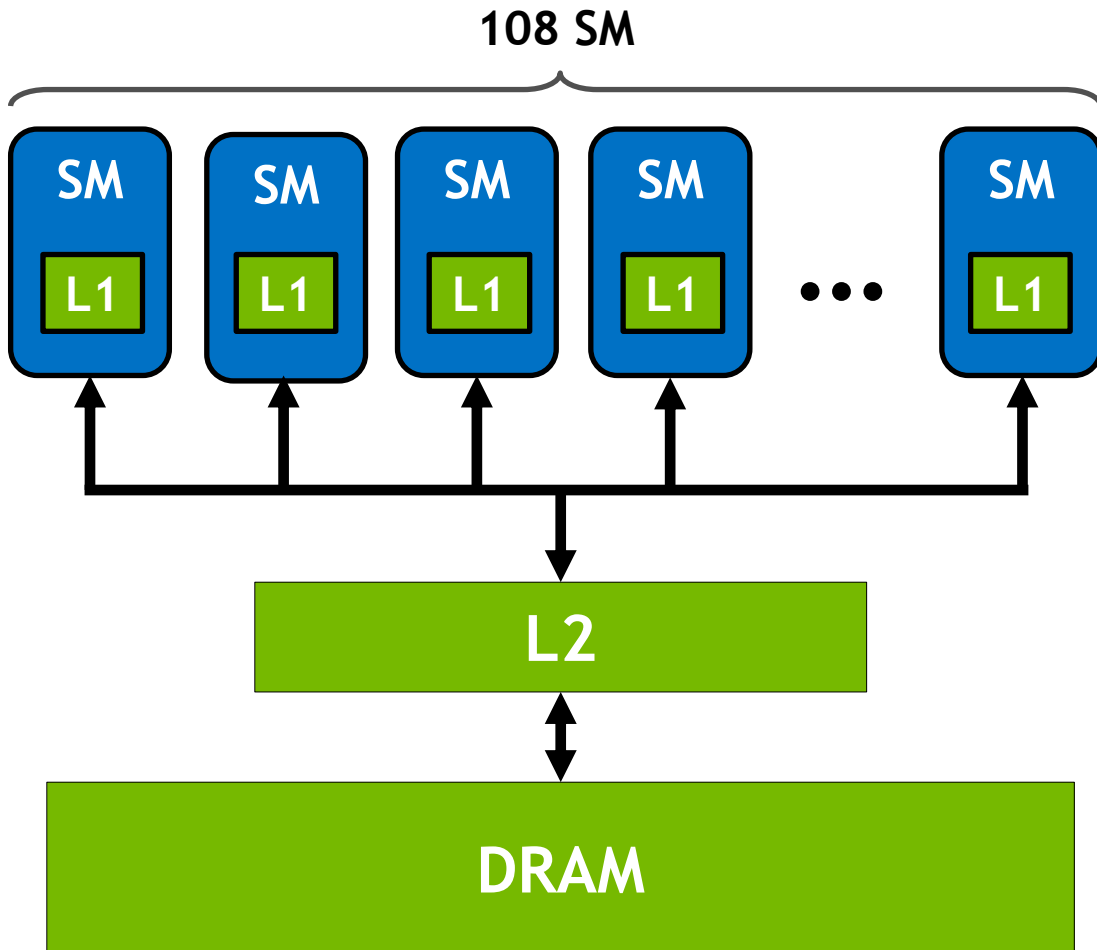




CUDA - Performance Optimization

Mohammad Motamedi - NVIDIA

Ampere A100



	P100	V100	A100
SMs	56	80	108
Memory	16GB	16/32GB	40/80GB
Bandwidth	720GB/s	900GB/s	1555GB/s
L2	4MB	6MB	40MB

STREAMING MULTIPROCESSOR

- 32 FP64 lanes
- 64 FP32 lanes
- 64 INT32 lanes
- 16 SFU lanes (transcendental)
- 32 LD/ST lanes
 - (G-mem/L-mem/S-mem)
- 4 Tensor Cores
- 4 TEX lanes



SM Resources

- Thread blocks require registers and shared memory.
- SM can schedule any resident warp without context switching.

Per SM	P100	V100	A100
Register File	256KB	256KB	256KB
Shared Memory	64KB	Configurable - Up to 96KB	Configurable - Up to 163KB
Max Threads	2048	2048	2048
Max Warps	64	64	64
Max Blocks	32	32	32

Performance Constraints

Compute bound - saturates compute units

- Reduce the number of instructions executed
 - Vector types, intrinsic operations, tensor cores, FMAs.

Bandwidth bound - saturates memory bandwidth

- Optimize access pattern.
- Use lower precision.

Latency bound

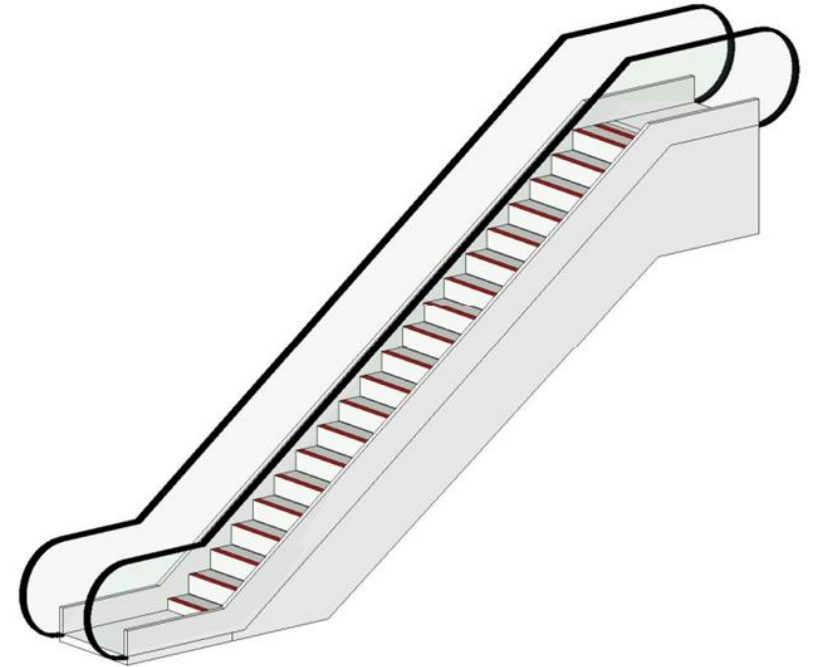
- Increase the number of instructions / mem accesses in flight.

Little's Law

For Escalators

Our escalator parameters

- 1 Person per step.
- A step arrives every 2 seconds
Bandwidth: 0.5 person/s.
- 20 steps tall
Latency = 40 seconds.

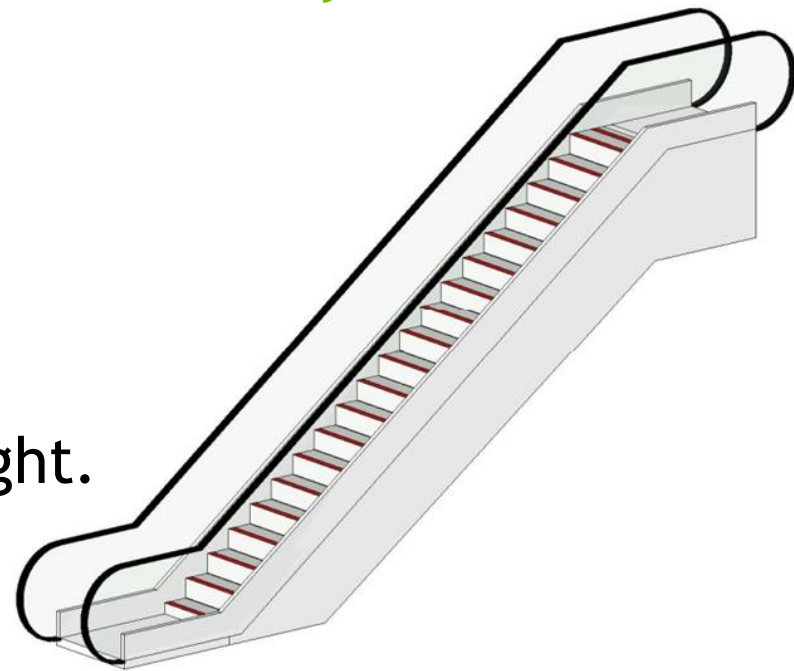


Little's Law

For Escalators

A step arrives every 2 seconds
Nominal Bandwidth: 0.5 person/s
20 steps tall : **Latency** = 40 seconds

- One person at a time?
Achieved bandwidth = 0.025 person/s.
- To saturate bandwidth
Need one person arriving with every step,
we need 20 persons in flight.
- Need **Bandwidth x Latency** persons in flight.



Optimization Goals

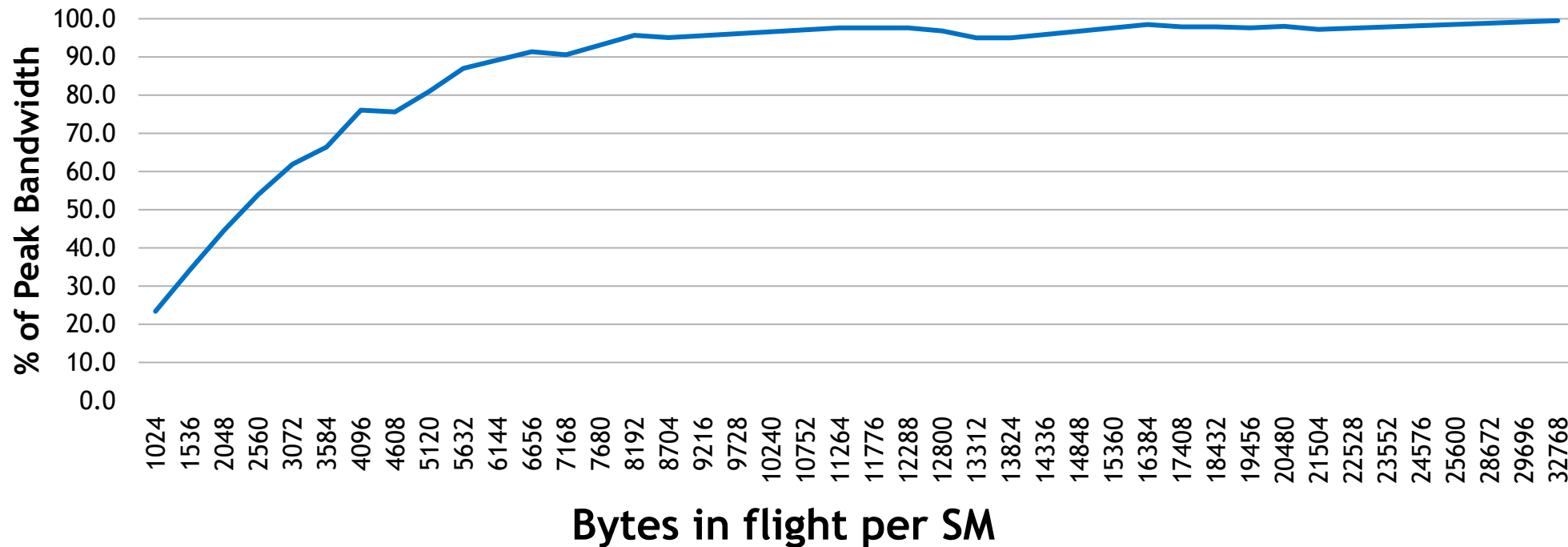
Saturating the compute units

- $a = b + c \rightarrow \text{arithmetic intensity} = \frac{1}{3 \times 4} F/B$
- $\text{GPU's capability} = \frac{15 \text{ TFps}}{900 \text{ GBps}} = 16.6 F/B$
- It is hard to saturate the compute units.
- One can improve it by saturating the memory bandwidth.

Saturating the memory bandwidth

- Hiding the latency is the key in achieving this goal.

Memory Bandwidth



Volta reaches 90% of peak bandwidth with ~6KB of data in flight per SM.

Little's law

Takeaways

Instructions in flight =

instructions in flight per thread x threads executing



Instruction Level Parallelism



Occupancy

Occupancy

Higher occupancy hides latency

SM has more warp candidates to schedule while other warps are waiting for instructions to complete.

$$\text{Occupancy} = \frac{\text{Achieved number of threads per SM}}{\text{Maximum number of threads per SM}}$$

- Use the profiler to compute it.

Achieved occupancy vs theoretical occupancy

Need to run enough thread blocks to fill all the SMs.

Be mindful of diminishing returns.

Instruction Issue

Instructions are issued in-order

If an instruction is not eligible, it stalls the warp.

An instruction is eligible for issue if both are true:

- **A pipeline is available** for execution

Some pipelines need multiple cycles to issue a warp.

- **All the arguments are ready**

Argument is not ready if a previous instruction has not produced it yet.

Instruction Issue Example

```
__global__ void kernel (float *a, float *b, float *c) {
```

```
    int i= blockIdx.x * blockDim.x + threadIdx.x;
```

```
    c[i] += a[i] * b[i];  
}
```

LDG.E R2, [R2];

LDG.E R4, [R4];

LDG.E R9, [R6];

stall!

FFMA R9, R2, R4, R9;

stall!

STG.E [R6], R9;

} 12B / thread
in flight

Computing 2 values per thread

```
__global__ void kernel (float2 *a, float2 *b, float2 *c) {
```

```
    int i= blockIdx.x * blockDim.x + threadIdx.x;
```

```
    c[i].x += a[i].x * b[i].x;  
    c[i].y += a[i].y * b[i].y;  
}
```

2 Independent instructions {

LDG.E.64 R2, [R2];

LDG.E.64 R4, [R4];

LDG.E.64 R6, [R8];

stall!

FFMA R7, R3, R5, R7;

FFMA R6, R2, R4, R6;

stall!

STG.E.64 [R8], R6;

} 24B/ thread
in flight

Control Flow

Blocks of threads, warps

- Single Instruction Multiple Threads (SIMT) model.
- CUDA hierarchy: **Grid -> Blocks -> Threads.**
- One **warp** = 32 threads.
- Why does it matter?
Many optimizations depend on behavior at the warp level.

