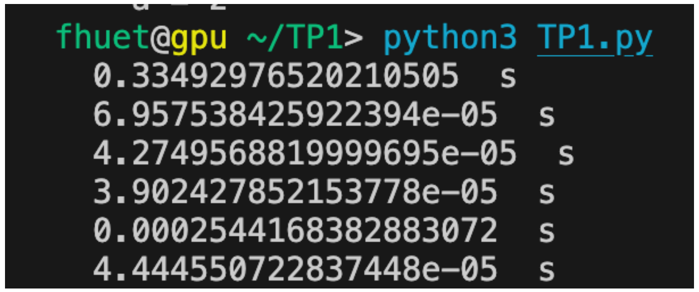
Introduction to GPU Programming - 2

Fabrice Huet

Take-away from Lab 1

- JIT Compiler
  - A kernel is compiled the first time it's run

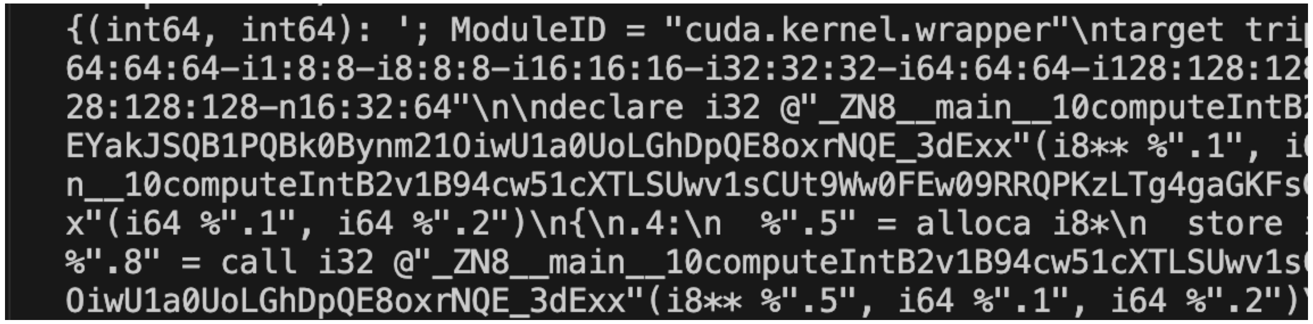


```
fhuet@gpu ~/TP1> python3 TP1.py
  0.33492976520210505  s
  6.957538425922394e-05  s
  4.2749568819999695e-05  s
  3.902427852153778e-05  s
  0.00025441683828830 72  s
  4.444507228374 48e-05  s
```

  - Either perform a dry run or exclude first measure

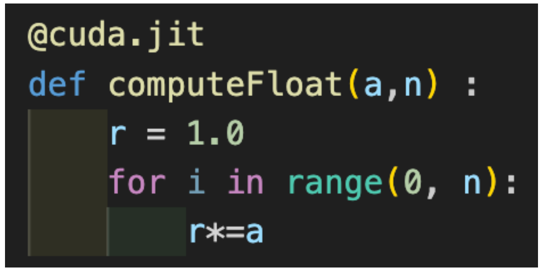
Take-away from Lab 1

- JIT Compiler
  - Typing is inferred by the compiler
  - Can be checked with *kernel_name.inpect_llvm()*

