INFO-H-503
CUDA GPGPU programming

≥ CUDA 6.0

Slides 01

TA: Daniele Bonatto

Teacher: Gauthier Lafruit

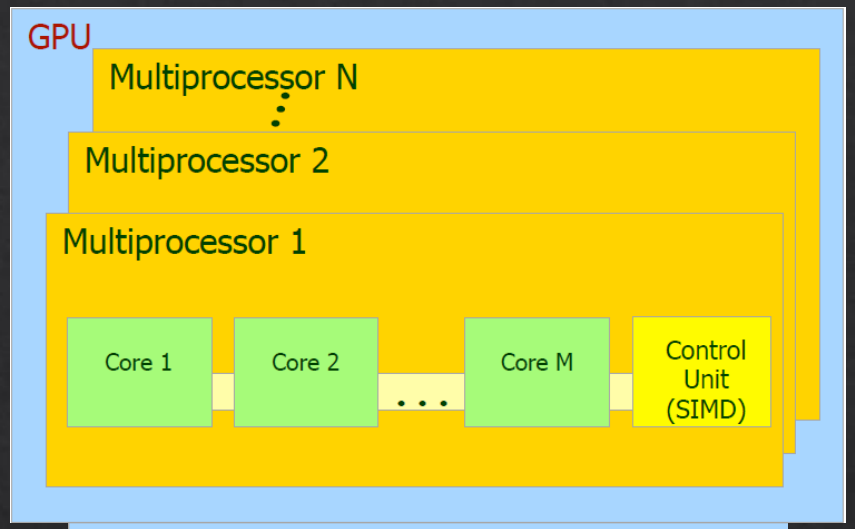
Acknowledgment

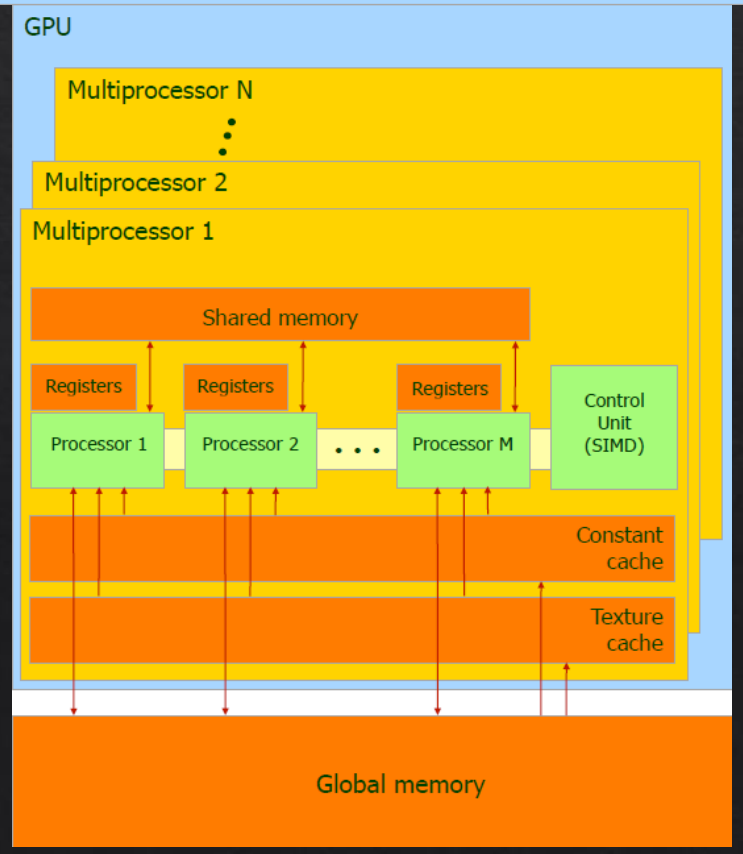
Those slides are partially based on:

- 9th Intl. Conference on Articulated Motion and Deformable Objects (AMDO'16) at Palma de Mallorca (Spain) by Manuel Ujaldón
  - http://amdo2016.uib.es/wp-content/uploads/2016/07/CUDA-Tutorial.pdf
- Programming Massively Parallel Processors, Third Edition: A Hands-on Approach 3rd Edition by David B. Kirk,Wen-mei W. Hwu
- CUDA by Example: An Introduction to General-Purpose GPU Programming 1st Edition by Jason Sanders, Edward Kandrot
- Professional CUDA C Programming 1st Edition by John Cheng, Max Grossman , Ty McKercher
- https://llpanorama.wordpress.com/2008/06/11/threads-and-blocks-and-grids-oh-my/
- https://www.pelagos-consulting.com/?p=503
- CUDA Programming – A Developer's Guide to Parallel Computing with GPUs – Shane Cook
- https://www.slideshare.net/npinto/harvard-cs264-05-advancedlevel-cuda-programming
- CSC 391/691 : GPU Programming – Fall 2011 – CUDA Thread Basics, Samuel S. Cho.
- https://www.youtube.com/watch?v=CZgM3DEBplE – CUDA Tutorial 9 Bank Conflicts
- CUDA Threads and Atomics - CME343 / ME339 - 25 April 2011, James Balfour

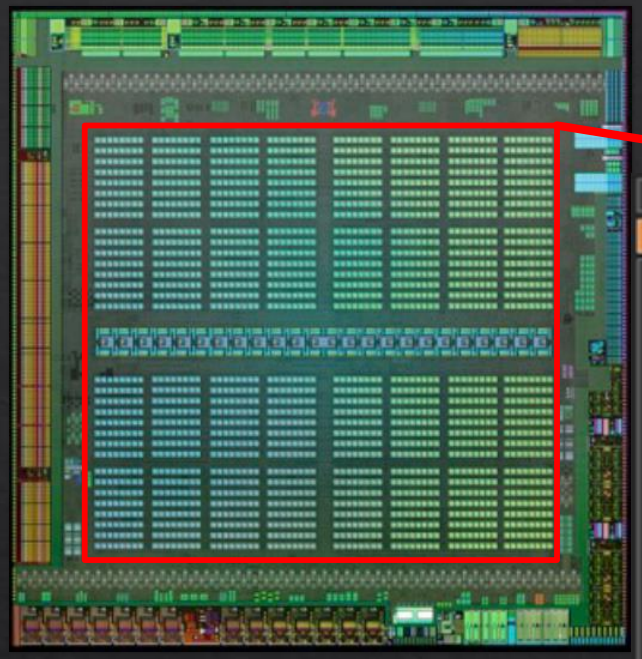
What is a GPU

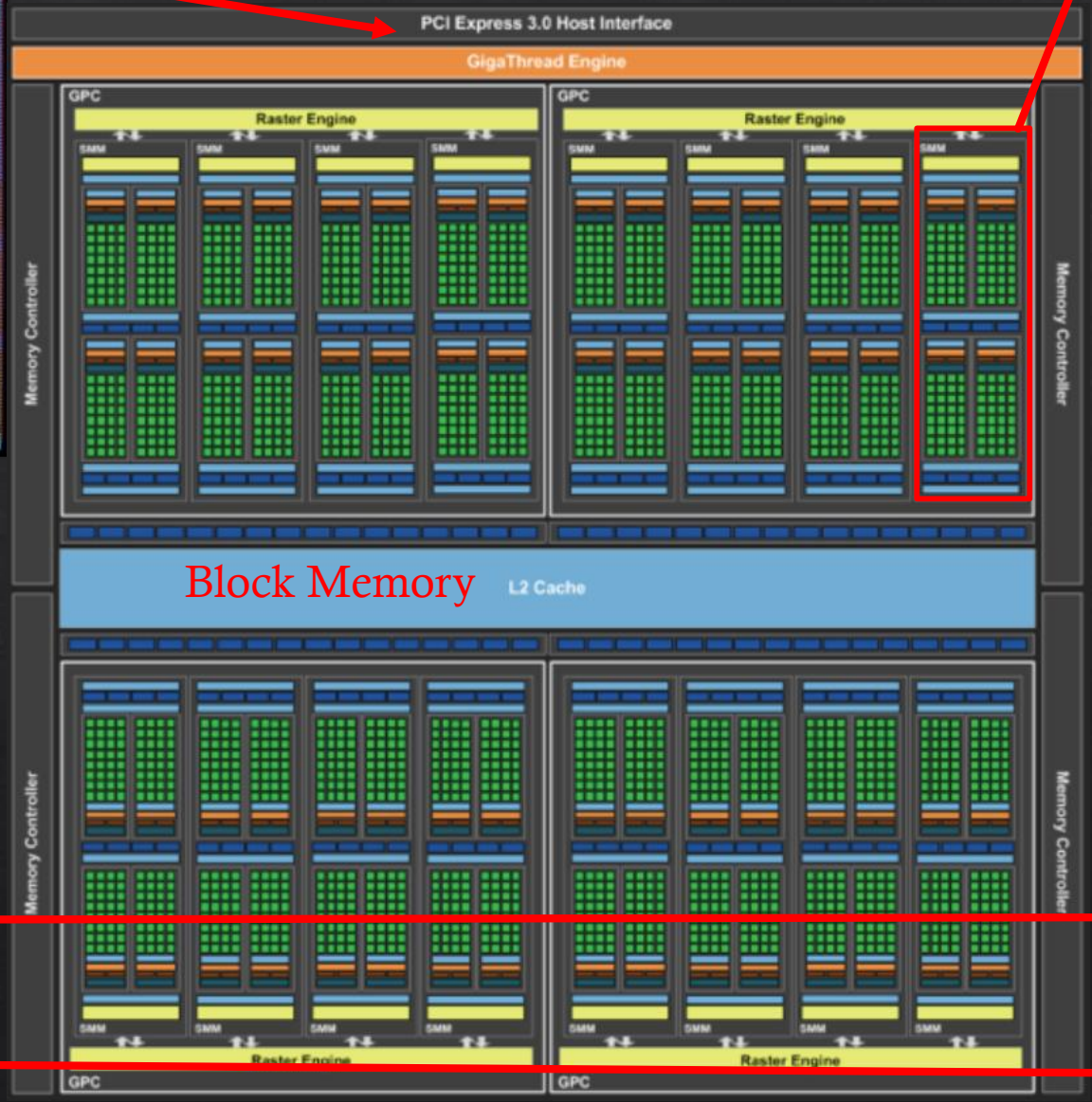
- A GPU consist of N Multiprocessors
- Each containing M Stream Processors (SM)
- Each containing K cores
- GM2000 (Maxwell)
  - 2015
  - 4 to 24 SMs
  - 128 cores/SM
  - Total number of cores: 512 to 3012
- We also have a memory hierarchy
  - Video memory (GDDR5), faster than DDR3
  - Each multiprocessor has :
    - a register file
      Very fast, each core in a SM has several registers.
    - shared memory, 500 x faster than the video memory.
      This memory is shared between all the cores in a SM.
    - constant cache and a texture cache (fast but READ ONLY)
      This memory is shared between all the cores in a SM.