Control Flow

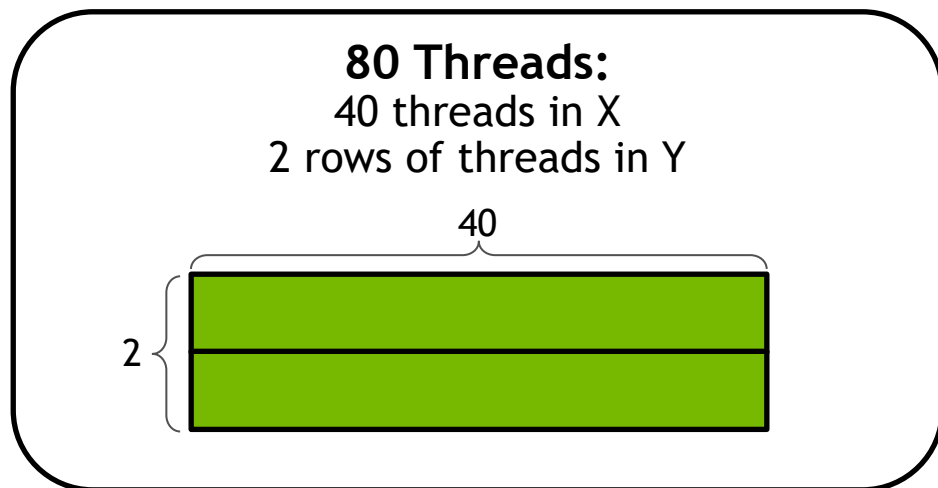
Divergence

- Different warps can execute different code
No impact on performance.
Each warp maintains its own Program Counter.
- Different code path inside the same warp?
Threads that do not participate are masked out,
but the whole warp executes both sides of the branch.

Control Flow

Mapping threads

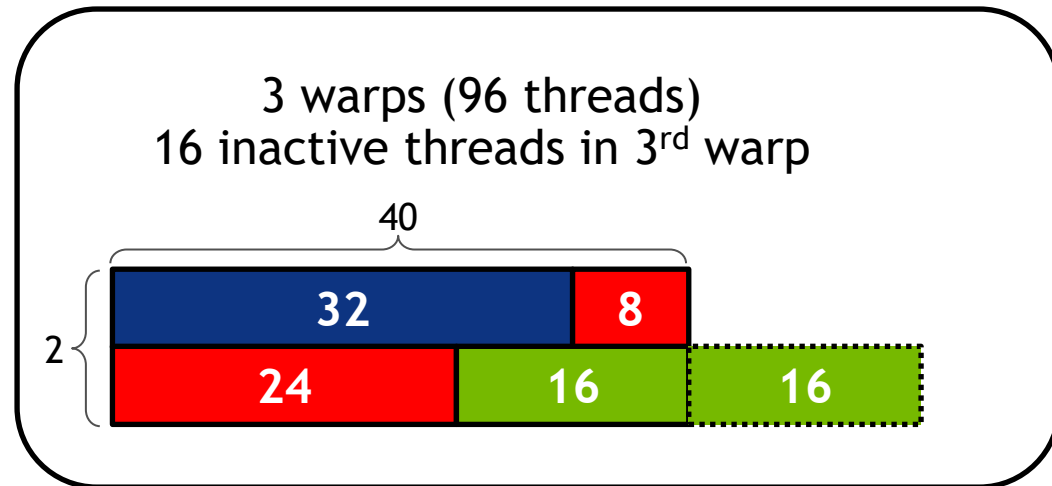
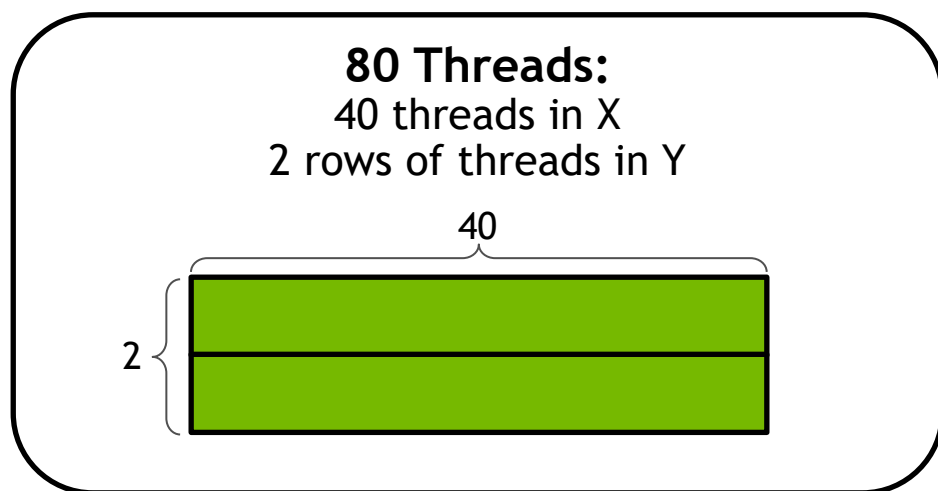
- **Thread blocks can have 1D, 2D, or 3D representation.**
Threads are linear in hardware.
- **Consecutive 32 threads** belong to the same **warp**.



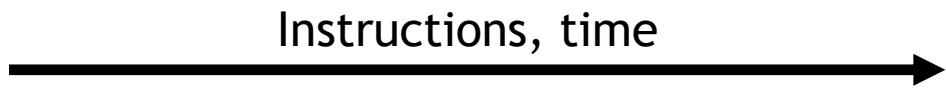
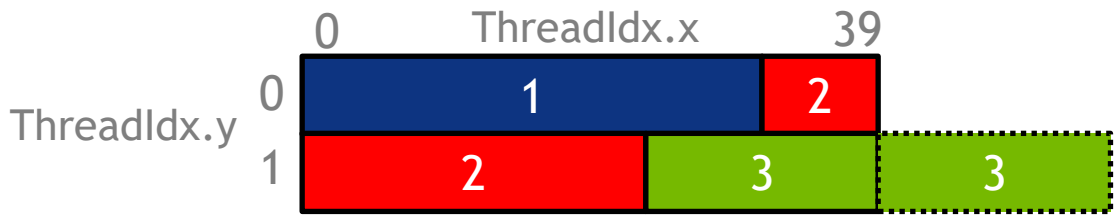
Control Flow

Mapping threads

- **Thread blocks can have 1D, 2D, or 3D representation.**
Threads are linear in hardware.
- **Consecutive 32 threads** belong to the same **warp**.



Control Flow



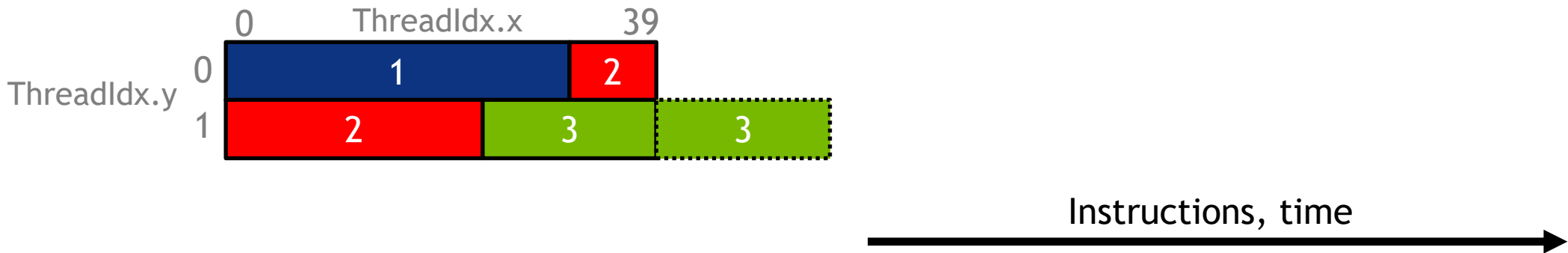
```

A;
if (threadIdx.y == 0)
    B;
else
    C;
D;

```

Warp 1 0 ... 31
 Warp 2 0 ... 31
 Warp 3 0 ... 31

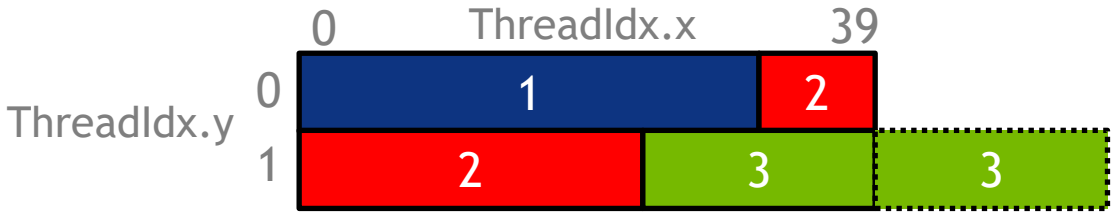
Control Flow



```
A;
if (threadIdx.y==0)
    B;
else
    C;
D;
```



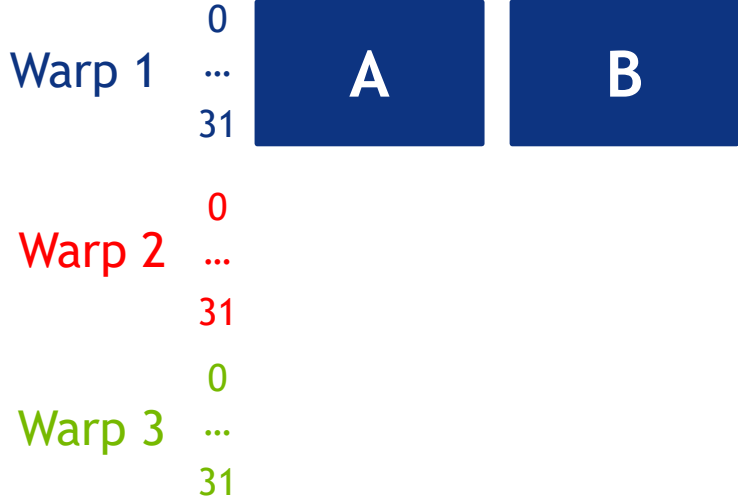
Control Flow



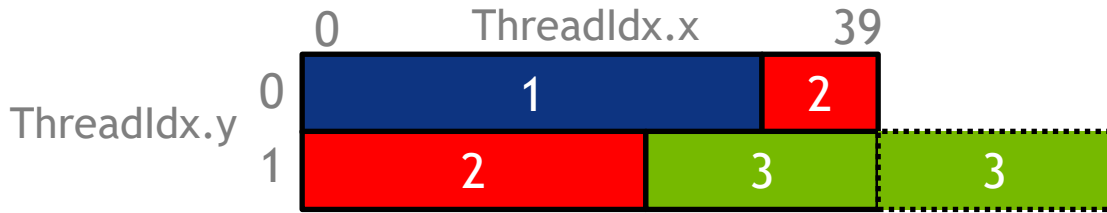
Instructions, time



```
A;
if (threadIdx.y == 0)
    B;
else
    C;
D;
```



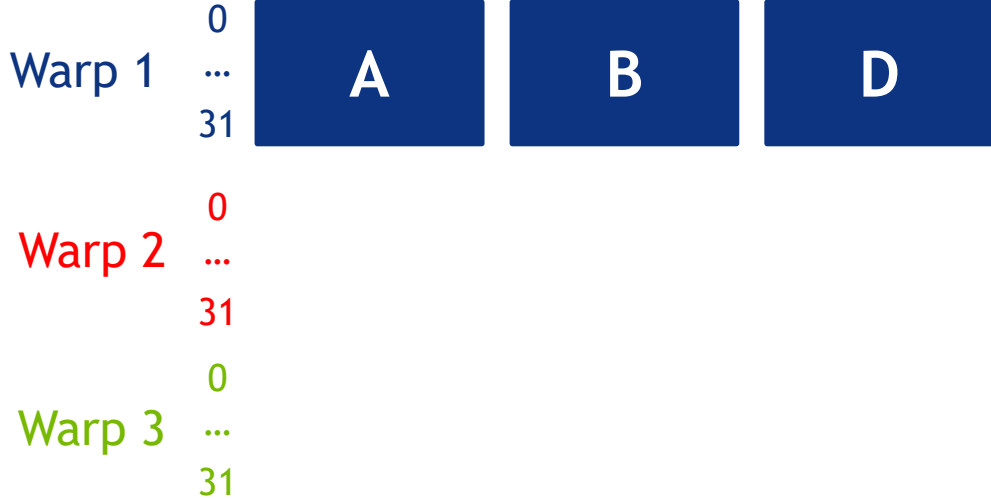
Control Flow



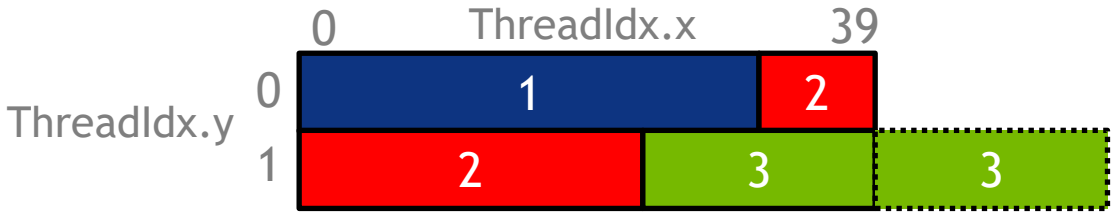
Instructions, time



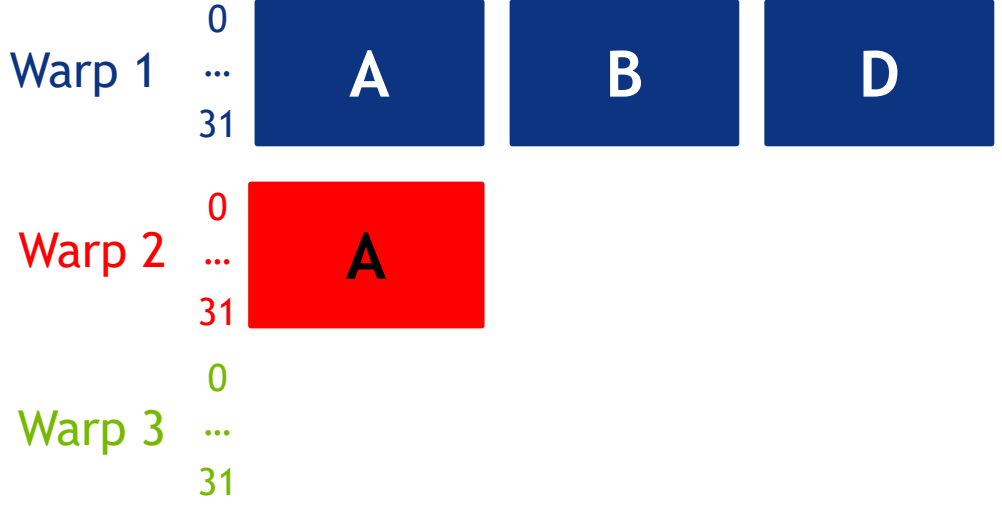
```
A;  
if (threadIdx.y==0)  
    B;  
else  
    C;  
D;
```