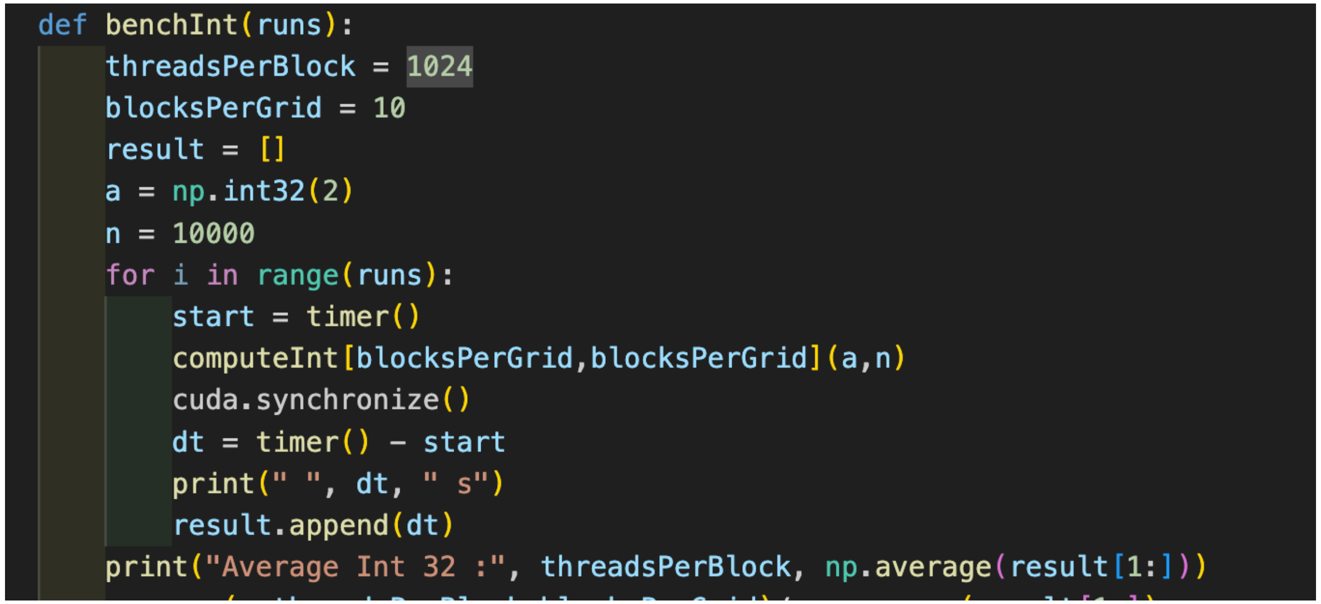


```
{(int64, int64): '; ModuleID = "cuda.kernel.wrapper"\ntarget trip
64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-i128:128:12
28:128:128-n16:32:64"\n\ndeclare i32 @"_ZN8__main__10computeIntB
EYakJSQB1PQBk0Bynm21OiwU1a0UoLGhDpQE8oxrNQE_3dExx"(i8** %".1", i
n__10computeIntB2v1B94cw51cXTLSUwv1sCUt9Ww0FEw09RRQPKzLTg4gaGKFs
x"(i64 %".1", i64 %".2")\n{\n.4:\n  %".5" = alloca i8*\n  store
%".8" = call i32 @"_ZN8__main__10computeIntB2v1B94cw51cXTLSUwv1s
OiwU1a0UoLGhDpQE8oxrNQE_3dExx"(i8** %".5", i64 %".1", i64 %".2")
```

- For benchmarking parameters
  - Write multiple kernels
  - Type parameters explicitly



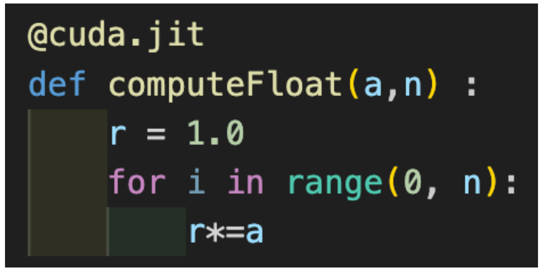
```python
@cuda.jit
def computeFloat(a,n) :
    r = 1.0
    for i in range(0, n):
        r*=a
```



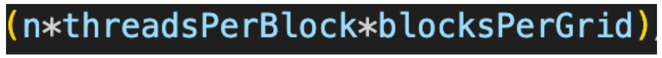
```python
def benchInt(runs):
    threadsPerBlock = 1024
    blocksPerGrid = 10
    result = []
    a = np.int32(2)
    n = 10000
    for i in range(runs):
        start = timer()
        computeInt[blocksPerGrid,blocksPerGrid](a,n)
        cuda.synchronize()
        dt = timer() - start
        print(" ", dt, " s")
        result.append(dt)
    print("Average Int 32 :", threadsPerBlock, np.average(result[1:]))
```

Take-away from Lab 1

- Execution time is not the best metric
  - Need long running kernels to measure execution time
  - Increase threads => increase parallelism
    - So no increase in execution time…
- Prefer other metrics like ops/s or GB/s
  - Compute nb of op



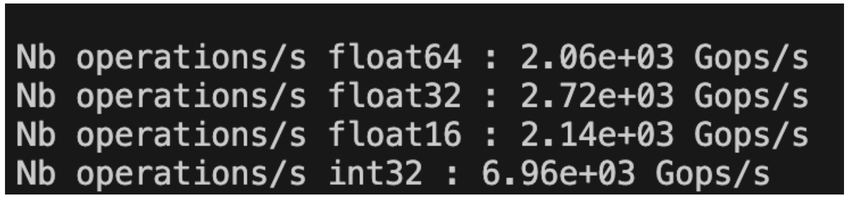
```python
@cuda.jit
def computeFloat(a,n) :
    r = 1.0
    for i in range(0, n):
        r*=a
```