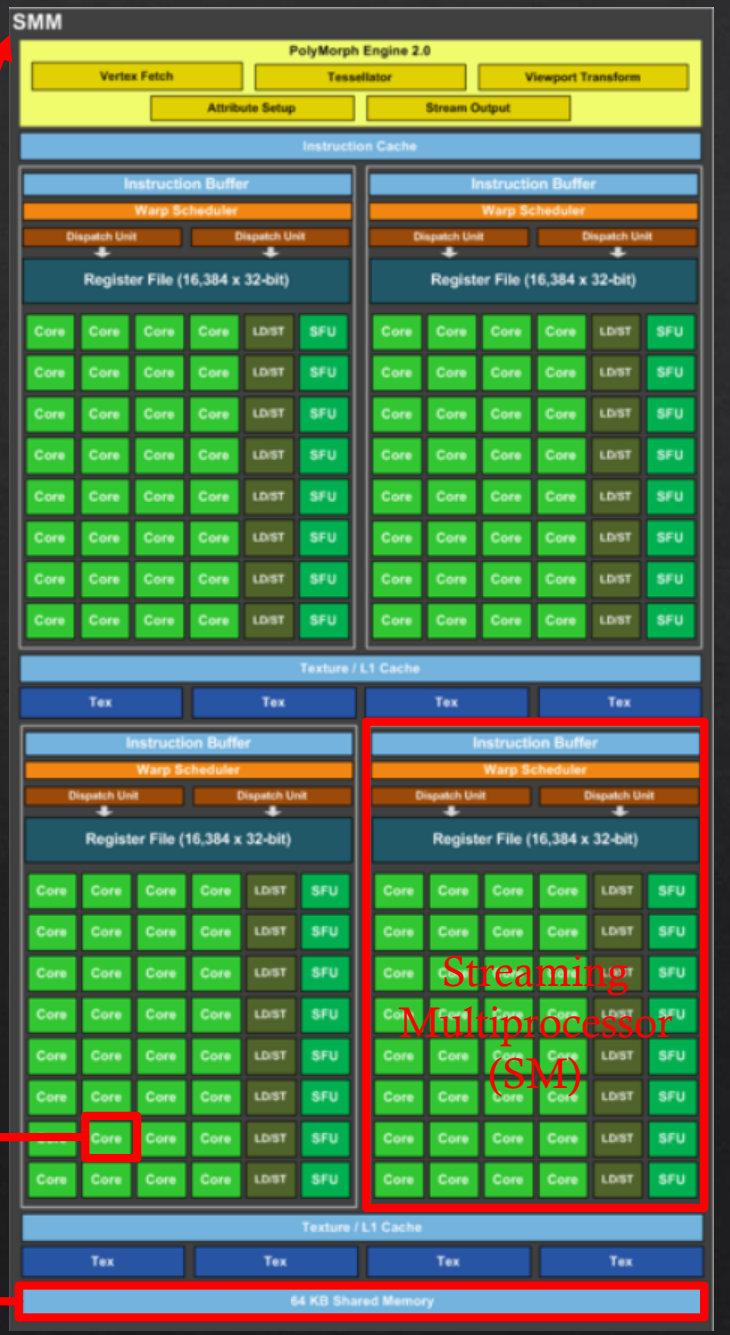


GTX 980 (Maxwell)





Multiprocessor



16 Multiprocessors
4 SM (Streaming processor)
32 Cores per SM
16 x 4 x 32 = 2048 cores!

1 core = 1 thread + local memory

Shared memory

Commercial models caracteristics (28nm)

GeForce	GTX 950	GTX 960	GTX 970	GTX980	GTX 980 Ti	Titan X
Release date	Aug'15	Aug'15	Sep'14	Sep'14	Jun'15	Mar'15
GPU (code name)	GM206-250	GM206-300	GM204-200	GM204-400	GM200-310	GM200-400
Multiprocessors	6	8	13	16	22	24
Number of cores	768	1024	1664	2048	2816	3072
Cores frequency (MHz)	1024-1188	1127-1178	1050-1178	1126-1216	1000-1075	1000-1075
DRAM bus width	128 bits	128 bits	256 bits	256 bits	384 bits	384 bits
DRAM frequency	2x 3.3 GHz	2x 3.5 GHz	2x 3.5 GHz	2x 3.5 GHz	2x 3.5 GHz	2x 3.5 GHz
DRAM bandwidth	105.6 GB/s	112 GB/s	224 GB/s	224 GB/s	336.5 GB/s	336.5 GB/s
GDDR5 memory size	2 GB	2 GB	4 GB	4 GB	6 GB	12 GB
Million of transistors	2940	2940	5200	5200	8000	8000
Die size	228 mm^2	228 mm^2	398 mm^2	398 mm^2	601 mm^2	601 mm^2
Maximum TDP	90 W	120 W	145 W	165 W	250 W	250 W
Power connectors	1 x 6 pines	1 x 6 pines	2 x 6 pines	2 x 6 pines	6 + 8 pines	6 + 8 pines
Price ($ upon release)	149	199	329	549	649	999

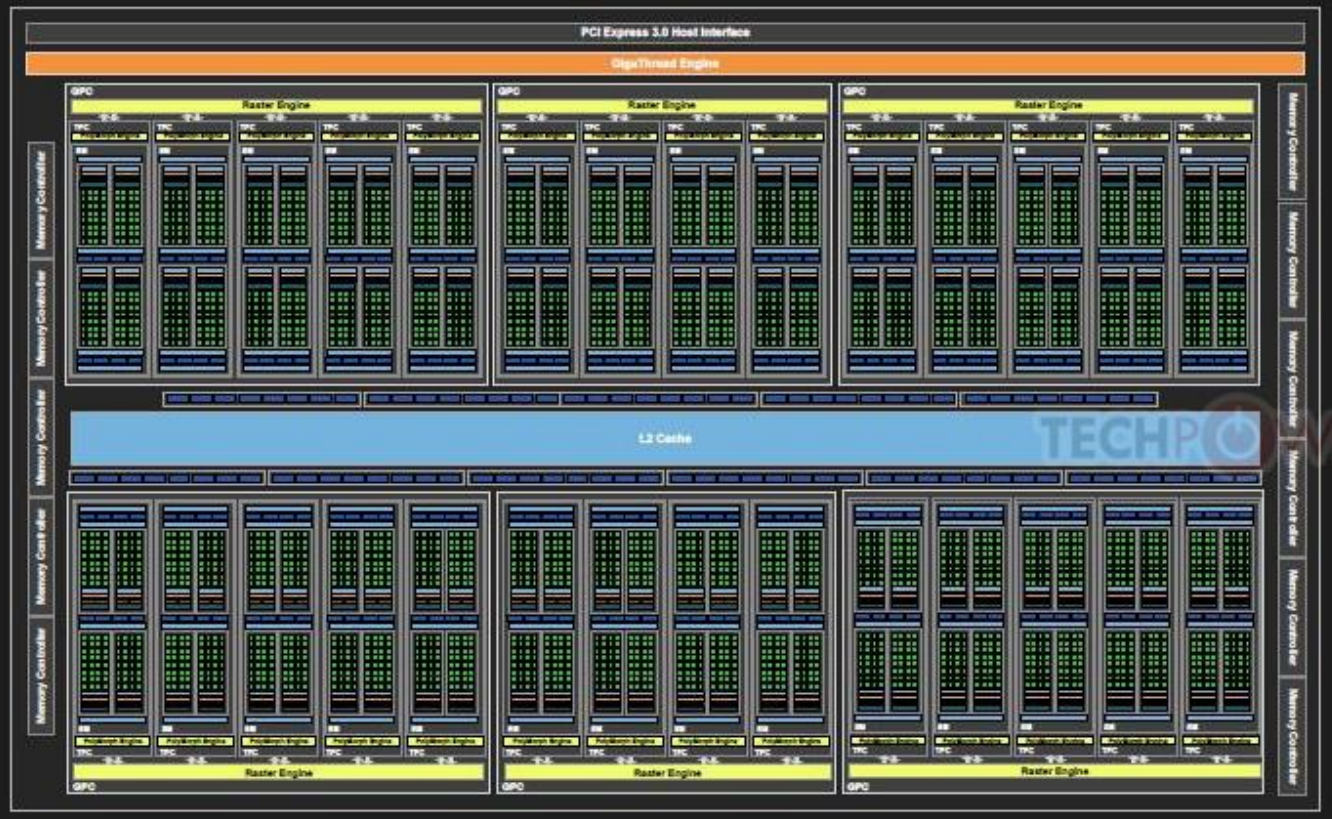
	GTX 680 (Kepler)	GTX 980 (Maxwell)	GTX 1080 (Pascal)
Year	2012	2014	2016
Transistors	3.54 B @ 28 nm.	5.2 B @ 28 nm.	7.2 B @ 16 nm.
Power consumption & die size	195 W & 294 mm^2	165 W & 398 mm^2	180 W & 314 mm^2
Multiprocessors	8	16	40
Cores / Multiproc.	192	128	64
Cores / GPU	1536	2048	2560
Clock (wo. and w. GPU Boost)	1006, 1058 MHz	1126, 1216 MHz	1607, 1733 MHz
Peak performance	1625 GFLOPS	4980 GFLOPS	8873 GFLOPS
Shared memory	16, 32, 48 KB	64 KB	
L1 cache size	48, 32, 16 KB	Integrated with texture cache	
L2 cache size	512 KB	2048 KB	
DRAM memory: Interface	256-bit GDDR5	256-bit GDDR5	256-bit GDDR5X
DRAM memory: Frequency	2x 3000 MHz	2x 3500 MHz	4x 2500 MHz
DRAM memory: Bandwidth	192.2 GB/s	224 GB/s	320 GB/s

For now, it is often better to have 980 Ti or Titan X than a GTX 1080 in term of computing cores. However, the frequencies are not the same

In the past

	Tesla		Fermi		Kepler				Maxwell				Pascal	
Architecture	**G80**	**GT200**	**GF100**	**GF104**	**GK104 (K10)**	**GK110 (K20X)**	**GK110 (K40)**	**GK210 (K80)**	**GM107 (GTX750)**	**GM204 (GTX980)**	**GM200 (Titan X)**	**GM200 (Tesla M40)**	**GP104 (GeForce GTX 1080)**	**GP100 (Tesla P100)**
Time frame	**2006 /07**	**2008 /09**	**2010**	**2011**	**2012**	**2013**	**2013 /14**	**2014**	**2014 /15**	**2014 /15**	**2016**	**2016**	**2016**	**2017**
CUDA Compute Capability	**1.0**	**1.3**	**2.0**	**2.1**	**3.0**	**3.5**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**5.3**	**6.0**	**6.0**
N (multiprocs.)	16	30	16	7	8	14	15	30	5	16	24	24	40	56
M (cores/multip.)	8	8	32	48	192	192	192	192	128	128	128	128	64	64
Number of cores	128	240	512	336	1536	2688	2880	5760	640	2048	3072	3072	2560	3584

GTX 1080 TI OVERVIEW



12B Transistors

1.6 GHz Boost, 2 GHz OC

28 SMs, 128 cores each

3584 CUDA cores

28 Geometry units

224 Texture units

6 GPCs

88 ROP units

352 bit GDDR5x

Terminology

- Host: The CPU and the main memory [DDR3/4/…]
- Device: The graphics card [set of multiprocessors + video memory (GDDR5/…)]
- Multiprocessor: Set of processors + shared memory
- Kernel: Steps that are run in sequence on the GPU ~function
- Stream: Queue of kernels that the hardware will schedule sequentially.
- SP – Symmetrical processor : Computational cores
- SM – Streaming machine – Can be thought of as single processor with N cores (SPs) embedded within it.
- Thread: An execution of a kernel with a given index. Each thread uses its index to access elements in arrays.
- Block: Group of threads, executed concurrently or serially and in no particular order (sync with _syncthreads(), wait all threads in the same block)
  - The threads in a block can communicate with shared memory.
- Grid: Group of blocks. On synchronization at all between the blocks.
- Warp size: 32. The threads in a block are organized in warps. This is the granularity of the scheduler for issuing threads to the execution units

From kernel to the execution

- Every kernel (program for the GPU) is ultimately executed on streaming multiprocessors (SMs).
- When a kernel is invoked, we specify how many threads we want to launch in a block and how many blocks we want to launch in the grid
  - These blocks gets ultimately mapped to one of the available SM's to execute
- All the threads in a block can share the memory on the SM as they are on the same SM.
- Now, we have blocks which execute on SM:
  - SM won't directly give the threads the execution resources.
  - It will try to divide the threads in the block into warps (32 threads)
  - The warps in each of the blocks exhibit SIMD execution (Single Instruction Multiple Data)
  - If there is a memory access to any thread in a warp, SM switches to next warp. This way, SM always have some work to do