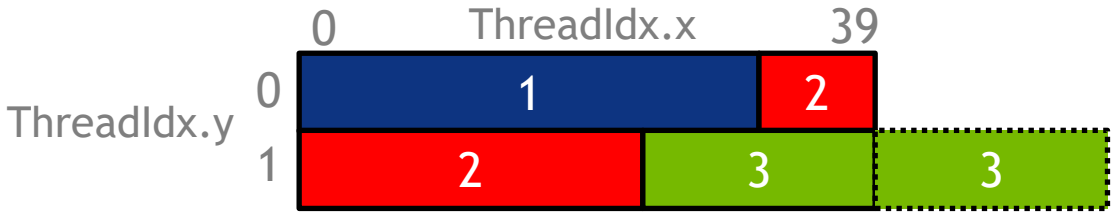
Control Flow



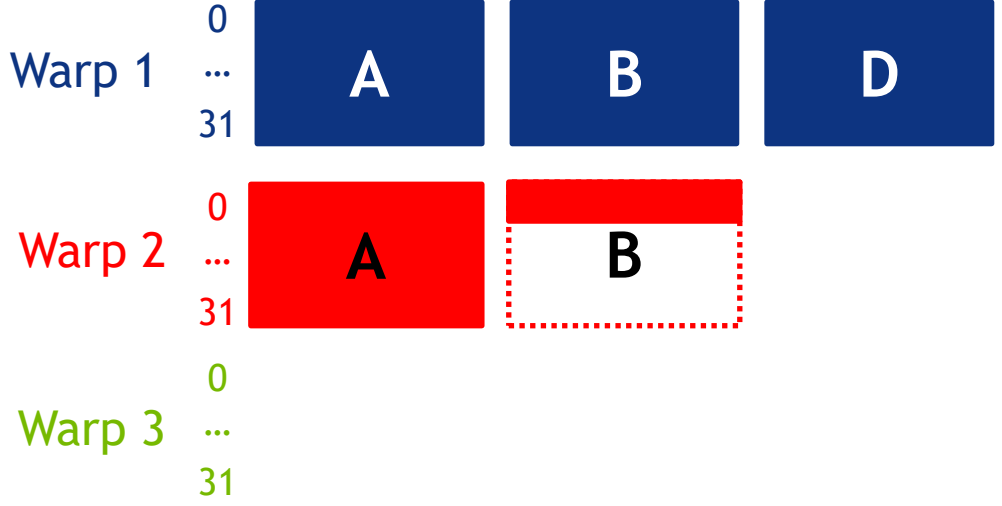
```
A;  
if (threadIdx.y == 0)  
    B;  
else  
    C;  
D;
```



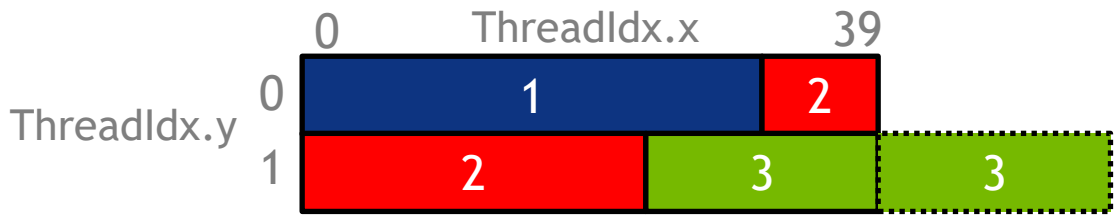
Control Flow



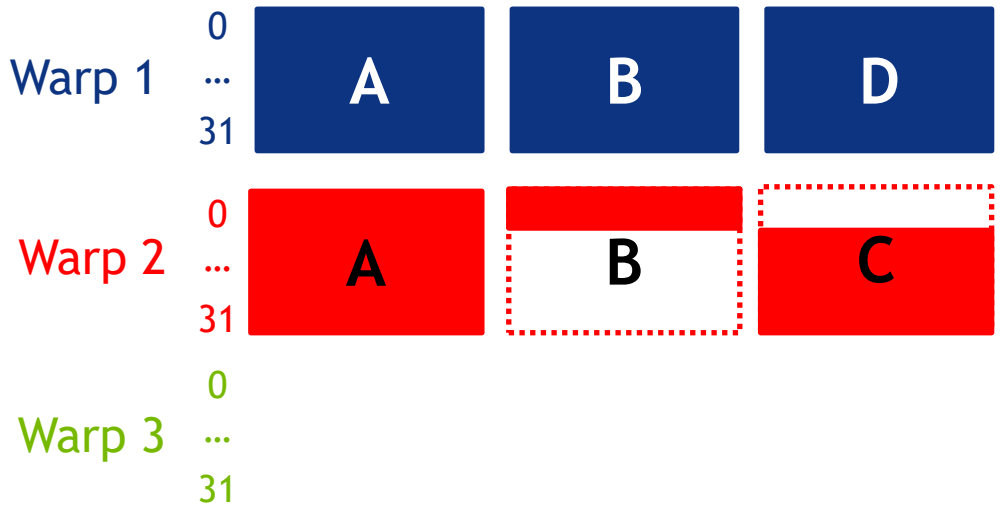
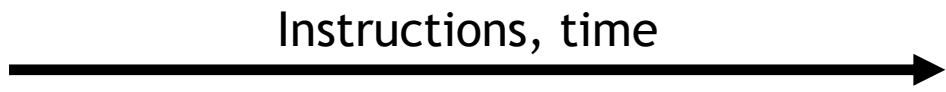
```
A;
if (threadIdx.y == 0)
    B;
else
    C;
D;
```



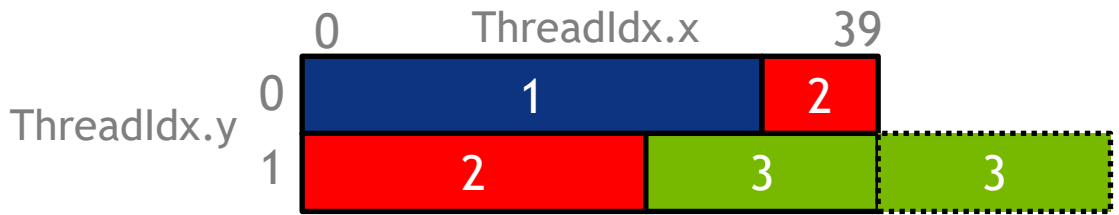
Control Flow



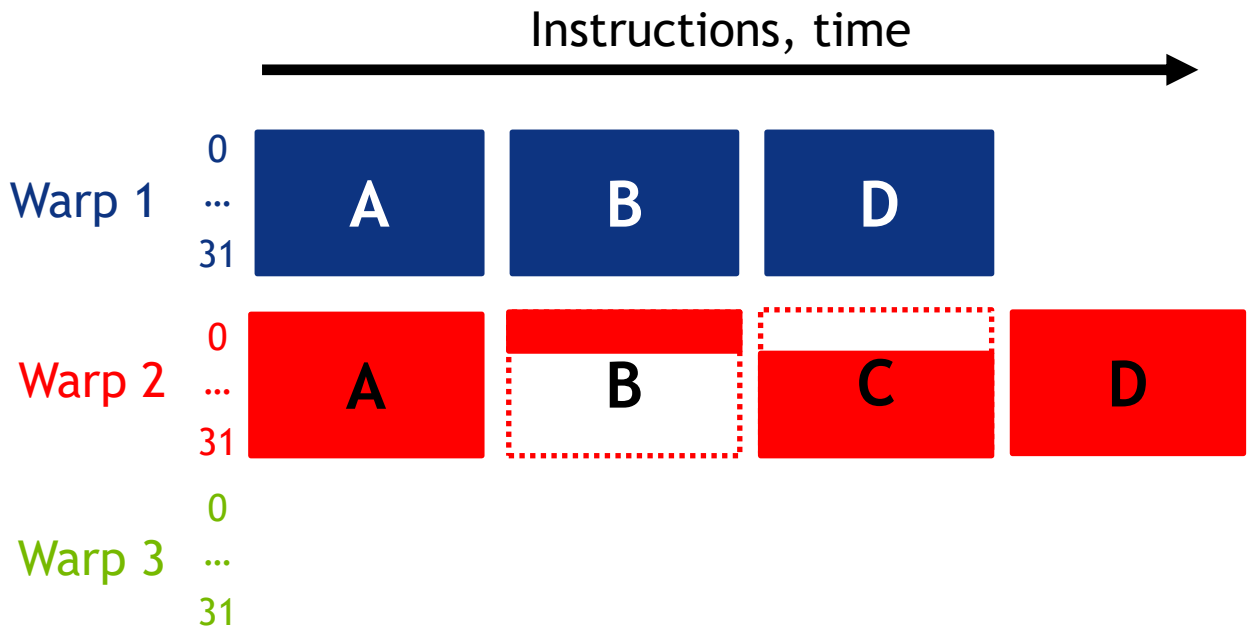
```
A;
if (threadIdx.y == 0)
    B;
else
    C;
D;
```



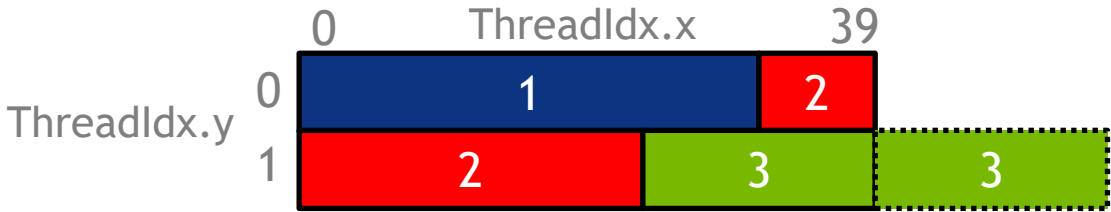
Control Flow



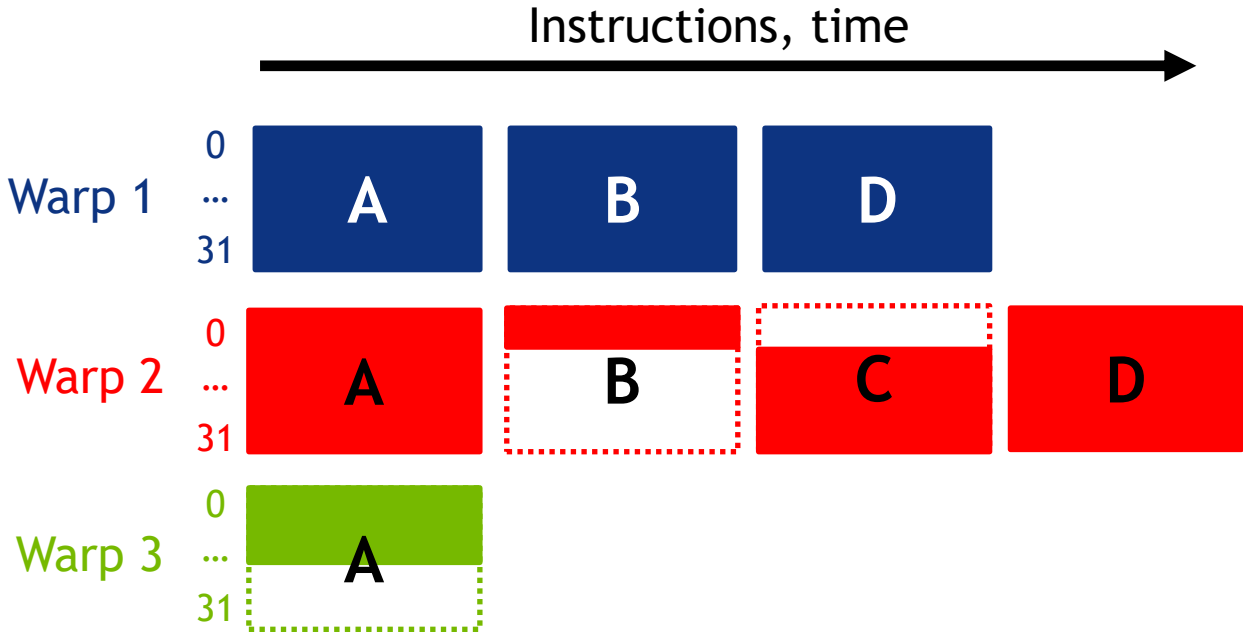
```
A;
if (threadIdx.y == 0)
    B;
else
    C;
D;
```



