10/08/2024

Introduction to cuda python with Numba

⇒ Objective - ~~perform~~ GPU accelerate python code that performs element-wise operations on Numpy arrays.

⇒ Numba: Numba is just-in-time, type-specializing, function compiler for accelerating numerically-focused Python either for CPU or GPU.

function-compiler. - Compiles python function to a faster function → compiles a machine code implementation of the function.

type-specializing : Speeds up function by generating a specialized implementation of ~~data type~~ specific data types you are using.

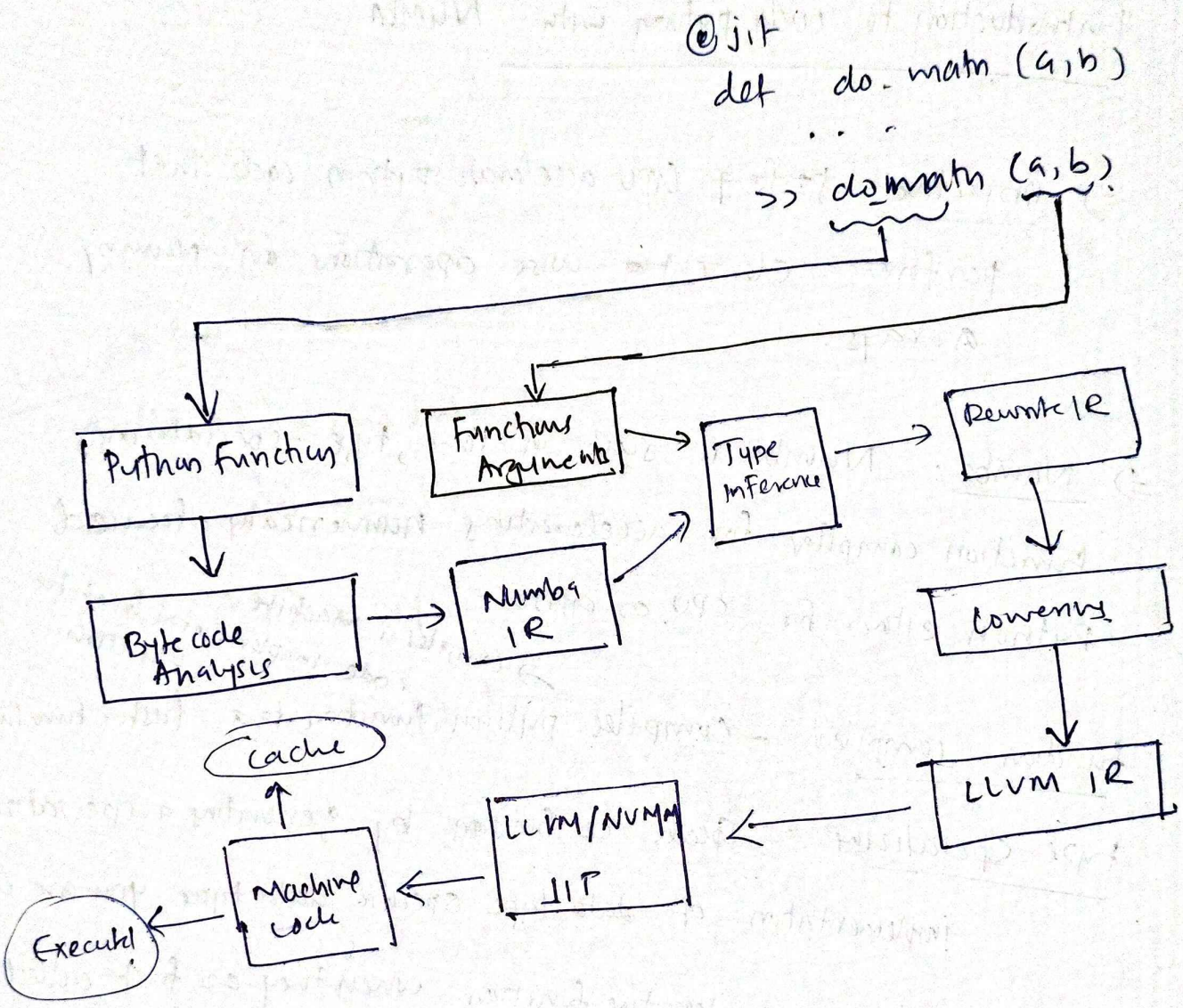
Just-in-time: compiles the function when they are first called.

numerically-focused: Focused on numerical data types like int, float & complex.

Note

- Python built-in functions are faster than "Numba" because 'numba' introduces some overhead to each function call that is larger than the ~~overhead of~~ function calls overhead of python itself.

JIT compilation works



Byte code Analysis: Byte code is a lower-level, platform-independent representation of your python code.

Numba IR : Intermediate representation of python code by Numba. (Simplified low-level representation easier to analyze and optimize)

Type-Inference: Finding the type of the function arguments. Numba needs to know the data types to generate machine code.

Rewrite IR ⇒ Numba IR is rewritten after Type inference.

Lowering: ⇒ Numba IR is "lowered" to even a lower-level representation that is closer to machine code, known as LLVM IR.

LLVM IR ⇒ Low-level Virtual Machine is a powerful compiler framework that Numba uses to generate highly optimized machine code

LLVM/NVVM JIT ⇒ Compiler translates LLVM IR into machine code that can run on the CPU (using LLVM) or GPU (using NVVM).

Machine code: machine code is specific to the architecture of the machine where code is being run.
The machine code is highly optimized for performance.

↳ If function is again called with same argument type, Numba can use cache machine code thing cache, avoiding need to go through the entire compilation process again.

note

* After @jit compilation the annotated source code could be accessed via "inspect_types()"

* Nopython and object mode

* @jit by default works in object mode, which does not do type-specialization.

* @jit(nopython = True) is called the nopython mode, that do type-specialization and tells whether type inference step fails or not.

note:

- Numba cannot compile every python data types.

  for eg:- Numba cannot compile, dictionaries

- @jit(nopython = True) == @njit

Numpy Ufuncs

⇒ Numpy ufuncs (or Universal functions) is a Numpy function that operates element-wise on arrays supporting array broadcasting, type casting & ~~several~~ other standard features. Such as vectorization and reduction operations.

eg:- ⇒ Arithmetic ufuncs ⇒ np.add, np.substract
np.multiply ... etc.

Trigonometric ufuncs ⇒ np.sin, np.cos, np.tan etc.

exponential and logarithmic ufuncs ⇒ np.exp,
np.log etc.

Comparison ~~oper~~ ufuncs ⇒ np.greater, np.less etc.

Bitwise operation ufuncs ⇒ np.bitwise_and
np.bitwise_or etc.

⇒ One can also create custom ufuncs using ~~"numpy.frompyfunc"~~ "numpy.frompyfunc" function or by using the @ vectorize. decorator.

⇒ numpy ufuncs are ideal to be carried out in GPU

- Since numpy ufuncs carry out same operation on different elements of the array, they are data parallel.
- GPU hardware is inherently data parallel. i.e, maximum throughput is achieved when ~~the~~ the GPU is computing the same operation on many different elements at once.

=) Using Numba also one can create ufunc that can be run on GPU

- this is by using the decorator @ vectorize (numba vectorize works in same manner as numpy ~~def~~ vectorize)

- For running it on GPU (CUDA enabled devices) we need to give an explicit type signature and target attribute.

~~a type signature is of this form:~~

- a type signature and traget is of this form:

@ vectorize ('return_value_type(argument1_value_type, argument2_value_type, ...)', target = 'cuda').

⇒ GPUs have advantage over CPUs on certain cases:

- Inputs have to very large.
- Advanced calculation. - Usuage of GPU requires execution overhead which makes execution of simple function in GPU costly compared to CPU.

CUDA Device functions

⇒ "Ufuncs" is fantastic when you want to perform elementwise operations and can be ~~accelerated~~ GPU accelerated easily using numba.

→ To compile functions that are not element wise, vectorized functions, we use "numba.cuda.jit"

↳ "numba.cuda.jit" can also be used in ~~helper functions~~ compile helper functions that ~~can be used inside~~ is used inside a GPU-accelerated "ufunc".