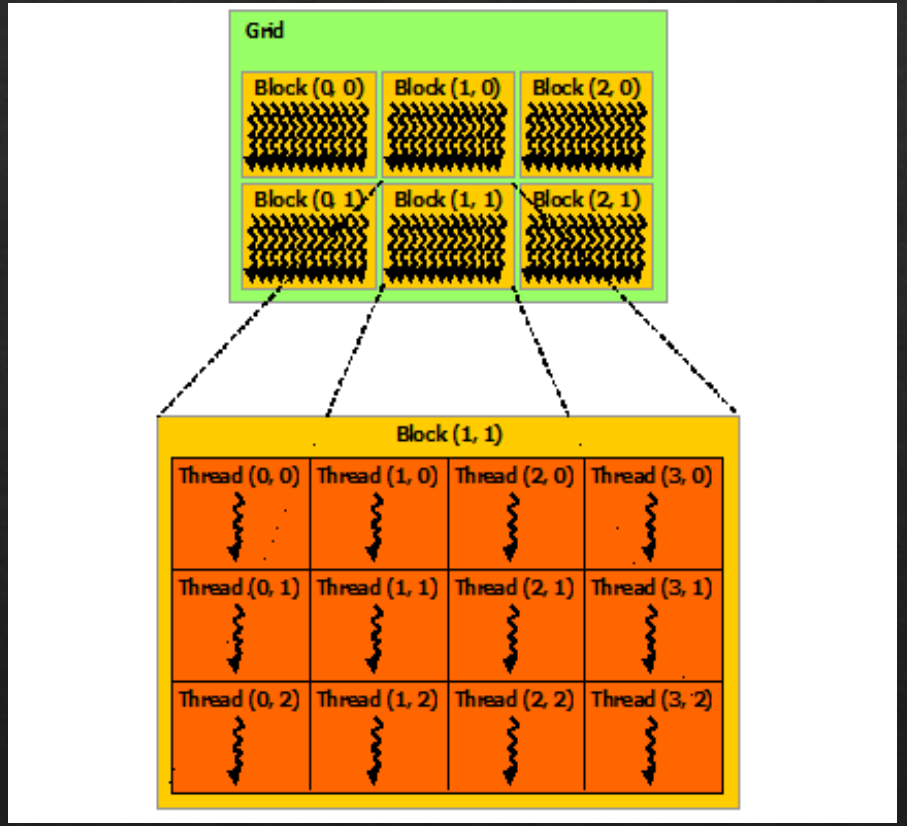
- https://www.quora.com/What-is-a-warp-and-how-is-it-different-from-a-thread-block-or-wave-in-CUDA

Grid? Blocks? Threads? Warps??

- A grid is composed of totally independent blocks
- A block is composed of threads which can communicate within their block
  - The threads can be paused and resumed at will
- 32 threads form a warp and the CUDA instructions are done in warp.
  - Thus:
    - Each warp threads execute parallelly and share the same code:
      - Follow the same execution path with minimal divergence
      - Is expected to stall at the same places
    - If a warp stall for any reason (typically operand not ready), we have a context switch and another warp will be executed
      - We pause all the threads in a warp and resume them at the same time due to the previous assumptions
    - REMARK: If the warp have divergences and/or stall in different places, we are going to have an underutilization of the hardware resources -> TRY TO AVOID IFs INSTRUCTIONS
    - REMARK: A warp is a scheduling unit for the SM
    - REMARK: CUDA works in SIMT Single Instruction Multiple Threads
- https://www.quora.com/What-is-a-warp-and-how-is-it-different-from-a-thread-block-or-wave-in-CUDA

How CUDA splits problems

- Grids of blocks of threads
- The blocks runs in any order
- The blocks are run on one SM with free slots
- A Warp is a set of 32 threads in a block that execute the same kernel at the same time
- Military analogy:
  - Army (grid) of soldiers (threads). The army is split into a number of units (blocks), each commanded by a lieutenant. The unit is split into squads of 32 soldiers (a warp), each commanded by a sergeant
  - To perform some action, central command (kernel/host) must provide some action plus some data. Each soldier (thread) works on his or her individual part of the problem. Threads may from time to time swap data with one another under the coordination of either the sergeant (warp) or the lieutenant (block). However, any coordination with other units (blocks) has to be performed by central command (kernel/host)



CPU vs GPU threading

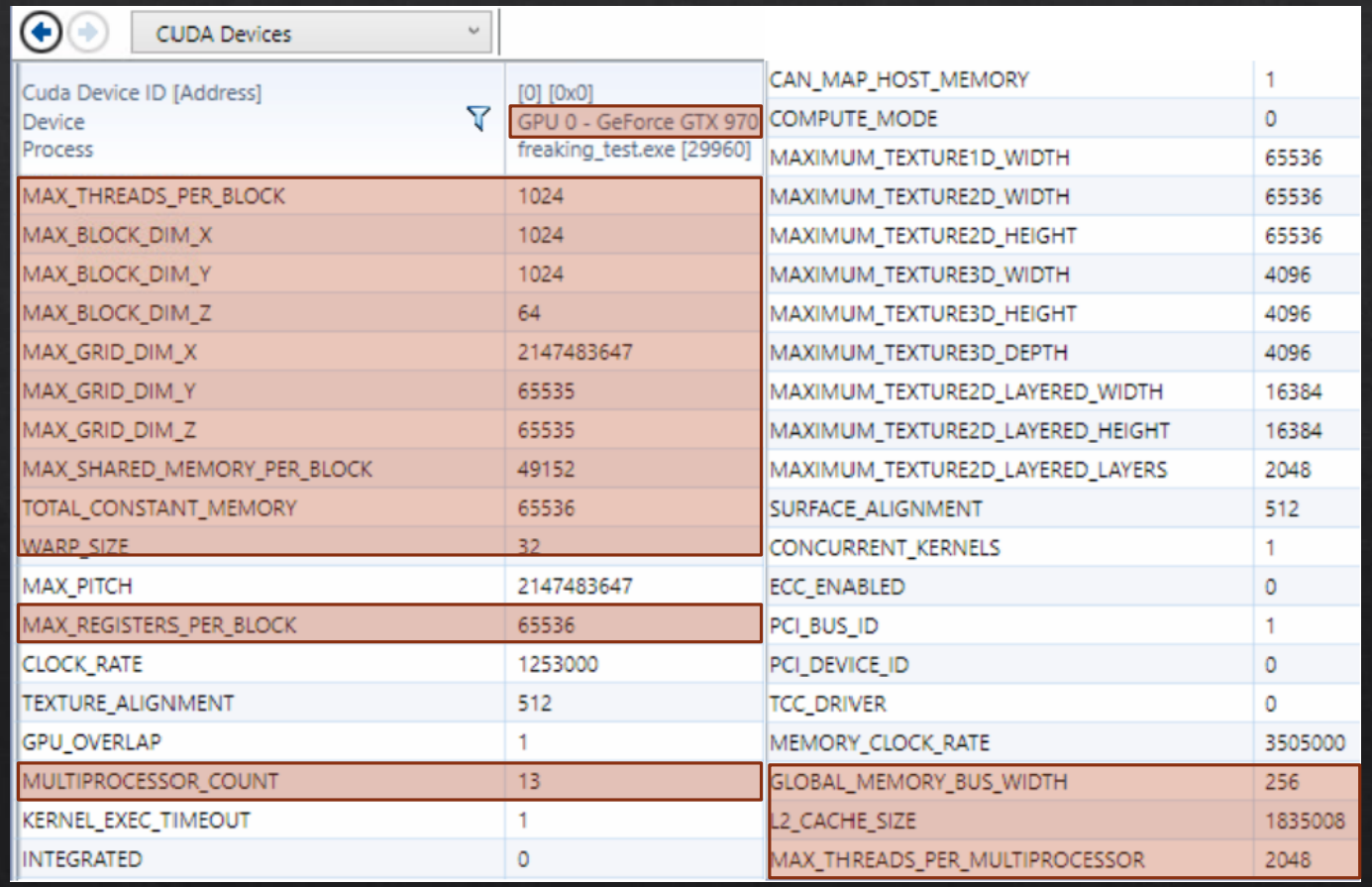
- In the CPU, threads are lighter than process but they remain slow
  - This is due to:
    - Their spawning and,
    - context switching.
- In the GPU the threads are extremely lightweight
  - Almost no overhead
  - Context switching is virtually free
- Thus in the GPU we want to make as much threads as we can
- Each SM process batches of blocks one after another.

Playing with parallel constraints on Maxwell to maximize concurrency

- We have hardware and software limits when we use a GPU
- By asking the card we can see that the limits within a SM in a Maxwell GPU are:
  - 32 concurrent blocks (blocks/SM in the doc)
  - 1024 threads/block (in the doc)
  - 2048 threads total (threads/SM in the doc)
- 1 block of 2048 threads. [2] (not feasible)
- 2 blocks of 1024 threads. Feasible on the same SM
- 4 blocks of 512 threads. Feasible on the same SM
- 4 blocks of 1024 threads. [3] but feasible involving two SM
- 256 blocks of 8 threads. [1] but feasible with 8 SM

GPU limits

- You have to check your GPU limits!



Cuda Device ID [Address]	[0] [0x0]
Device	GPU 0 - GeForce GTX 970
Process	freaking_test.exe [29960]
MAX_THREADS_PER_BLOCK	1024
MAX_BLOCK_DIM_X	1024
MAX_BLOCK_DIM_Y	1024
MAX_BLOCK_DIM_Z	64
MAX_GRID_DIM_X	2147483647
MAX_GRID_DIM_Y	65535
MAX_GRID_DIM_Z	65535
MAX_SHARED_MEMORY_PER_BLOCK	49152
TOTAL_CONSTANT_MEMORY	65536
WARP_SIZE	32
MAX_PITCH	2147483647
MAX_REGISTERS_PER_BLOCK	65536
CLOCK_RATE	1253000
TEXTURE_ALIGNMENT	512
GPU_OVERLAP	1
MULTIPROCESSOR_COUNT	13
KERNEL_EXEC_TIMEOUT	1
INTEGRATED	0
CAN_MAP_HOST_MEMORY	1
COMPUTE_MODE	0
MAXIMUM_TEXTURE1D_WIDTH	65536
MAXIMUM_TEXTURE2D_WIDTH	65536
MAXIMUM_TEXTURE2D_HEIGHT	65536
MAXIMUM_TEXTURE3D_WIDTH	4096
MAXIMUM_TEXTURE3D_HEIGHT	4096
MAXIMUM_TEXTURE3D_DEPTH	4096
MAXIMUM_TEXTURE2D_LAYERED_WIDTH	16384
MAXIMUM_TEXTURE2D_LAYERED_HEIGHT	16384
MAXIMUM_TEXTURE2D_LAYERED_LAYERS	2048
SURFACE_ALIGNMENT	512
CONCURRENT_KERNELS	1
ECC_ENABLED	0
PCI_BUS_ID	1
PCI_DEVICE_ID	0
TCC_DRIVER	0
MEMORY_CLOCK_RATE	3505000
GLOBAL_MEMORY_BUS_WIDTH	256
L2_CACHE_SIZE	1835008
MAX_THREADS_PER_MULTIPROCESSOR	2048

GTX 970

CUDA is almost C

- CUDA is very similar to C, we only add some keywords:
  - Type qualifiers:
    - __global__ to declare a function to be executed on the GPU
    - __device__ to declare a function to be called by another function on the GPU
    - __shared__ to declare shared memory in a kernel
    - __local__ [see doc]
    - __constant__ [see doc]
  - Keywords: threadIdx, blockIdx, gridDim, blockDim to access elements in a kernel
  - Intrinsics:
    - __syncthreads(); // to synchronize threads between warps (within a block) and in all the warps (thus all threads..)
    - atomicAdd(); // to synchronize blocks (see atomic ops: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions)
    - dim2, dim3, uint2, uint3 // units for cuda : dim2 hello(16,32);
  - Runtime API: to allocate memory on the GPU and transfer data, check errors, etc.. [doc]
  - Kernel functions to launch code to the GPU from the CPU: <<<X,Y,Z>>>(.) (special syntax detailed later)

How to solve a problem on the GPU

- Think of a formula that represents each output point as some function of the input data.
- For algorithms that have multiple steps: (and are hard to express as a function)
  - Write several kernels for each step
    - Usually those are simple kernels
    - The harder one is the bottleneck which will need some thought
  - Put those kernels on a queue (Stream)

How to divide a problem in CUDA

- When we are working with data (1D like vectors, 2D like matrices, 3D like spatial data), it is often bigger than the quantity of threads available
  - Image: 2D, 1920x1080 = 2.073.600 pixels (2Mp)
  - GTX 1080: 40 SM, 64 cores/SM, Total: 2560 cores available
  - Thus : 2.073.600/2560 = image 810 times bigger than the available number of cores
- We are going to setup a grid of the size of the image (closest upper power of 2 in reality)
- CUDA will work with 40 SM/blocks at a time before going to the next 40 blocks of the grid

- Another strategy is to use only 40 blocks for the entire image and make each thread in the kernel do 2.073.600/40 = 51.840 operations… this is clearly suboptimal in this case but using threads to treat multiple part of an image could be successfully used in some algorithms!