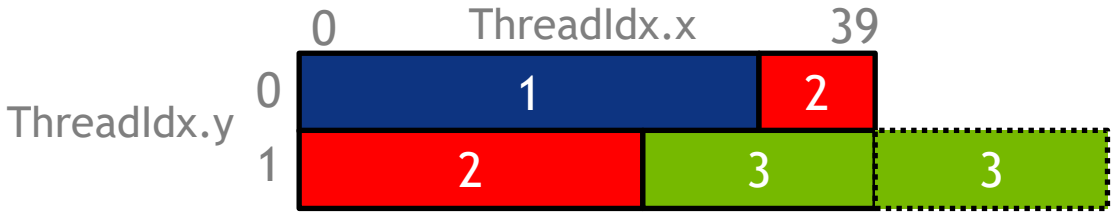
Control Flow



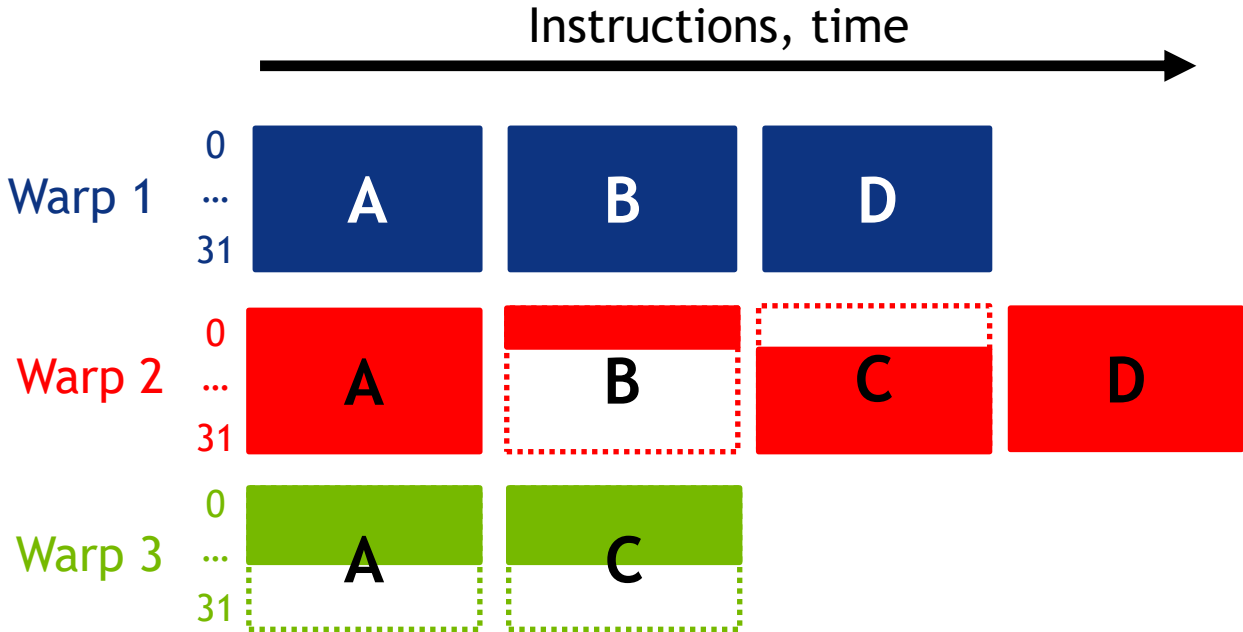
```
A;
if (threadIdx.y == 0)
    B;
else
    C;
D;
```



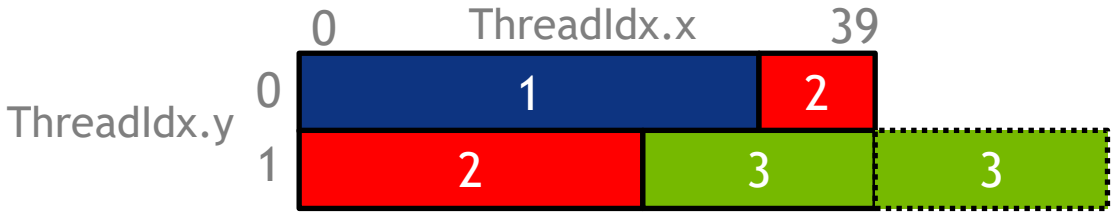
Control Flow



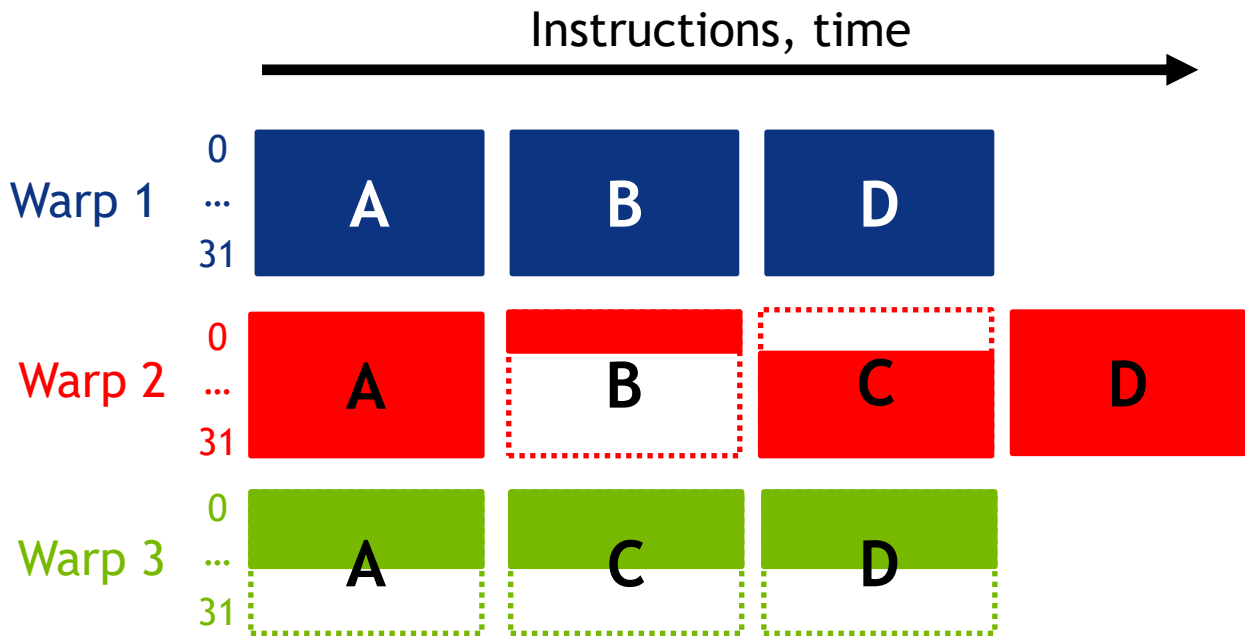
```
A;
if (threadIdx.y == 0)
    B;
else
    C;
D;
```



Control Flow



```
A;
if (threadIdx.y == 0)
    B;
else
    C;
D;
```

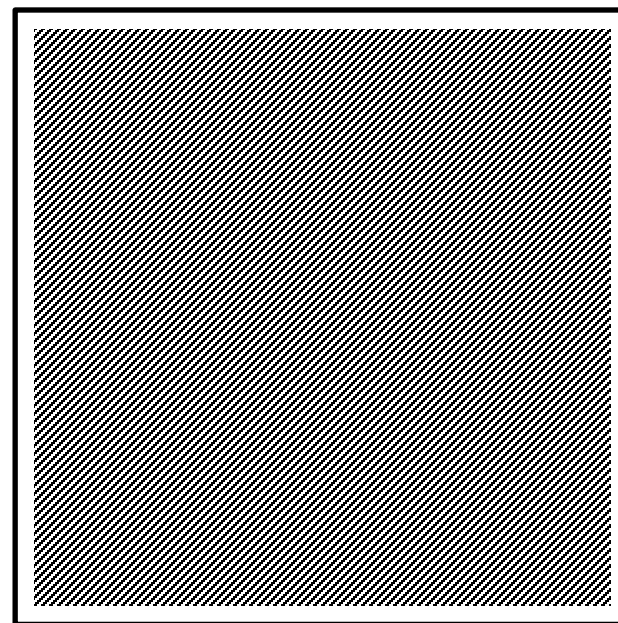


Control Flow

Case study: Thread divergence

Problem: 2D convolution with padding on an input image of size 1024x1024.

- Number of divergence in rows
 - 2-way divergence in first and last warp of each row.
 - Total warps with divergence: $1024 \times 2 = 2048$.
 - 6% of threads will have a 2-way divergence.



Control Flow

Takeaways

- Not every branch is a divergence.
- Minimize thread divergence inside a warp.
- Divergence between warps is fine.
- Maximize “useful” cycles for each thread (maximize # of threads executing + minimize thread divergence).
- Do not call a warp-wide instruction on a divergent branch (e.g. `__syncthreads()`).

Intrinsic Functions

- Fast but less accurate math intrinsic functions are available.
- 2 ways to use the intrinsic functions
 - Whole file: compile with `--fast-math`
 - Individual calls
E.g. `__sinf(x)`, `__logf(x)`, `__fdivide(x,y)`
- The programming guide has a list of intrinsic functions and their impact on accuracy.

Tensor Cores

Volta+

Dedicated matrix multiplication pipeline.

Input precision: FP16.

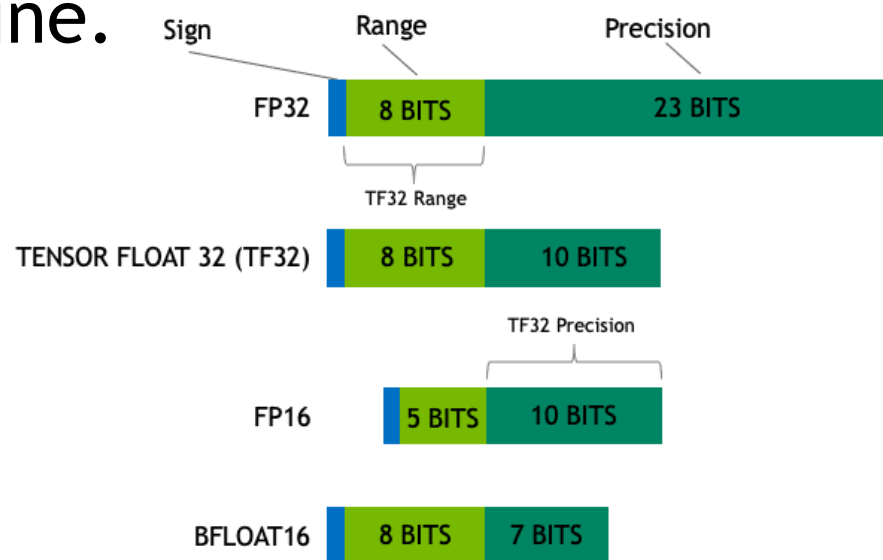
Peak Performance

V100: 125 TFLOPS, A100: 312 TFLOPS.

Used in CUBLAS, CUDNN, CUTLASS.

Optimized libraries can reach ~90% of peak.

Exposed in CUDA.



<https://blogs.nvidia.com/>

Tensor Cores

Using Tensor Cores in your CUDA code

Warp Matrix Multiply Add (WMMA)

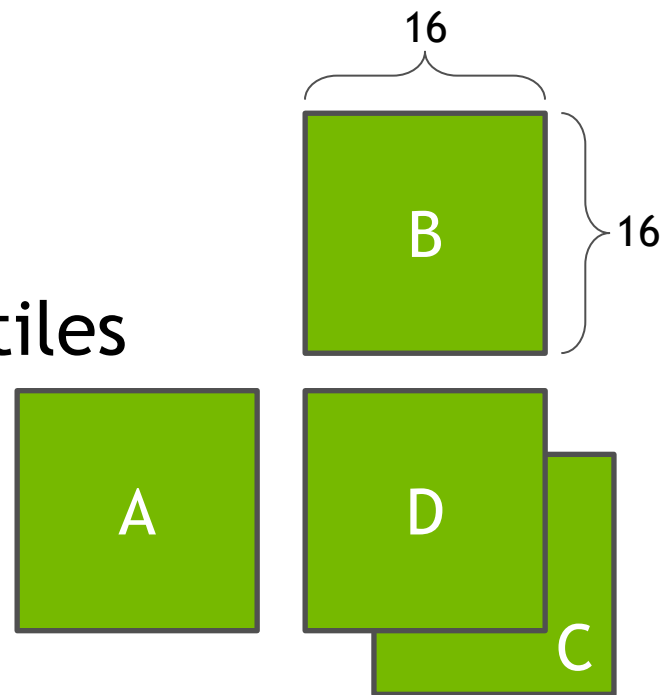
- Warp-wide macro-instructions.
- All threads in the warp must be active.

Performs matrix multiplication on 16x16 tiles
(8x32x16 and 32x8x16 tiles also available)

$$D = A \times B + C$$

A and B: FP16 only

C and D: Same, either FP16 or FP32.



Tensor Cores

Typical use

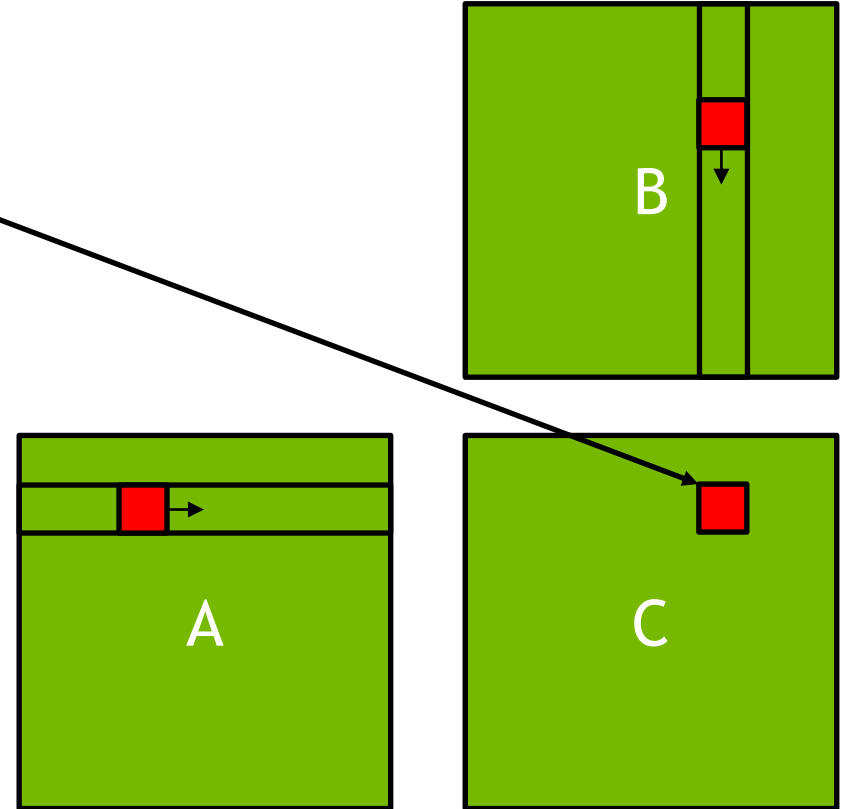
Each warp processes a 16x16 output tile

Each warp:

Loop on all input tiles A_k and B_k

$$C = C + A_k \times B_k$$

Write the output tile.