Syntax:

```
from cuda import jit

@cuda.jit (device = True)
~~def ufunc_to_accelerate (rho, theta):~~
def _helper_func( ):
    :
    return ( )

@vectorize ([ 'float32 ( . . . . )' ], target = 'cuda')
def _ufunc: ( ):
    :
    return ( )
```

Note

* The argument "device = True" ensures that the decorated function (decorated by cuda.jit) could be only used inside a function that is running on a GPU. (~~ufun~~ GPU accelerated Ufunc) and not on CPU hosted code.

* During helper function calls, there is very little overhead since, CUDA compiler aggressively inlines device functions.

* Allowed Python on the GPU:

1. if | elif | else
2. while and for loops
3. Basic math operations
4. selected functions from the math and cmath modules
5. Tuples.

Managing GPU Memory

⇒ Numba transfers the data to GPU when it undergo operation from host (CPU)

⇒ After the operation the data gets automatically transfered back to the host / CPU

⇒ Data is transfered to and from GPU while performing some computation and it is handled by Numba.

⇒ But data transfer is a time-intensive operation

⇒ But we can prevent the automatic data transfer from GPU to host and store it GPU for additional process and copy it back to the host only after end of process in GPU.

⇒ The way to do it is to create "CUDA Device Arrays", and pass them to our GPU functions.

↳ Device arrays will not be automatically transfered back to the host after processing, and can be reused as we wish on the device before ultimately, and only if necessary, sending them, or parts of them, back to host.

syntax

x_device = cuda.to_device(x)

empty cuda device array: out_device = cuda.device_array(shape = (n,), dtype = float32)

out_host = out_device.copy_to_host()
↳ to copy the data back to the host //.

Custom CUDA kernels in python with Numba

⇒ custom CUDA kernels require more work to be implemented on GPU.
↳ we cannot use the decorator @vectorize as in the case of Ufunc

- Ufuncs are limited to ~~appli~~ scalar case of scalar ~~function~~ operation that is operated on element wise data.
- There are more class of problems which cannot be solved by applying the same function to each element on dataset.

examples of such cases: stencil algorithms or reduction algorithms

- These algorithms are inherently parallelizable but cannot be done using ufunc

CUDA kernels

* Function written for GPU (while programming in CUDA) is called "kernel"
* Execution of Function ⇔ Launching a kernel.
* Kernels are launched on the GPU's many cores using "parallel" threads. ⇒ "Execution configuration" or "launch configuration" is used to describe parallel execution configuration.

GPU - accelerated v/s CPU - only application

* In CPU - only application data is allocated on the CPU and all the work is performed serially.
* In GPU system there is both host and device memory. Data initialized on CPU can be copied to GPU where it can be worked on in parallel.
* Work on GPU is asynchronous to host (CPU), thus it can happen in same time.
  - to synchronize GPU and CPU one could use: cuda.synchronize()
  - data then can be copied back to the CPU for verification.

Execution in GPU

- GPU do work in parallel and work is done in "thread" and many threads run in parallel.
- A collection of thread is called a block
- A collection of blocks with a grid kernel launch is called a grid.
- To launch a kernel we require a execution configuration which defines the number of blocks in a grid and number of threads in each block.
- Every block in the grid contains the same number of threads.

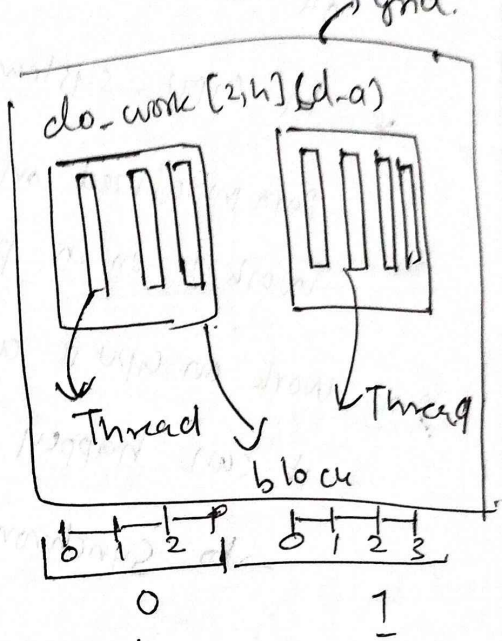
-) Inside ~~CUDA~~ kernel definitions, CUDA - provided variables describes its executing thread, block and grid.

↳ gridDim.x - number of blocks in the grid.

↳ blockIdx.x → Index of the current block

↳ blockDim.x ⇒ number of threads in each block (all blocks will have same number of threads).

↳ threadIdx.x ⇒ describes the index of the thread within a block.



⇒ Accessing individual threads (for a parallel mapping of data)

" threadIdx.x + blockIdx.x * blockDim.x "

will returns the thread's unique index in the whole grid which can map to data elements.

↳ ~~cuda~~ provide a unique way to access individual thread using "cudaGetThreadIndex()"

It gives unique indices for each thread.

⇒ 'i' implies one-dimensional grid of threads.

⇒ cuda allows to construct 1, 2, and 3D grids.

⇒ During launching if kernel we pass execution configuration that contains information about the number of threads and blocks for the problem.

⇒ To accelerate custom functions using numba in GPU we use "cuda.jit" or @cuda.jit

⇒ note: while using functions do not returns an output value we pass the it will be written to an output array.

note: Taking copies of arrays from device to host memory implies implicit synchronization and thus manual synchronization is not required in such cases

⇒ Deciding the very best size for the CUDA thread grid is a complex problem and depends upon both the algorithm and the specific GPUs compute capability.

Rough Heuristics that we tend to follow are these:

(1) Size of a block should be a multiple of 32 thread (since each streaming multiprocessors in GPUs are made up of 32 warps")

with typical block sizes btw 128 to 512 threads per block.

- Size of grid should ensure the full GPU is utilized where possible. Some thing in range b/w 200-100' blocks is a good starting place.

- CUDA launch overhead increases with block size.

  - So when the input size is very large it is best not to launch a grid where number of threads equal to the number of input elements.

    ↳ this would increase number of blocks In such cases approaches like Grid stride looping is performed.

Grid stride looping

⇒ Used when the number of threads is less than the data elements to be processed.

⇒ Grid striding looping is same as a ~~custom stepsize~~ setting ~~step~~ stepsize while using for loop in python:

Syntax:

For loop with step size:

for i in range (start, stop, stepsize)

Grid stride looping

for i in range (start, data.shape[0], stride).

⇒ "start" is given by: cuda.grid(1)

stride is given by: cuda.gridsize(1)
↳ number of threads in the grid

cuda.gridsize(1) = blockDim.x + gridDim.x

⇒ Without Grid stride looping all data elements will not get processed by threads and some data elements will be left unprocessed.

⇒ ~~with Grid stride looping~~, the device coalesces memory reads/writes into as few transactions as possible for performance

⇒ This is Grid stride looping:

Each thread processes multiple elements by iterating over them in steps that are equal to the total number of threads (grid size). This allows the entire dataset to be processed even when the grid size is smaller than the total number of elements.

eg:- Assume you have 1024 threads.

~~Grid stride with~~ Grid stride loop with stride = 1024.

