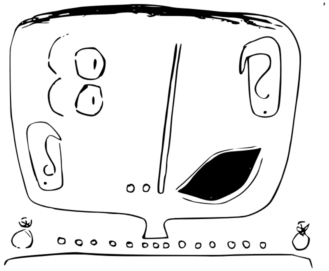


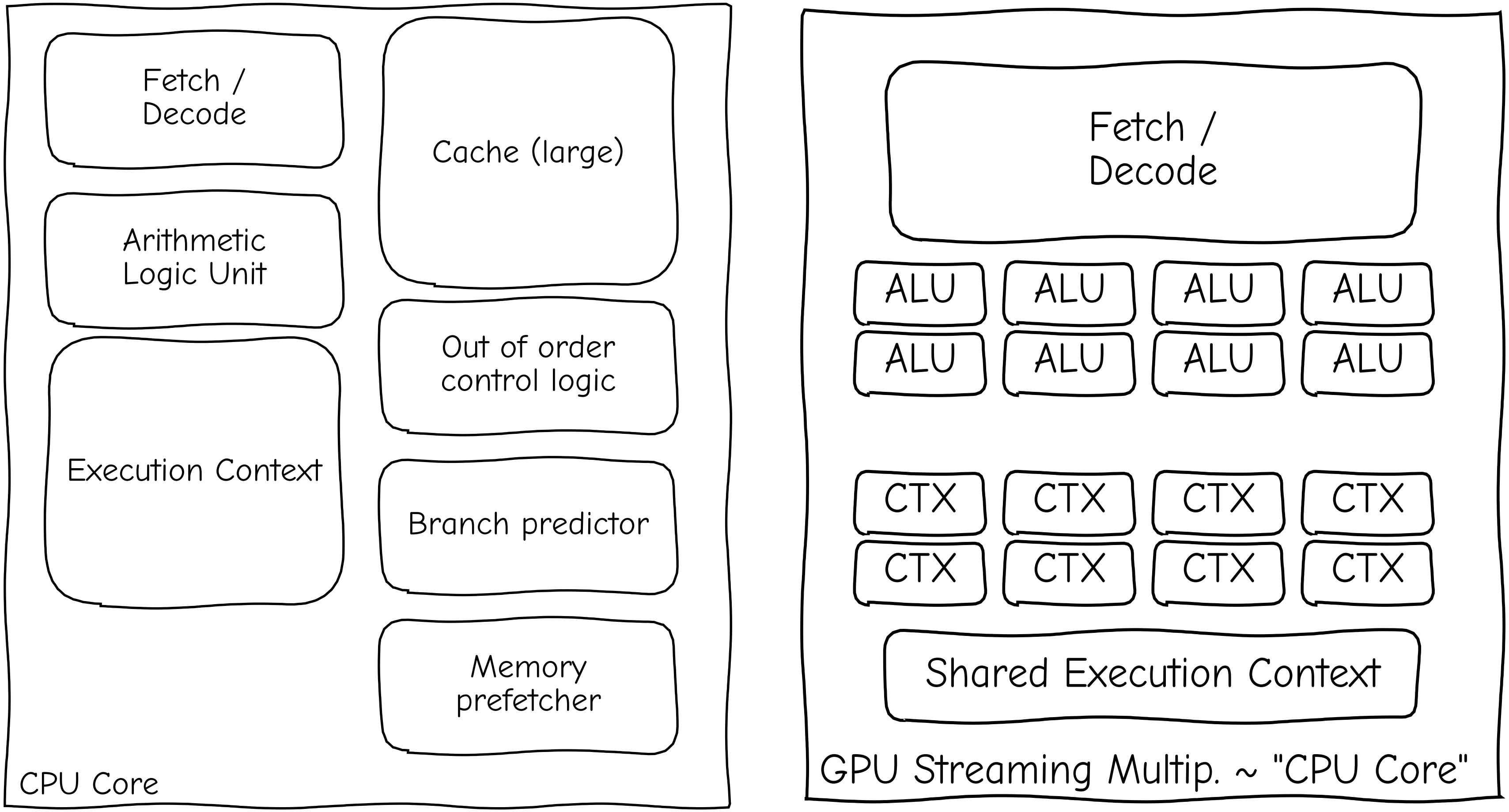
CUDA-Mode Session 4:
Compute and Memory basics
(based on ch 4 + 5 of the PMPP book)

3 February 2024

Thomas Viehmann, tv@lernapparat.de, MathInf GmbH
And starting next week: ...

Chapter 4: Compute Architecture and Scheduling
aka: How to keep all of the GPU busy

CPU core ~ GPU Streaming Multiprocessor (comic version)



Actually 1SM ~ 4 “CPU-Cores”,
originally (<=Pascal) GPU threads in a warp shared a Program Counter, now each has one