Grid example in 3Ds Max rendering

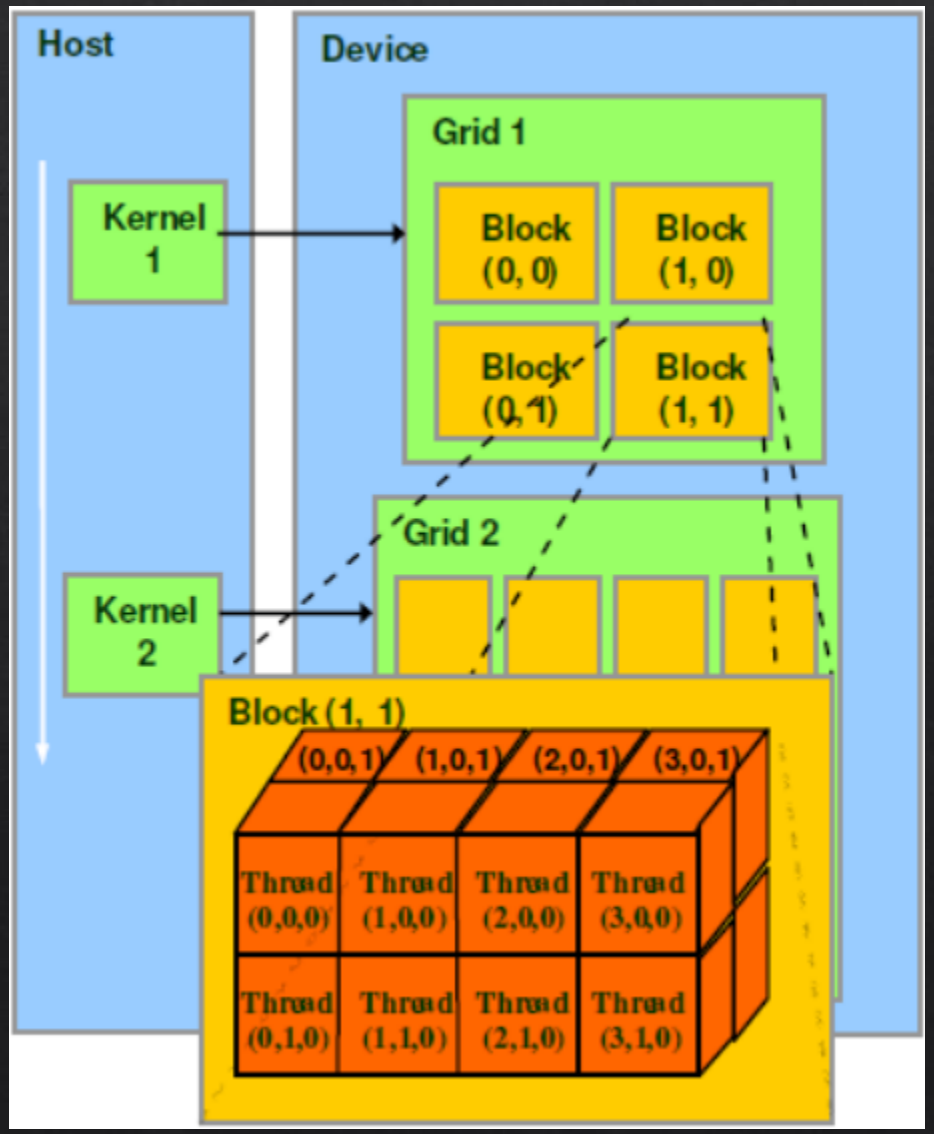
We can clearly see the blocks working

More cores, less time by Duy Le
https://www.youtube.com/watch?v=MiZDtax_O4U

Rendering Evermotion on 360 cores by MRIYArender
https://www.youtube.com/watch?v=XPXiITMpHGE

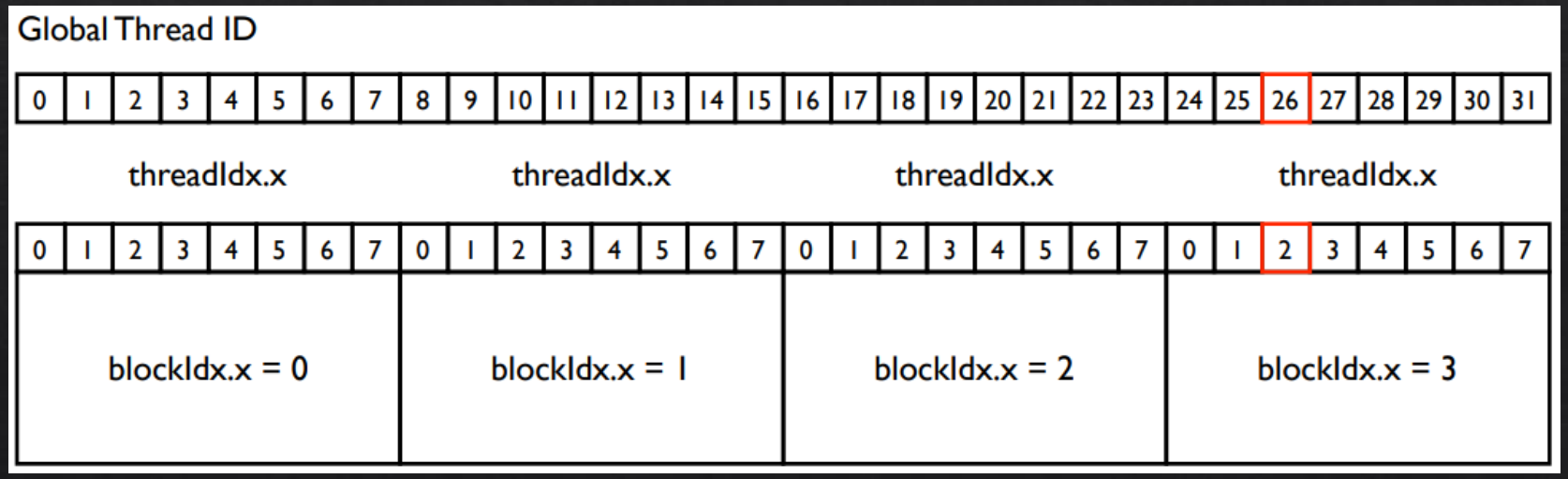
Grid and Blocks dimensions

- A lot of problems in science are not expressed in 1D but are rather natural in 2 or 3 dimensions.
  - Images work well in 2D
  - Medical scans work well in 3D
- CUDA allows the user to make 1D or 2D or 3D grids and 1D or 2D or 3D blocks!
- We have special variables in the kernels:
  - gridDim (How many blocks in the grid)
  - blockIdx (Index of the working block)
  - blockDim (How many threads in this block)
  - threadIdx (local index of the thread within the block)
  - For all 3 variables, we can acces at least x,y and often z
    - Eg: blockIdx.x * blockDim.y + threadIdx.z (nonsense here)
- REMARK: CUDA WORKS IN COLUMN MAJOR ORDER



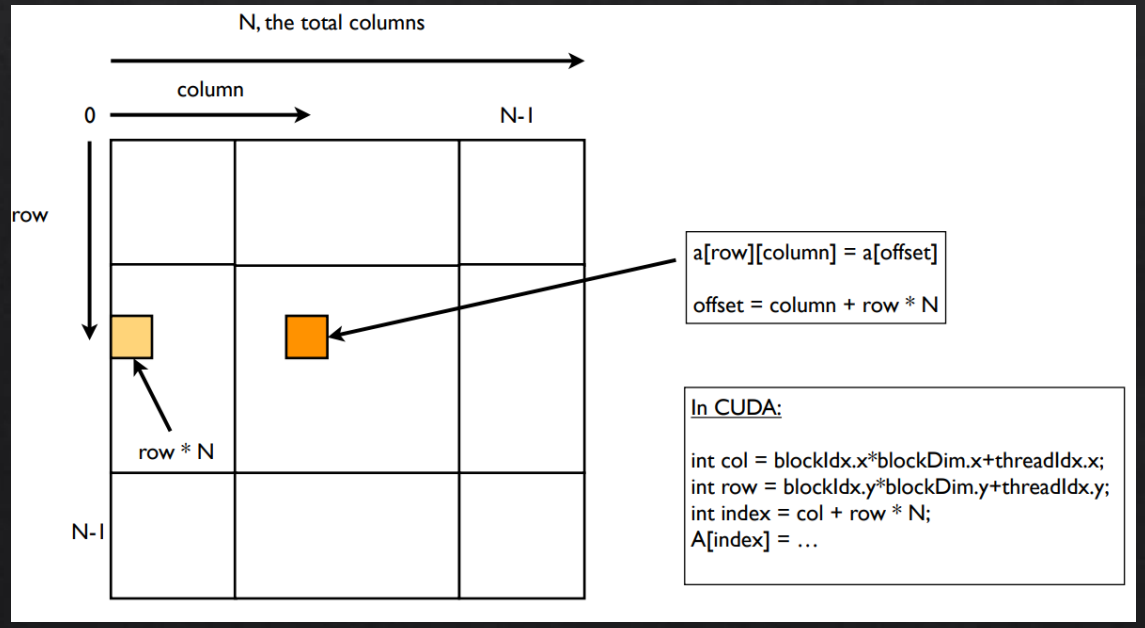
Indexing

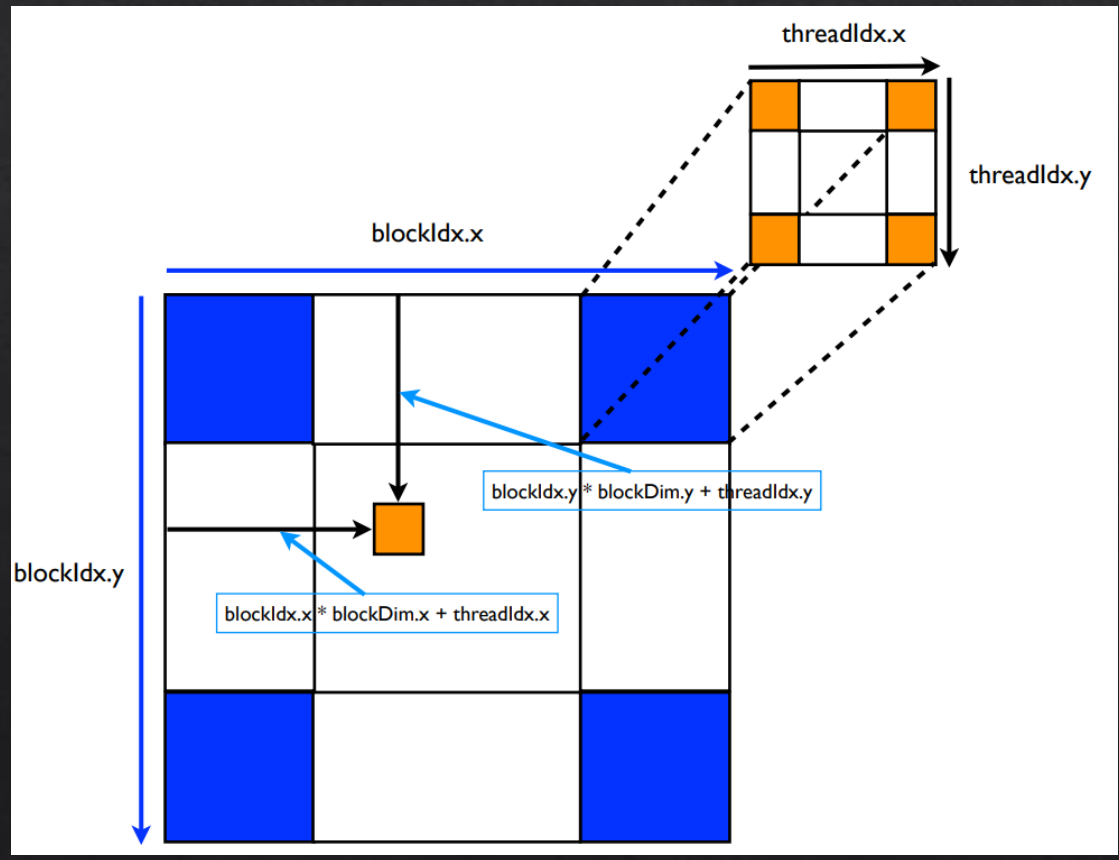
- CUDA Built-In Variables
  - blockIdx.{x,y,z} Built-in variables that returns the block ID in the {x,y,z}-axis of the block.
  - threadIdx.{x,y,z} Built-in variables that return the thread ID in the {x,y,z}-axis of the thread.
  - blockDim.{x,y,z} Built-in variables that return the block dimension.
- The full global thread ID in x dimension can be computed by
  - x = blockIdx.x * blockDim.x + threadIdx.x;