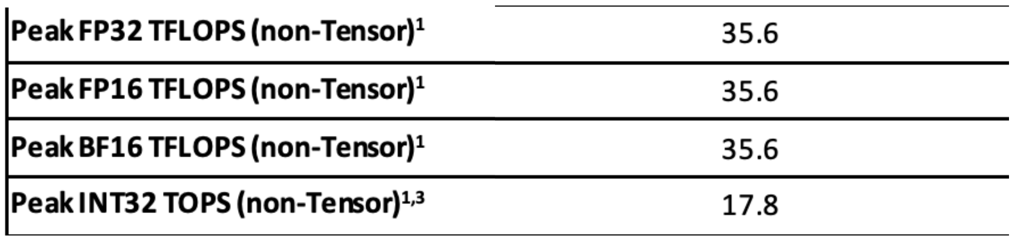
```
(n*threadsPerBlock*blocksPerGrid)
```

Take-away from Lab 1

- Micro-benchmarks are hard!
  - Measurements vary widely from one run to another
  - Impact of other kernels running concurrently
  - Depends on the architecture (here RTX 3090)



```
Nb operations/s float64 : 2.06e+03 Gops/s
Nb operations/s float32 : 2.72e+03 Gops/s
Nb operations/s float16 : 2.14e+03 Gops/s
Nb operations/s int32 : 6.96e+03 Gops/s
```



Peak FP32 TFLOPS (non-Tensor)[1]	35.6
Peak FP16 TFLOPS (non-Tensor)[1]	35.6
Peak BF16 TFLOPS (non-Tensor)[1]	35.6
Peak INT32 TOPS (non-Tensor)[1,3]	17.8

https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

Cuda & Numba

Device function

Kernel and device functions

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

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

- A device funtion is similar
  - A device function can only be called **from** a kernel

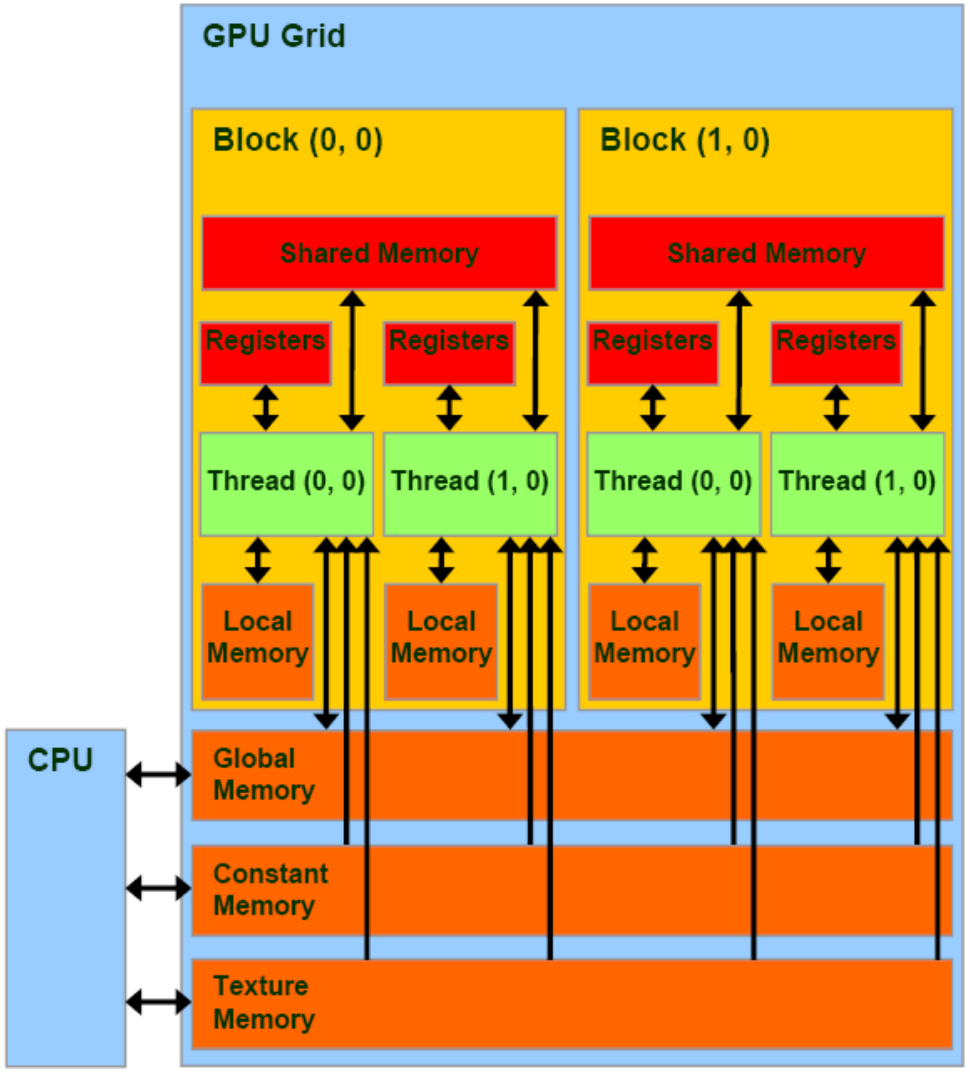
```
@cuda.jit(device=True)
def deviceFunction(tab, index):
```