Tensor Cores

Typical use

Each warp processes a 16x16 output tile

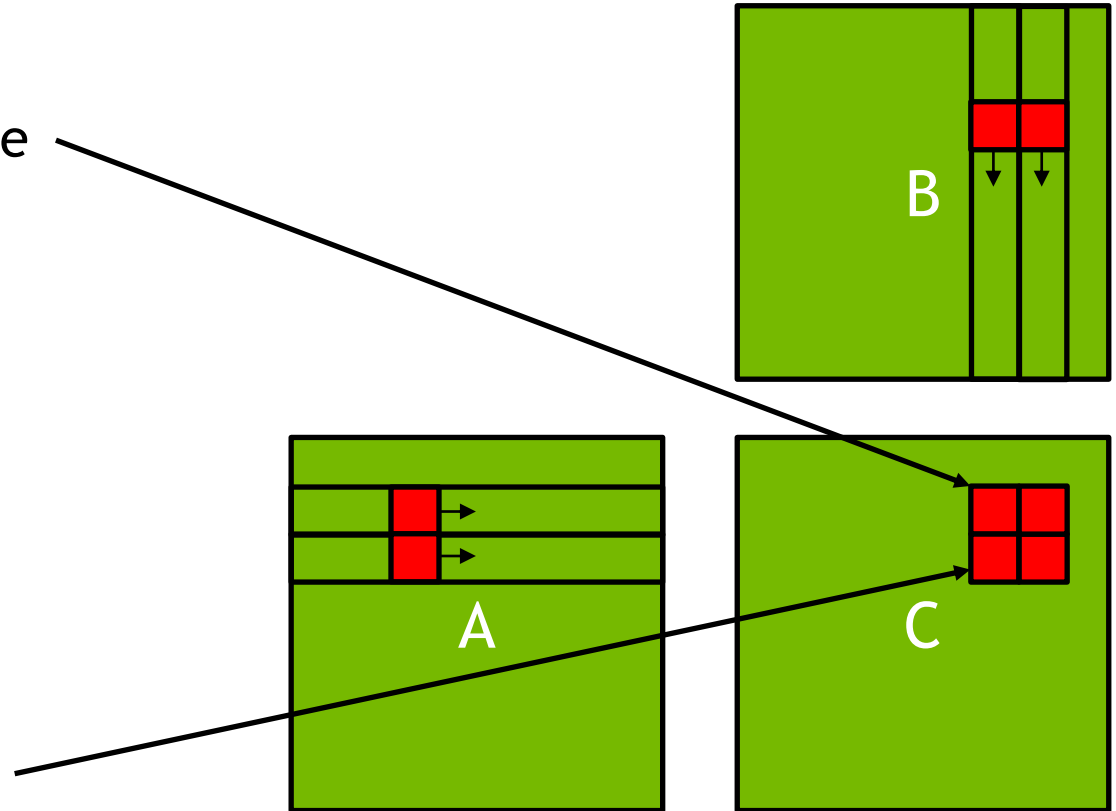
Each warp:

Loop on all input tiles A_k and B_k

$$C = C + A_k \times B_k$$

Write the output tile.

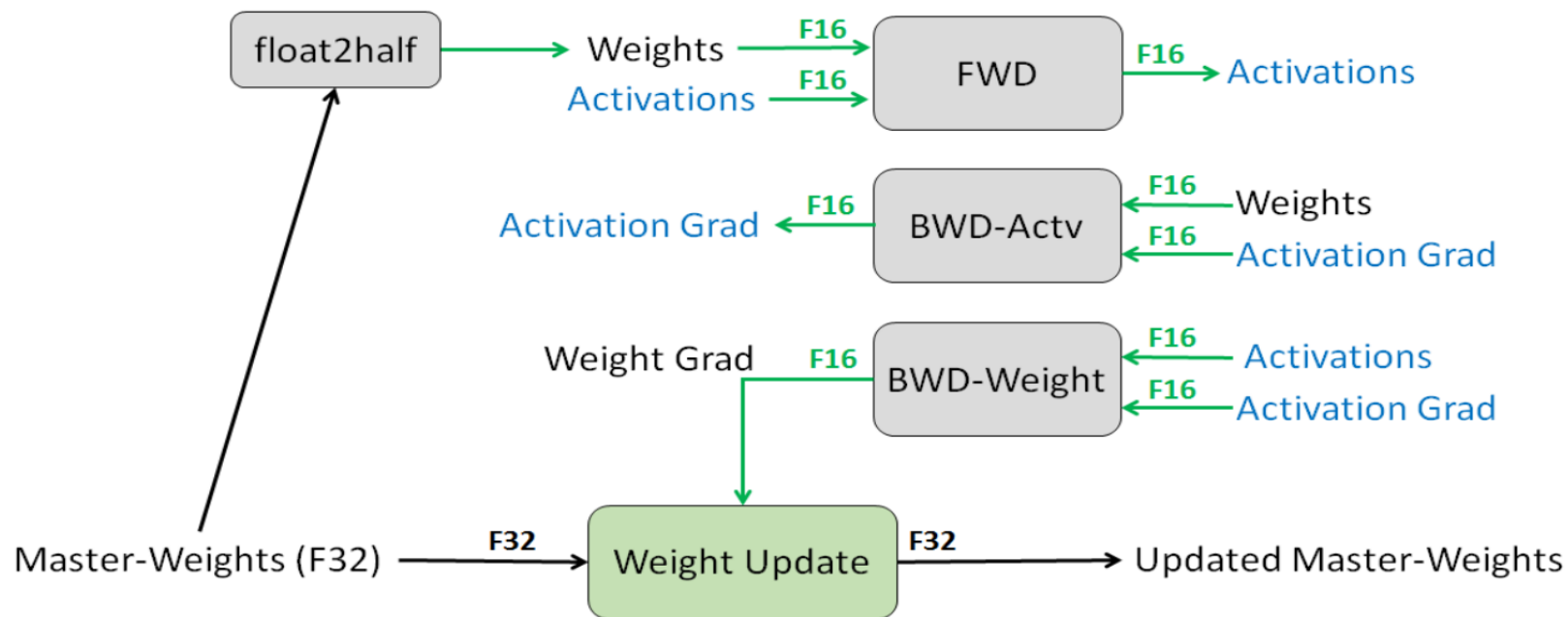
Can compute several tiles per block,
with inputs staged in shared memory.



Tensor Cores

Case Study: Deep Learning

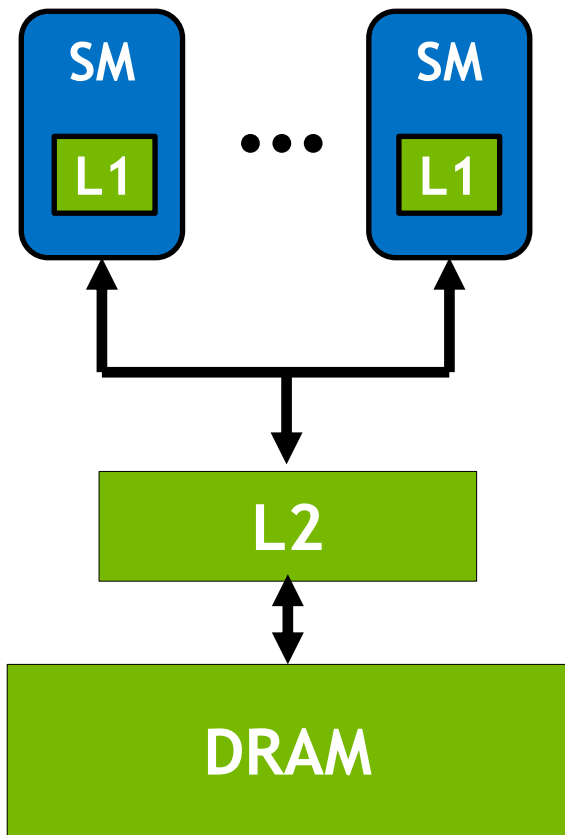
Scaling is required in the backward.



Source: <https://arxiv.org/pdf/1710.03740.pdf>

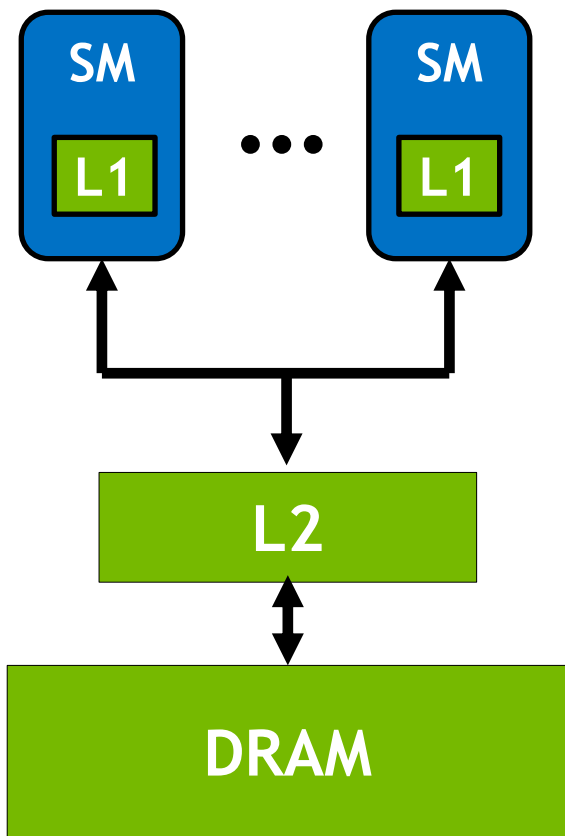
Memory Reads

Getting Data from Global Memory



- Checking if the data is in L1 (if not, check L2).
- Checking if the data is in L2 (if not, get in DRAM).
- Unit of data moved: Sectors.

Memory Writes



Before Volta : Writes were not cached in L1.

Volta+ : L1 will cache writes.

L1 is write-through: Write to L1 AND L2.

L2 is write back : Will flush data to DRAM only when needed.

Partial writes are supported (masked portion of sector, but behavior can change with ECC on/off).

L1, L2 Caches

Why Does GPU Have Caches?

In general, not for cache blocking

- 100s ~ 1000s of threads running per SM.
Tens of thousands of threads sharing the L2 cache.
L1, L2 are small per thread.
E.g., at 2048 threads/SM, with 80 SMs: 64 bytes L1, 38 Bytes L2 per thread.
Running at lower occupancy increases bytes of cache per thread.
- Shared Memory is usually a better option to cache data explicitly:
User managed, no evictions out of your control.

L1, L2 Caches

Why Does GPU Have Caches?

Caches on GPUs are useful for:

- “Smoothing” irregular, unaligned access patterns.
- Caching common data accessed by many threads.
- Faster register spills, local memory.
- Fast atomics.
- Codes that do not use shared memory (naïve code).

Cache Lines and Sectors

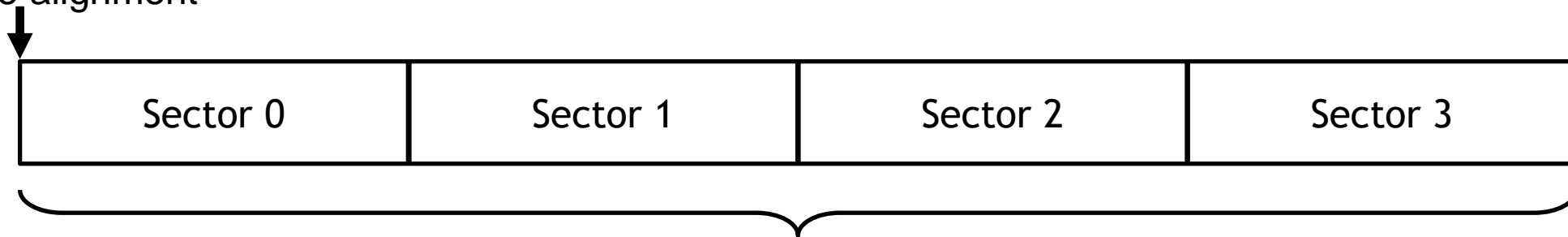
Moving data between L1, L2, DRAM

Sector-level memory access granularity.

- Sector size in Maxwell, Pascal, and Volta: 32B.
- Sector size in Kepler and before: variable (32 or 128).

A cache line is 128 Bytes, made of 4 sectors.

128-Byte alignment



128 Byte cache line

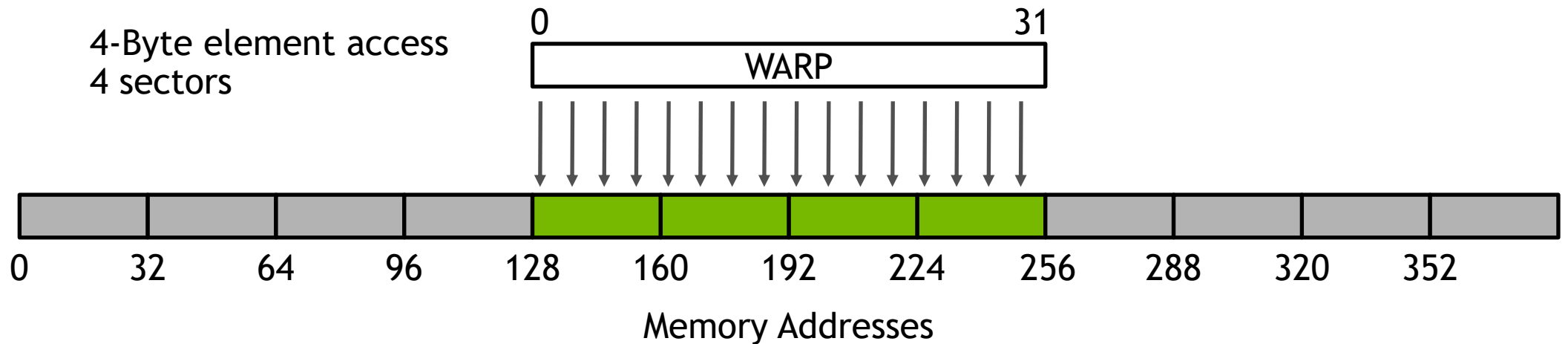
Access Patterns

Warps and Sectors

For each warp: How many **sectors** needed?