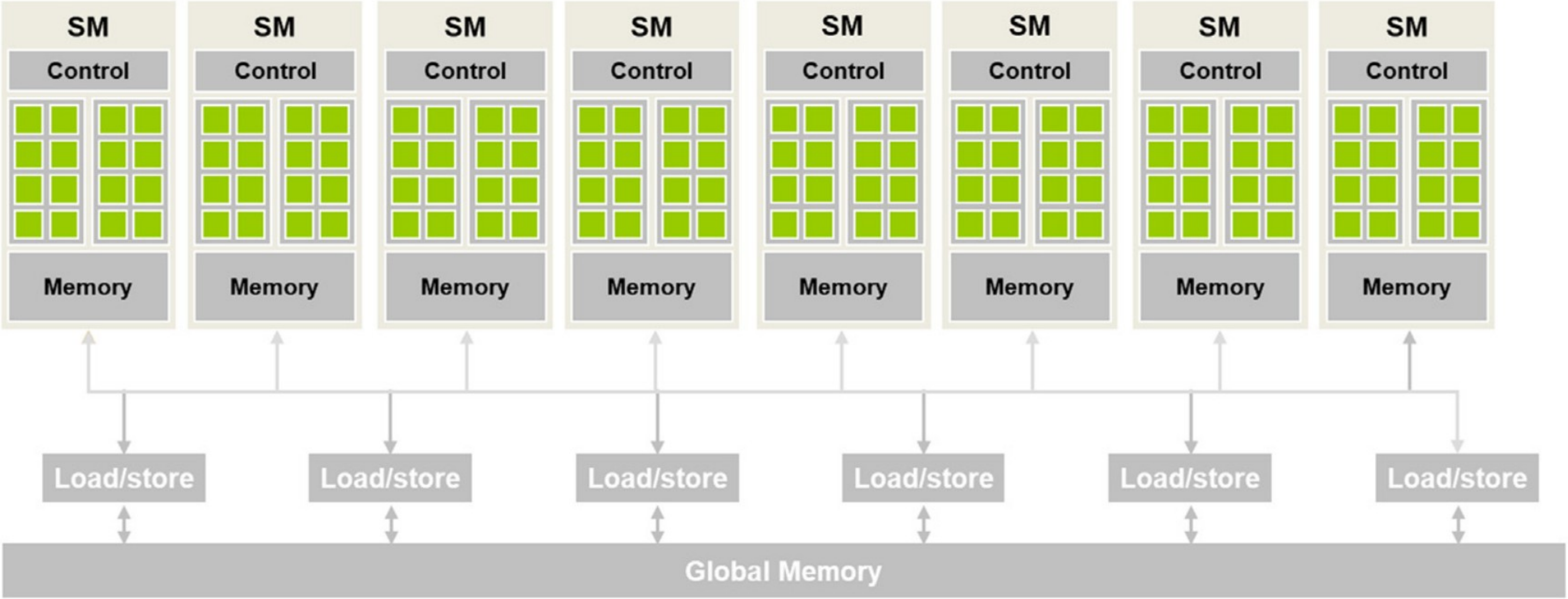


FIGURE 4.1

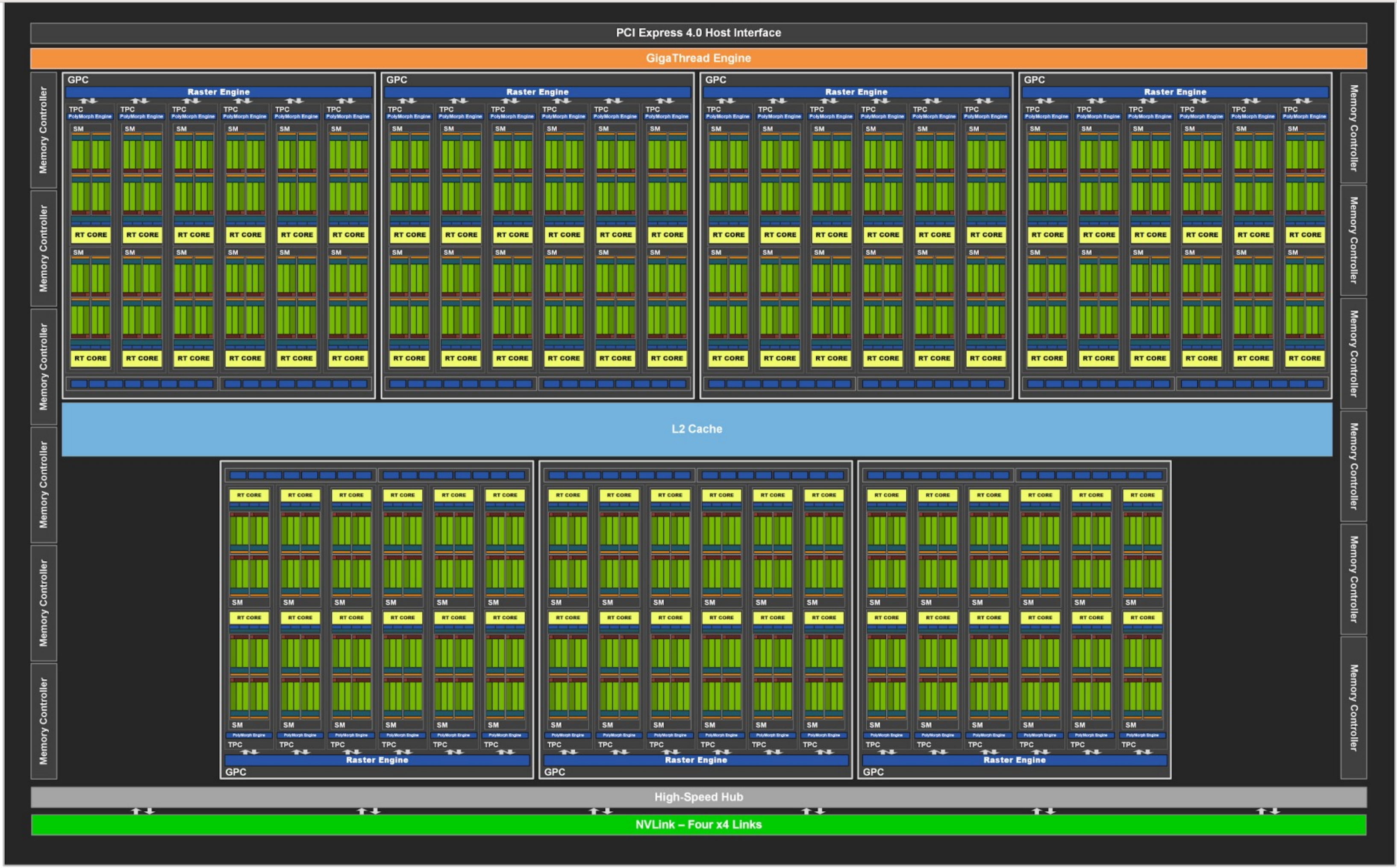
Architecture of a CUDA-capable GPU.

Inside the GPU
(here RTX3090)

82 SMs
(=Streaming
Multiprocessor)

1 common
L2 cache

Almost no FP64
in consumer/
non-datacenter
GPUs
(2 FP64 per SM vs.
128 FP32)



Note: The GA102 GPU also features 168 FP64 units (two per SM), which are not depicted in this diagram. The FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations. The small number of FP64 hardware units are included to ensure any programs with FP64 code operate correctly, including FP64 Tensor Core code.

Streaming Multiprocessor

- A thread block is assigned to one SM
(max 1536 threads assignable to SM)
NO control which block in a grid when where
(ok, in Hopper+ we can have thread block groups)

- 4 warps or”part-warp” can compute at a cycle, those SHARE one instruction (but Volta+ does have per-thread program counter)

- 32 FP32 units (one per thread),
16 of which know INT32

- 16k 32-bit registers shared between things scheduled on the same block)

- L1 cache and shared memory share hardware (128KB) directly on the SM
shmem can be 0/8/16/32/64/100KB
L1 Cache the remainder (>=28KB)

computation units

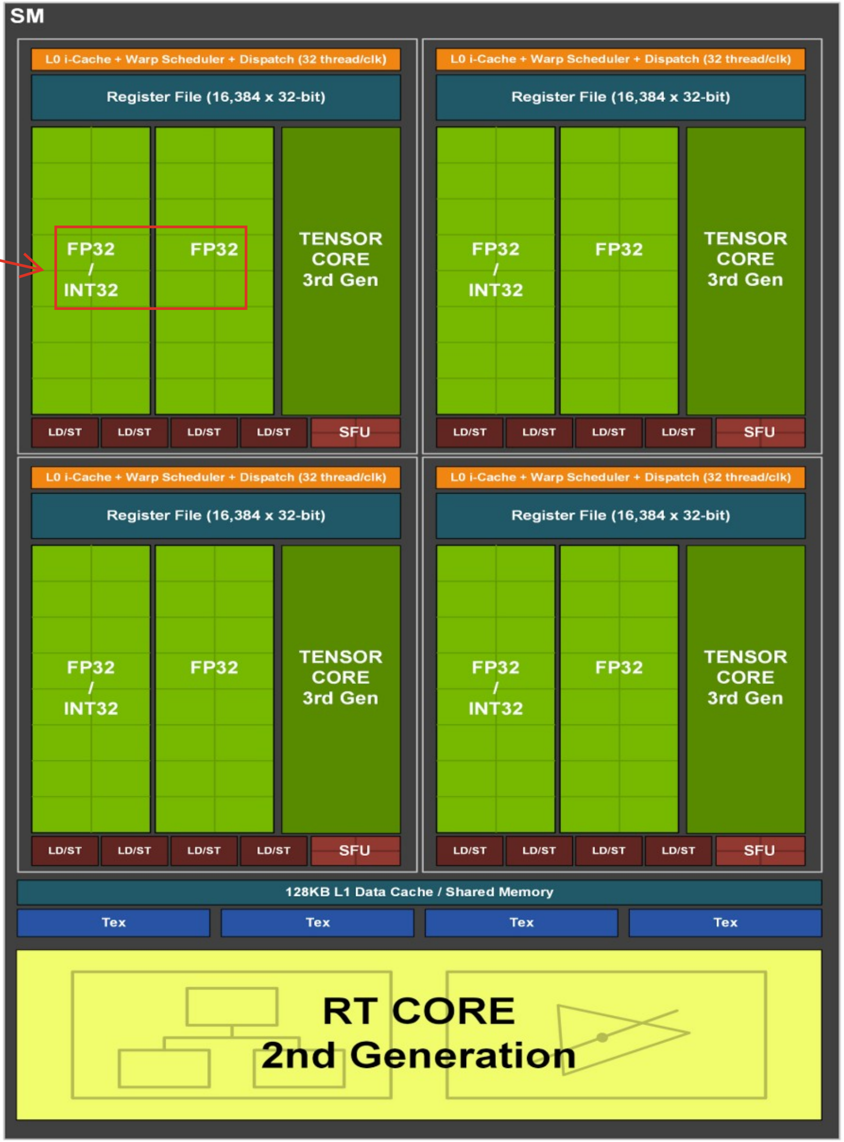
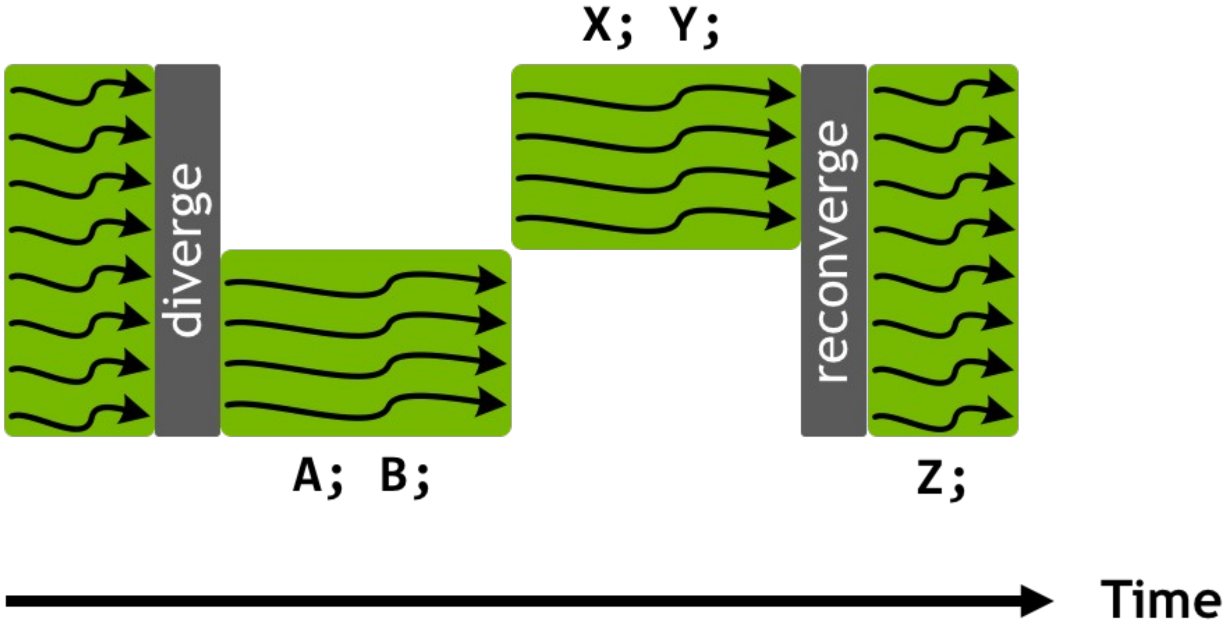

SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

Figure 3. GA10x Streaming Multiprocessor (SM)

Warp Divergence (<= Pascal)

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



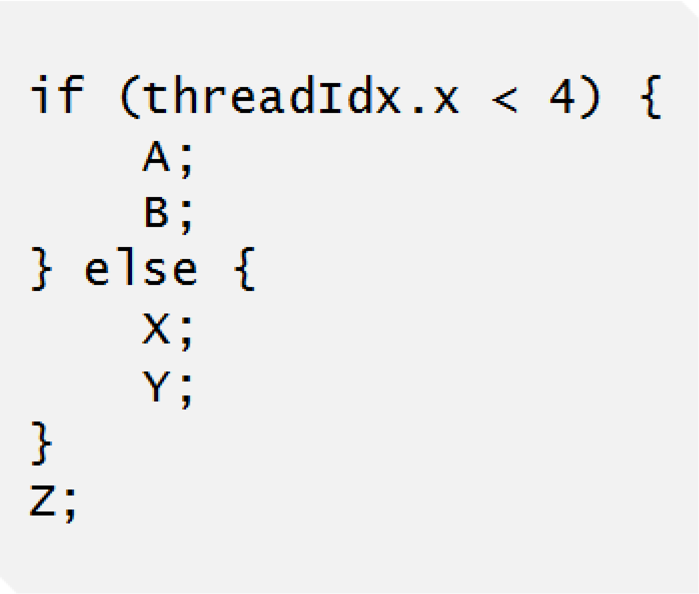
Old way: threads share program counter, but have an "active mask".

Careful not to do inter-thread communication / sync inside if (without mask)!

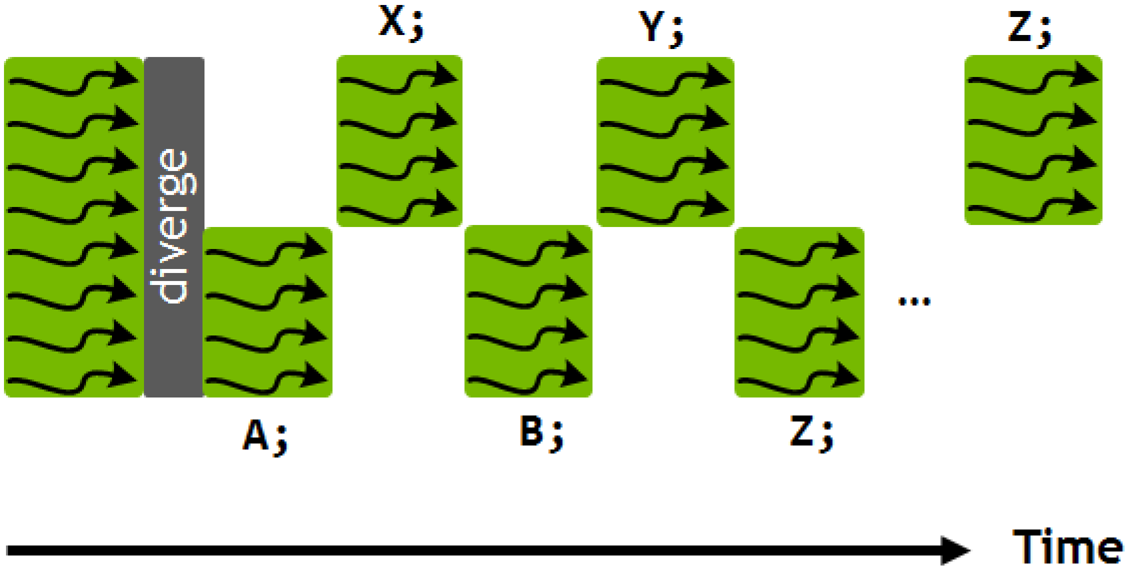
Automatic reconvergence.

N.B.: As there are load/store instructions the typical pattern (cond ? x[i]: 0f) will not cause divergence.

Warp Divergence (>= Volta)



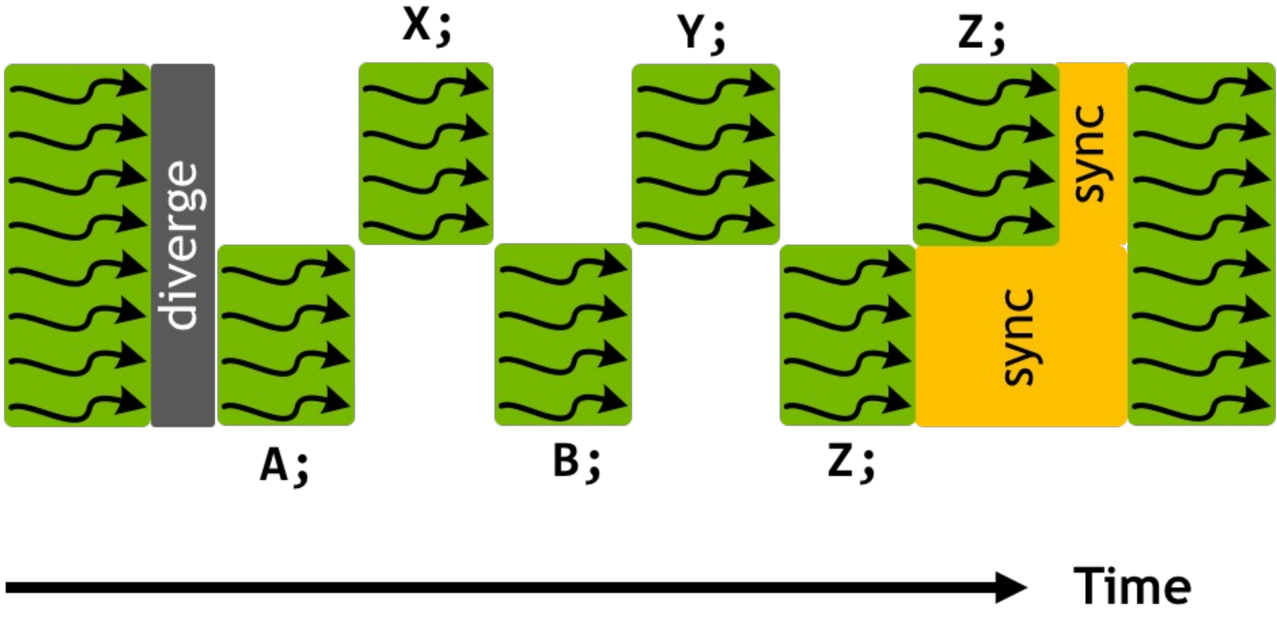
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



New way: threads have individual program counter, GPU will threads of a warp by PC for execution.
No automatic reconvergence, but potentially better latency hiding (e.g. if both parts load from dram).