Cuda & Numba

Memory organization

NUMA

- Non Uniform Memory Access
- Host and device memory are separated
    - Explicit memory transferr
- Hierarchical memory
    - Different size, different access pattern, different performance



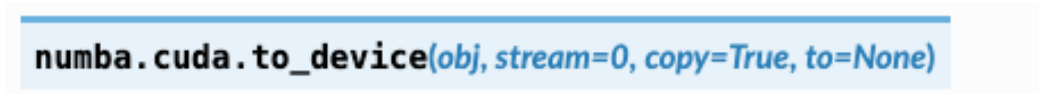
https://www.3dgep.com/cuda-memory-model/

Memory

- Global Memory
  - Located on the device
  - Accessible from host or device
  - Used for transferring data between host and device
  - Limited by the amount of memory on the device
- Shared memory
  - Located on the device, roughly 100x faster than global
  - Only accessible from inside the device
  - Shared memory inside a SM
    - Sharing between threads of the same block
  - Very limited (64KB-100KB)
    - Shared with L1 cache

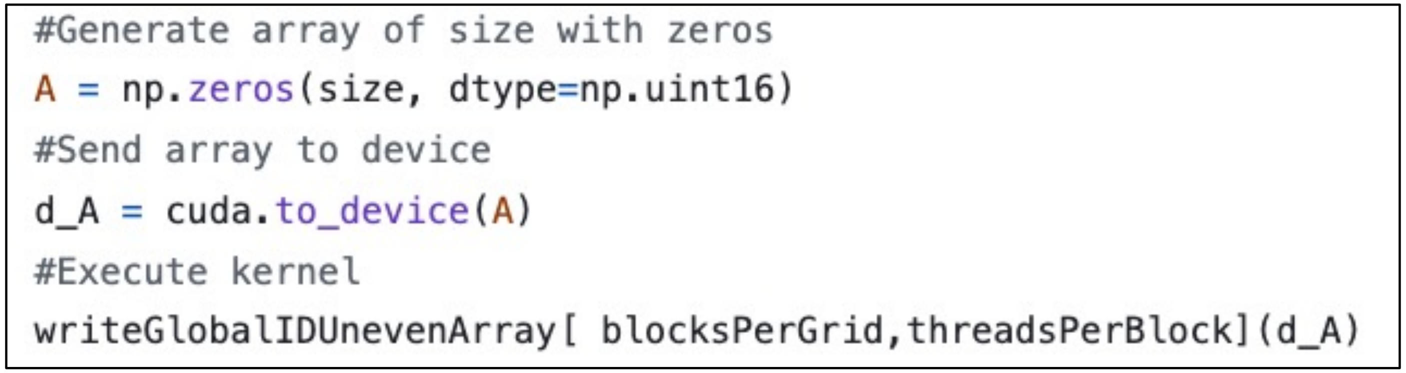
Memory management

- Explicite memory management
  - Close to CUDA/C
  - Based on arrays
- Transfering an ndarray



```
numba.cuda.to_device(obj, stream=0, copy=True, to=None)
```

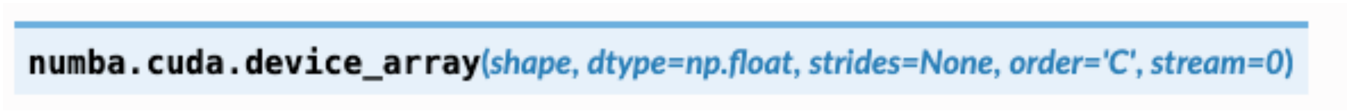
  - Returns a DeviceNDArray
- DeviceNDArray
  - Reference to an ndarray on the device
  - Must be passed as argument to the kernel



```
#Generate array of size with zeros
A = np.zeros(size, dtype=np.uint16)
#Send array to device
d_A = cuda.to_device(A)
#Execute kernel
writeGlobalIDUnevenArray[ blocksPerGrid,threadsPerBlock](d_A)
```