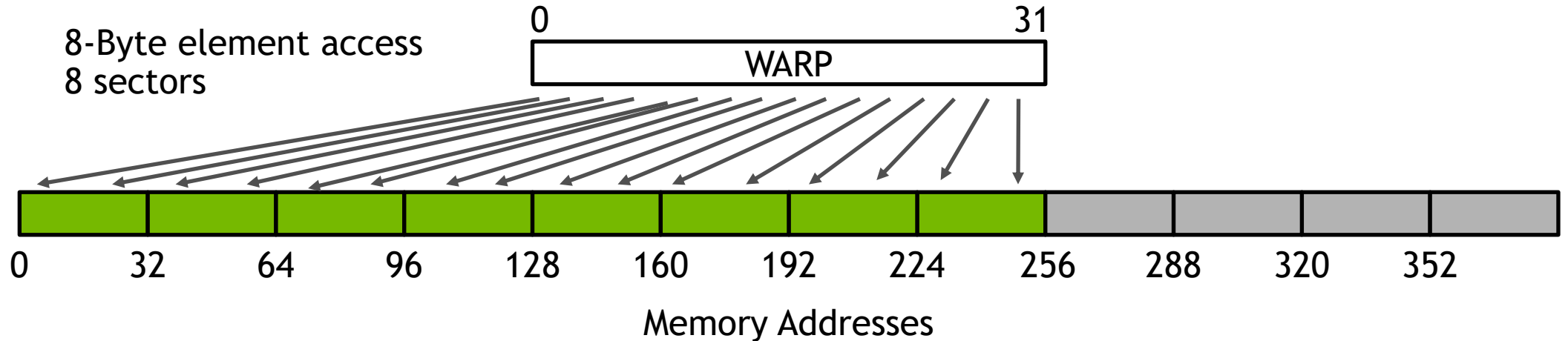
Depends on addresses, active threads, access size.

Natural element sizes = 1B, 2B, 4B, 8B, 16B.



Access Patterns

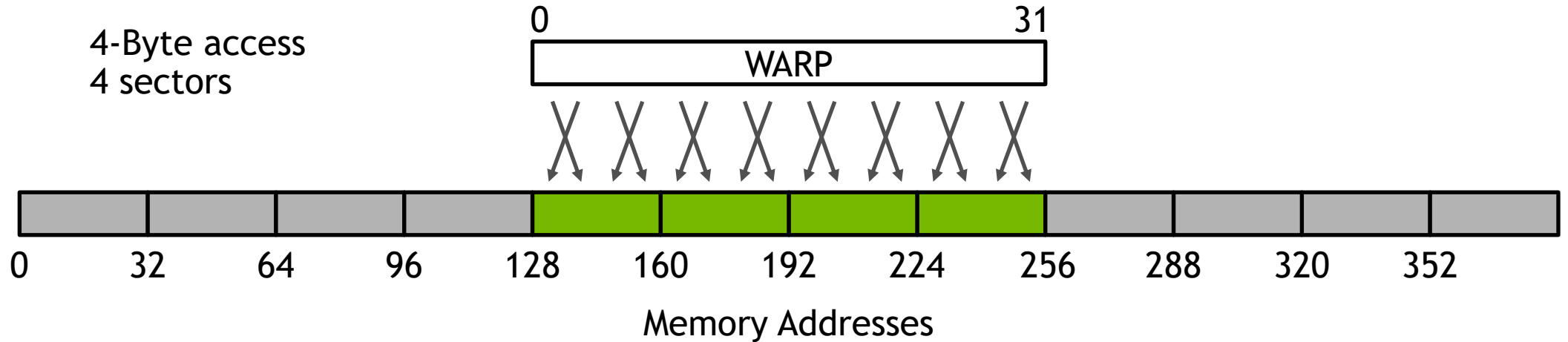
Warps and Sectors



Examples of 8-byte elements: long long, int2, double, float2.

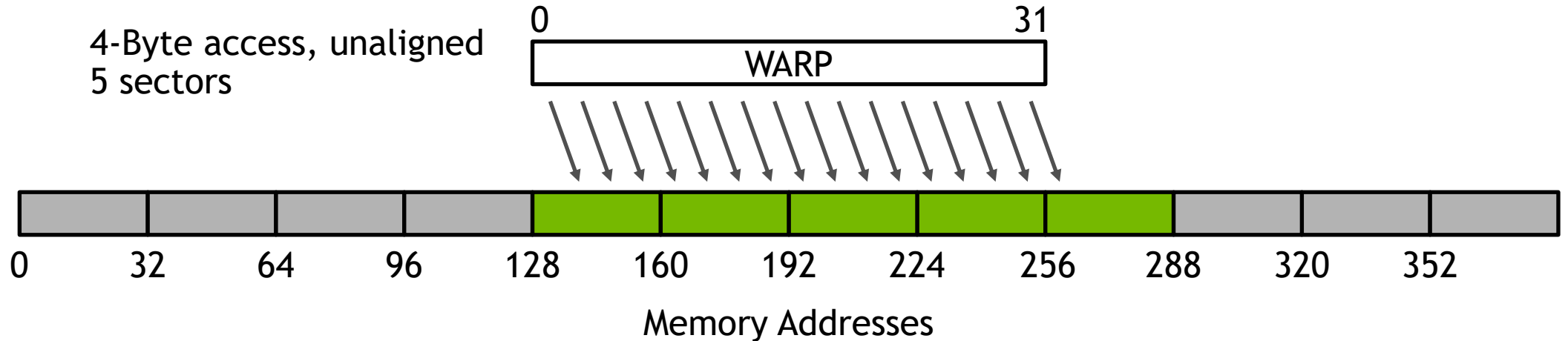
Access Patterns

Warps and Sectors



Access Patterns

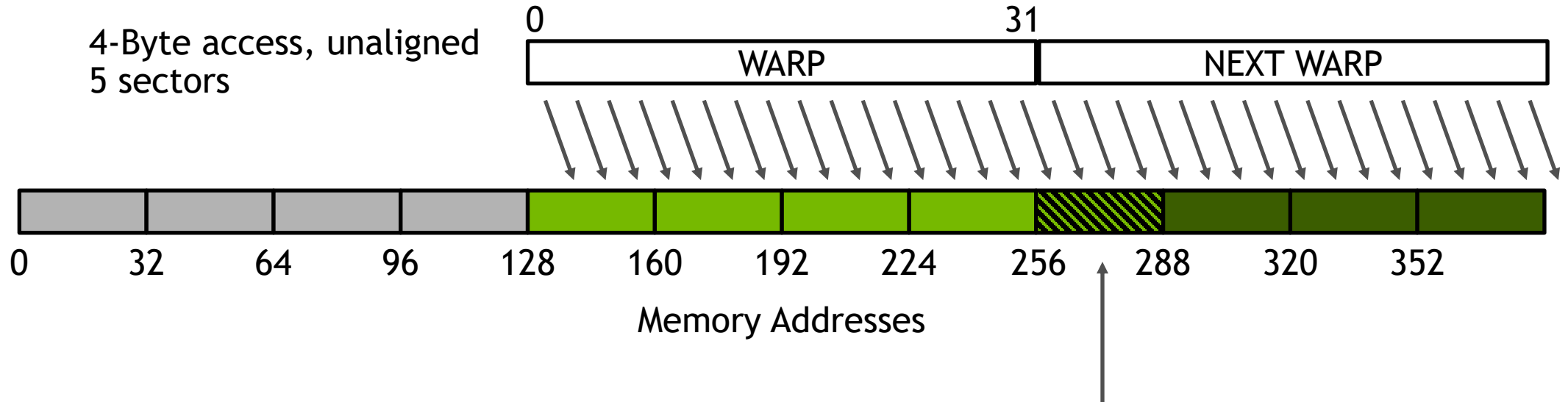
Warps and Sectors



128 Bytes requested; 160 bytes read (80% efficiency).

Access Patterns

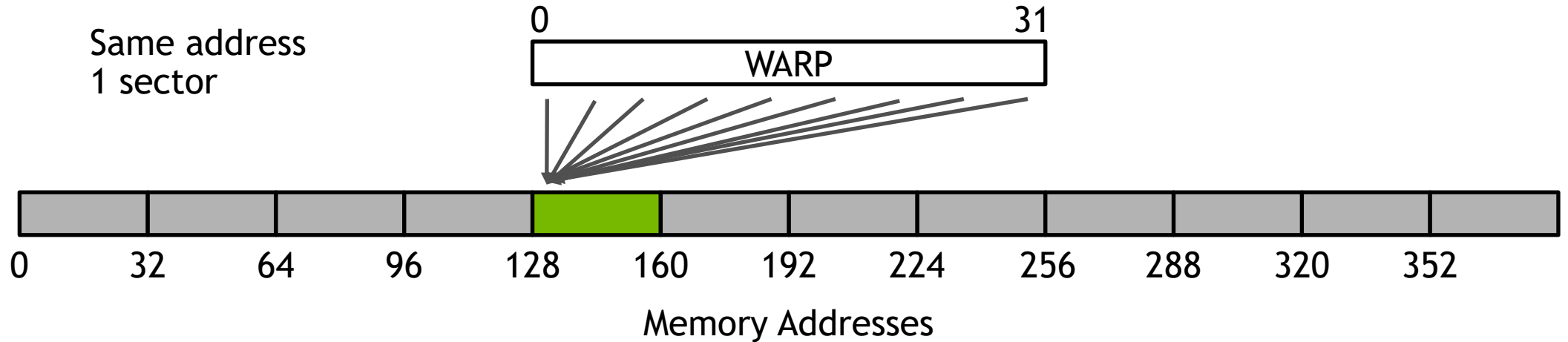
Warps and Sectors



With >1 warp per block, this sector might be found in L1 or L2.

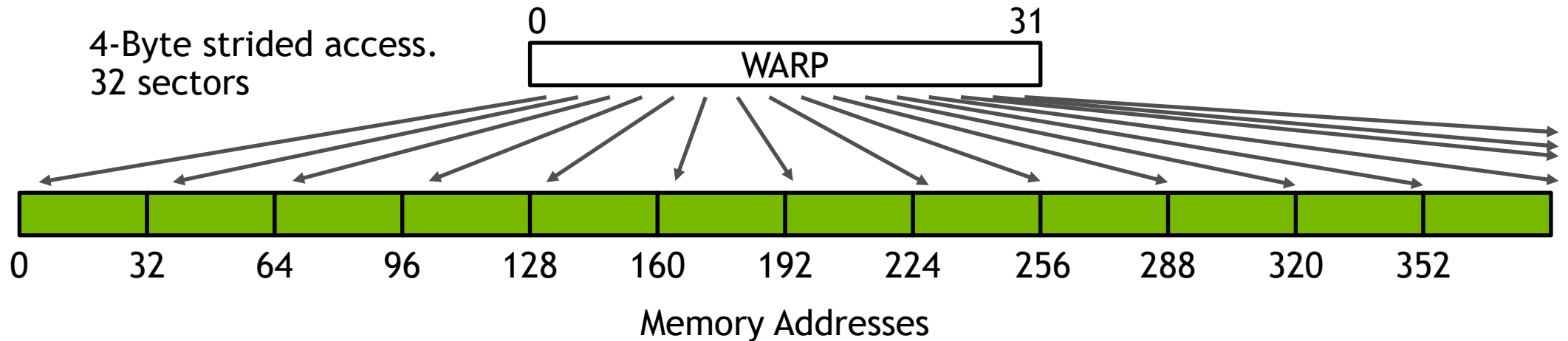
Access Patterns

Warps and Sectors



Access Patterns

Warps and Sectors



128 bytes requested; 1024 bytes transferred.
Using only a few bytes per sector. Wasting lots of bandwidth.

Access Patterns

Takeaways


- Know your access patterns.
- Use the profiler (metrics, counters) to check how many sectors are moved. Is that what you expect? Is it optimal?
- Using the largest type possible (e.g., float4) will maximize the number of sectors moved per instruction.

Memory System

Shared Memory Split

Shared Memory / L1 Split	
Volta	Ampere
	163KB/0KB*
	132KB/32KB
96 KB / 32 KB	100KB/64KB
64 KB / 64 KB	64KB/100KB
32 KB / 96 KB	32KB/132KB
16 KB / 112 KB	16KB/148KB
8KB / 120KB	8KB/156KB
0KB / 128KB	0KB/164KB

By default, the driver is using the configuration that maximizes the occupancy.

Examples


Examples	Volta	Ampere
0 KB Shared Mem. Other resources: 16 Blocks/SM.	Config: 0/128 Blocks/SM: 16	Config: 0/164 Blocks/SM: 16
45 KB Shared Mem. Other resources: 4 Blocks/SM.	Config: 96/32 Blocks/SM: 2	Config: 163/0 Blocks/SM: 3

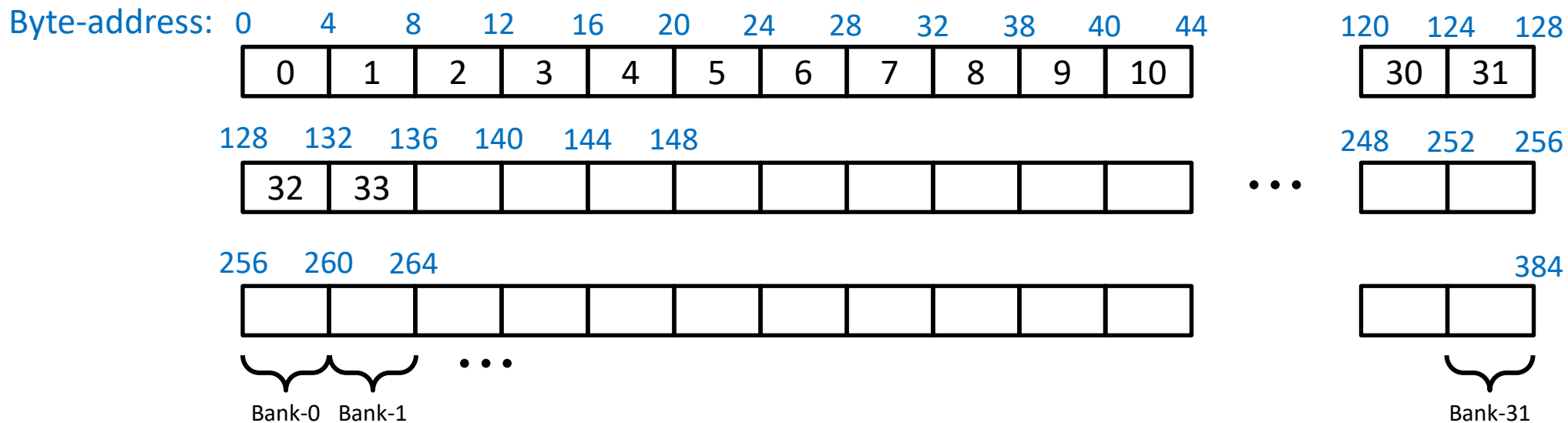
* 1KB is reserved.