1st thread access elements 0, 1024, 2048

2nd thread access elements 1, 1025, 2049

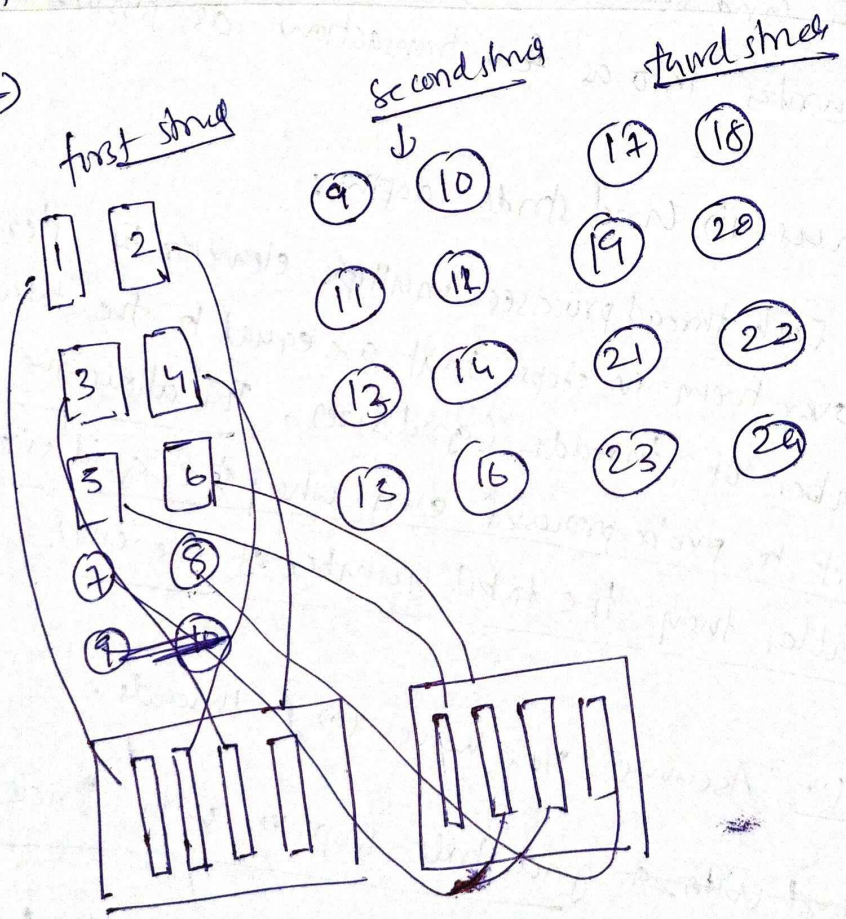
goes like this

Memory coalescing

⇒ Memory coalescing occurs when threads in a warp (sub system of streaming Multiprocessors containing 32 threads) access memory in a way that these access can be combined into a single or a few memory transactions.

⇒ Unde stride loaping facilitate memory coalescing and allows read/writes with few transactions.

⇒



⇒ Cond stride loaping ensures that consecutive threads in a warps access consecutive memory locations even when they are looping over multiple elements

NVIDIA - TASK 3.

Effective use of Memory subsystem

Global Memory Coalescing

Contiguous Memory Location

Refers to a sequence of memory addresses that are next to each other without any gaps.

eg:- array = {10, 20, 30, 40};

these integers in the array might be stored in contiguous memory locations like this:

Value 10 stored at address 0x1000
value 20 stored at address 0x1004
Value 30 stored at address 0x1008
Value 40 stored at address 0x100C

⇒ Contiguous memory allows for efficient data access and transfer.

If a program reads data from contiguous memory, it can retrieve a large block of data in one operation, rather than multiple smaller operations.

- Memory Bus is the connection b/w CPU and RAM (Main memory). Now, data is transferred in blocks across the memory bus.
- Modern computers are also equipped with cache lines (a type of small, fast memory close to the CPU) that stores data temporarily to speed up access.

↳ A cache block is a type of contiguous memory, of 64 bytes (or more) in size.

⇒ While Reading contiguous memory, the if the data is stored in contiguous locations, the CPU can load the data ~~in one~~ to cache in one operation.

↳ For instance, if the CPU needs the first element of an array, it might load the entire cache line (which could include several elements of the array) into cache. thus subsequent access to other elements of the array can be satisfied directly from the cache., without needing additional trips to the slower main memory.

Memory coalescing in GPU

⇒ In GPU computing threads are organized in Blocks and Blocks are further divided in "warps" which ~~consist~~ consist of 32 threads.

⇒ Instructions are issued parallely at warp level., i.e, each thread in a warp will ~~carry~~ carry out "same instruction of different data elements."

⇒ Memory transfer b/w global device memory and GPU occur in segments, typically 32 byte segments

↳ For data stored in L1 cache ~~it~~ ~~will~~ data transfers occure in 128 byte segments → Cache memory closest to GPU/CPU

⇒ The objective of Memory coalescing is to reduce the number of memory lines (segments) needed to fulfill the read/write requirements of a warp.

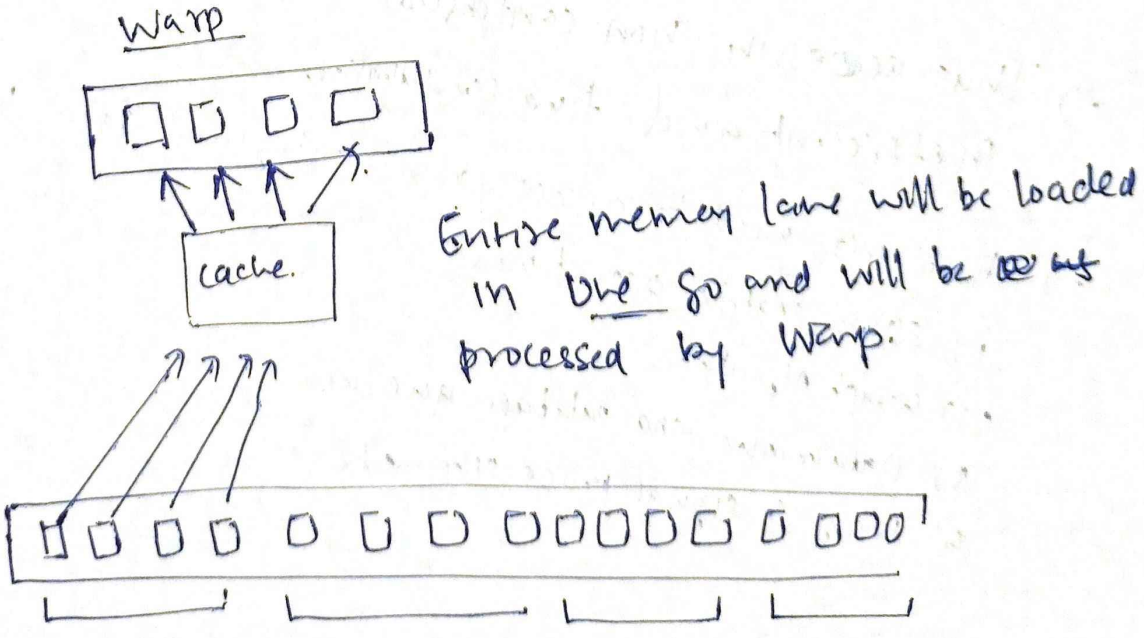
↳ memory coalescing done properly leads to efficient memory transfer.

Contiguous Memory Access

⇒ If the data requested by the warps are stored at contiguous memory location, the memory access is considered to be fully coalesced.

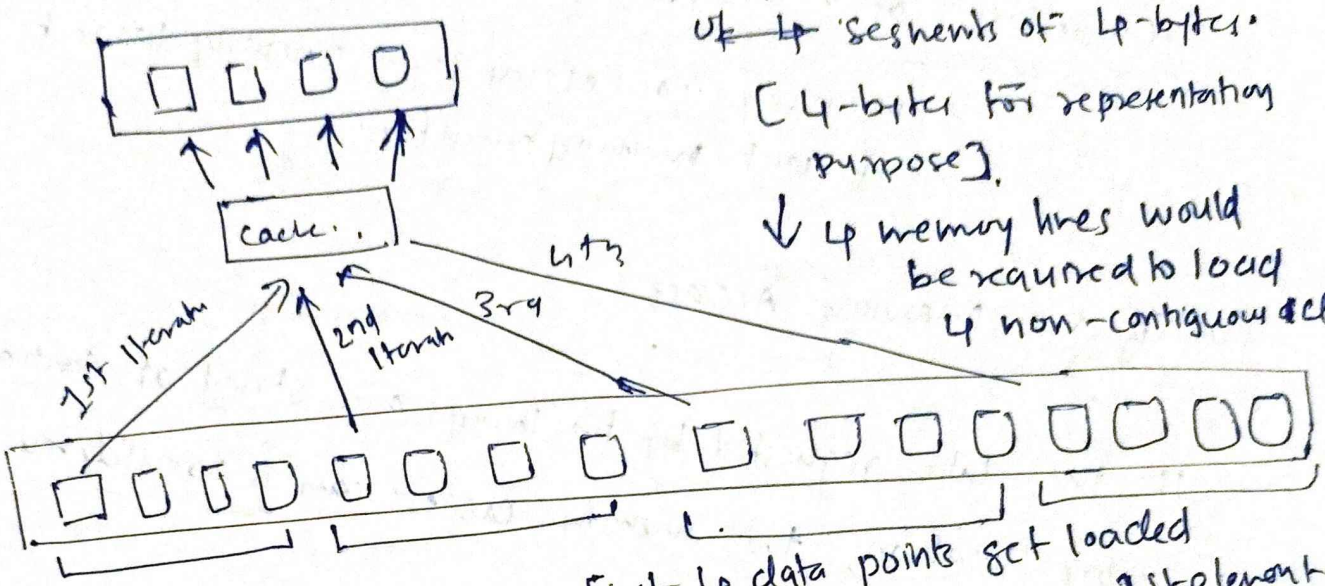
↳ i.e, all the data in the memory line will be used and data transfer occurs in few operation.