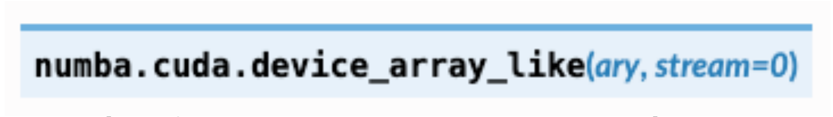
Memory management

- Creating an array directly on the device



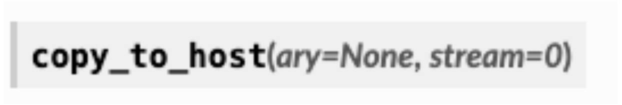
```
numba.cuda.device_array(shape, dtype=np.float, strides=None, order='C', stream=0)
```

- By default use float
  - Type numpy
- Array_like allocation



```
numba.cuda.device_array_like(ary, stream=0)
```

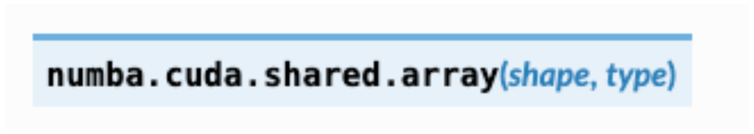
- Copy device to host (bring back the array)



```
copy_to_host(ary=None, stream=0)
```

Memory management – Shared array

- Allocating shared array
  - Fast memory
  - Shared among all threads of the same block

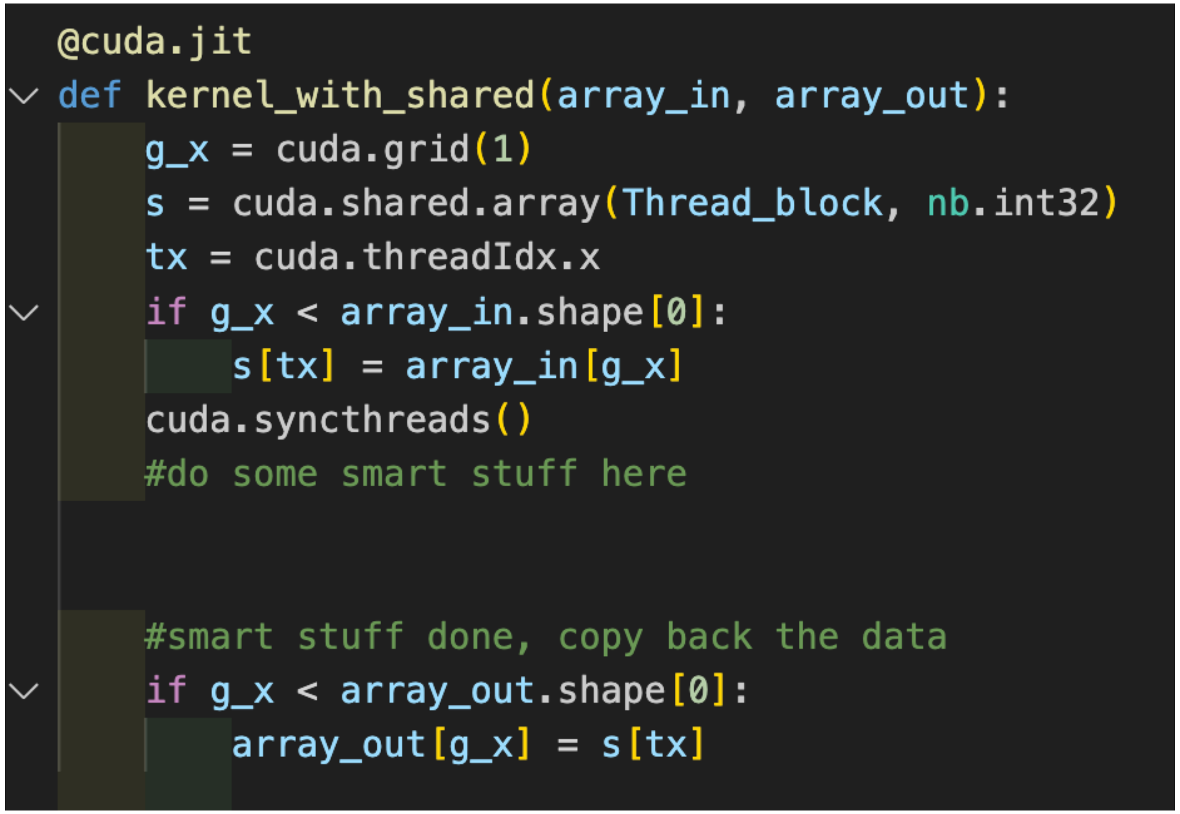


```
numba.cuda.shared.array(shape, type)
```