Shared Memory Addressing

- Number of banks: 32, bandwidth: 4B.
- Mapping addresses to banks
 - Successive **4B** words go to successive banks
 - Bank index computation examples:
 - (4B word index) % 32.
 - ((1B word index) / 4) % 32.
 - 8B word spans two successive banks.

Logical View Of Shared Memory Banks

With 4-Byte data



Shared Memory Bank Conflicts

A **bank conflict** occurs when, **inside a warp**:
2 or more threads access within **different** 4B words in the **same bank**.
Think: 2 or more threads access different “rows” in the same bank.

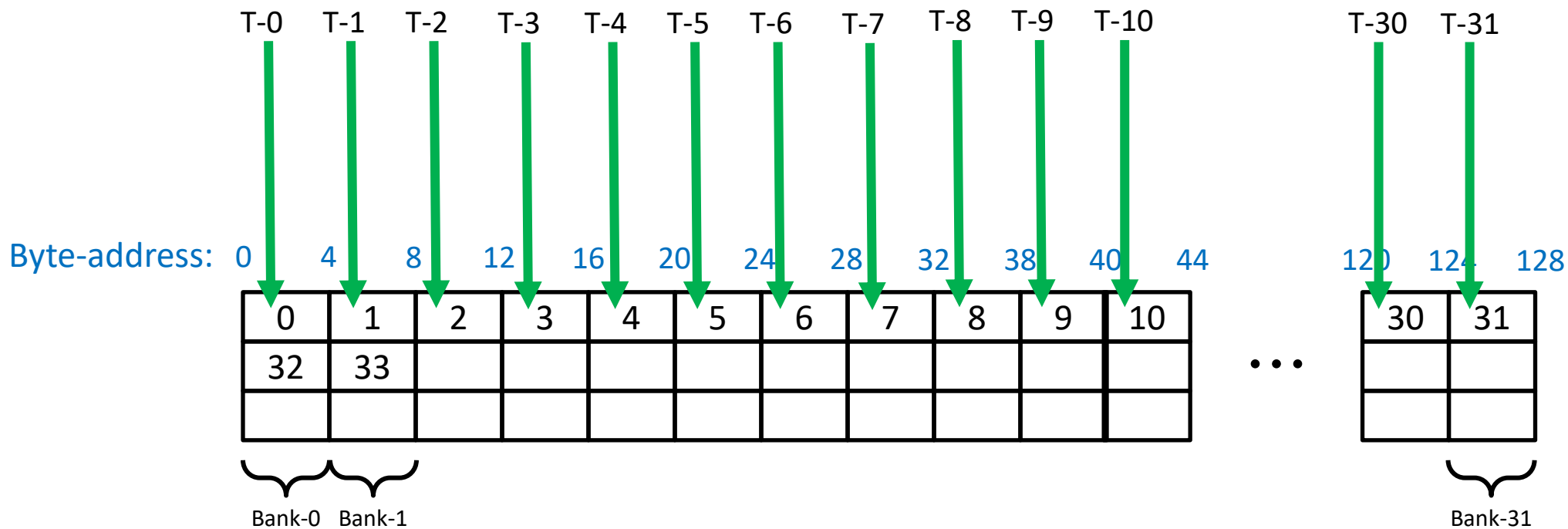
N-way bank conflict: **N** threads in a warp conflict

- Increases latency.
- Worst case: 32-way conflict → 31 replays.
- Each replay adds a few cycles of latency.

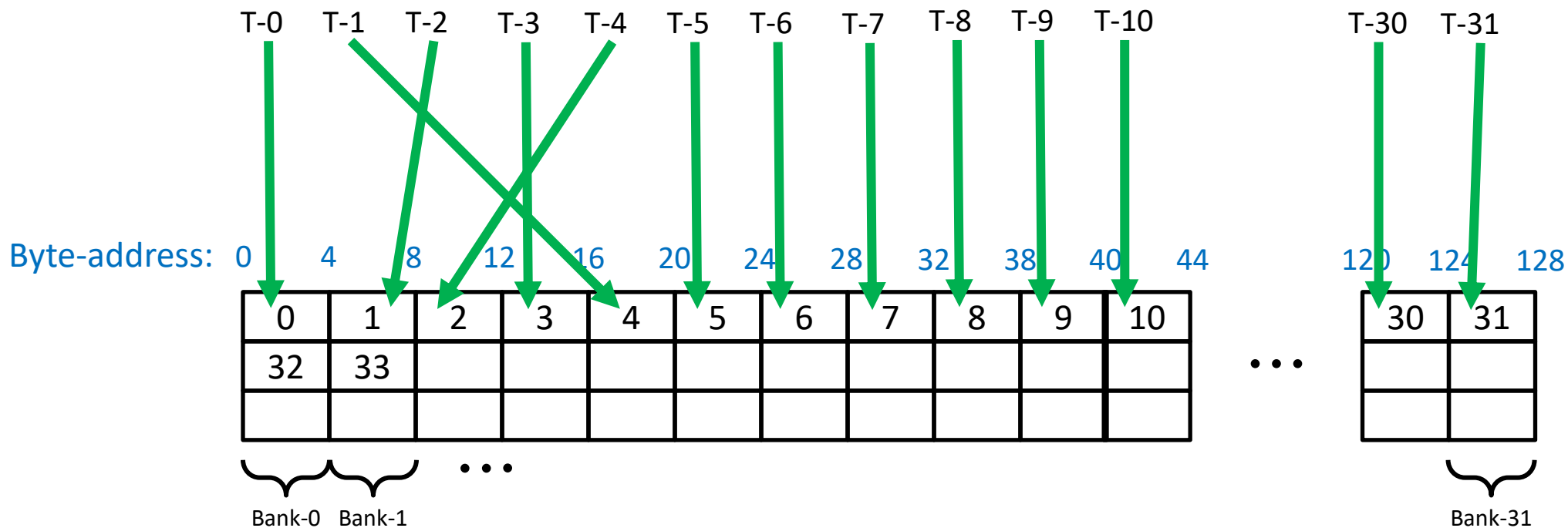
There is **no bank conflict** if:

- Several threads access the same 4-byte word.
- Several threads access different bytes of the same 4-byte word.

