Parallel programming

Python Numba - 1

FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

What is “numba”?

- Numba is a powerful Python library that ***compiles*** Python code to machine code ***on-the-fly***, enhancing execution speed
- It eliminates the need for manually rewriting code in a lower-level language, making it accessible and user-friendly
- Numba's ***just-in-time compilation*** optimizes your Python code without the need for external compilation steps, resulting in faster execution
- It supports CPU and ***GPU acceleration***, making it a versatile tool for performance enhancement

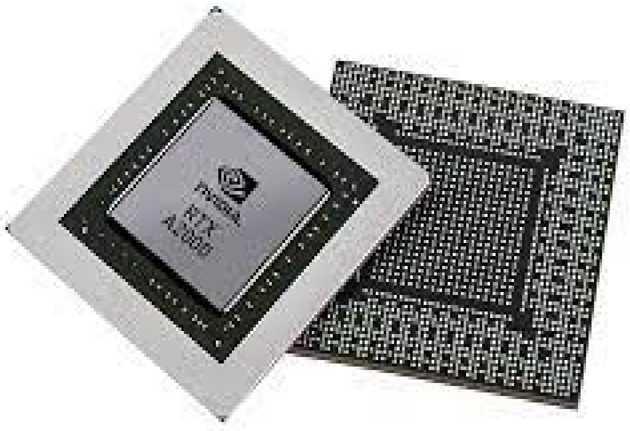
Numba

Why using “numba”?

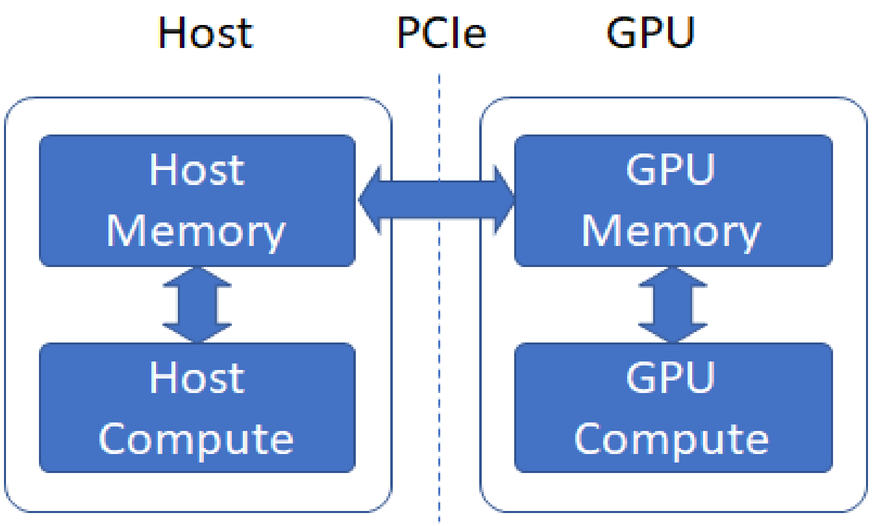
- ***Speed Up*** Your Python Code
- Numba isn't just about speed; it's about breaking free from Python's Global Interpreter Lock (GIL), enabling ***multi-threaded*** Python ***code execution***
- Numba's optimization capabilities result in significant speed improvements, making it a preferred choice for scientific computing
- You can apply Numba to data-intensive tasks like simulations, numerical computations, and more

Numba & CUDA GPU Programming

- CUDA is a parallel computing platform and API created by NVIDIA for GPU acceleration, and Numba ***seamlessly integrates with it***
- Numba extends its capabilities to GPU programming, allowing you to harness the ***massive parallel processing potential*** of GPUs
- With Numba and CUDA, you can accelerate data-intensive tasks, such as ***image processing and simulations***, by orders of magnitude



Terminology



- host: the CPU
- device: the GPU
- host memory: the system main memory
- device memory: onboard memory on a GPU card
- kernels: a GPU function launched by the host and executed on the device
- device function: a GPU function executed on the device which can only be called from the device (i.e. from a kernel or another device function)

Setting up python numba

- You can install the NVIDIA bindings with:

  $ conda install nvidia::cuda-python

- Or if you are using pip:

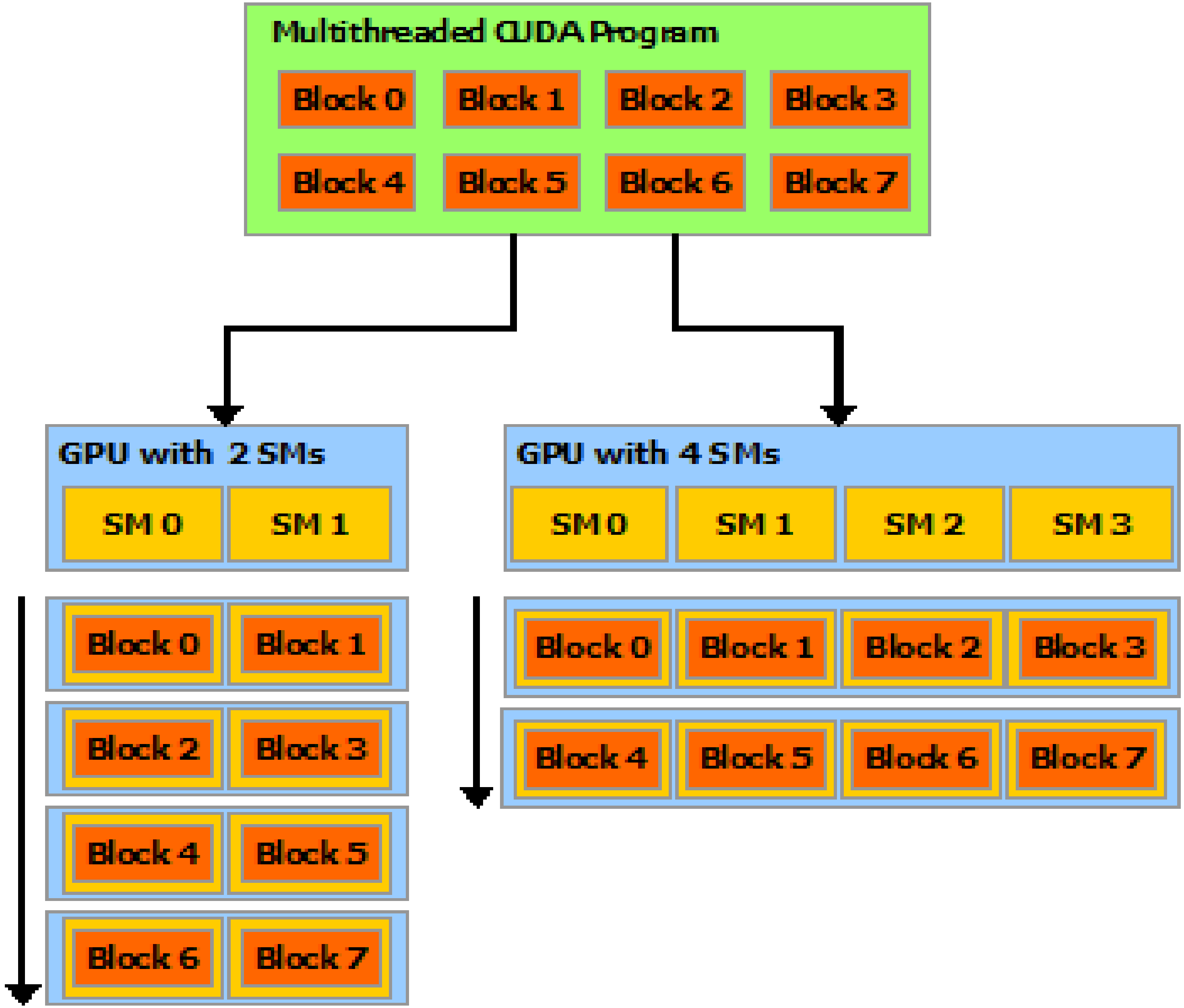
  $ pip install cuda-python

- Easy to work in Google Colab:
  https://colab.research.google.com

- Additional info:

  https://numba.readthedocs.io/en/stable/cuda/overview.html

CUDA recap



Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 SMs
SM 0
SM 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 SMs
SM 0
SM 1
SM 2
SM 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

CUDA Kernels

- A kernel function is a ***GPU function*** that is meant to be called from CPU code
- Kernels cannot explicitly return a value: all result data must be ***written to an array*** passed to the function
- Kernels explicitly declare their thread hierarchy when called: the number of thread blocks, the number of threads per block
- While a kernel is compiled once, it can be called multiple times with different block sizes or grid sizes
- *See the example of kernel declaration and invocation in the first&second sections of the provided .ipynb notebook*

Blocks of threads

- The block size (the number of threads per block) is often crucial:

  \- <u>Software side</u>: the block size determines how many threads access a given area of shared memory

  \- <u>Hardware side</u>: the block size must be large enough for full occupation of execution units (recommendations can be found in the CUDA C Programming Guide)

Threads & Blocks positioning

Inside block/grid:

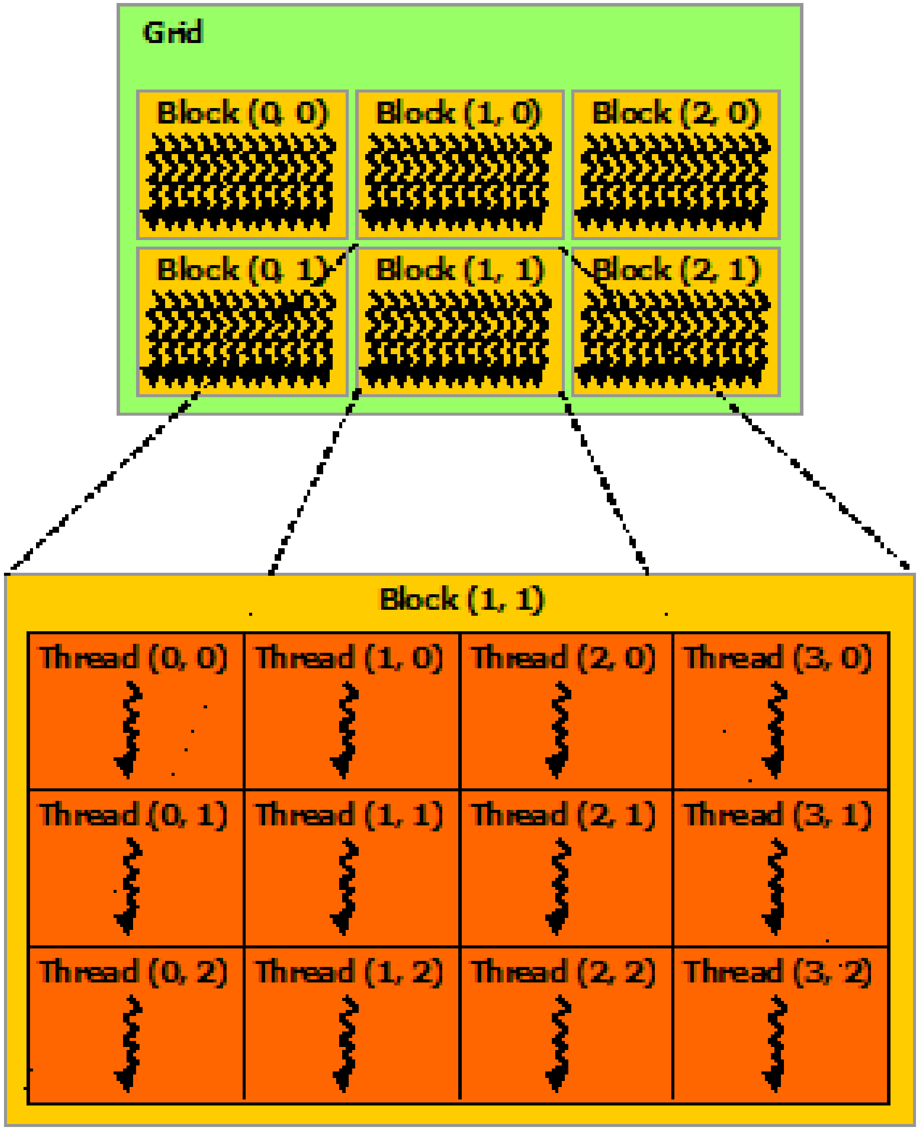
- numba.cuda.threadIdx
- numba.cuda.blockIdx

Dimensions:

- numba.cuda.blockDim
- numba.cuda.gridDim

Absolute positions:

- numba.cuda.grid(ndim)
- numba.cuda.gridsize(ndim)



Data transfer

- **Allocate device array:**

  - numba.cuda.device_array(…)

  - numba.cuda.device_array_like(…)

- **Copy the data from host to device:**

  - numba.cuda.to_device(…)

- **Copy the data from device to host:**

  - numba.cuda.copy_to_host(…)