accessing contiguous locations.



non-contiguous memory

Memory transfer occurs in ~~4 bytes~~ 4 segments of 4-bytes.
[4-bytes for representation purpose].
↓ 4 memory lines would be required to load 4 non-contiguous data



⇒ During 1st transfer ⇒ First 4 data points get loaded but GPU or warp only process 1st element and remaining 3 are discarded

During 2nd transfer ⇒ second 4 data points get loaded and warp only process the 1st element and remaining 3 are discarded.

⋮

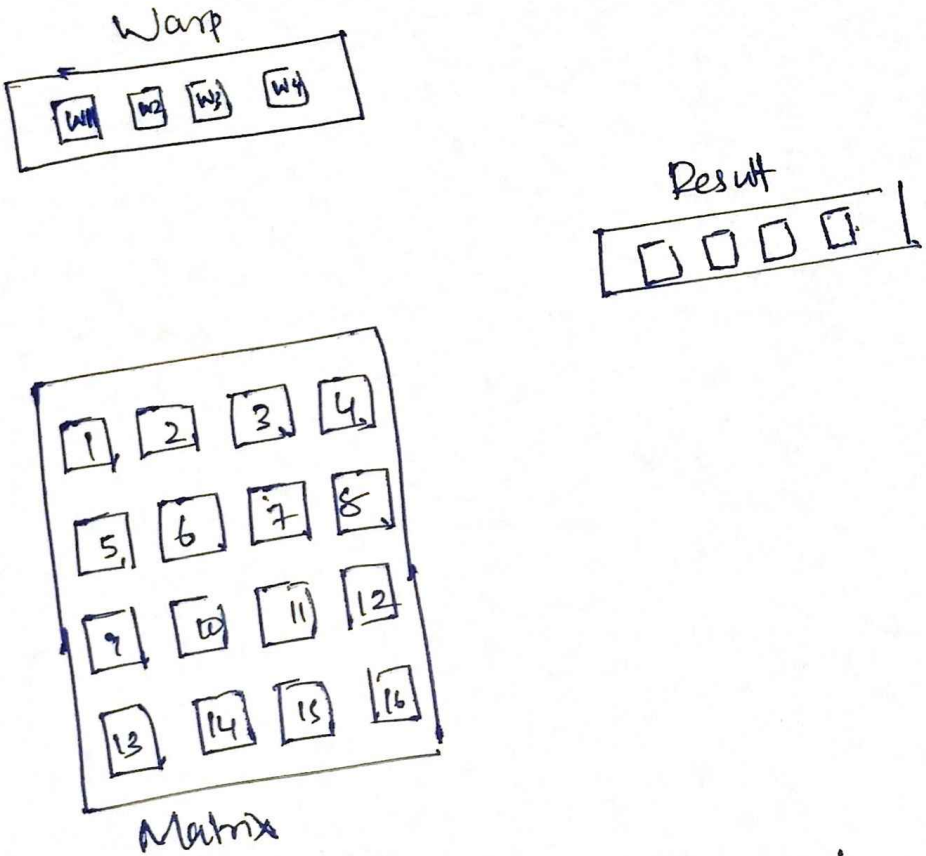
It goes like this

⇒ Thus accessing non-contiguous memory location is inefficient and time consuming.

⇒ But we need to access non-contiguous data in cases like:
(1) Sparse Matrix operations
(2) Graph Algorithms
(3) Data mining and database queries.
(4) Image & signal processing etc.

Memory Coalescing while working with Matrix

Example (1): Consider a kernel that stores the sum of each row in a matrix (which is depicted as 4 contiguous data elements) in a result vector.



⇒ In this case there are two options available.

Option - I: A single thread can iterate over a row summing it and storing the result into a vector.

~~Option - 2 : Parallelising the operation ⇒ i.e, W1 access~~

⇒ 1st row of matrix will be transferred to CPU. and W1 access the elements iteratively.

Illy, 2nd row will be transferred to GPU W2 will sums the rows elements iteratively:

This process is repeated:

option (2): Accessing Row elements parallely by a thread in warp

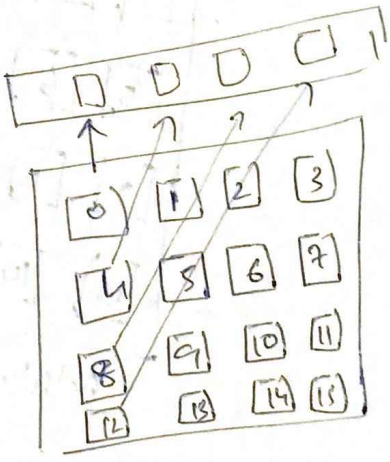
i.e,

(1) W[1] loads first row to GPU access first element and discard all other element

(2) W[1] loads second row to GPU access first element and discard a

option (2): Parallelising the process of summation. Each thread in warp is requesting for data in different line of memory

- W[1] load first row access only first element and discard all other element
- W[2] load second row access only first element and discard all other ele



⇒ In this process, in Each Iteration 4 data lines will be transferred loaded to GPU gpu but 75% data is discarded

⇒ Now this process is repeated to access all other elements of row in subsequent iteration

⇒ It also follows uncoalesced pattern. Thus it will load the during summation, there would be 16 memory line transfers in total and each use only 25% of data for each line transferred.

Example (2) : If each thread sums a column, the memory access pattern can be coalesced

Summing each column and storing the resultant into a vector.

Option (1) : Iterating over elements of column one by one.

w[1] loads first row access first element
w[2] loads second row access first element.
w[3] loads third row access first element.
w[1] loads fourth row access first element.

This process is repeated for all, by all threads. This process involves transfer of 16 memory lines and at each transfer only 25% of data transferred is used.

Option (2) : Iteration parallel execution:

First row is loaded to GPU and w[1] access first element,
now, w[2] → 2nd element
w[3] → 3rd element
w[4] → 4th element

This process is repeated for other rows.

⇒ In this way memory access will be coalesced.

⇒ The computation requires only transfer of 4 memory lines and 100% of data transferred is used in each iteration.

Conclusion

~~⇒ While performing row operations~~

⇒ If we assume the row elements forms a contiguous memory operations like summing ~~of~~ elements in row is best to be performed iteratively rather than parallely (in terms of memory coalescing) in GPU. ~~For operations~~

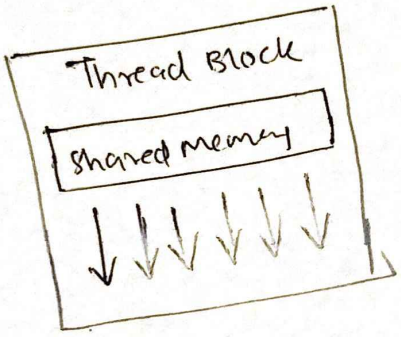
⇒ Operations like summing elements in a column is best to be performed parallely.

⇒ Note: ~~To Me~~ The way to perform memory coalescing depends upon the task in hand.

