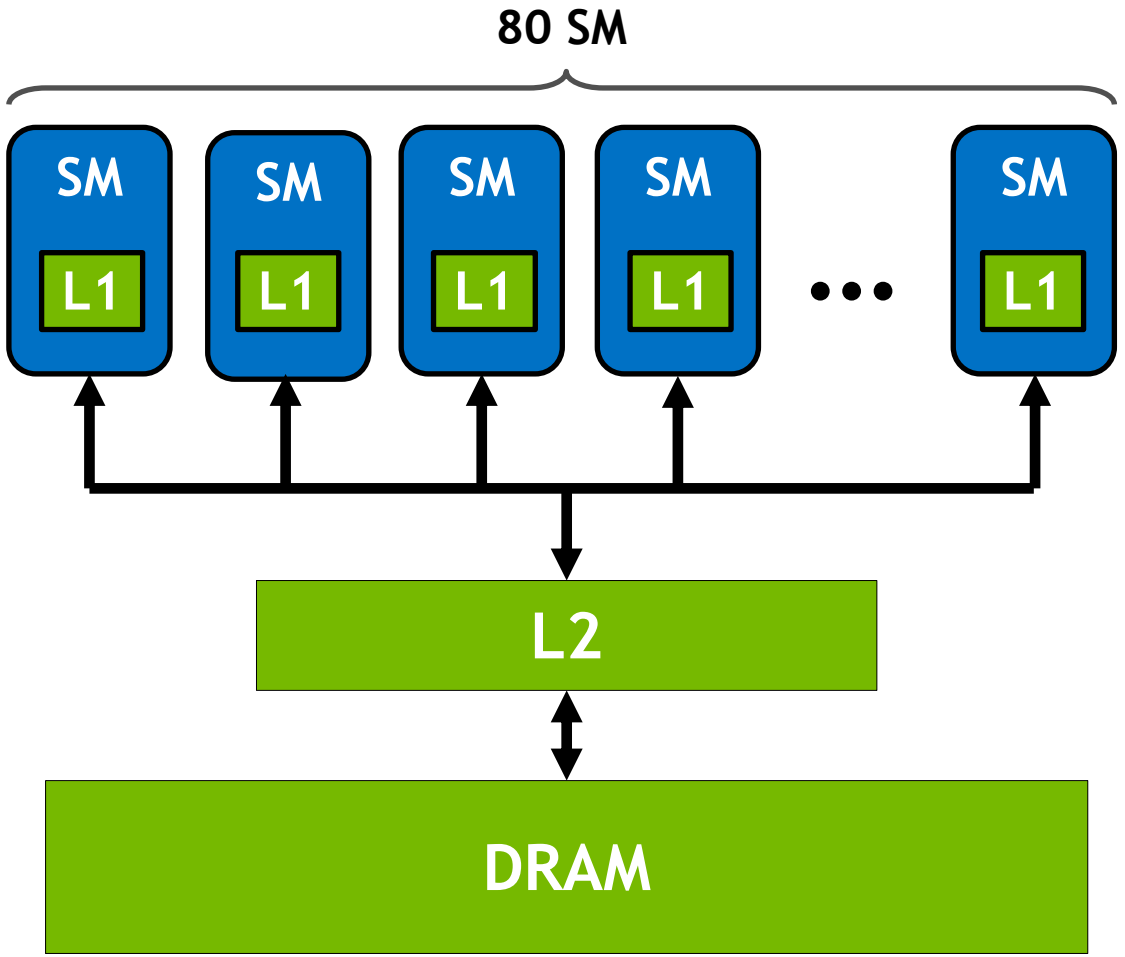
CUDA - Performance optimization

Hugo Braun
AI Developer Technology
Slides courtesy of:
Guillaume Thomas-Collignon

VOLTA V100



Per Streaming Multiprocessor:

- **64 FP32** lanes
- **32 FP64** lanes
- **64 INT32** lanes
- **16 SFU** lanes (transcendentals)
- **32 LD/ST** lanes (Gmem/Lmem/Smem)
- **8 Tensor Cores**
- **4 TEX** lanes

SM Resources

Each thread block needs:

- **Registers** (#registers/thread x #threads)
- **Shared memory** (0 ~ 96 KB)

Volta limits per SM:

- 256KB Registers
- 96KB Shared memory
- 2048 threads max (64 warps)
- 32 thread blocks max

Can schedule any resident warp without context switch

Running Faster

Solving the bottlenecks

A piece of code can be:

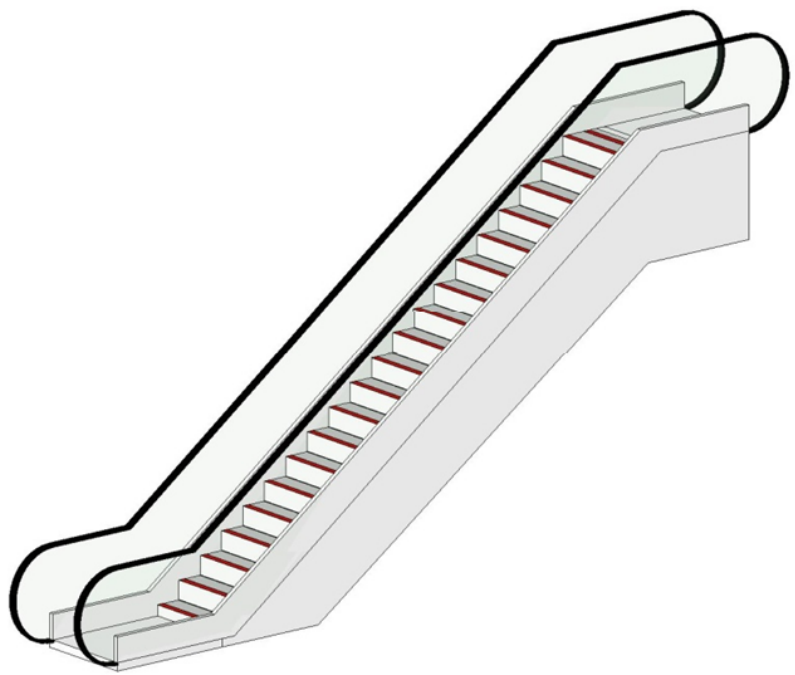
- **Compute bound** (saturating compute units)
  Solution: Reduce the number of instructions executed
  Using vector types, intrinsics, tensor cores, FMAs
- **Bandwidth bound** (saturating memory bandwidth)
  Solution: Reduce the amount of data transferred
  Optimal access patterns, using lower precision
- **Latency bound**
  Solution: 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