Indexing – Higher dimensions

- Number of blocks in 2D grid: gridDim.x * gridDim.y
- Number of threads in 3D block: blockDim.x * blockDim.y * blockDim.z
- Example of initialization:
  - dim3 grid(16,16); // grid = 16x16 blocks
    dim3 block(32,32); // block = 32x32 threads
    myKernel<<<grid, block>>>(…);
  - x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;





THINK BY YOURSELF THERE MAY BE ERRORS THERE: send me a mail ;)

Playing with indexes

Often we want to find the current thread in absolute position with respect to the blocks and the grid

1D grid of 1D blocks:

```
threadId = blockIdx.x * blockDim.x
         + threadIdx.x;
```

1D grid of 2D blocks:

```
threadId = blockIdx.x * blockDim.x * blockDim.y
         + threadIdx.y * blockDim.x
         + threadIdx.x;
```

1D grid of 3D blocks:

```
threadId = blockIdx.x * blockDim.x * blockDim.y * blockDim.z
         + threadIdx.z * blockDim.y * blockDim.x
         + threadIdx.y * blockDim.x
         + threadIdx.x;
```

2D grid of 1D blocks:

```
blockId   = blockIdx.y * gridDim.x + blockIdx.x;
threadId = blockId * blockDim.x + threadIdx.x;
```

2D grid of 2D blocks:

```
blockId = blockIdx.x + blockIdx.y * gridDim.x;
threadId = blockId * (blockDim.x * blockDim.y)
         + (threadIdx.y * blockDim.x)
         + threadIdx.x;
```

2D grid of 3D blocks:

```
blockId = blockIdx.x + blockIdx.y * gridDim.x;
threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
          + (threadIdx.z * (blockDim.x * blockDim.y))
          + (threadIdx.y * blockDim.x)
          + threadIdx.x;
```

3D grid of 1D blocks:

```
blockId = blockIdx.x  + blockIdx.y * gridDim.x
                + gridDim.x * gridDim.y * blockIdx.z;
threadId = blockId * blockDim.x + threadIdx.x;
```

3D grid of 2D blocks:

```
blockId = blockIdx.x  + blockIdx.y * gridDim.x
                + gridDim.x * gridDim.y * blockIdx.z;
threadId = blockId * (blockDim.x * blockDim.y)
                + (threadIdx.y * blockDim.x)
                + threadIdx.x;
```

3D grid of 3D blocks:

```
blockId = blockIdx.x
                + blockIdx.y * gridDim.x
                + gridDim.x * gridDim.y * blockIdx.z;
threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                + (threadIdx.z * (blockDim.x * blockDim.y))
                + (threadIdx.y * blockDim.x)
                + threadIdx.x;
```

CPU-GPU Memory Management

- As we are going to deal with pointers to the CPU and to the GPU main memory, it is useful to start every pointer name by h_ if it is a pointer to the host's memory (CPU) or d_ if it is a pointer to the GPU's memory

```
int N = 16;
int num_bytes = N*sizeof(int);
int *d_a, *h_a = 0;

h_a = (int*) malloc(num_bytes); // allocate the memory on the CPU
cudaMalloc ( (void**) &d_a, num_bytes); // allocate the memory on GPU

cudaMemset( d_a, 0, num_bytes); // set all the values to 0
cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost); // copy back the values

free(h_a); // free the memory
cudaFree(d_a); // free the memory on GPU
```

We often use flags such as:
cudaMemcpyDeviceToHost or cudaMemcpyHostToDevice

Memory & Data Locality

- We have multiple types of memory:
  - Global memory/Constant memory
  - Shared memory
  - Local memory
- We want to exploit multilevel caches by using
  - Spatial locality (address space)
  - Temporal locality (data accessed before will be likely accessed again)
  - Tasks repeated many times (last accessed)
- On GPU YOU are responsible for the content of the caches!
  - Advantage Control when the writes happen.

- Tip: It's a perfectly valid approach to develop a program, prove the concept, and then deal with locality issues

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Task execution model

- A group of N streaming processors execute in a lock-step basis, running the same program but on different data
  - Each instruction in the instruction queue is dispatched to every SP within an SM.
- We have a huge register file, switching threads has effectively zero overhead
  - GPU supports a very high number of threads!

How to declare and call a kernel

- Declare a kernel:
  - You simply put __global__ in the beginning of the prototype of a function!
  - The Nvidia compiler (nvcc) will know what to do

```
__global__ void mySuperKernel(float * A, float * B, float *C) {
  // here you can use blockIdx, threadIdx, etc..

  [your parallel algorithm]
}
```

- Run a kernel:
  - You use the special operator <<< >>>
  - mySuperKernel <<<1, 1>>>(A, B, C); // executes 1 block composed of 1 thread (no parallelism)
  - mySuperKernel<<<B, 1>>>(A, B, C); // executes B blocks composed of 1 thread. Inter-SM parallelism
  - mySuperKernel<<<B, M>>>(A, B, C); // Executes B blocks composed of M threads each. Inter- and Intra- SM parallelism

More on kernel invocation

- Kernel_function<<<num_blocks, num_threads>>>(params…)
  - Use variables for num_blocks and num_threads, you are going to tune them.
  - num_threads is the number of threads in each block we want
    - The number of threads in a block are limited: 512, 1024, …
  - Parameters can be passed via registers or constant memory
    - If passed by registers: eg: 128 threads + 3 parameters : 128 x 3 = 384 registers
      - A GPU has a lot of registers, at least 8192 per SM. So potentially you could use 8192/128=64 registers per thread. But it is a bad idea: prefer using multiple blocks (otherwise the SM are going to be idle as soon as you access the memory)
  - The number of threads in a block should ALWAYS be a multiple of the warp size (32, 64, …)

More on kernel invocation – Shared memory

- Dynamic
  - Kernel_function<<<num_blocks, num_threads, shared_memory_size * sizeof(int)>>>(params…)
  - __global__ void Kernel(params) {
    extern __shared__ int a[];
    }
- Static
  - #define CONSTANT_SIZE 100
  - __global__ void Kernel(params) {
    __shared__ int a[CONSTANT_SIZE];
    }
- See: https://stackoverflow.com/questions/5531247/allocating-shared-memory

Is 512 threads enough?

- In the GPU world… NO
- Take for eg: blurring a 1024 x 1024 image: 1048576 pixels
- How to handle this?
  - Add more blocks (limit: 65 536 blocks!)
  - You have potentially 65 536 x 512 = 33 554 432 threads!
  - In reality, you will have up to three blocks per SM. This limit is based on the total number of threads per SM (1536 Fermi)
  - Currently the maximum number of SM is 30 in any card
  - Eg: if you schedule 65 536 blocks with 1024 threads each, you can process up to ~64 millions elements. Assuming each element is a single-precision floating-point number, requiring 4 bytes of data: you need around 256MB.
  - TODO: SEE EXAMPLE CODE PG 81

More on Blocks

- A Kernel is executed in batches of 32 threads called warps
- Threads are referenced according to 3D rectangular cuboid called a block with axes (x,y,z). At present, blocks are limited to a maximum 1024 threads. The maximum length of a block in the z direction is 64 threads and any of the axes for the block may have a length of 1 thread.

More on Grids

- A grid is simply a set of blocks it's like a 3D rectangular cuboid with axes x,y,z. The maximum number of blocks along a dimension is 65 536 and the minimum is 1
- Eg: Image 1920 x 1080 pixels: 2 073 600 pixels
  - Lets say we want 192 threads -> 6 * 32 threads = 6 warps per block
  - This gives 10 blocks on each row of the image (192*10 = 1920)
    - Using a thread size that is a multiple of the X axis and the warp size makes life a lot easier
  - Total number of blocks needed: 1080 x 10 = 10 800 blocks
  - Find the number of blocks by SM : (fermi 8/SM)
  - How many SM needed -> #blocks/#blocks_by_SM = 10800/8 = 1350 SM)
  - Find how many SM we have (Fermi 16 SM)
  - Find how many blocks will be used by SM -> #blocks/#SM = 10800/16 = 675 blocks/SM

Mapping order

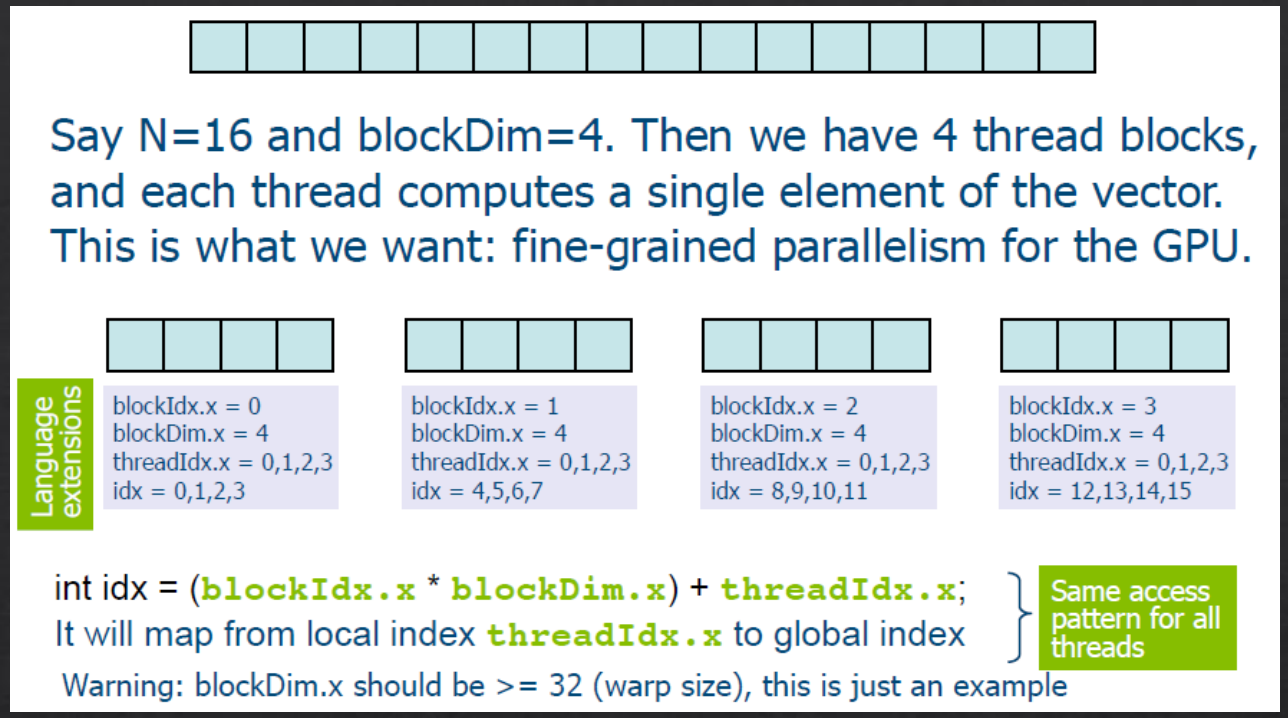
- The order of threads in blocks and the order of blocks in grids are both arranged according to a FORTRAN-style column-major format.
- The way you arrange the data in memory is independently on how you would configure the threads of your kernel.
- The memory is always a 1D continuous space of bytes. However, the access pattern depends on how you are interpreting your data and also how you are accessing them by 1D, 2D and 3D blocks of threads.
- dim3 is an integer vector type based on uint3 that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1.
  - dim3 blockdim(x,y,z);