Warp Divergence (>= Volta)

```
if (threadIdx.x < 4) {
     A;
     B;
} else {
     X;
     Y;
}
Z;
__syncwarp()
```



No automatic reconvergence → use __syncwarp to rejoin warp.
Inter-Thread communication (e.g. shuffle) will also sync the participating threads.
__syncthreads() syncs an entire block, not just warps.

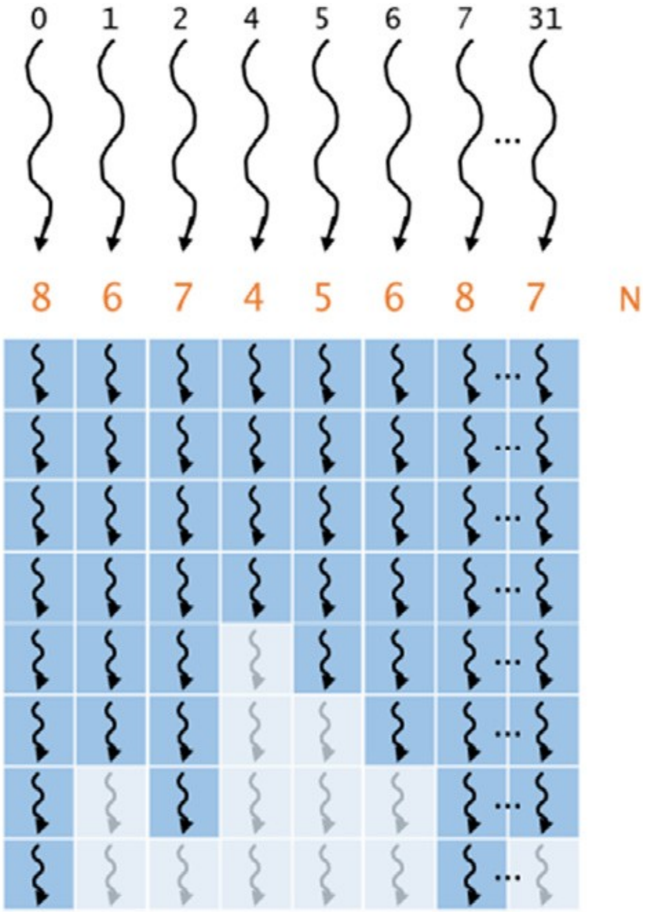
We’re not done with a warp until everything is done

(note: this is a pre-volta image)

```
N = a[threadIdx.x];
for(i = 0; i < N; ++i) {

    A

}
```



Loops with variable upper bound also cause divergence.

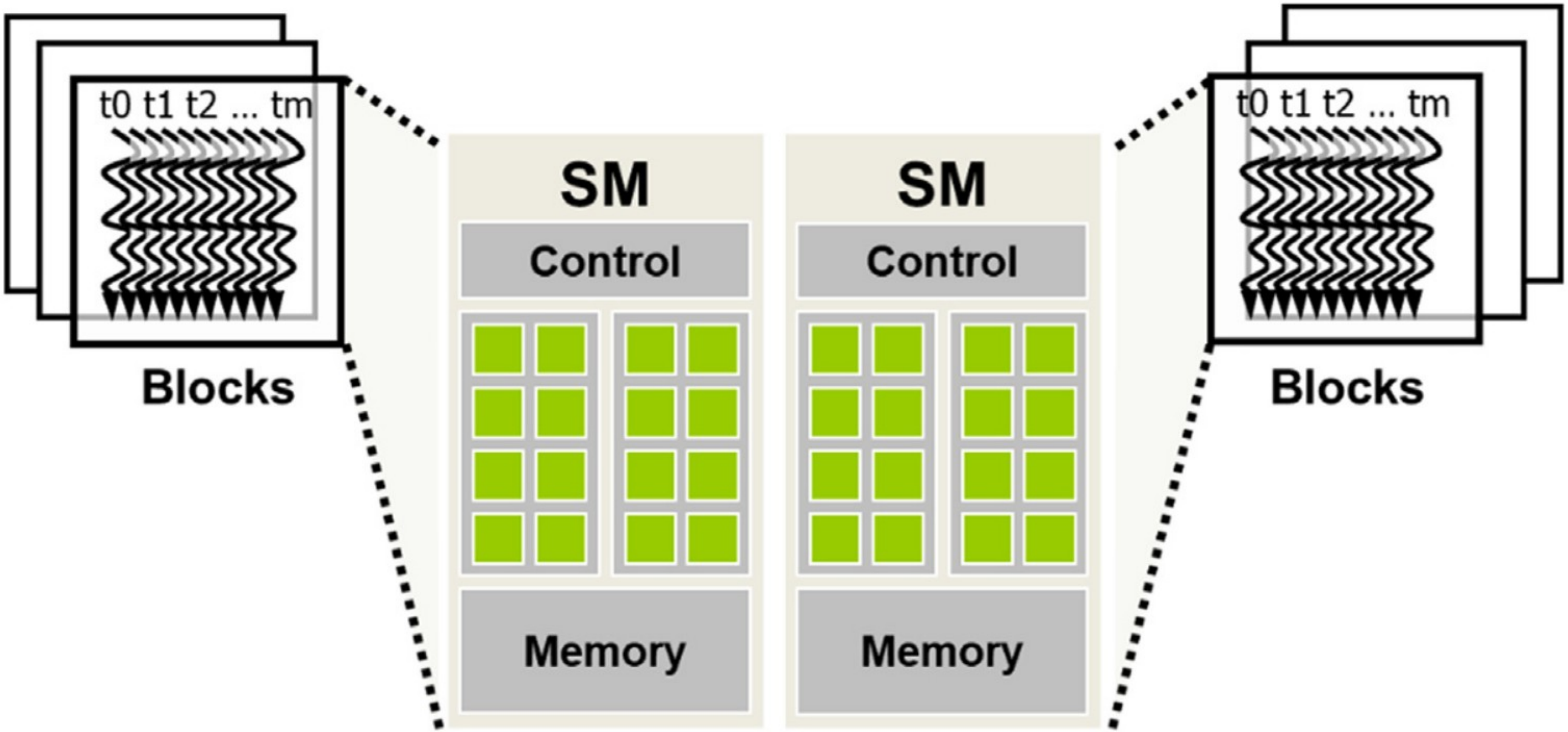


FIGURE 4.2

Thread block assignment to streaming multiprocessors (SMs).