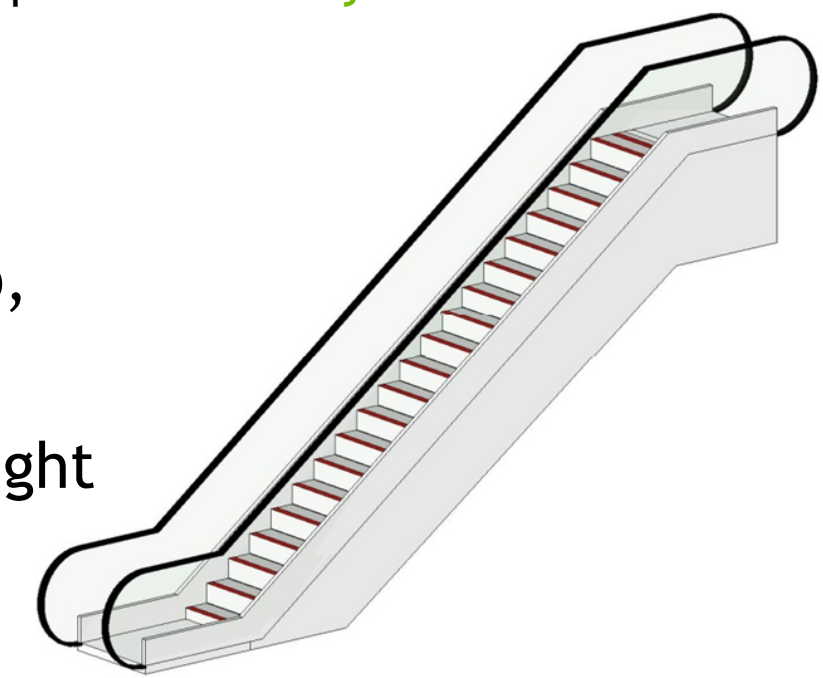


Little's Law

For Escalators

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

- One person in flight ?
  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



Little's law

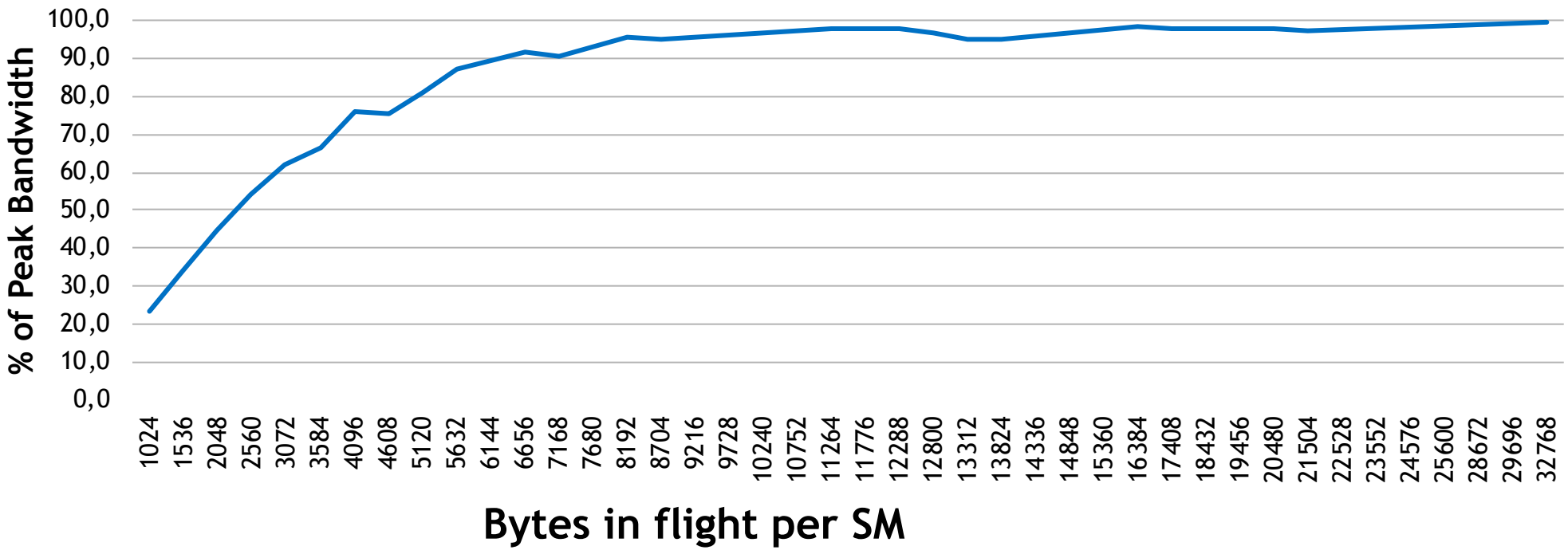
For GPUs

Optimization goals:

1. **Saturate Compute units**
   Accelerate **computing**
   Get close to the peak performance
2. **Saturate Memory Bandwidth**
   If compute density too low to saturate computation

