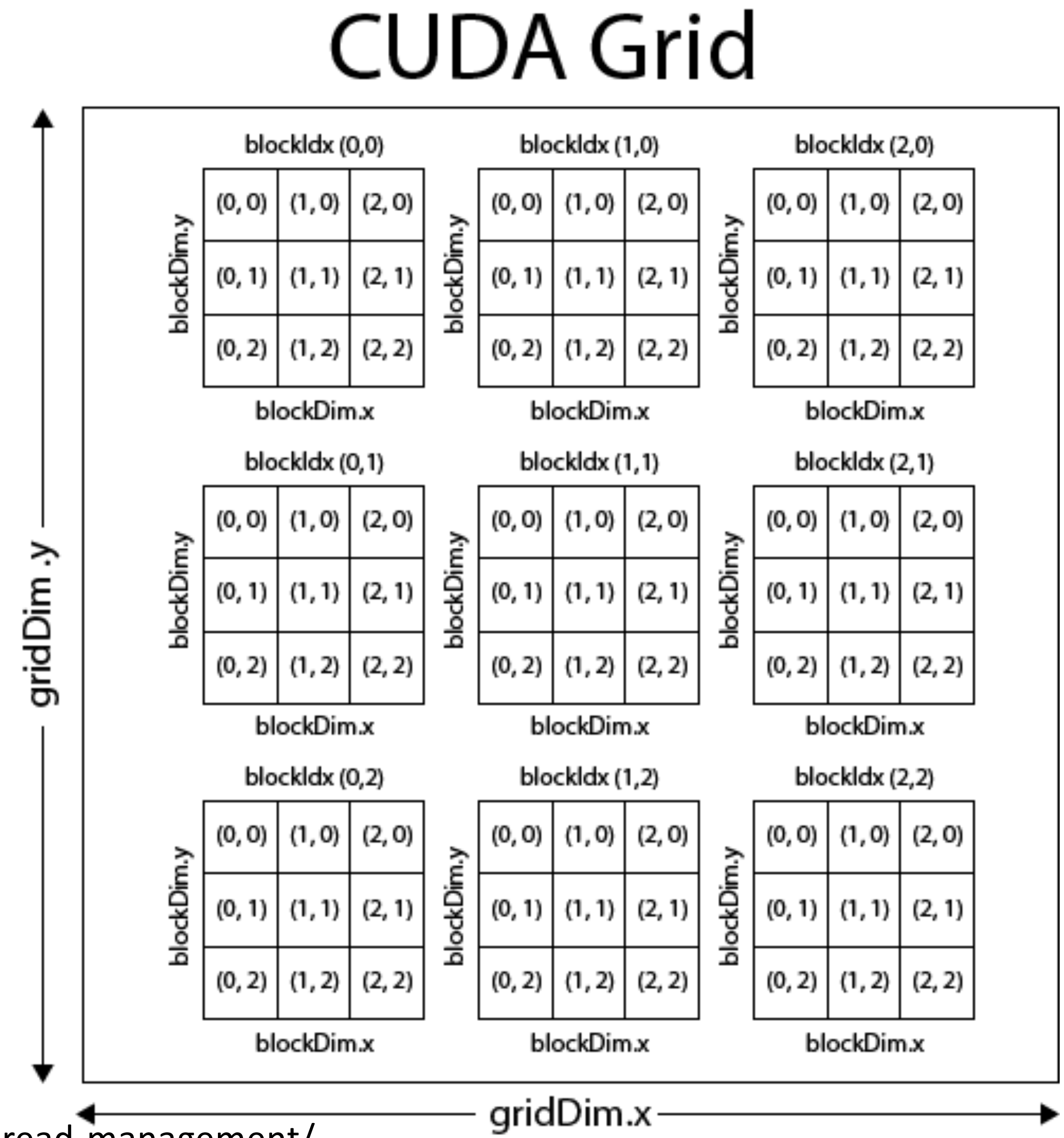
- How to execute more than 512/1024 threads ?
  - Use multiple thread blocks
- A Grid is a group of thread blocks
- A Grid is also structured as 1D, 2D or 3D
  - Independent from the thread block structure
  - Dimensions limited to (2^{31}-1,65535,65535)
- Each block inside a Grid has its own coordinates
  - blockIdx.x , blockIdx.y, blockIdx.z