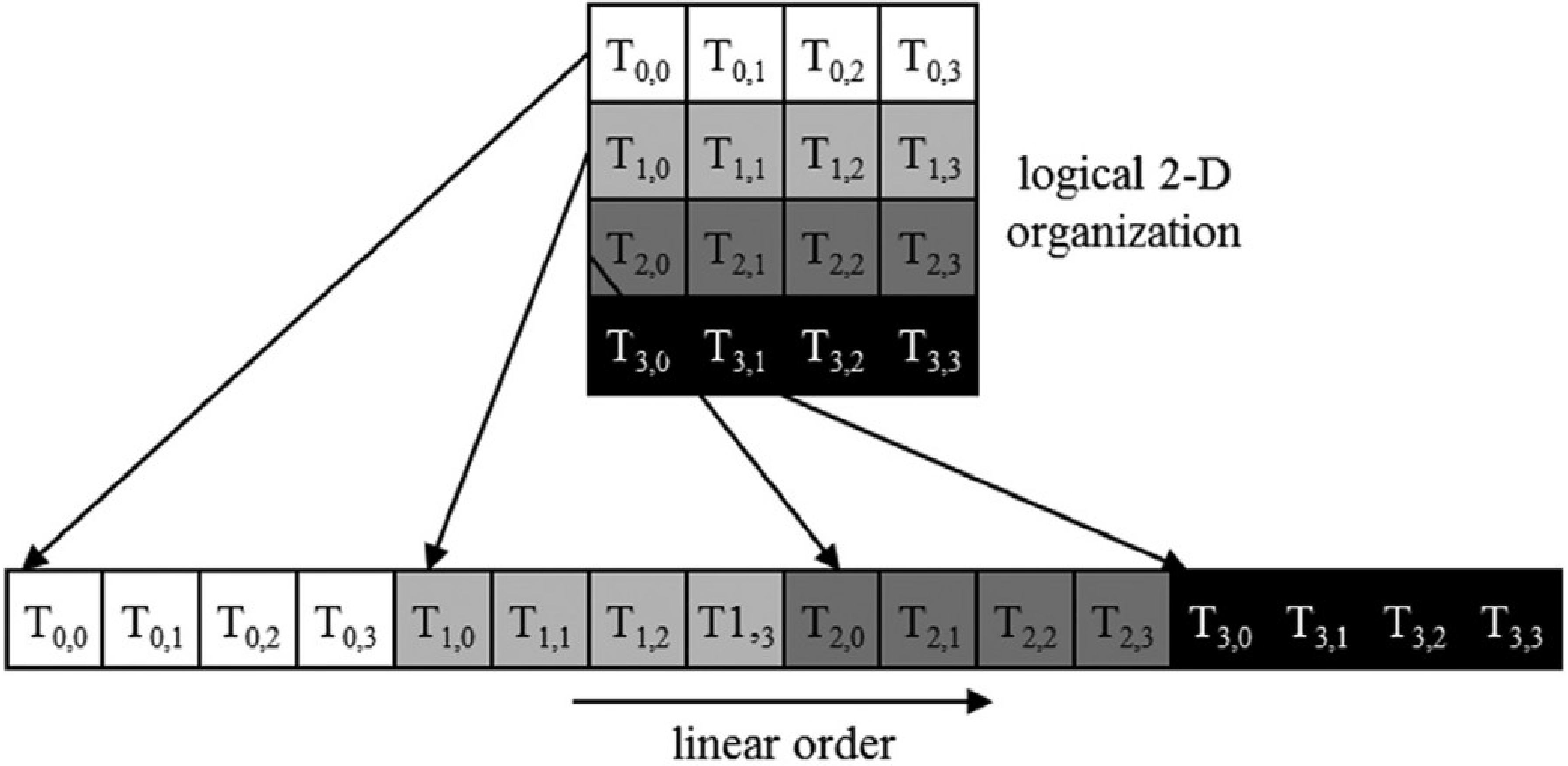

$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
logical 2-D organization
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$ $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ T1,3 $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$ $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
linear order

FIGURE 4.7

Placing 2D threads into a linear layout.

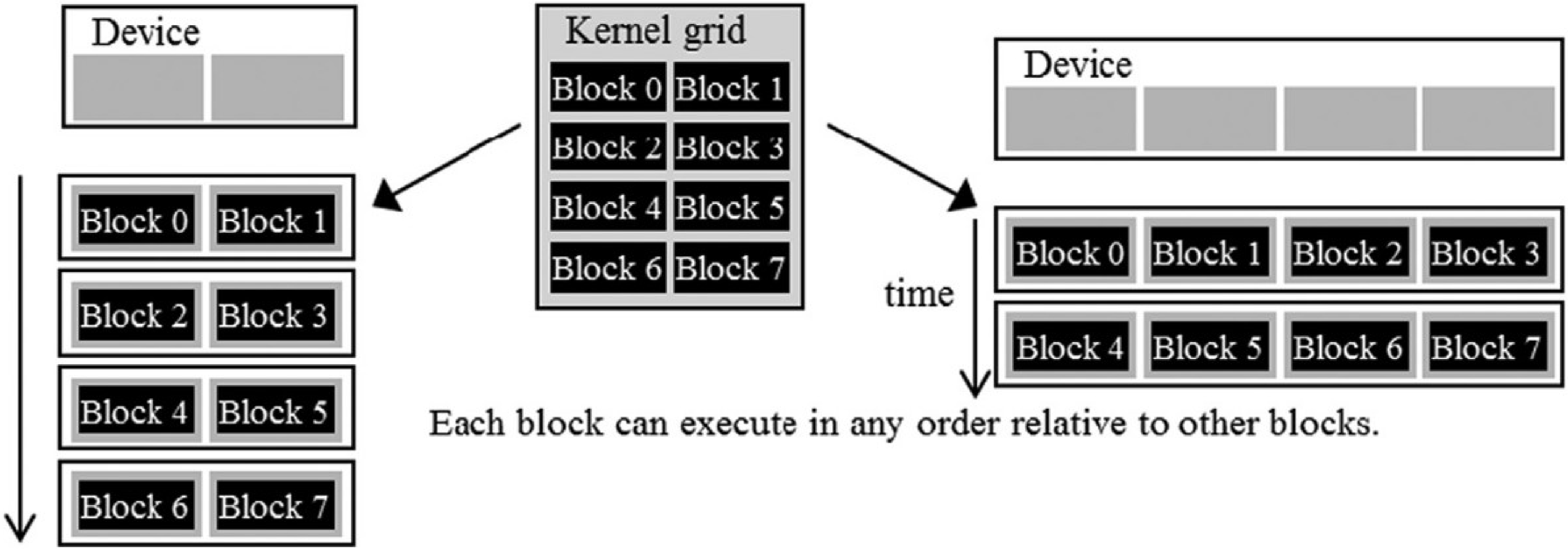


FIGURE 4.5

Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

Getting good occupancy – balance resources

- Have 82 SM → many blocks = good
  (for comparison Jetson Xavier has 8 Volta SM)
- Can schedule up to 1536 threads per SM
  → power of two block size <512 desirable
  (some other GPUs 2048)
- Avoid divergence to execute an entire warp (32 threads) at each cycle
- Avoid FP64/INT64 if you can on Gx102 (GeForce / Workstation GPUs)
- Shared Memory and Register File → limits number of scheduled on SM
  (use __launch_bounds__ / C10_LAUNCH_BOUNDS to advise compiler of # of threads for register allocation, but register spill makes things slow)
- Previously, there was an excel sheet for occupancy calc, now it is in the Nsight Compute
- Use torch.cuda.get_device_properties(<gpu_num>) to get properties (e.g. max_threads_per_multi_processor)
  even more in CUDA than in PyTorch (
  https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/html/group__CUDART__DEVICE_g5aa4f47938af8276f08074d09b7d520c.html
)

Chapter 5: Memory architecture and data locality

aka the basics of getting fast kernels

How do PyTorch programs spend their time

At a very high level:

- Python processing,
- Data “administrative overhead” (allocate Tensor structures etc.),
- Data acquisition (I/O), ← check this before diving into GPU optimization
- The GPU computation
  - Fixed cost (kernel launches etc.)
  - Memory access (read inputs / write results), ← This is us, chapter 5, i.e. now.
  - "real" computation (“FLOPs”). ← occupancy a key part, chapter 4

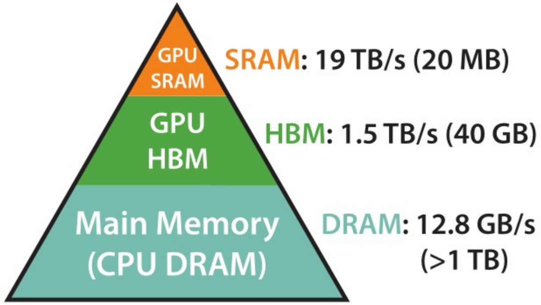
Thomas Rules of Thumb

- As long as you don’t have close to 100% GPU utilization in nvidia-smi, work on data acquisition etc.
- As long as you have Tensors with a few 100s of elements, “Python is slow” and data administrative over head is single digit percentages.
- Obviously the algorithms also matter (parallel algorithms in the following chapters)

But we are beyond this here!