- Must be inside a kernel or a device funtion
  - Shape must be known at compile time
    - Typically a global variable is used
  - Type is a numba type
    - Close to numpy types

Memory management – Shared array

- Principle
  - Declare the array at beginning of kernel
  - Use threads to copy from global memory to shared array
  - Sync threads
  - Do some smart stuff …
  - Copy back data from shared array to global memory
- Beware
  - Shared memory accessed using local ID
  - Global memory accessed using global ID



```python
@cuda.jit
def kernel_with_shared(array_in, array_out):
    g_x = cuda.grid(1)
    s = cuda.shared.array(Thread_block, nb.int32)
    tx = cuda.threadIdx.x
    if g_x < array_in.shape[0]:
        s[tx] = array_in[g_x]
    cuda.syncthreads()
    #do some smart stuff here


    #smart stuff done, copy back the data
    if g_x < array_out.shape[0]:
        array_out[g_x] = s[tx]
```

Cuda & Numba

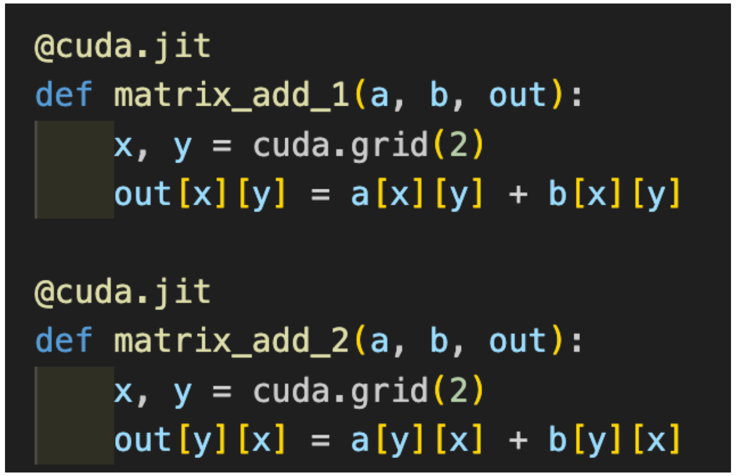
Optimizing memory access

Memory Coalescing

- CUDA tries to optimize access to global memory
  - Groups memory access from the same warp (32 threads)
  - Very efficient if consecutive memory addresses
- Ideally in a warp
  - Thread id 0 accesses memory N
  - Tread id 1 accesses memory N+1
  - ...
- Allows for DRAM burst (good)

Example Matrix addition

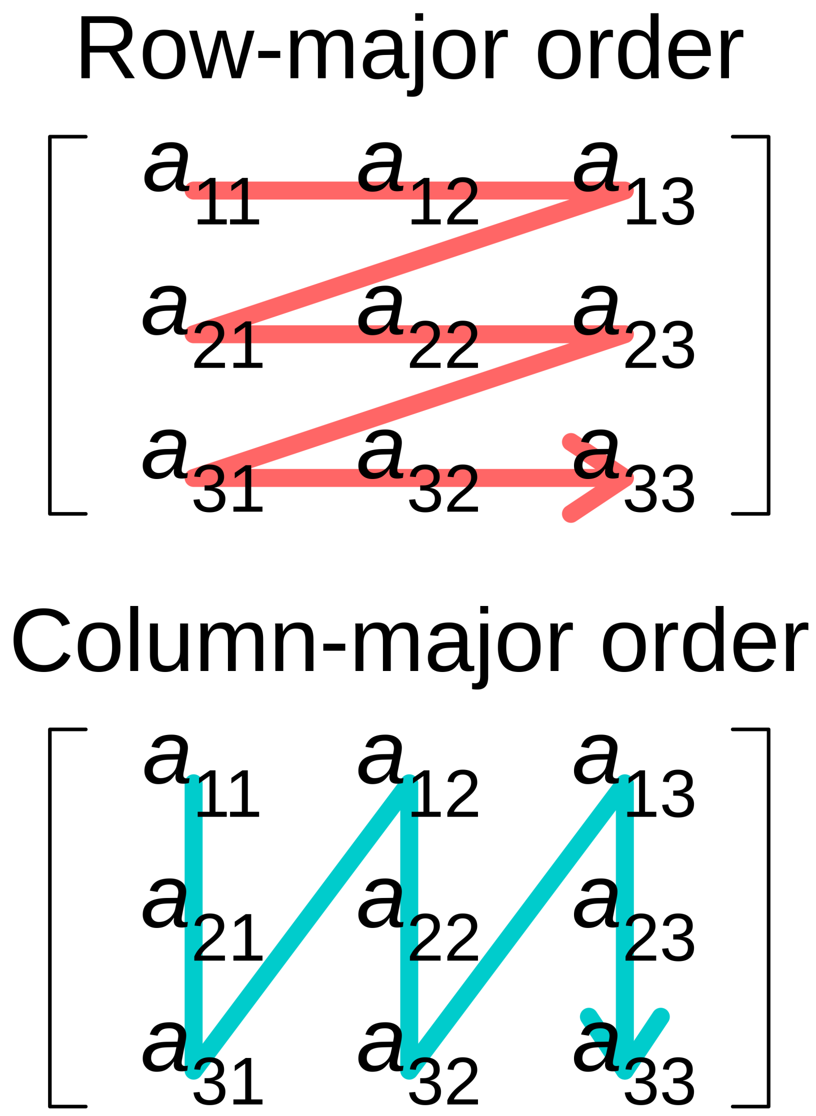
- We want to have a kernel to perform
  - D = A +B
- Straightforward ....



```
@cuda.jit
def matrix_add_1(a, b, out):
    x, y = cuda.grid(2)
    out[x][y] = a[x][y] + b[x][y]

@cuda.jit
def matrix_add_2(a, b, out):
    x, y = cuda.grid(2)
    out[y][x] = a[y][x] + b[y][x]
```