Example 1D-1D



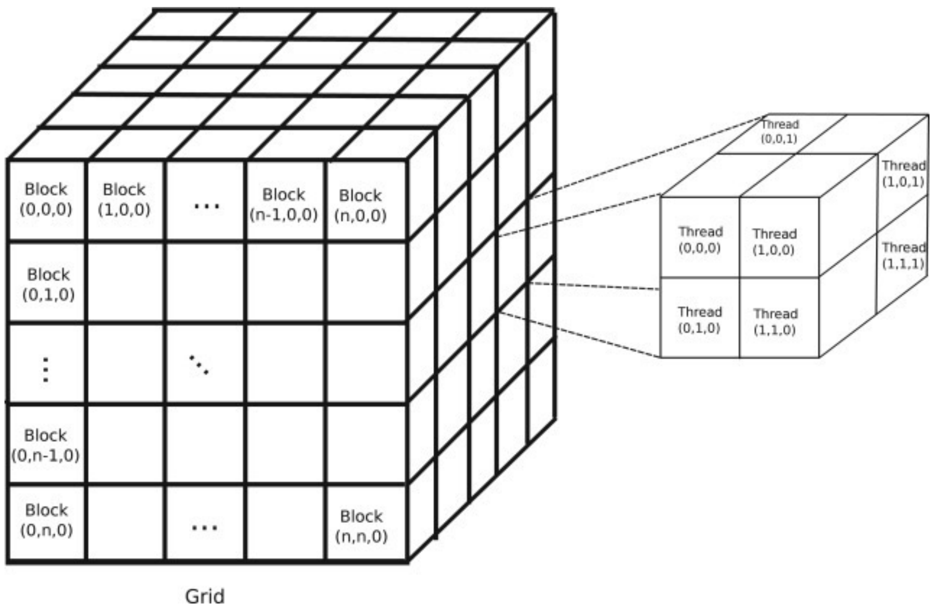
https://acenet-arc.github.io/ACENET_Summer_School_GPGPU/06-all-together/

Example 2D-2D



https://www.microway.com/hpc-tech-tips/cuda-parallel-thread-management/

Example 3D-3D



https://www.sciencedirect.com/science/article/pii/S0377042715001247

Kernel

- A kernel is composed of
  - The code to execute on the GPU
  - A single Grid
    - Specified as a triplet with the number of blocks on each dimension
  - Blocks
    - Specified as a triplet with the number of threads on each dimension
- Everything must obey the hardware limits
- A kernel does not return a value

Execution

- A kernel is executed per block
- As many blocks as possible executed in parallel
  - No specific order
- A block is executed piece-wise
  - Threads are grouped as warp of 32
- Warp == scheduling unit at the hardware level
  - All memory access of a warp are done at together

Example of hardware limitations

Technical specifications	Compute capability (version)																			
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128	16	128		
Maximum dimensionality of grid of thread blocks	2				3															
Maximum x-dimension of a grid of thread blocks	65535					$2^{31} - 1$														
Maximum y-, or z-dimension of a grid of thread blocks	65535																			
Maximum dimensionality of thread block	3																			
Maximum x- or y-dimension of a block	512				1024															
Maximum z-dimension of a block	64																			
Maximum number of threads per block	512				1024															
Warp size	32																			