Memory access as a bottleneck

- Eager PyTorch does “load input, compute, store output” for each operation.
- Obviously faster if we can combine kernels:
  ”load inputs, compute, compute, compute, compute, compute, compute, store outputs”
- Has been in the focus of PyTorch for a long time
  - original reason for the PyTorch JIT – fusing pointwise operations into one kernel has been key e.g. to get LSTMs close to CuDNN perf.
  - 2nd gen PyTorch JIT fusers added contractions etc. (NVFuser going beyond PyTorch in https://github.com/NVIDIA/Fuser and learning new things every week)
  - Today’s inductor / Triton based optimizations are also partly with that, but supports more complex ops
- Also core ingredient of flash attention (along with other things)
  (graphic on the right from Dao et al., Flash Attention…) →



Memory Hierarchy with Bandwidth & Memory Size

Memory and computation

- Take rgb2gray: For each pixel
  - Load 3 bytes,
  - Compute I (1 mul + 1 add in 32 bit integer)
  - compute 5 ops (3 mul, 2 add – ideally in 32 bit) + data conversion,
  - Store 1 byte

What speed can we expect maximally for a 2048 x 2048 image?
RTX3090:

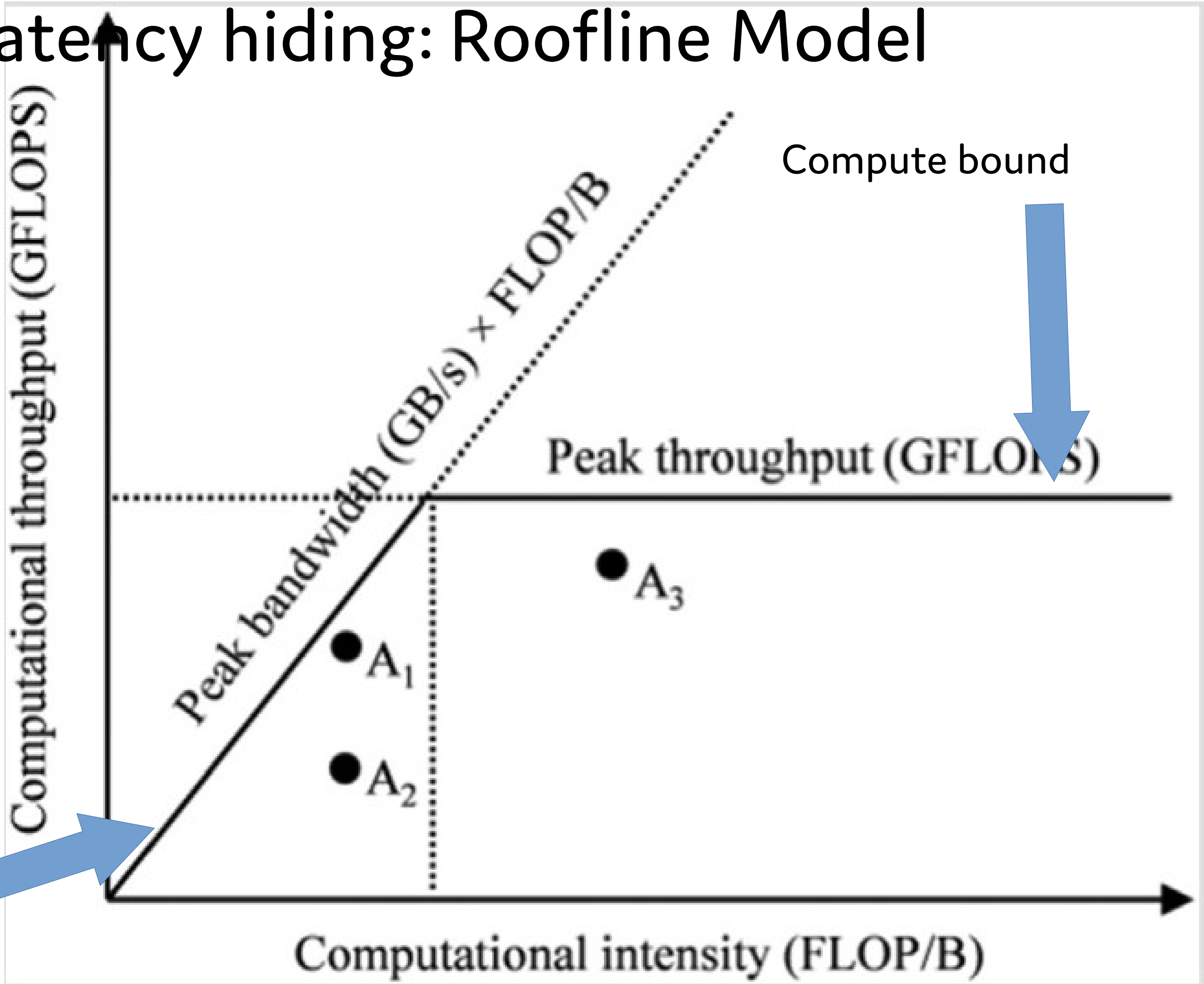
- NVidia lists ~900GB/s memory bandwidth 4*4M bytes transfer: ~18µs (16M/900GB) s ("speed of light")
- Compute: 35.6FP32 TFLOP/s 16.8 Int32 TFLOP/s ~2µs (generous).
  (Does not include latency hiding across warps.)
- If measuring with %timeit: Kernel Launch: ~3µs (see with empty kernel)
- (don't forget the element size if using 32bit or 16bit)

Measured time of kernel (with "f" for the consts): 27µs ~ 74% of theoretically possible.
Note: I have made an "out" function to split out the memory allocation, but it is relatively fast as long as you have the caching allocator.
N.B. Alignment helps… (try copy kernel with offset)

Similar model with latency hiding: Roofline Model

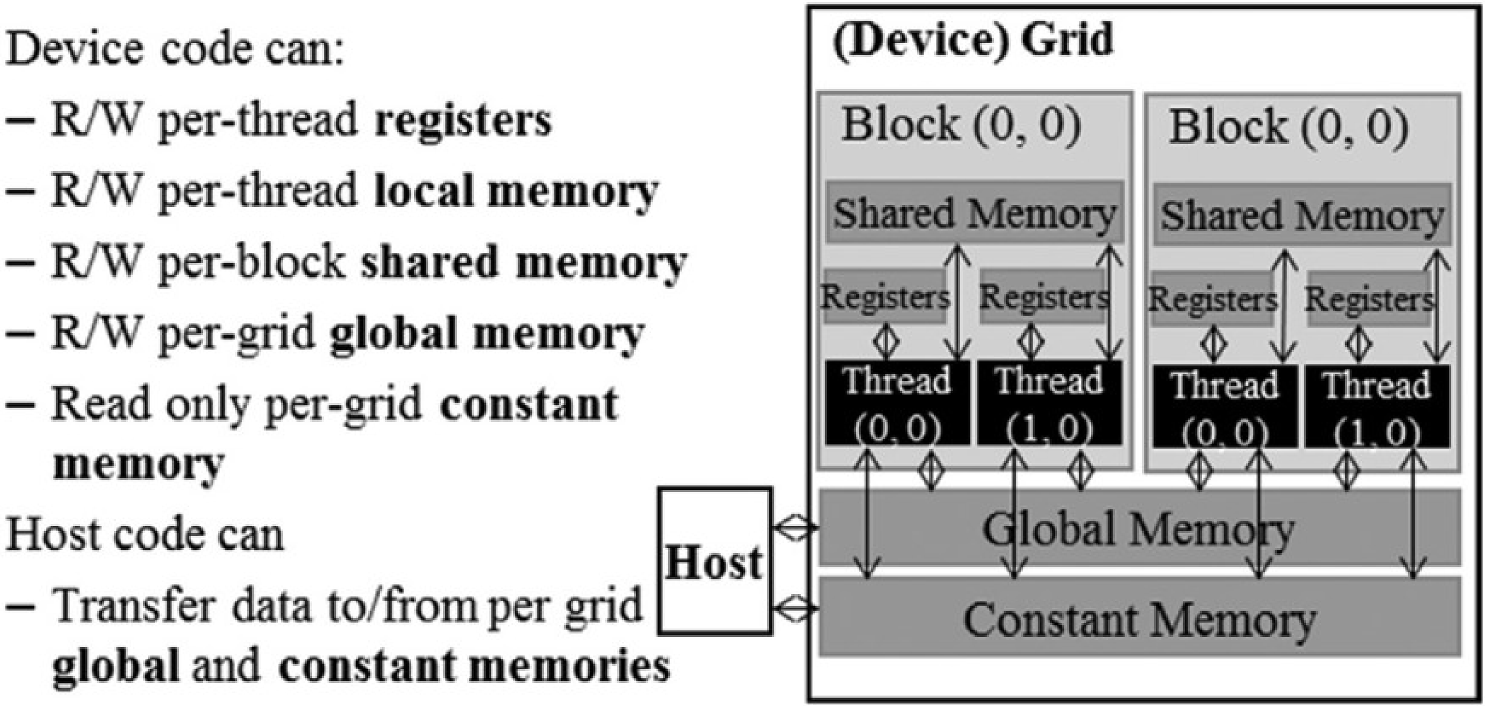
Key quantity: **computational intensity**: FLOP/Byte of memory transfer

Latency hiding: having multiple warps on the SM allows warps to compute while others wait
(i.e. the memory transfer "+" compute becomes a max(..., ...))
→ roofline model



Compute bound

Memory bound



Device code can:

- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read only per-grid **constant memory**

Host code can

- Transfer data to/from per grid **global** and **constant memories**

FIGURE 5.2

An (incomplete) overview of the CUDA device memory model. An important type of CUDA memory that is not shown in this figure is the texture memory, since its use is not covered in this textbook.