Coding exercise

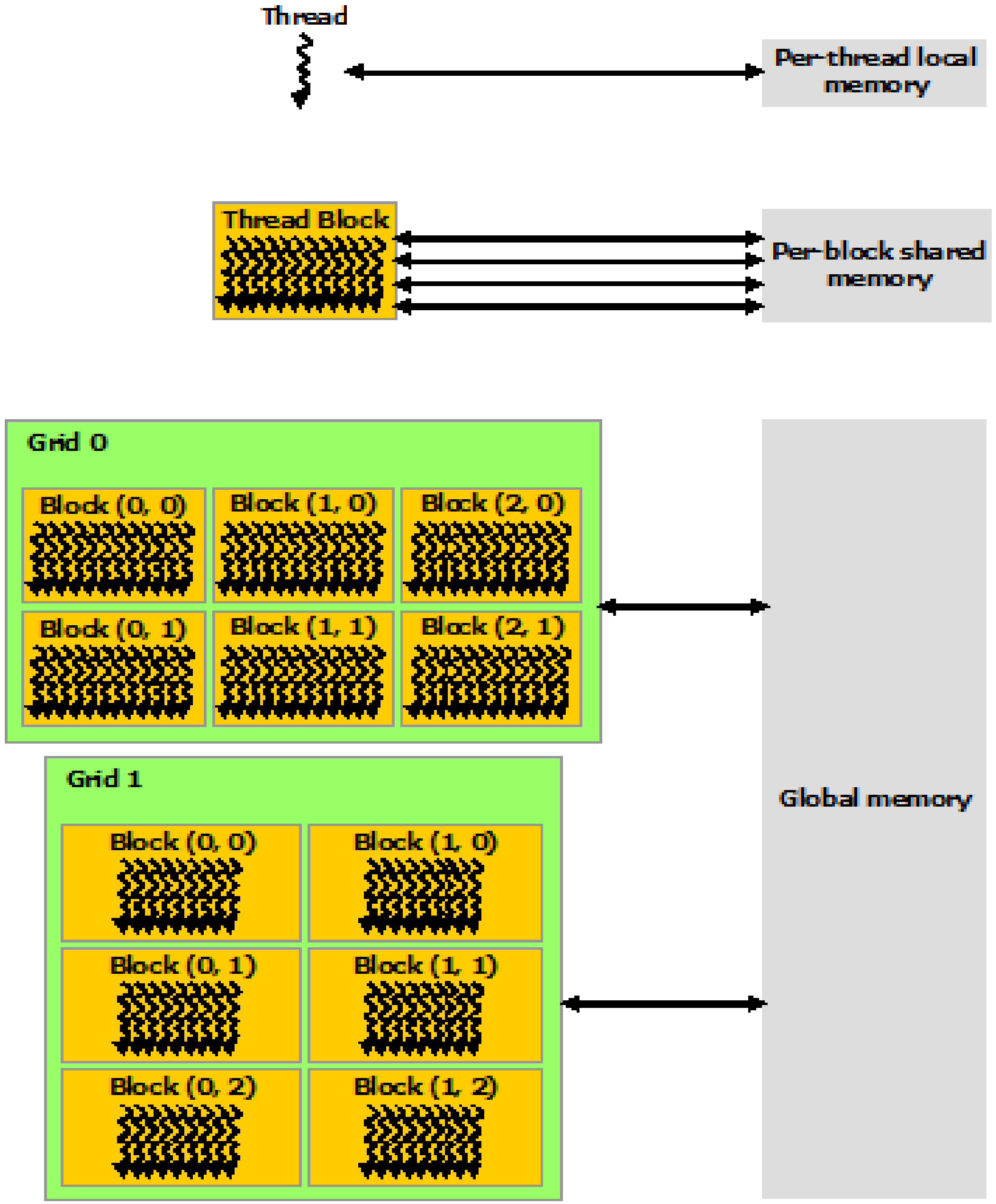
- Implement matrix-matrix multiplication using Python Numba:

  - transfer the data to device

  - declare and invoke the kernel

  - receive the result from device

Different GPU memory types



Shared memory

- A limited amount of shared memory can be allocated ***on the device*** to speed up access to data
- That memory will be shared (i.e. both readable and writable) amongst ***all threads belonging to a given block*** and has faster access times than regular device memory
- It also allows threads ***to cooperate*** on a given solution. You can think of it as a manually-managed data cache
- The memory is ***allocated once*** for the duration of the kernel

Shared memory & synchronization

- *numba.cuda.shared.array(shape, type)*

  \- Allocate a shared array of the given shape and type on the device

  \- The function must be called from the device

- *numba.cuda.syncthreads()*

  \- Synchronize all threads in the same thread block

  \- This function implements the pattern of barrier

Local memory

- Local memory is the memory area ***private to a thread***:

  \- *numba.cuda.local.array(shape, type)*

- Using local memory helps to allocate some ***scratchpad area*** when scalar local variables are not enough

- The memory is ***allocated once*** for the duration of the kernel

Constant memory

- Constant memory is an area of memory that is read only, cached and off-chip: numba.cuda.const.array_like(arr)

- Accessible ***by all threads***

- Allocated from the host

Coding exercise

- Implement the vector normalization using Python Numba:

  - transfer the data to device

  - declare and invoke the kernel

  - make each thread responsible for a separate part of a vector

  - use the shared memory

References

- Fundamental tutorial on numba:

  https://numba.readthedocs.io/en/stable/cuda/index.html

- Selected pages:

  https://numba.readthedocs.io/en/stable/cuda/kernels.html

  https://numba.readthedocs.io/en/stable/cuda/memory.html