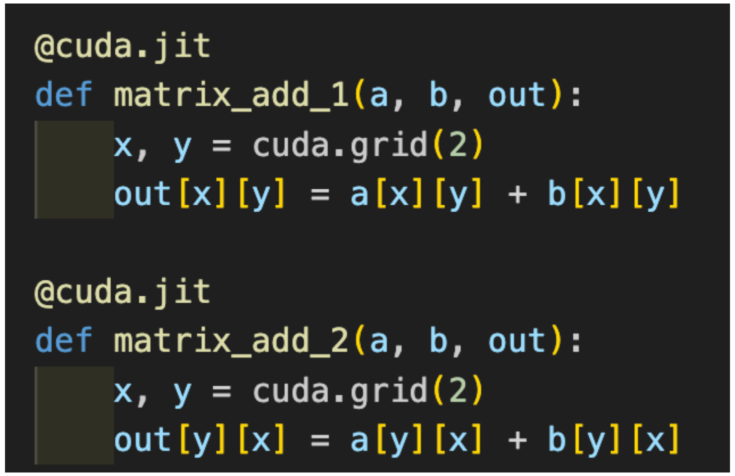
- Is there one better ?
  - Depends on how matrix are organized in memory


Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

https://en.wikipedia.org/wiki/Row-_and_column-major_order

Example Matrix addition

- Numba relies on Numpy arrays
  - Numpy uses default C order
  - Row Major (https://numpy.org/doc/1.14/glossary.html)
- What about threads ?
  - Threads in a warp are linearized on Row-major also (x, y, z)
- Now let’s go back to the kernels

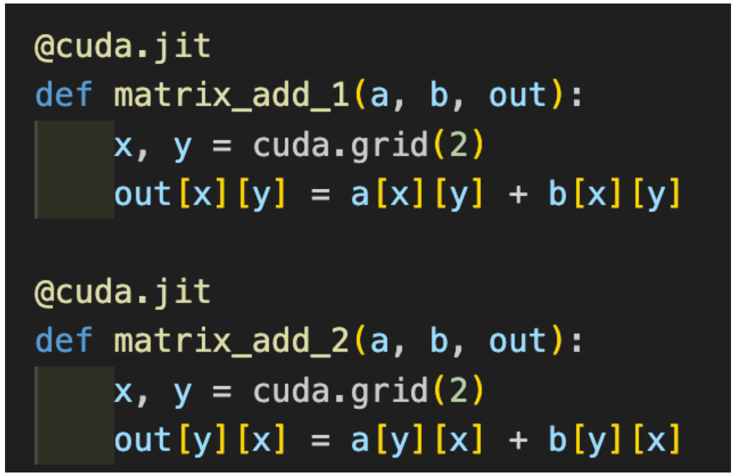


```python
@cuda.jit
def matrix_add_1(a, b, out):
    x, y = cuda.grid(2)
    out[x][y] = a[x][y] + b[x][y]

@cuda.jit
def matrix_add_2(a, b, out):
    x, y = cuda.grid(2)
    out[y][x] = a[y][x] + b[y][x]
```