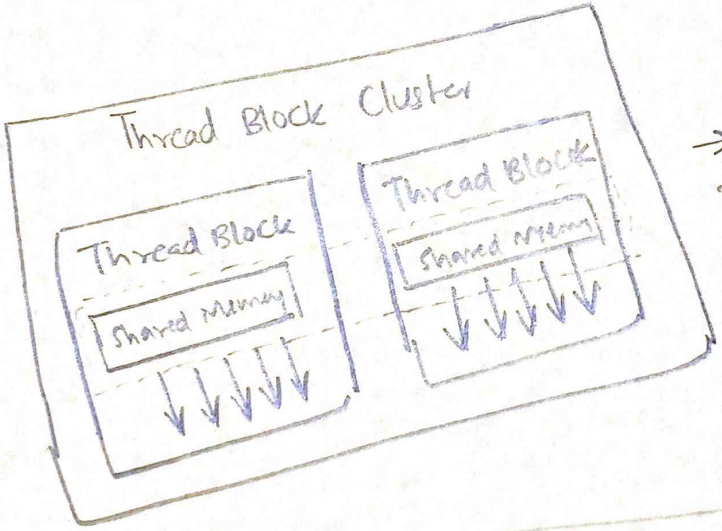
⇒ for more info: check the code work.

Memory Hierarchy of CUDA

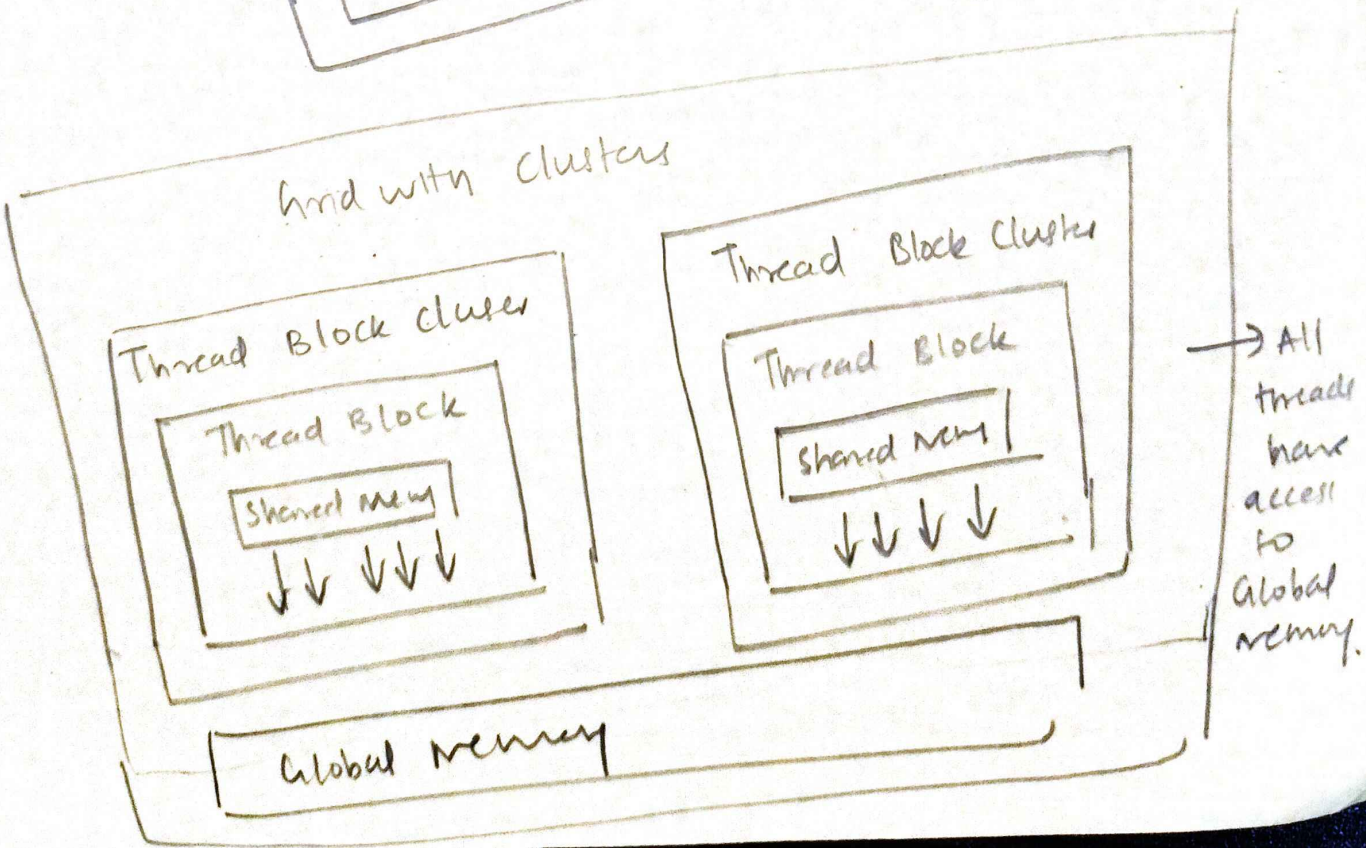
→ Each thread ~~have~~ has its private local memory.



→ Each thread block has a shared memory visible to all threads of the block with the same lifetime as the block



→ Thread blocks in a Thread blocks cluster can perform, read, write and atomics operations on each other's shared memory.



→ All threads have access to Global memory.

Further, there are also two-additional read-only memory spaces accessible by all threads: the constant & texture memory spaces

↳ The global, constant & texture memory spaces are optimized for different memory usages.

Global Memory V/s Shared Memory

* Global memory is accessible to any thread or block on the device and can persist for the lifetime of the application, and is relatively large memory space.

* Shared memory → A user defined memory which has very wider bandwidth than global memory that is shared between all threads in a block.
  - ↳ It does not persist after a kernel finishes executing.
  - ↳ It can be considered as a programmable cache.
  - ↳ higher bandwidth implies that shared memory allows data to be transfered to and from the processing cores at much faster rate than Global memory.

* How to use shared memory

(1) caching memory: Read from Global memory

↳ threads within a block may need to access same data multiple times during execution. If the data is stored in global memory, each access is slow and costly in terms of latency.

(2) Buffering output from threads for coalesced writes to Global memory

⇒ When the thread write results to global memory, writing one value at a time will be inefficient. Thus the results can be staged in the shared memory or in other words it could be used as temporary buffers and they be written back to the global memory in coalesced manner.

(3) staging data for scatter/data operations within a Block

⇒ In some algorithms, different thread need to read/write data to non-contiguous regions of memory.

⇒ direct access of global memory will be inefficient.

⇒ Hence also, we can use shared memory as a staging area for loading or storing data efficiently.

* setting up shared Memory

⇒ As said earlier, shared memory is user defined and persist on till the execution of kernel is completed.

⇒ Numba provides functionalities to define shared memory and also synchronize threads automatically which is often required while reading/writing to a shared memory.

⇒ While declaring a shared memory ⇒ we need to specify

(1) the size of shared array

(2) type of data in the array (using numba type)

Note: while declaring the size/shape of array, we only can pass Constant value as argument. Thus one cannot pass variables of a function and other variables provided by

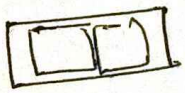
numba (eg:- ~~cuda~~ numba.cuda.blockDim.x), or the calculated ...

of ~~(eg:- cu~~ cuda.griddim.