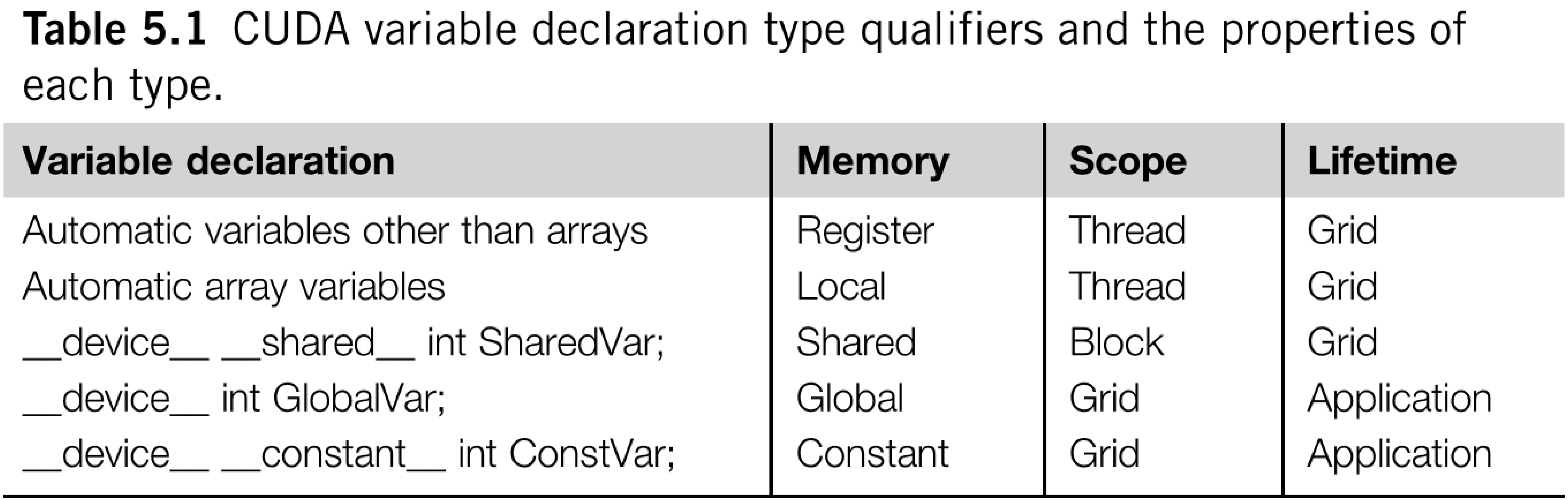


Table 5.1 CUDA variable declaration type qualifiers and the properties of each type.

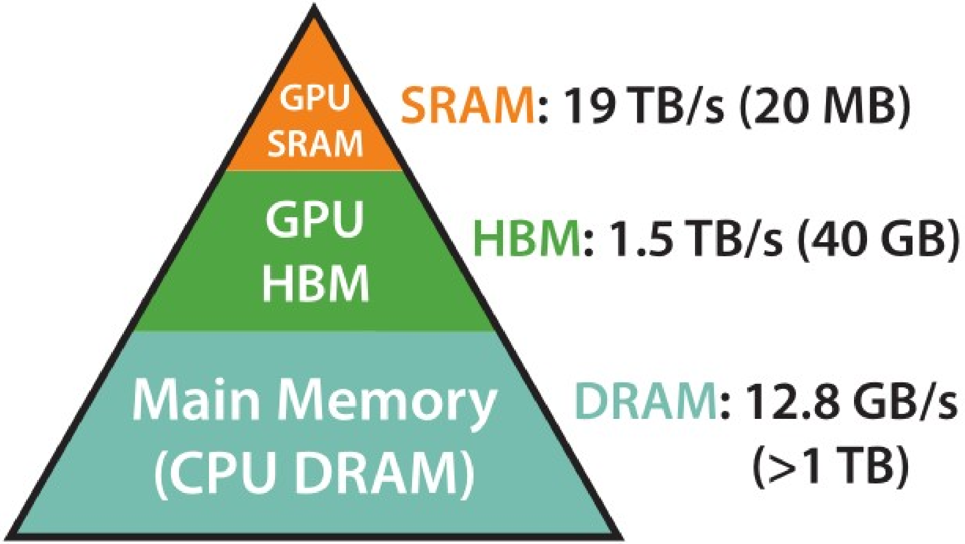
Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

What kinds of memory does the rgb2gray kernel use?

Tiling – why?

- In Matmul: each of the n^2 outputs uses 2n inputs
- every input used n times, naively n times from main memory
  → inefficient
  → try to reuse param

  Similar in Convolution, FlashAttention,...



Memory Hierarchy with Bandwidth & Memory Size

Dao et al. Flash Attention...

Tiling

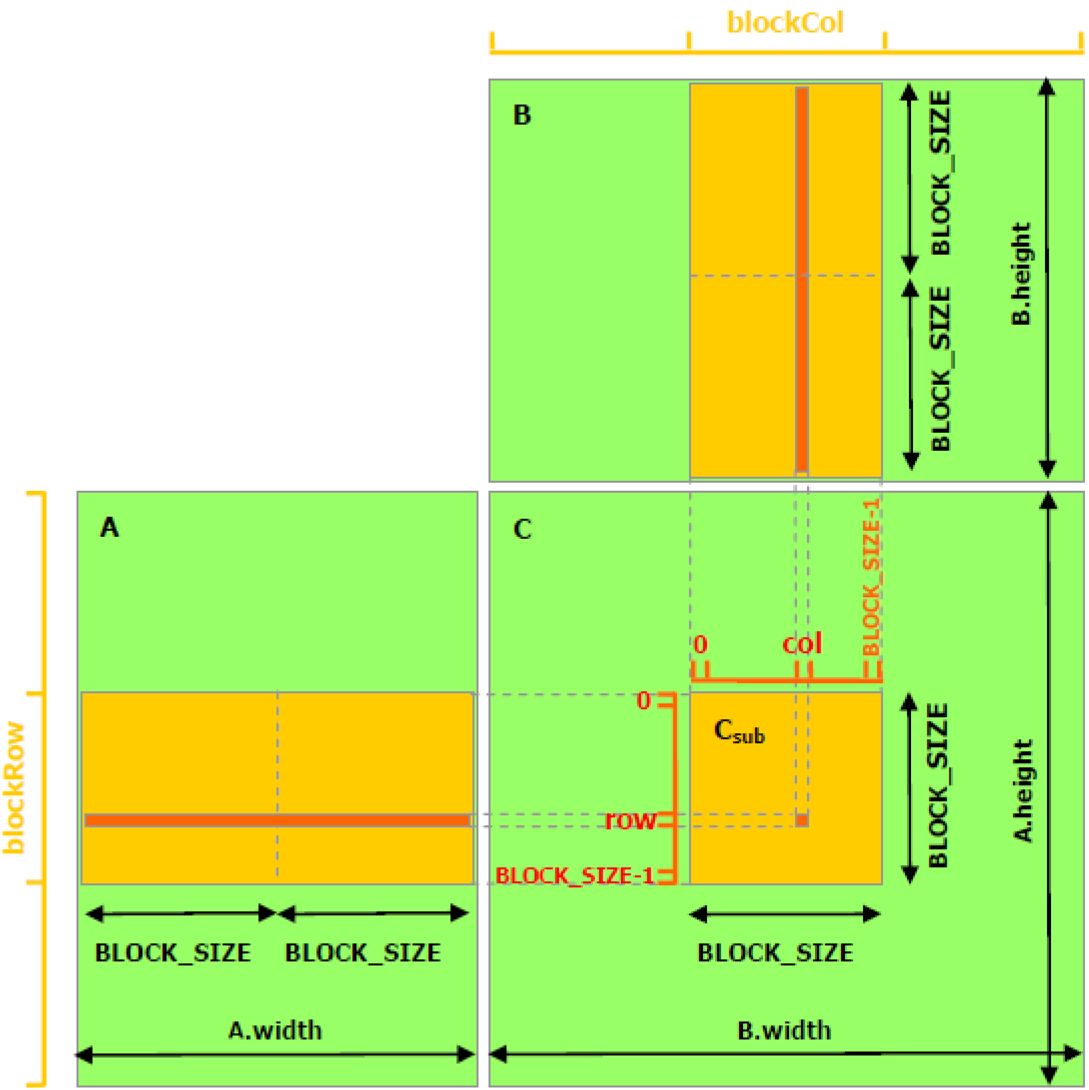
Divide output and input matrix into “tiles” (e.g. 16x16).