Example Matrix addition

- Assume square matrices of size > 32
  - 32 threads will run at once
  - At first all 32 threads will have x ==0 and y variable
  - a[x][y] accesses row x and column y
- Matrix_add_1
  - Consecutive threads will access different row
  - No coalescing
- Matrix_add_2
  - Consecutive threads access different columns
    - Same row
  - Colaescing!
- In practice Matrix_add_2 is 2x faster



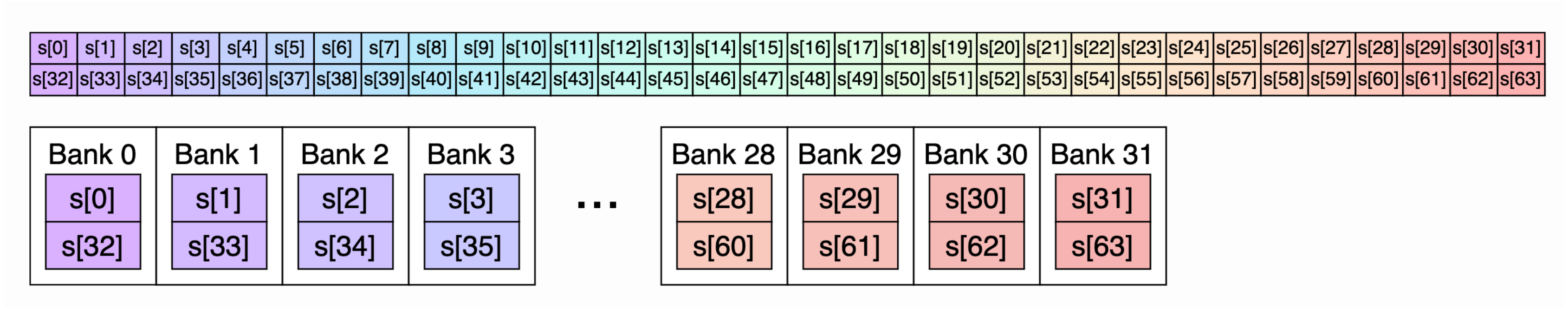
```python
@cuda.jit
def matrix_add_1(a, b, out):
    x, y = cuda.grid(2)
    out[x][y] = a[x][y] + b[x][y]

@cuda.jit
def matrix_add_2(a, b, out):
    x, y = cuda.grid(2)
    out[y][x] = a[y][x] + b[y][x]
```

What about shared memory ?

- Shared memory is organized into 32 banks
  - 1 load/bank/clock cycle
  - Successive 32-bit words are stored in successive banks
- Banks can be accessed in parallel
  - So threads should avoid accessing the same shared memory address
- Otherwise bank conflict

```
__shared__ float s[64];
```


s[0] s[1] s[2] s[3] s[4] s[5] s[6] s[7] s[8] s[9] s[10] s[11] s[12] s[13] s[14] s[15] s[16] s[17] s[18] s[19] s[20] s[21] s[22] s[23] s[24] s[25] s[26] s[27] s[28] s[29] s[30] s[31]
s[32] s[33] s[34] s[35] s[36] s[37] s[38] s[39] s[40] s[41] s[42] s[43] s[44] s[45] s[46] s[47] s[48] s[49] s[50] s[51] s[52] s[53] s[54] s[55] s[56] s[57] s[58] s[59] s[60] s[61] s[62] s[63]
Bank 0
Bank 1
Bank 2
Bank 3
...
Bank 28
Bank 29
Bank 30
Bank 31

https://feldmann.nyc/blog/smem-microbenchmarks

Cheat sheets

Don’t panic

Before writing some code

- Identify which part will be executed on the GPU
  - This is your kernel
- Identify the data needed by the kernel
  - Size and shape
- Assume a “safe” thread block size to start
  - 32 usually good
- Deduce your grid size
  - Typically data size/thread block

Writing some code

- Initialize/create your data on the host
- Transfer data to device
  - *cuda_to_device(…)*
- Call the kernel with the correct grid and thread block sizes
- Transfer data back to the host
  - *copy_to_host(…)*

Writing a kernel

- You have to think like a thread
  - As a thread, what should you do ?
- Compute your global ID
  - Usually you need it
- Always check you don’t read/write outside of your data
  - More on this during lab sessions
- Usually a for loop in an algorithm can be removed
  - Each index of the loop is handled by a thread
  - Data parallelism