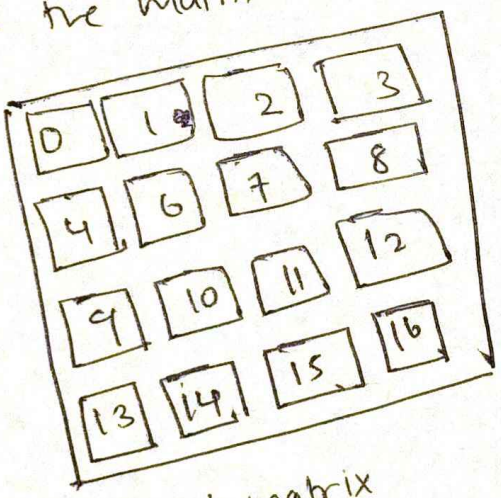
Using shared Memory for Memory Coalescing

⇒ ~~(use)~~ Problem with naive approach of not using shared memory

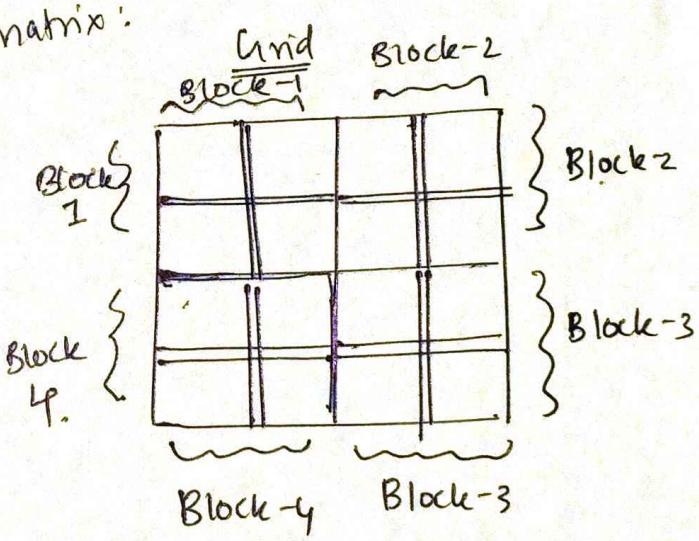
Let, the warpe size be threads of 2.



Let the matrix be a 4×4 matrix:



Input matrix

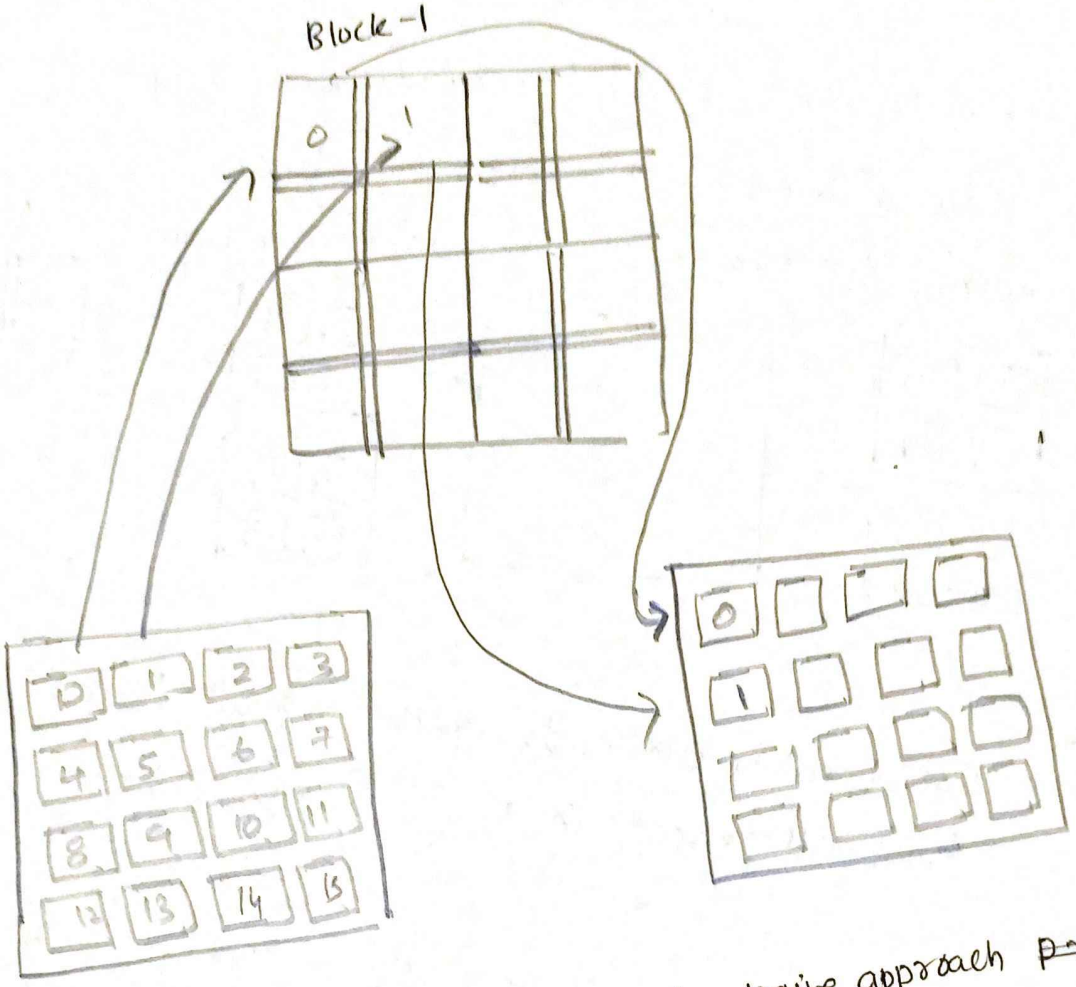


* ~~Assume the task~~ Let task be taking transpose of a matrix

- naive approach is to make each thread read a element in the matrix

⇒ Then write the element in the output ~~matrix~~ matrix in the transposed location.

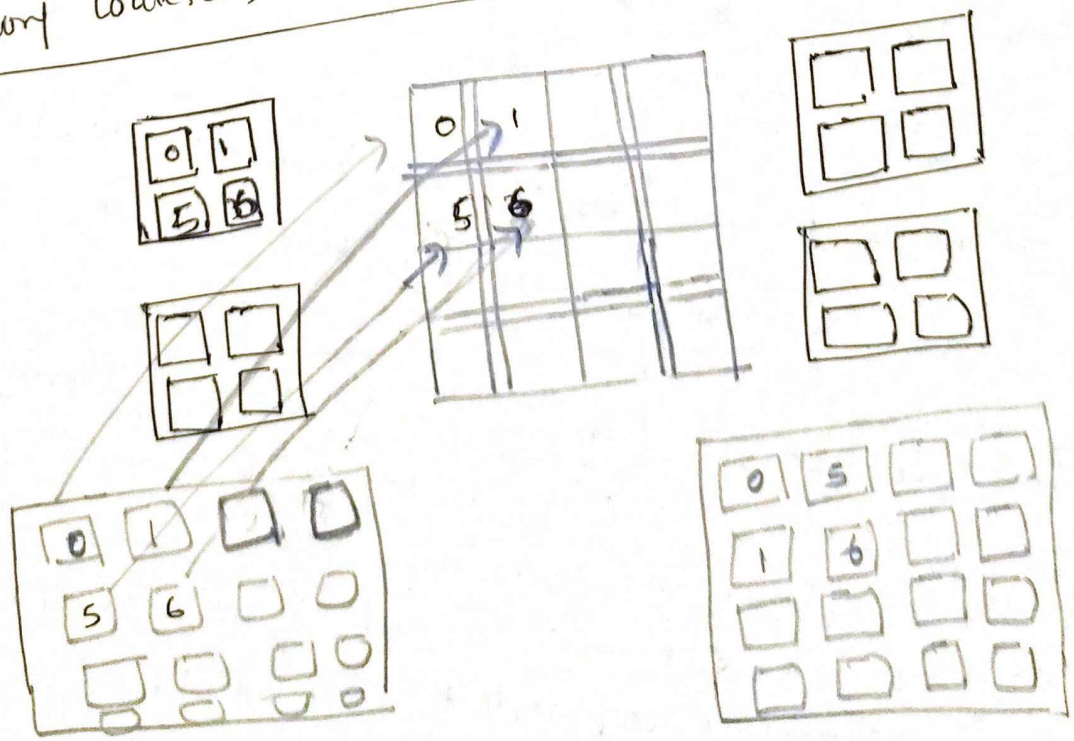
- This scheme makes read in coalesced fashion and writing in uncoalesced fashion.
- ~~Note: Accessing~~