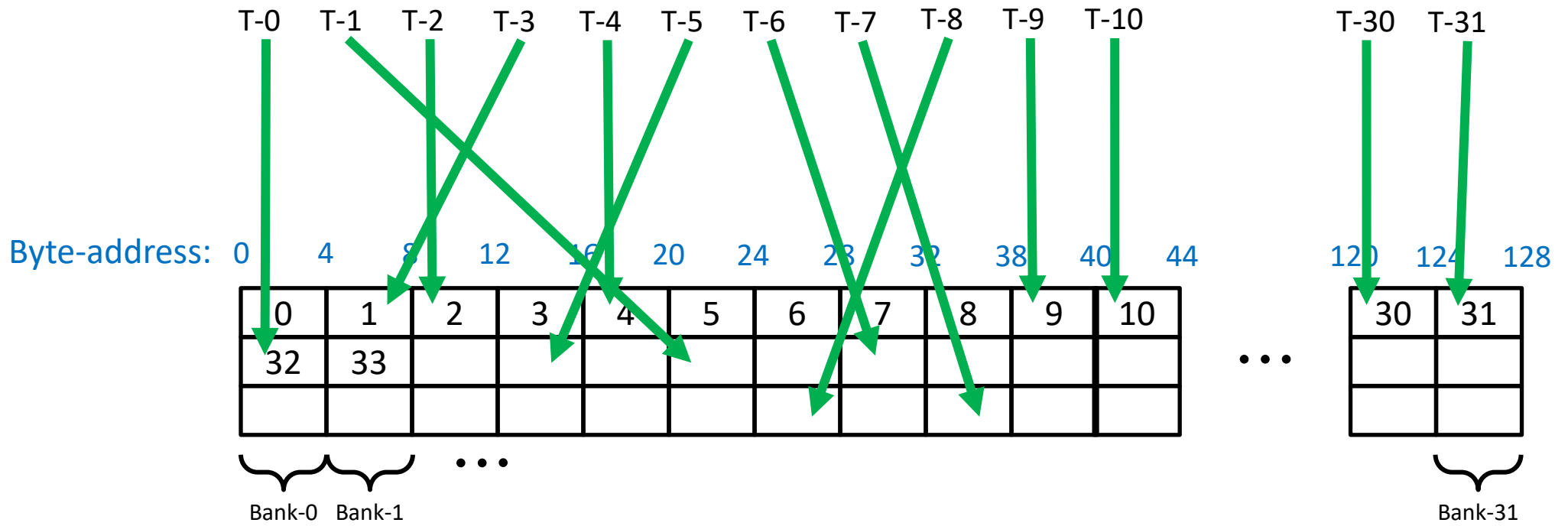
No Bank Conflicts



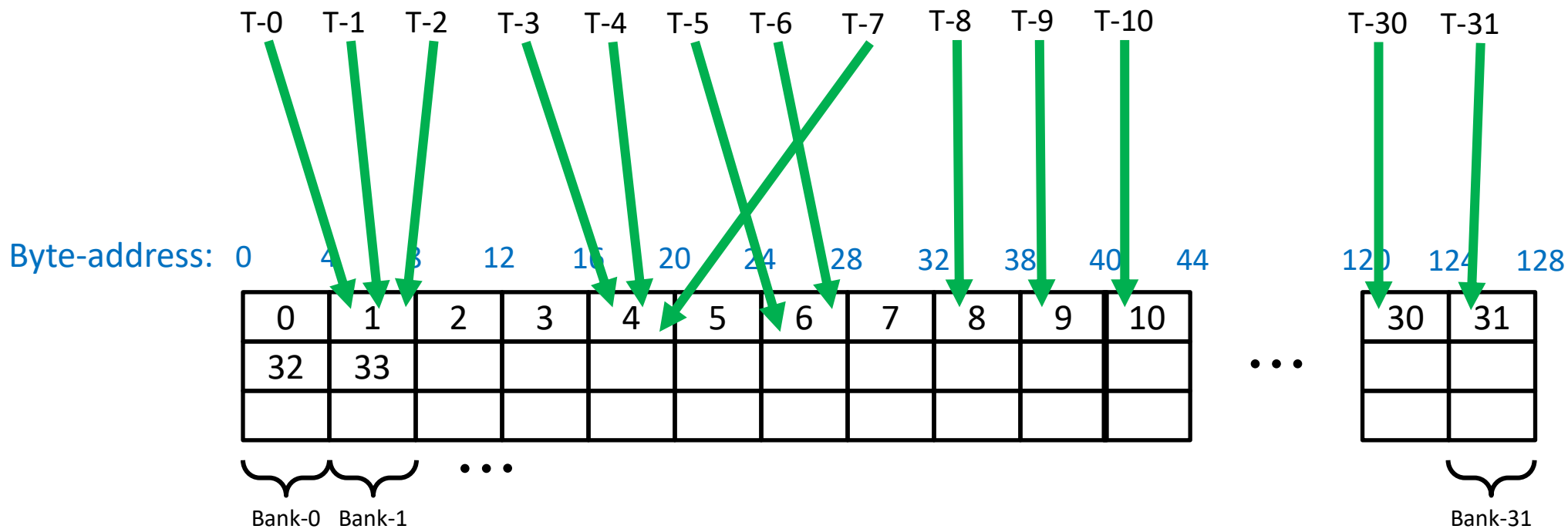
No Bank Conflicts



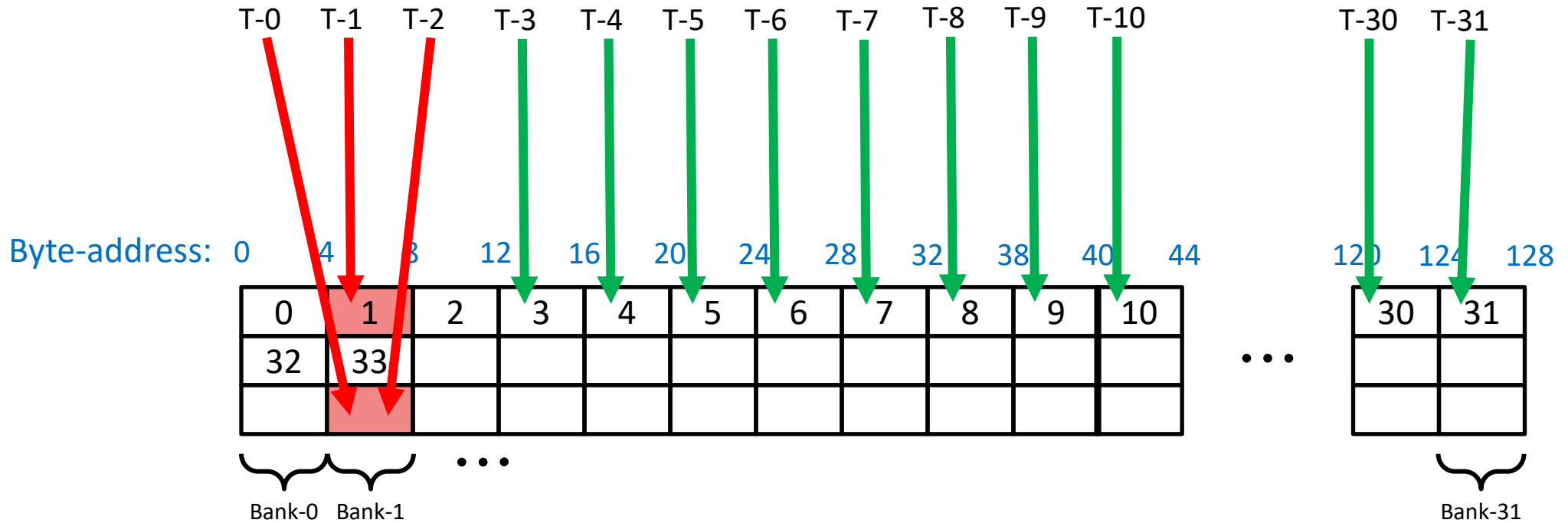
No Bank Conflicts



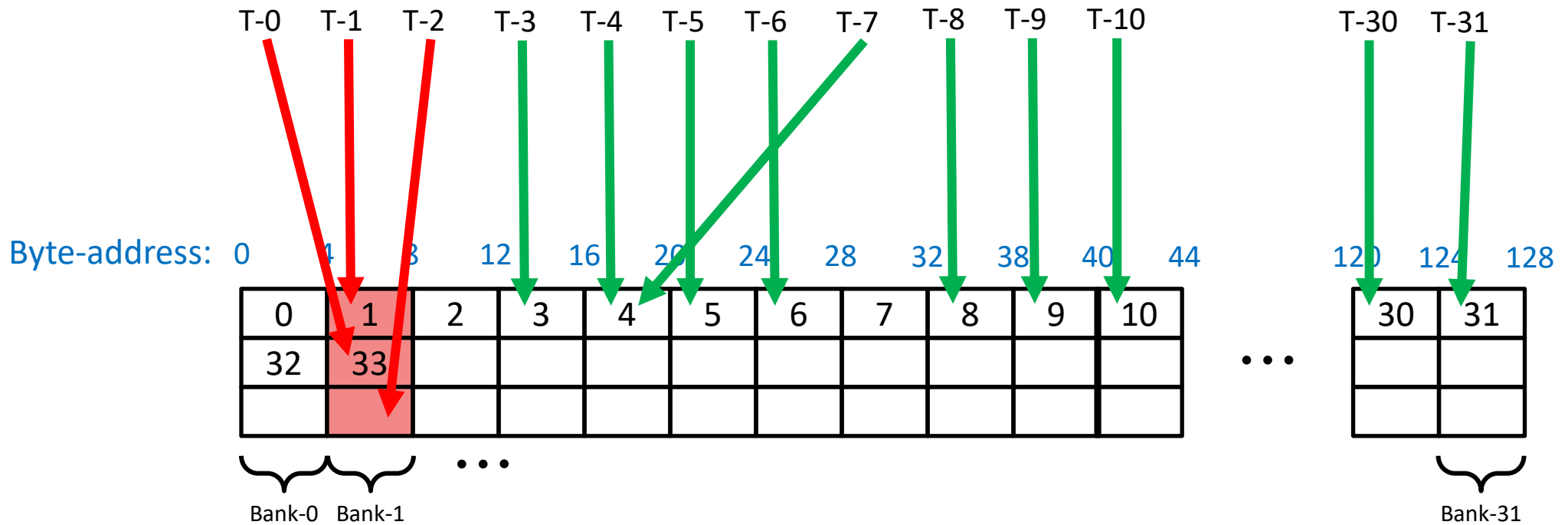
No Bank Conflicts



2-way Bank Conflict



3-way Bank Conflict



Bank Conflict and Access Phase

4B or smaller words

- Process addresses of all threads in a warp in a single phase.

8B words are accessed in 2 phases

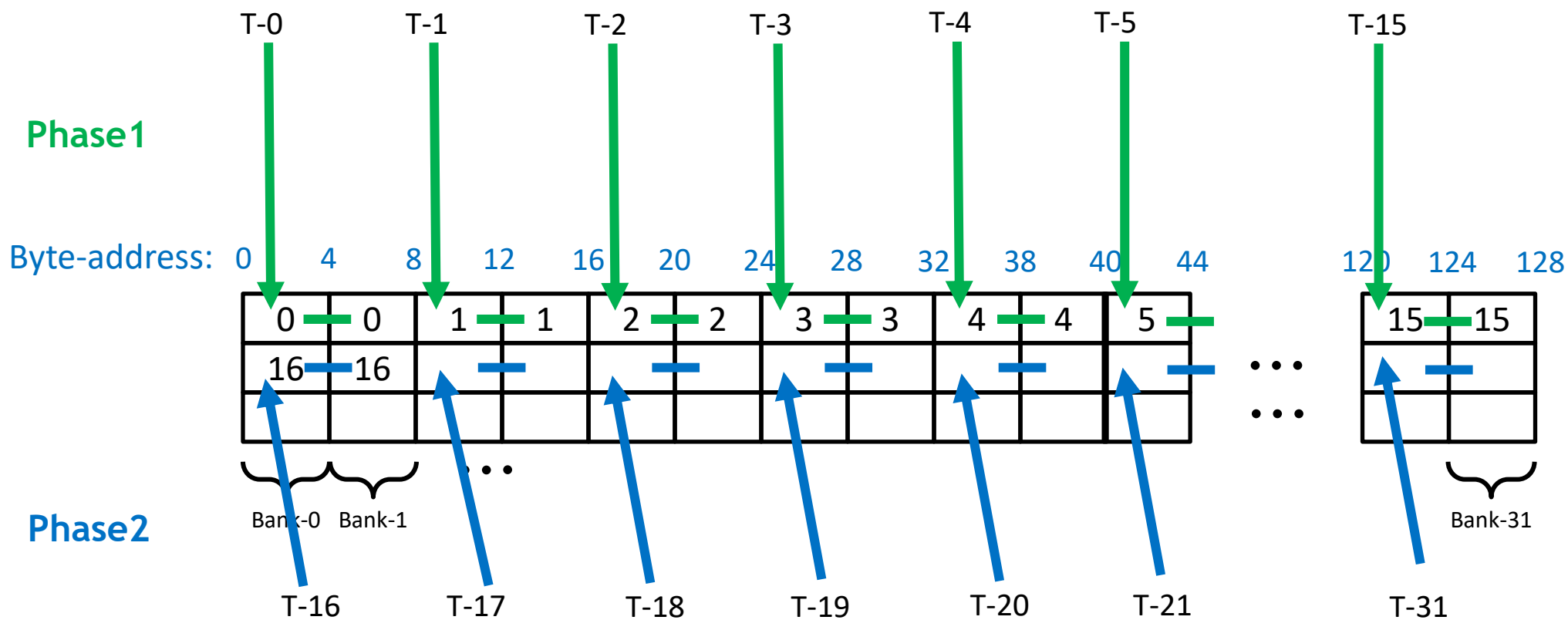
- Process addresses of the **first 16** threads in a warp.
- Process addresses of the **second 16** threads in a warp.

16B words are accessed in 4 phases

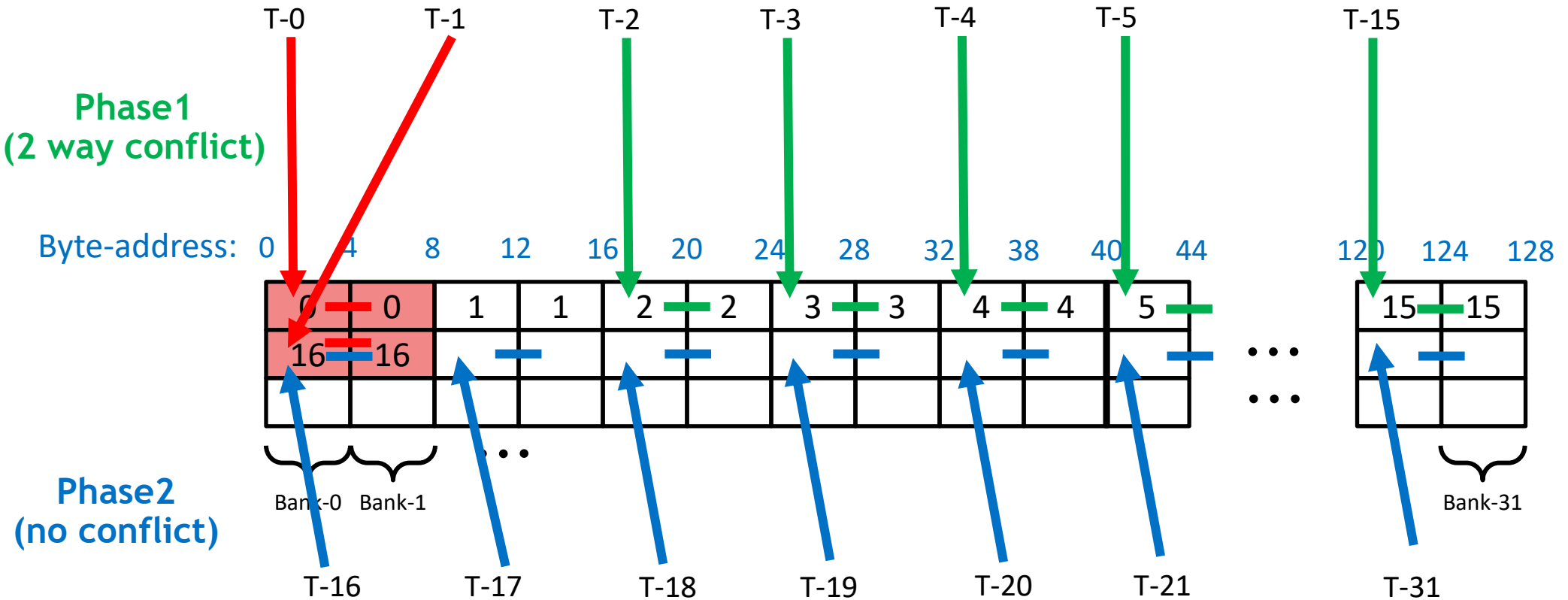
- Each phase processes a quarter of a warp.

Bank conflicts occur only between threads in the same phase

8B words, No Conflicts



8B words, 2-way Conflict



Case Study: Matrix Transpose

Staged via SMEM to coalesce GMEM addresses

32x32 blocks, single-precision values.

32x32 array in shared memory.

Initial implementation:

A warp reads a row from GMEM, writes to a row of SMEM.

Synchronize the threads in a block.

A warp reads a column of from SMEM, writes to a row in GMEM.

Case Study: Matrix Transpose

32x32 SMEM array (.e.g. `__shared__ float sm[32][32]`)

Warp accesses a row : No conflict.

Warp accesses a column : 32-way conflict.

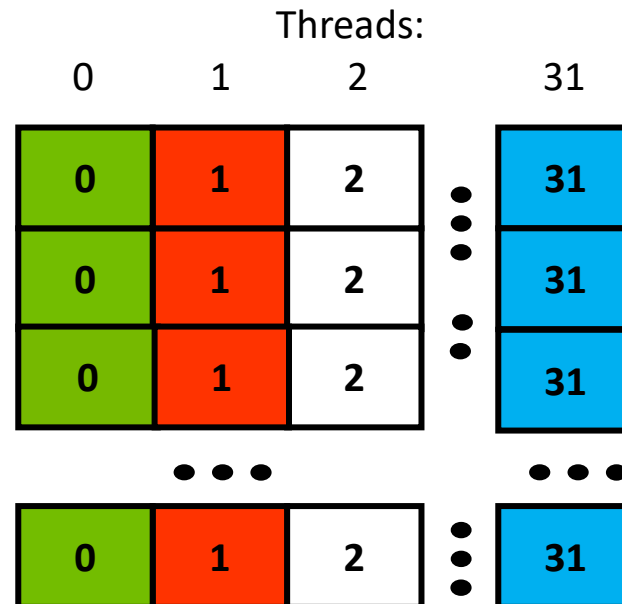
Number identifies which warp is accessing data.
Color indicates in which bank data resides.

Bank 0

Bank 1

...

Bank 31



Case Study: Matrix Transpose

Solution: add a column for padding: 32x33
(.e.g. `__shared__ float sm[32][33]`)

Warp accesses a row or a column: **no conflict.**

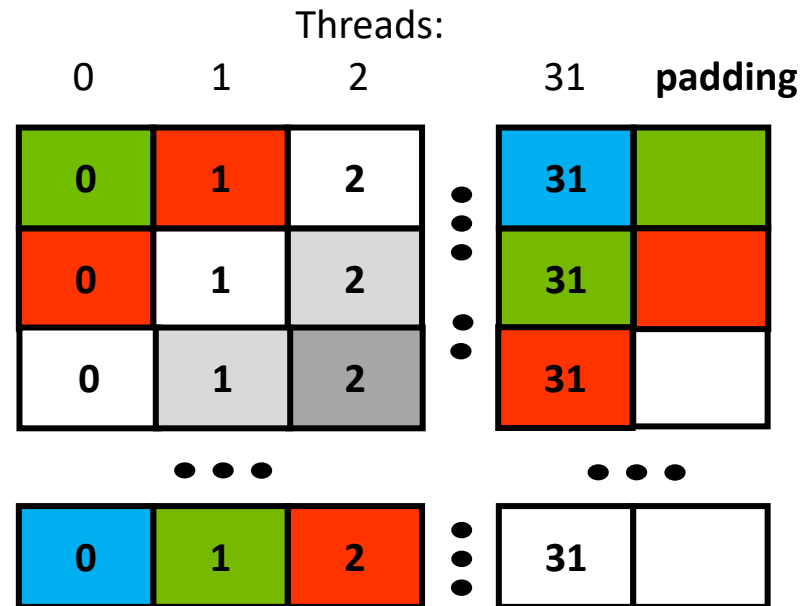
Number identifies which warp is accessing data.
Color indicates in which bank data resides.

Bank 0

Bank 1

...

Bank 31



Speedup
1.3x

Summary: Shared Memory

Shared memory is a precious resource

Very high bandwidth (14 TB/s), much lower latency than Global Memory.

Data is programmer-managed, no evictions by hardware.

Volta/Ampere: up to 96/164KB of shared memory per thread block.

4B granularity.

Performance issues to look out for

Bank conflicts add latency and reduce throughput.

Use profiling tools to identify bank conflicts.

10 Commandments of Performance Optimizations

1. Efficient Algorithm
2. Thread Divergence
3. Coalesced Memory Accesses
4. Shared Memory
5. Bank Conflict
6. Intrinsic Functions
7. Correct Data Types
8. Vectorized Operations
9. Using Tensor Cores
10. Using Streams