Handling arbitrary vector sizes

- Typical problems are not friendly multiples of blockDim.x. We have to prevent accessing beyond the end of arrays:

- And now, update the kernel launch to include the "incomplete" block of threads:
  - mySuperKernel<<<(N + M-1)/256, 256>>>(A,B,C, N);
    - Number of threads/block = 256
    - Number of blocks = (N + M-1)/256 = (numberOfElements + NumberOfThreads-1)/NumberOfThreads
      = (N + 255)/256 blocks
  - ```
    __global__ void addKernel(int *c, const int *a, const int *b) {
      int i = threadIdx.x;
      c[i] = a[i] + b[i];
    }
    addKernel<<<1, size>>>>(d_c, d_a, d_b);
    ```
  - ```
    __global__ void addKernel(int *c, const int *a, const int *b) {
      int i = threadIdx.x;
      if (i < size)
        c[i] = a[i] + b[i];
    }
    addKernel<<< (size + (N-1))/N, N >>> (d_c, d_a, d_b);
    ```

A first example: increment a vector




```
void increment_cpu(float *a, float b, int N) {
  for (int idx = 0; idx < N; idx++)
    a[idx] = a[idx] + b;
}

void main() { …; increment_cpu(a, b, N); }
```

```
__global__ void increment_gpu(float *a, float b, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N)
    a[idx] = a[idx] + b;
}

Void main() {
  …;
  dim3 dimBlock(blocksize);
  dim3 dimGrid(ceil(N/(float)blockSize));
  increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

```
__global__ void increment_gpu(float *a, float b) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  a[idx] = a[idx] + b;
}

Void main() {
  int N = 16;
  dim2 blockDim(4,0);
  unsigned int numBytes = N * sizeof(float);
  float * h_a = (float*) malloc(numBytes);
  float * d_a = 0; cudaMalloc(&d_A, numBytes);
  cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
  increment_gpu<<<N/blockSize, blockSize>>>(a, b);
  cudaFree(d_A);
}
```

Steps for building the CUDA source code

1. Identify the parts with a good potential to run in parallel exploiting SIMD data parallelism
2. Identify all data needed for the computations
3. Move data to the GPU
4. Call to the computational kernel
5. Establish the required CPU-GPU synchronization
6. Transfer results from GPU back to CPU
7. Integrate the GPU results into CPU variables

Coordinated efforts in parallel

- Parallelism is given by block and threads
- Threads within each block may require an explicit synchronization, as only within a warp it is guaranteed its joint evolution (SIMD):