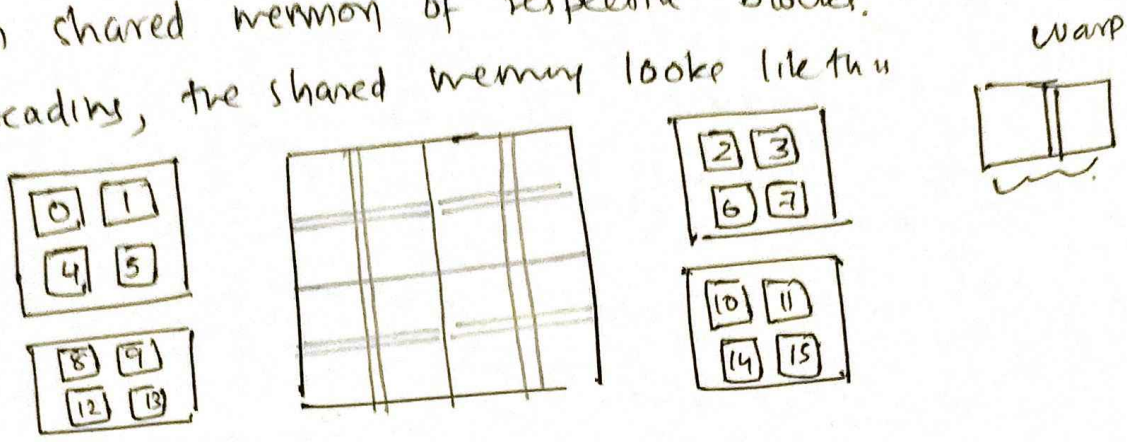
=> Here we can see, ~~thread per~~ naive approach ~~provide~~ performs coalesced reads while do not perform coalesced write //.

=> This can be rectified by using Shared memory.

Memory coalescing using Shared Memory



⇒ Every thread in block makes coalesced reads and stored the data in shared memory of respective blocks.
After reading, the shared memory looks like this



Note: we need to use cuda.synchronize all threads such that all threads in block is synchronized..

⇒ To perform coalesced write, each warp reads column elements from shared memory and write it row of output array (or global memory in general)

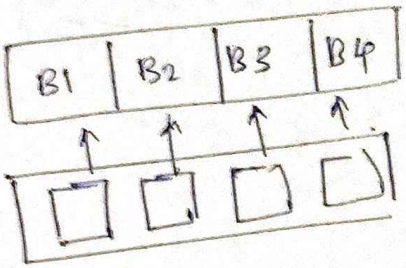
↳ In a row major order convention followed by CUDA, writing to a row is coalesced since it forms continous memory.

⇒ This we can take transpose of a matrix using coalesced reads & writes.

Note
⇒ Here they performs column wise read from shared memory and write it the appropriate row of output matrix (global memory)
⇒ I don't how to column wise read is uncoalesced.
⇒ Further, Columnwise read leads to some thing knows as "Bank conflict".

Shared Memory Bank conflicts

* Banks are the subunits of shared memory that allows parallel access
* In most of Hardware a shared memory will be divided to 32 banks
  * Since a warp has 32 threads, the shared memory is divided to 32 banks such that each thread can access one bank parallely.

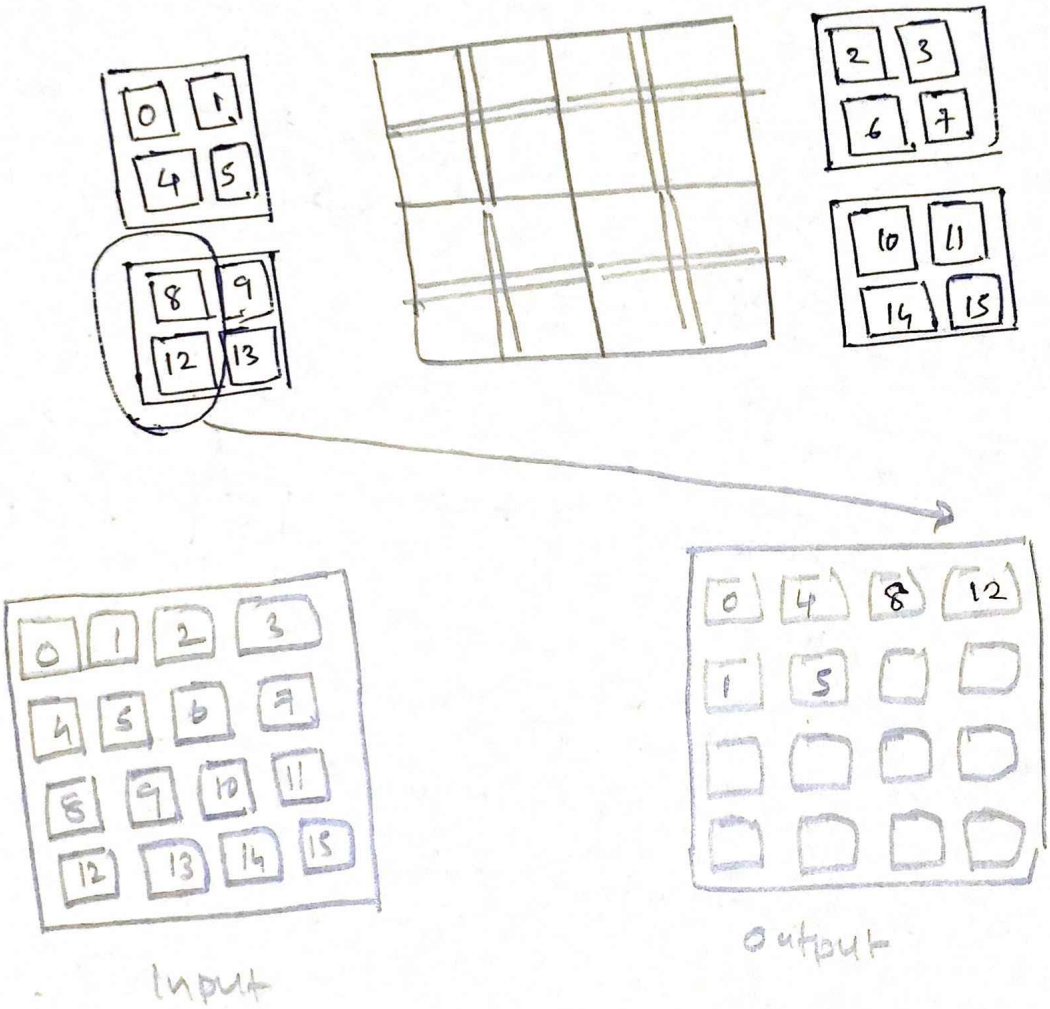


* Banks in a shared memory living contingeous memory.
* Each bank can serve one thread per cycle.
  i.e, If you have 32 threads in a warp and each thread accessing a different bank, all 32 accesses will occur in parallel leading to high efficiency.
* In CUDA, addresses are mapped to banks in a cyclic maner,
  If you declare a shared memory array, first element of array goes in to bank 0, the second into bank 1 and so on. once it reaches last bank, it wraps arounds and start at 0 again