Assuming TILE_SIZE divides n:
Output tile depends on 2n/TILE_SIZE input tiles of size TILE_SIZE*TILE_SIZE.

Have (n/TILES_SIZE)^2 tiles:
read each input only n/TILE_SIZE times from main memory.

...need to store input tiles in shmem, so they can be used in TILE_SIZE computations by various threads in the block.

Easiest setup: TILE_SIZE^2 threads



Practical implementation of tiling

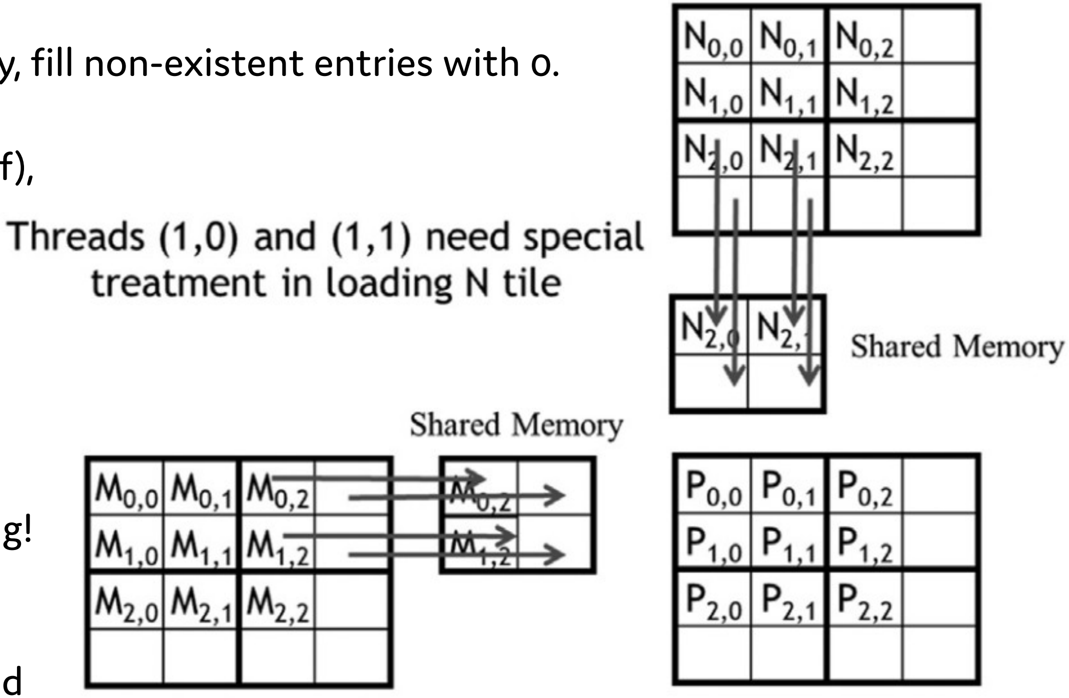
Tile padding if TILE_SIZE does not divide n

- Do this when reading / writing global memory, fill non-existent entries with 0.
- I like to write this as a tenary (valid ? x[i] : 0.0f), makes me feel good w.r.t. divergence (will be translated into conditional load also if you use if)

Balance shmem use for occupancy!

Don’t forget **__synthreads before and after** reading! (Q: why?)

Will later consider how much work to do per thread (thread coarsening).


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$
$N_{2,0}$ $N_{2,1}$
Shared Memory
Shared Memory
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$
$M_{0,2}$
$M_{1,2}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$

Threads (1,0) and (1,1) need special treatment in loading N tile

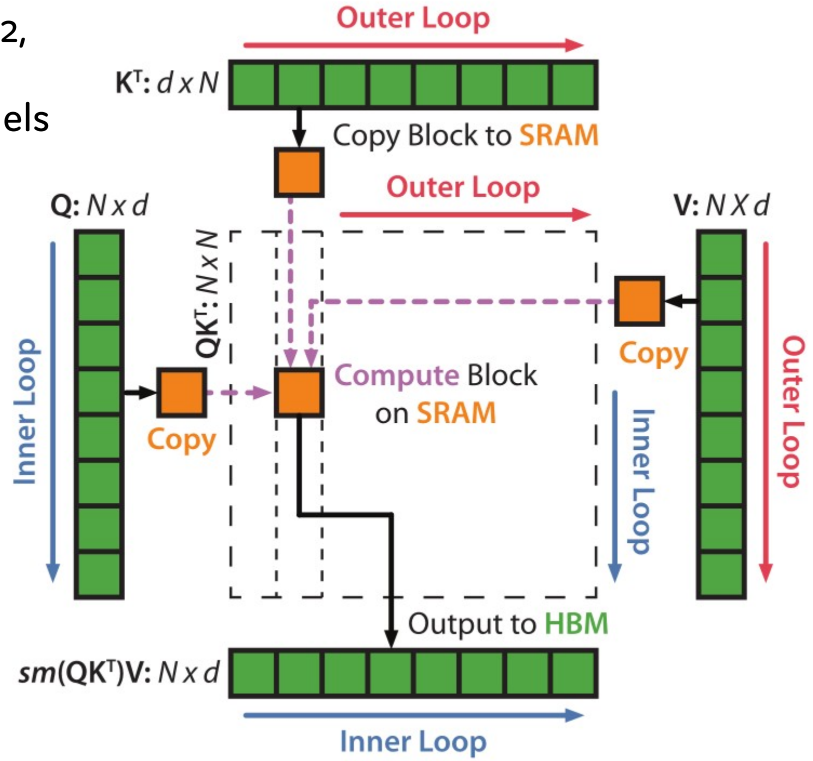
Threads (0,1) and (1,1) need special treatment in loading M tile

Advanced tiling and fusing: FlashAttention

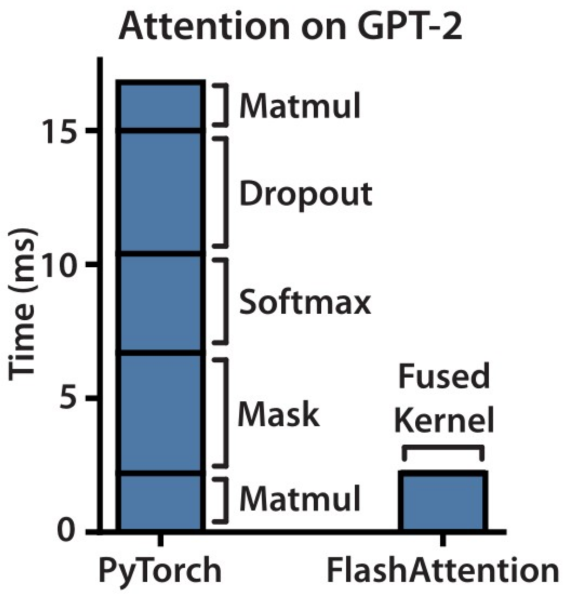
3x end-to-end speedup on GPT2,

high impact also for larger models

Exercise: Implement
flash attention from scratch.



FlashAttention



Conclusion

Key takeaways from ch 4 and 5:

- how the GPU organizes the computation in threads / warps / blocks
- use as much of the hardware as possible (occupancy), balance bottlenecks
- avoid thread divergence
- roofline model and "theoretical maximum speed"
- try to not read/write too much from/too global memory.

Next chapter: read/write consecutive and aligned global memory locations (**coalesced memory access**)
(and a lot about how this works in detail)

Sources

- The book (Wen-mei W. Hwu, David B. Kirk, Izzat El Hajj: Programming Massively Parallel Processors)
- GA102 Whitepaper (but check properties against query) https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf
- Inside Volta for the independent thread scheduling https://developer.nvidia.com/blog/inside-volta/
- Tri Dao et al.: FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, https://arxiv.org/abs/2205.14135