```
a[i] = b[i] + 7;
_syncthreads();
x[i] = a[i-1]; // the warp 1 reads here the value of a[31] which should have been written by warp 0 before
```

- Kernel borders place implicit barriers:
  - Kernel1 <<<nblocks, nthreads>>> (a, b, c);
  - Kernel2 <<<nblocks, nthreads>>> (a,b);
- Blocks can coordinate using atomic operations:
  - Eg: Increment a counter “atomicInc();”

Why do we need threads?

- Threads add a level of complexity without contributing with new features.
- However, unlike parallel blocks, threads can:
  - Communicate (via shared memory)
  - Synchronize (for example, to preserve data dependencies)
- We can share data between threads by using shared memory within the block
  - Shared memory is user-managed: declare with __shared__ prefix in a kernel
  - Data is allocated per block
  - Shared memory is extremely fast (500x faster than global memory)
  - Shared memory is more versatile than registers (registers are private to each thread)

Limits of the shared memory and registers

- Kepler: max of 64K registers and 48Kb of shared memory per SM
  - 2 blocks per SM: max 32K registers, 24Kbytes of shared memory
  - 3 blocks per SM: max 21,3K registers and 16Kbytes of shared memory
  - 4 blocks per SM: max 16 K registers and 12Kbytes of shared memory

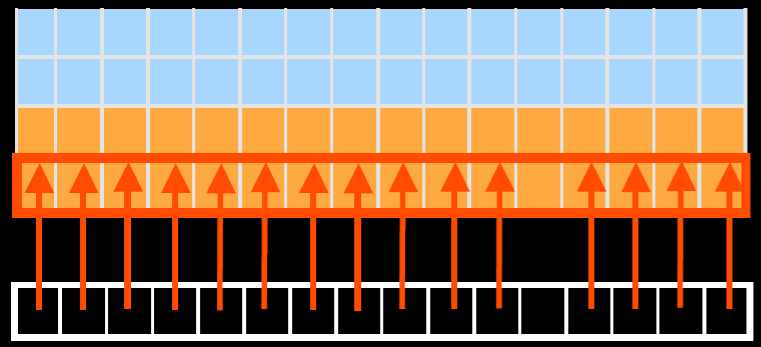
- You can find on internet the « CUDA Occupancy Calculator » to figure out the limits!

- More on the shared memory on the next exercise session!

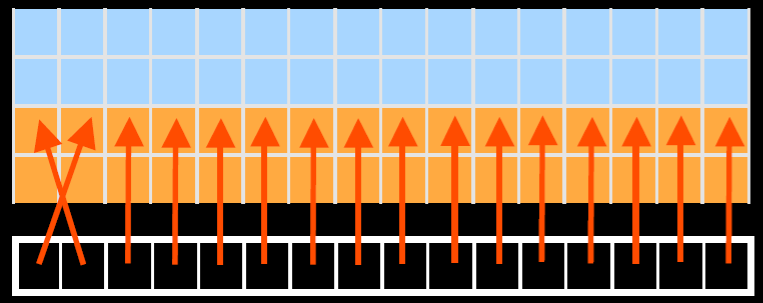
Coalescence

- **K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate**

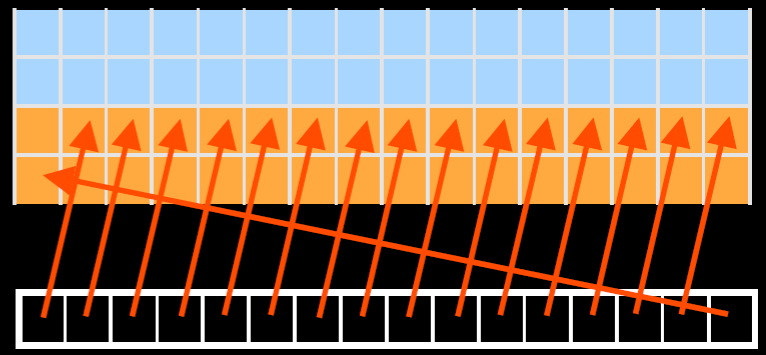
Coalesces – 1 transaction



Out of sequence – 16 transactions



Misaligned – 16 transactions



Divergences

- Control Flow Divergences
  - Avoid divergences (conditionals where threads in a warp do different things)
- Data Address Divergences
  - Hardware is optimized for accessing contiguous blocks of global memory when performing loads and stores.
  - Global memoy blocks are aligned to multiples of 32, 64, 128 bytes
  - If requests from a warp span multiple data blocks, multiple data blocks will be fetched from memory.
  - Entire block is fetched even if only a single byte is accessed, which can waste bandwidth
  - Hardware may need to issue multiple loads and stores whan a warp accesses addresses that are far apart

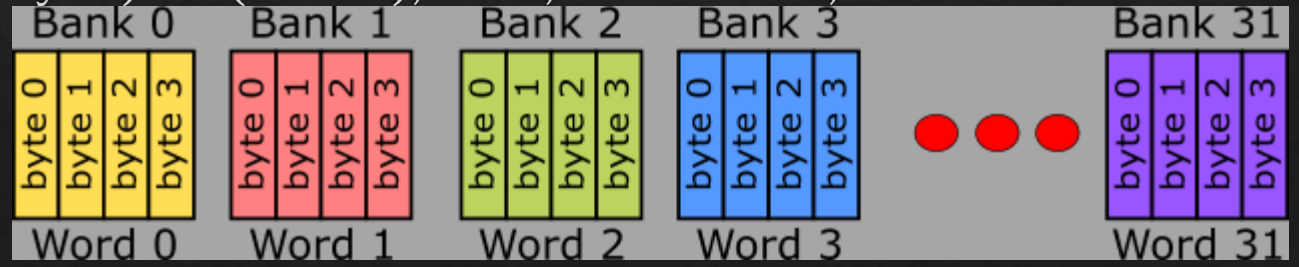
Occupancy

- Maximize the arithmetic intensity
  - Math/memory
  - You want to maximize the compute operations per thread
    - Every thread are working
    - Avoid divergences
    - Every thread terminate at approximatively the same time
  - Or
  - You want to minimize the time spent on memory per thread
    - Memory coalescing, spatial, temporal optimizations, etc..

Bank Conflicts

- To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called memory banks
- Each bank can only address one dataset at a time
- Bank conflicts arise because of some specific access patterns of data

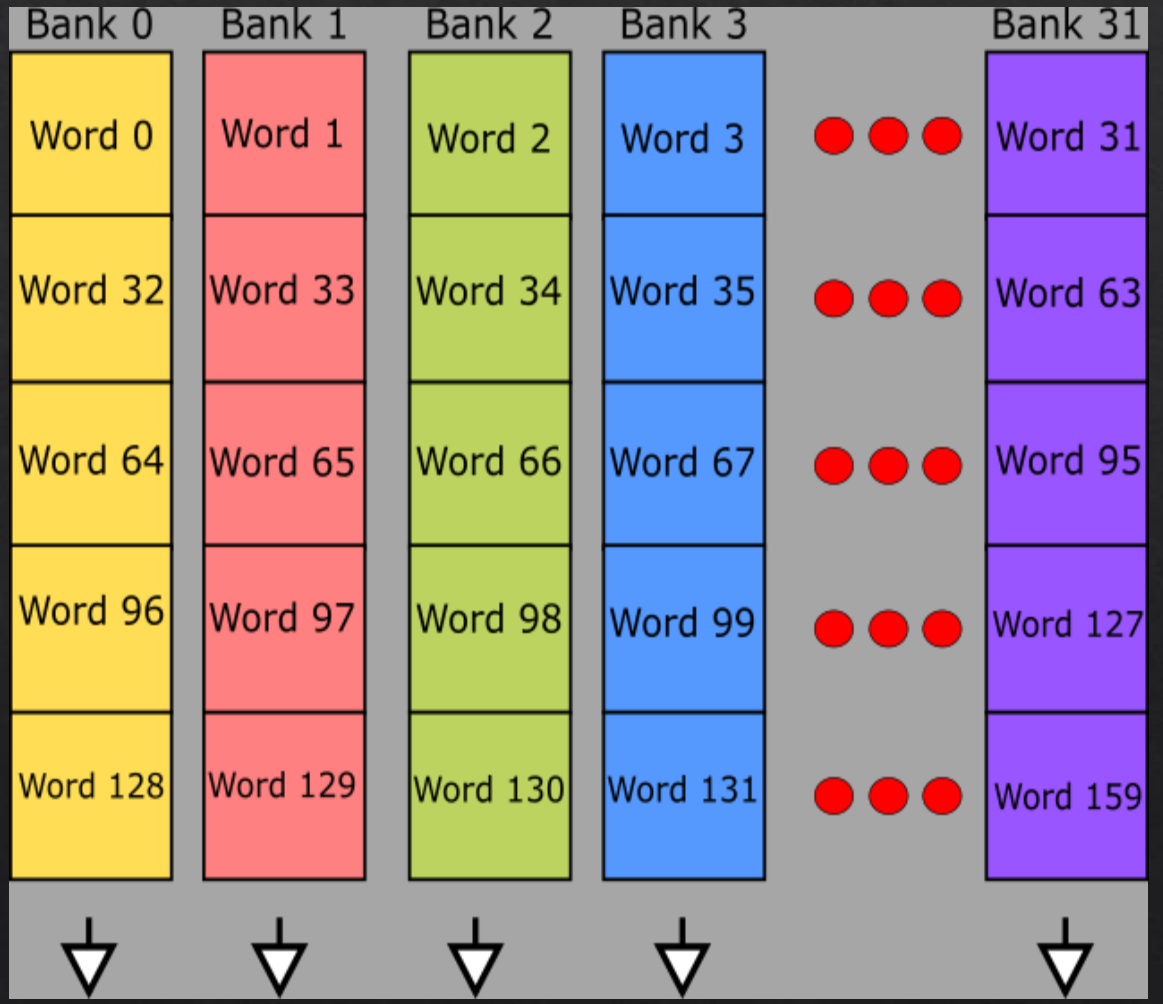
- We have 32 banks in the shared memory*
- Everything is split in words (4bytes) int (32 bits), float, half double, …
- word[0] is in bank 0