Bank conflict

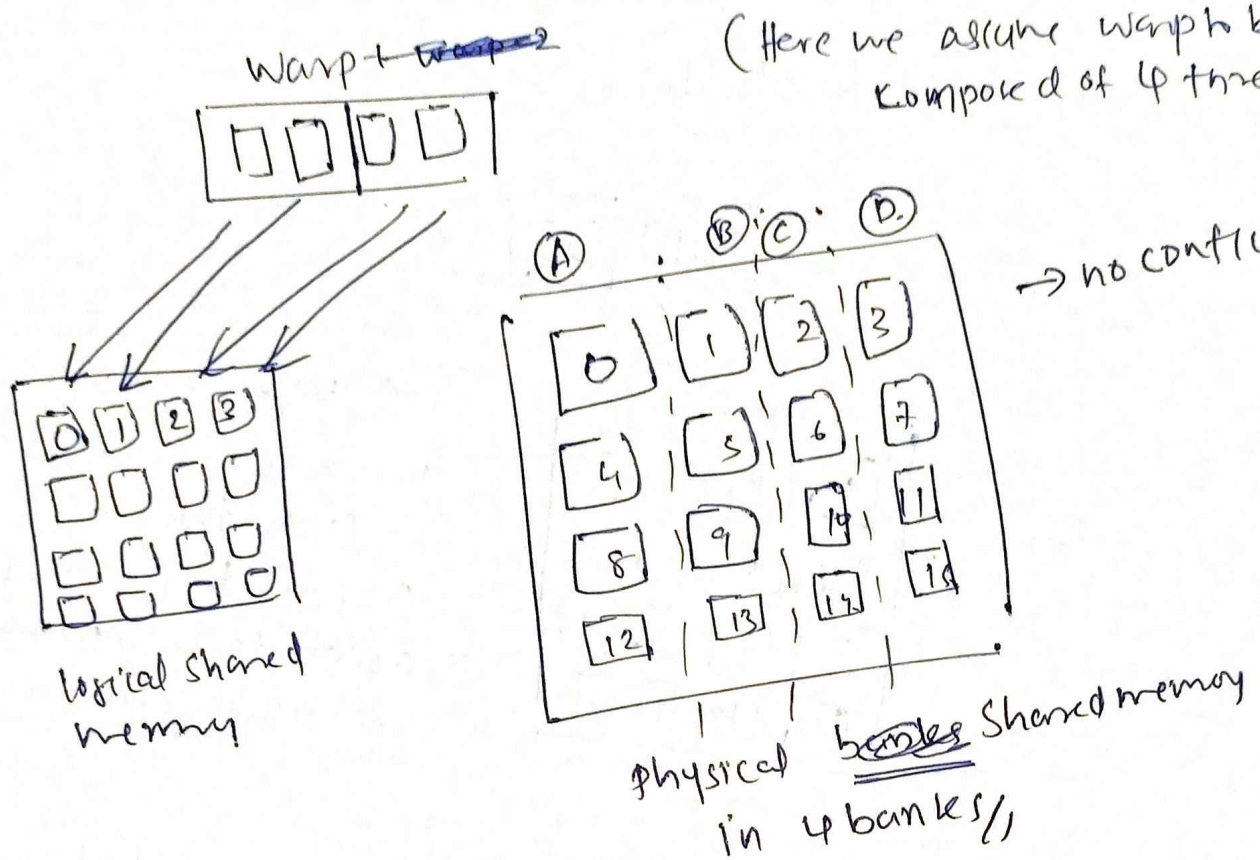
* Bank conflict arise when two or more threads tries to access same bank.
* In such cases, the memory access becomes serialized, meaning the threads have to wait and access the bank one after the other, reducing the performance.

Bank conflict in the case of transpose of Matrix



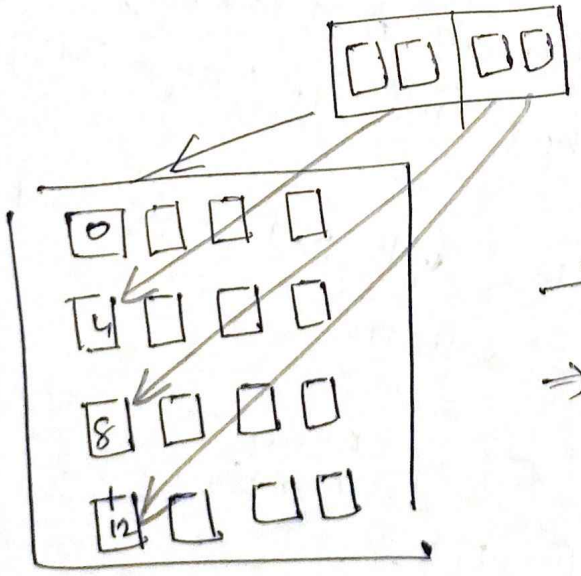
⇒ ~~In Previous ex~~ In transpose of a matrix, ~~we~~ using shared memory to ~~avoi~~ make reading and writing coalesced,

- we first read the data in coalesced manner (row wise) and store it in appropriate shared memory.
- Then we read data column wise ~~from~~ from shared memory and write it in the transposed location at output array.
- It ~~we~~ leads to Bank conflicts.



⇒ Elements in a column are stored in same bank.

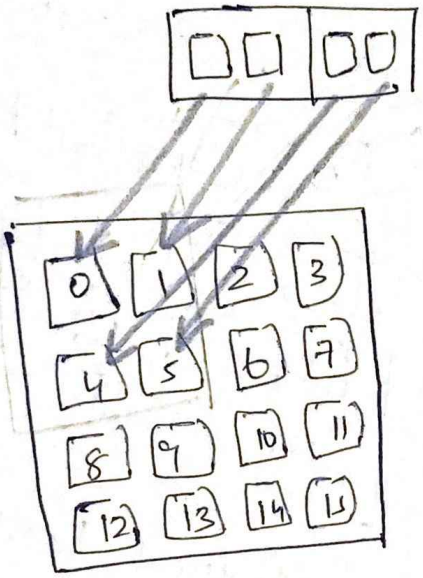
⇒ Thus accessing elements in a column by more than one thread leads to Bank conflict.



→ leads to 4-way bank conflict.

⇒ the memory access will be serialized over 4 cycles

⇒ Similarly consider the situation.



⇒ ~~use~~ In this case, we have 2-way bank conflict that would require the memory access to be ~~serialize~~ serialized over 2 cycles

⇒ so in the previous example, ~~there occurs~~ of transposing a matrix there occurs 2-way bank conflict.

Solving ~~Bank 2~~ Bank conflict

⇒ In the case of transposing a matrix (or the cases where we need to ~~acce~~ multiple elements in a column) we can use a simple trick of padding to prevent Bank conflict.

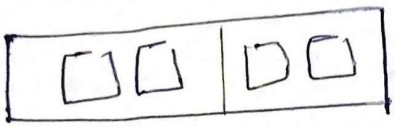
⇒ For doing padding, we will pad the shared memory with an extra column.

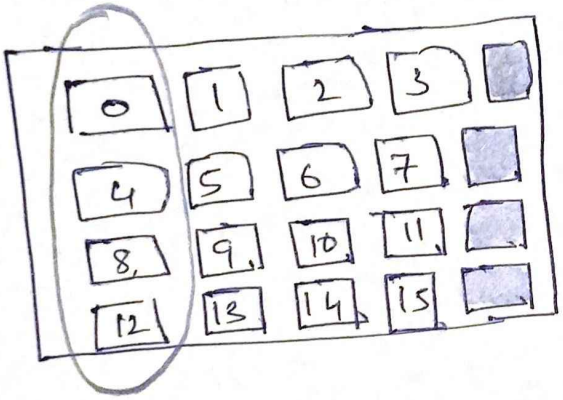
i.e, if the shared memory array size = 4 x 4 (or tile), we will make it 4 x 5

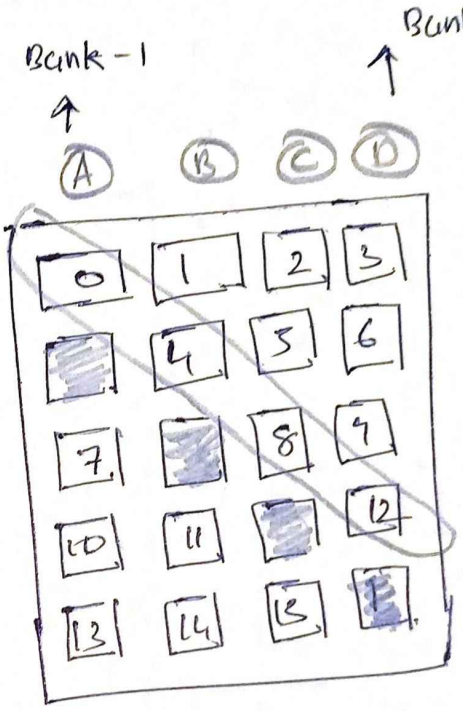
- The we write elements of input array to memory tile to addresses in the range [0:4][0:4]

- ~~This when~~ it

⇒ This padded memory tile will be stored in the physical shared memory with 4 banks. in such a way that we can access multiple elements in a column of memory tile without facing bank conflict.







⇒ ~~Here one col~~ Here accessing the first column of memory tile will be possible without memory bank conflict since ~~it~~ elements are stored in different banks.

⇒ Rowwise access also will be devoid of bank conflicts.