https://en.wikipedia.org/wiki/CUDA

Cuda

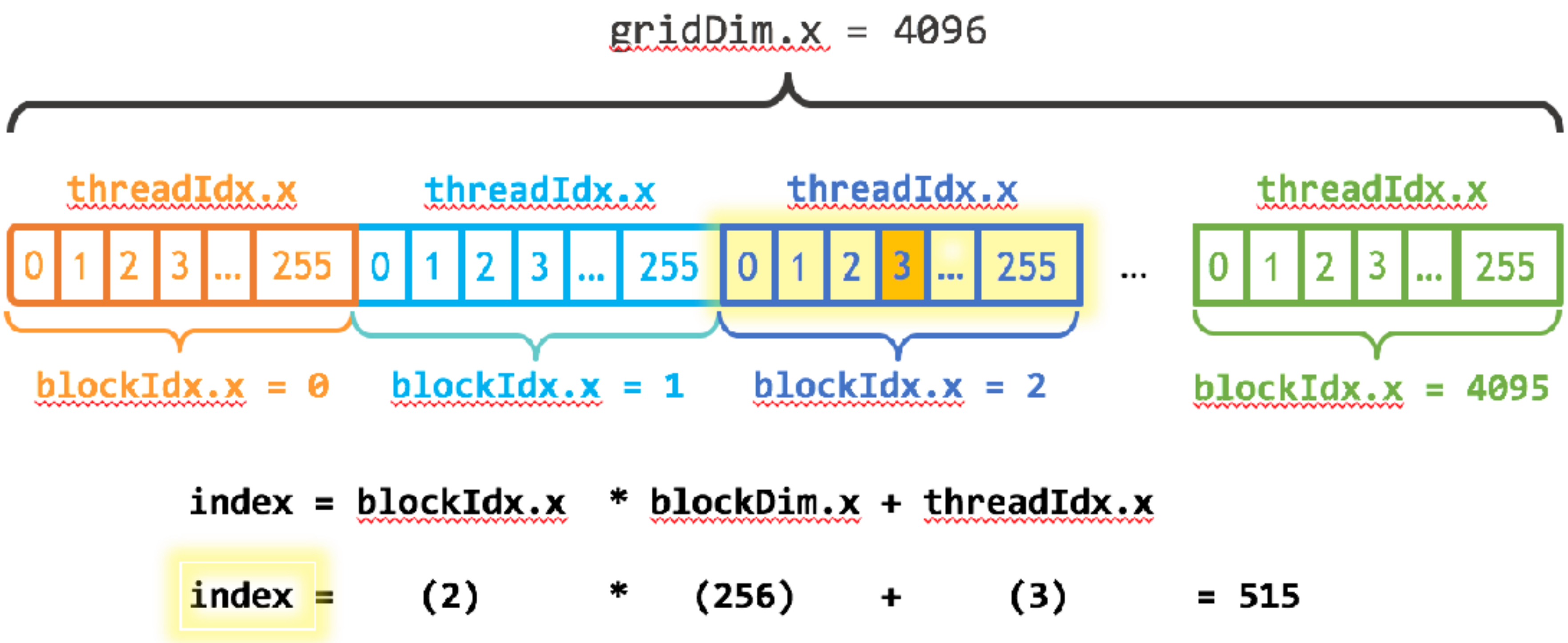
Managing coordinates

Local vs global coordinates

- Each thread has local coordinates
  - Inside its block
- Each block has local coordinates
  - Inside its grid
- In data parallelism, each thread should manage some data globally
  - So we need to transform local (block) coordinates to global ones

Local vs global coordinates

- Example : Processing a 1024 cells array
- The data is 1D, so we will use 1D blocks and grid
  - Size of a block : (64,1,1)
  - We then **compute** the Grid : (16,1,1)
- In each block threads will have an ID from 0 to
  - Ideally, we want all 1024 threads to have an ID between 0 and 1023
- How to obtain a global ID ?



https://acenet-arc.github.io/ACENET_Summer_School_GPGPU/06-all-together/

Local vs global coordinates

- Example : Processing a B&W image of size 1024x1024
- 2D topology is more natural
  - Block size : (32,32,1)
  - Grid size : (32,32,1)
- How to obtain a global ID ?

Cuda

Writing kernels

Introduction

- Cuda supports C/C++
  - Low level languages
  - Performance oriented
- Structure
  - Application code in .c or .cpp files
  - Kernel in .cu or directly in application files
- Compilation
  - nvcc
- Executing a cuda program
  - Like usual

High level languages

- First approach
  - Kernel in C
  - Host code in higher level languages
    - On the fly translation to C followed by nvcc
    - Direct compilation to binary and linking
- Second approach
  - Everything including kernel in HL language
  - Must wrap all the CUDA API
  - Translation to C or direct compilation

Numba

Writing CUDA in Python

Numba

- Full Python approach
- Python code is compiled to native code
  - Use of LLVM
- Decorator based
  - Code that should be compiled is annoted
- Installation
  - Easy if you have installed Anaconda first
  - Anaconda : Python for Data Science
- https://numba.pydata.org/

```
from numba import cuda
import numba as nb
```

Kernel

- A kernel is written in Python
  - Not all Python API is supported
- Add @cuda.jit in front of function

```
@cuda.jit
def writeGlobalIDUnevenArray(array):
```

Kernel

- Calling a kernel is similar to any function, just add
  - Grid size
  - Thread block size

```
writeGlobalIDUnevenArray[ blocksPerGrid,threadsPerBlock](d_A)
```

- Beware, execution is asynchronous
  - Call cuda.synchronize() to block
  - Or perform any memory transfer

Global and local coordinates

- Local

  `numba.cuda.threadIdx`

  `numba.cuda.blockIdx`

  `numba.cuda.blockDim`

  `numba.cuda.gridDim`

- Global

  `numba.cuda.grid(ndim)`

  - ndim : dimensions of your Grid

Global and local coordinates

- Example

```
@cuda.jit
def increment_by_one(an_array):
    pos = cuda.grid(1)
    if pos < an_array.size:
        an_array[pos] += 1
```

```
@cuda.jit
def increment_a_2D_array(an_array):
    x, y = cuda.grid(2)
    if x < an_array.shape[0] and y < an_array.shape[1]:
        an_array[x, y] += 1
```

https://numba.readthedocs.io/en/stable/cuda/kernels.html#absolute-positions

Limits

- Not all CUDA functions are supported
- No support for Python exceptions
- No support for most Python modules from kernel/device functions
  - But math is supported, good enough
- print(…) support
  - Only support for scalar types, no tuple or array
  - Can overload memory and cause a kernel crash
- No dynamic allocation on the device
  - Hence no variable size array, no append…