  word[1] is in bank 1
  …



*The programming guide indicates 16 banks for compute capability 1.x, and 32 banks for compute capability 2.x and 3.x.

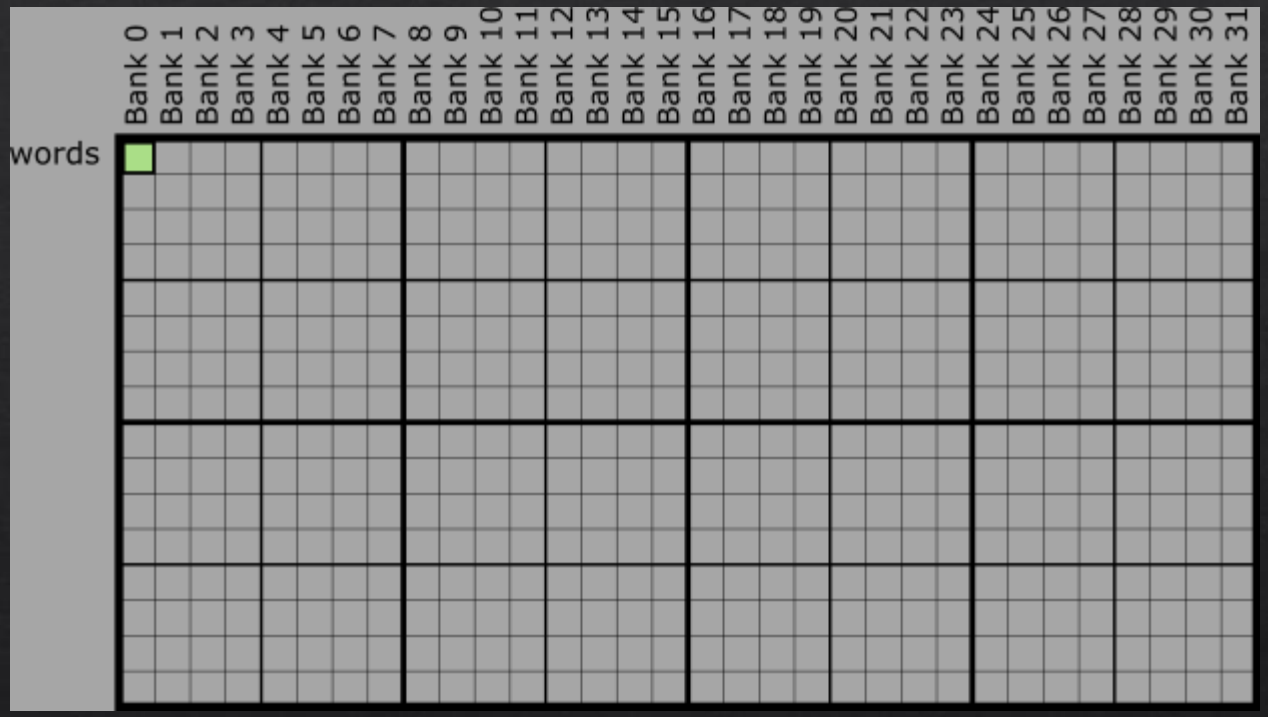
Bank Conflicts

- If the threads of a warp all request the same word, we have a broadcast
- If several threads request the same value from a particular bank, we have a multicast
- Eg:
  - 5 threads reading bank 2 word 2
  - 5 threads reading bank 30 word 30
    - Same word for each 5 threads
- The moment two or more threads reads different words from the same bank, we have a bank conflict



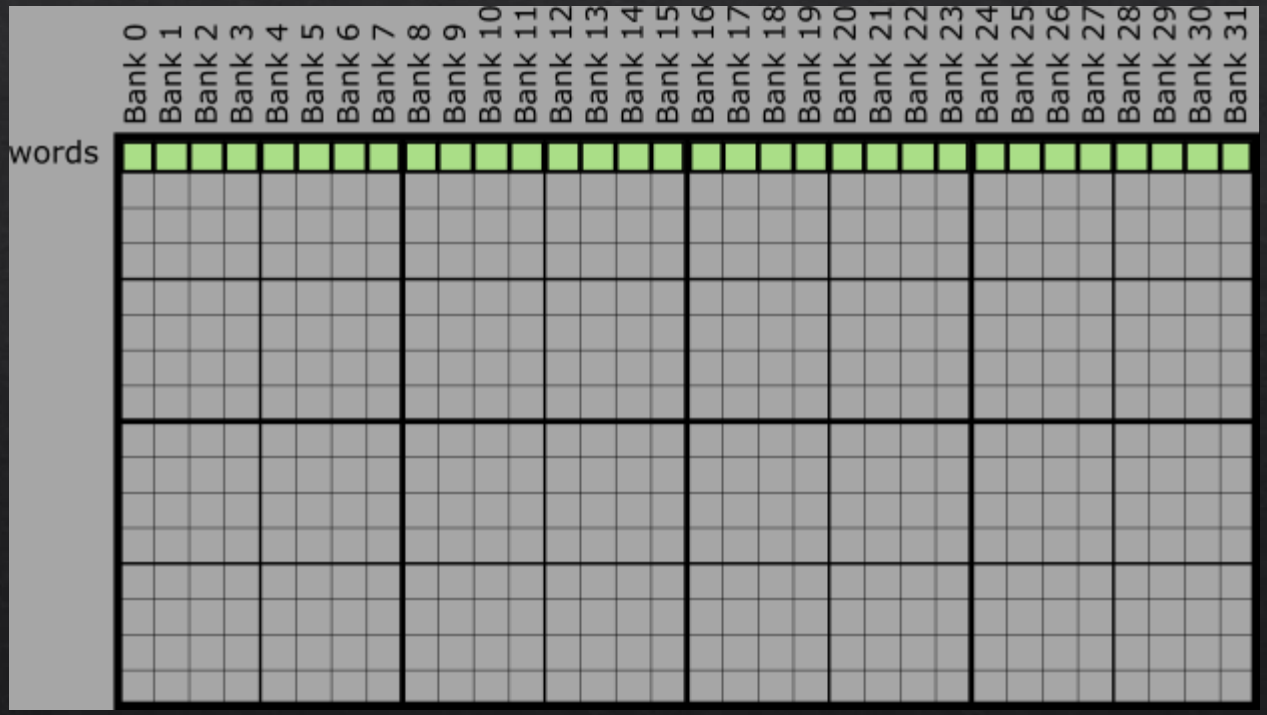
Bank Conflicts - Patterns

- Broadcast: Every expression which results in a single value for all threds in each warp
  - arr[threadIdx.x*0]
  - arr[12]
  - arr[blockIdx.x*3]



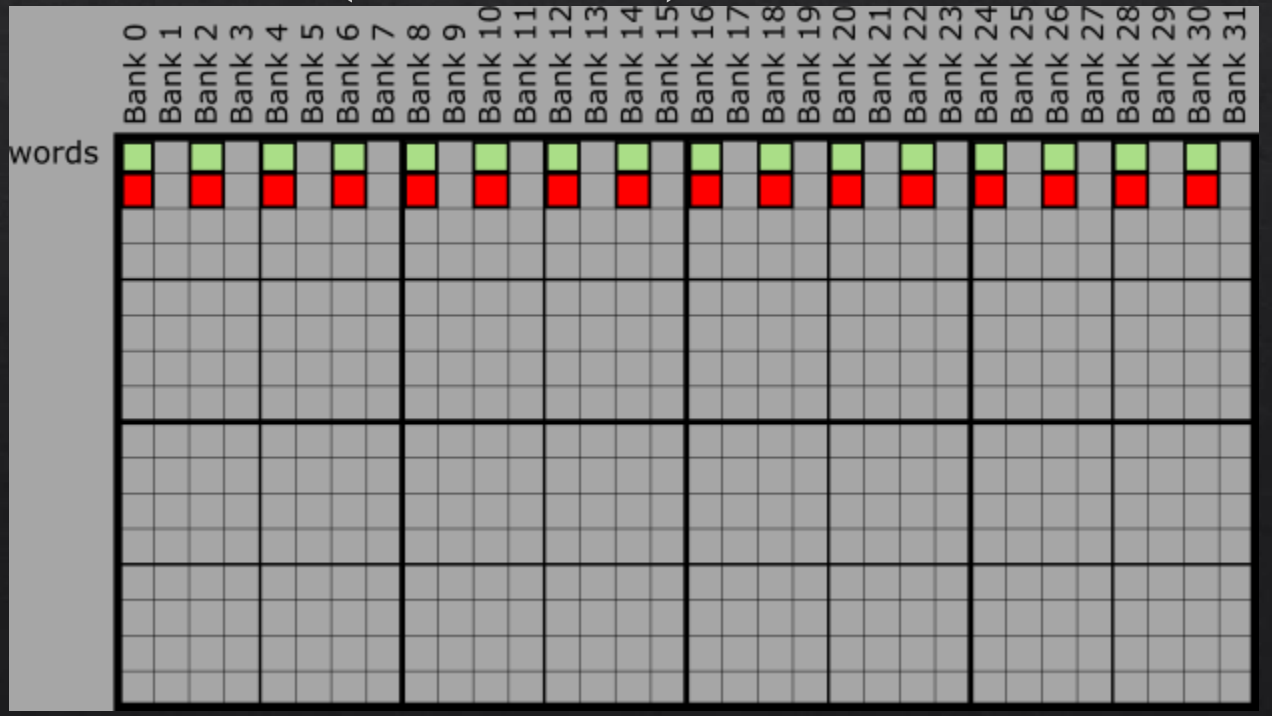
Bank Conflicts - Patterns

- Each thread requests a word based on its threadIdx
  - arr[threadIdx.x]
- Each bank is only accessed for a single value



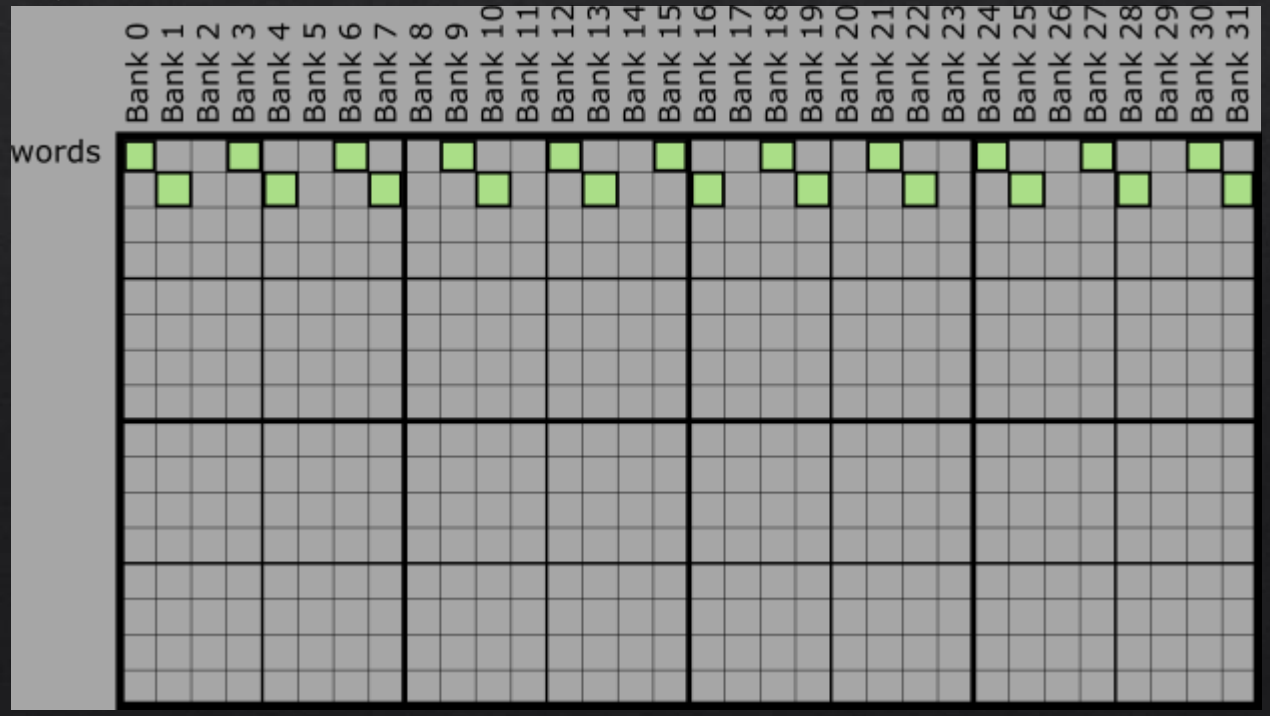
Bank Conflicts - Patterns

- Threads of a warp request words which are double of their threadIdx
- We are working with doubles, structures of 2 floats, 8 bytes data, …
  - arr[threadIdx.x*2]
- We have a 2way bank conflict! (twice as slow!)



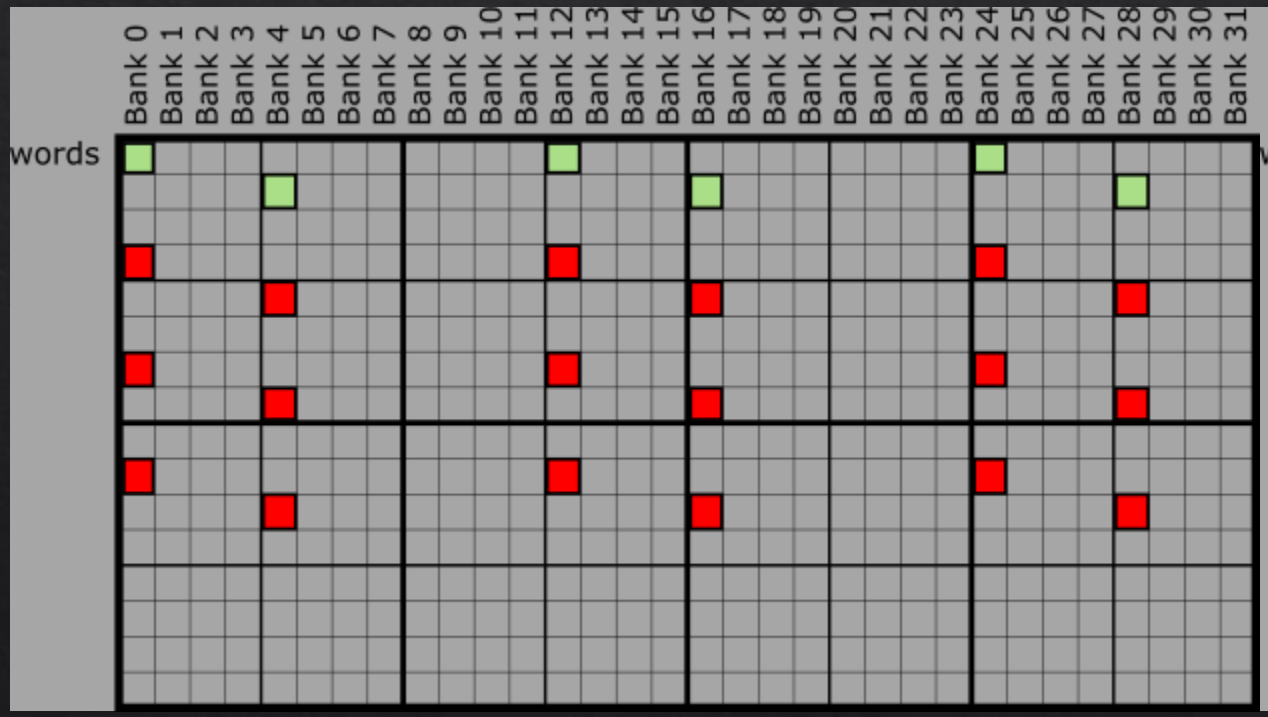
Bank Conflicts - Patterns

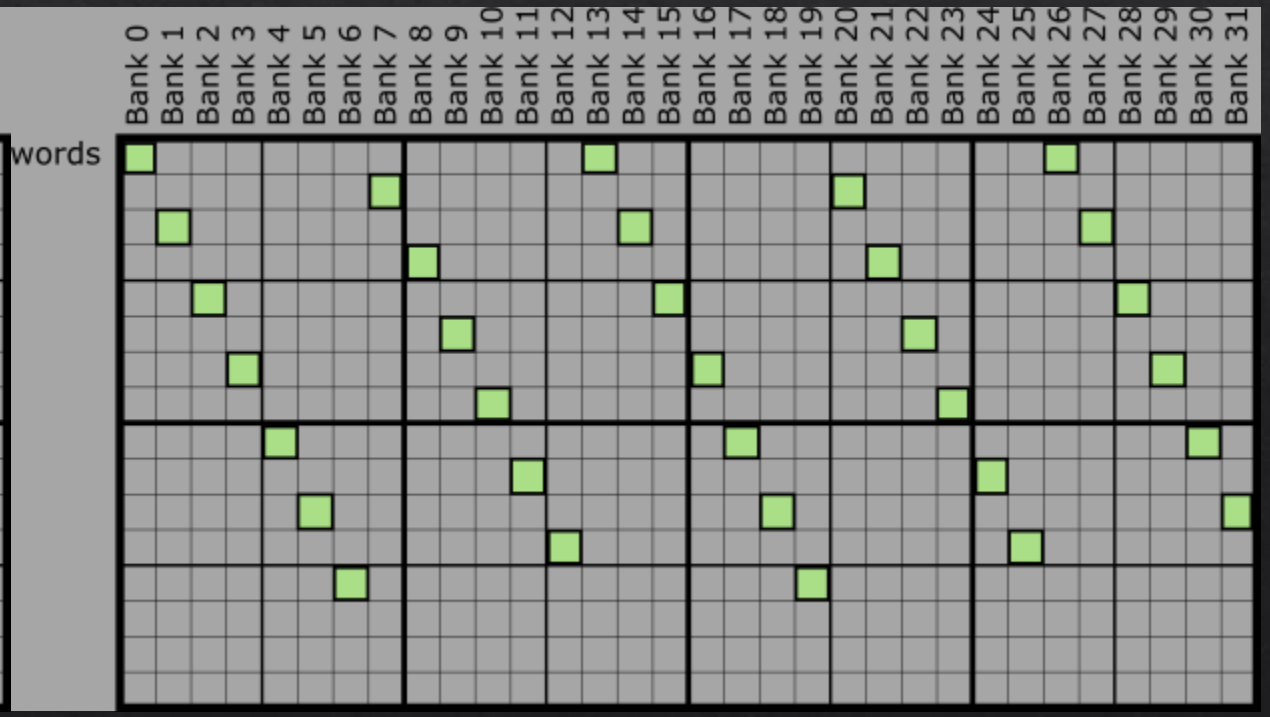
- When we are working with 12 bytes data (eg: Spatial points X,Y,Z, floats)
- We don’t have bank conflicts!
- So when we use doubles we can add an extra padding to avoid bank conflicts!
- ! This is not always the case, on some architectures, doubles are as fast as this!



Bank Conflicts - Patterns

- 4 way bank conflicts – With 12 bytes apart
  - arr[threadIdx.x*12]
- We can solve this by adding a pad of 1



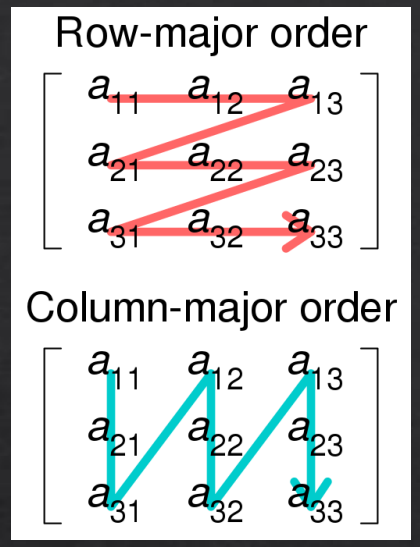


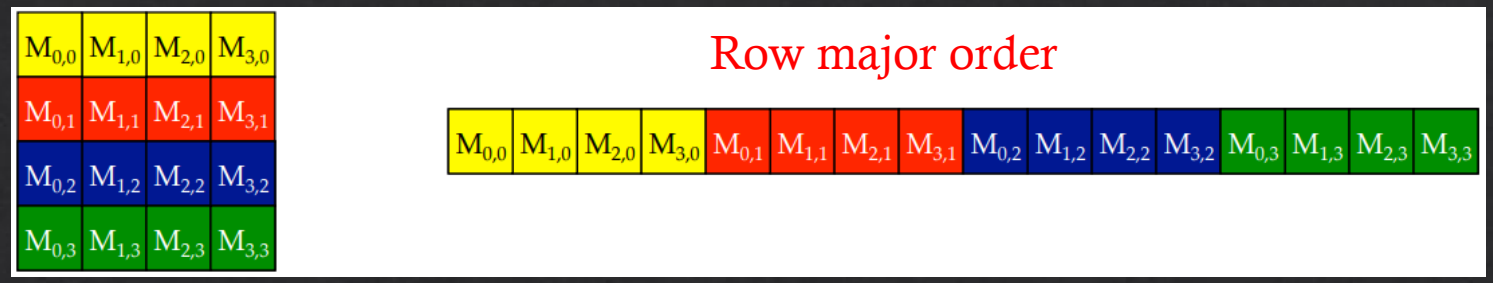
Bank Conflicts – Final words

- Latency hiding
  - Even with bank conflicts, if many threads are running on a SM, the scheduler can simply switch to another warp while the bank figure out the requested data.

- The only real way to know if bank conflicts are slowing an algorithm is to experiment with padding and access patterns.

Appendix: Row-major vs Column-major





- Generally we cannot use two-dimensional indices A[row][column] to access matrices in GPU
- We will need to know how the matrix is laid out in memory and then compute the distance from the beginning of the matrix
- Eg
  - int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
  - int linear_index_col_major = x + y * ROW_SIZE;
  - int linear_index_row_major = y + x * COL_SIZE;

Appendix: Rules of thumb

- Prefer thread block sizes that result in mostly full warps
  - Bad: kernel<<<N, 1>>> (…)
  - Okay: kernel<<<N/32, 32>>> (…)
  - Better: kernel<<<N/128, 128>>>(…)
- Prefer to have enough threads per block to provide hardware with many warps to switch between
- When number of threads is not a multiple of preferred block size, insert bounds test into kernel
  - ```
    __global__ void kn(int n, int* x) {
      int i  = threadIdx.x + blockDim.x * blockIdx.x;
      if ( i < n) { [CODE] }
    }
    ```
    ```