3. Need to hide the latencies to achieve this

Memory Bandwidth



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

Little's law

Takeaways

- Maximize # instructions in flight =

  # instructions in flight / thread * # threads executing

  Instruction parallelism **Occupancy + Control flow**

SM Resources

Limiting the number of threads executing

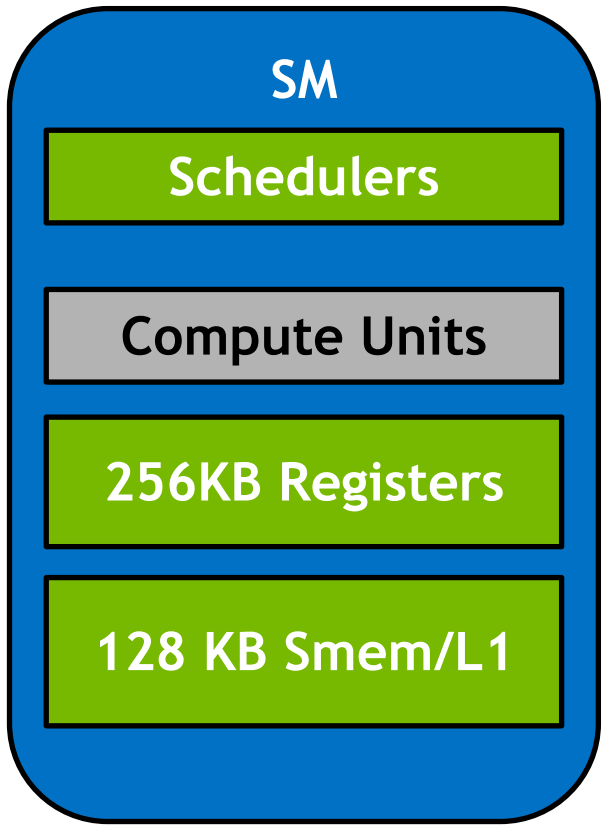
Each thread block needs:

- **Registers** (#registers/thread x #threads)
- **Shared memory** (0 ~ 96 KB)

Volta limits per SM:

- 256KB Registers
- 96KB Shared memory
- 2048 threads max (64 warps)
- 32 thread blocks max

Can schedule any resident warp without context switch



Occupancy

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

Use the profiler !

Higher occupancy can help to hide latency!
SM has more warp candidates to schedule while other warps are waiting for instructions to complete

Achieved occupancy vs **theoretical occupancy**
Need to run enough thread blocks to fill all the SMs!

Control Flow

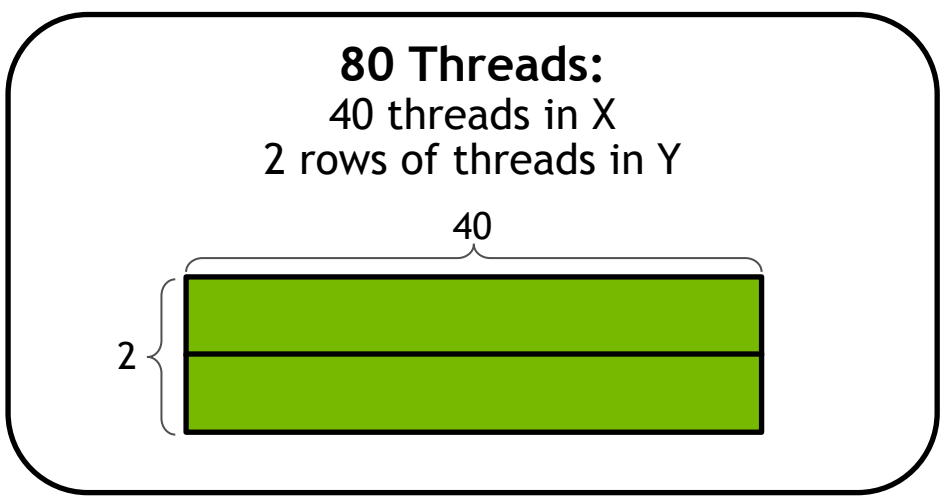
Blocks of threads, warps

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

Control Flow

Mapping threads

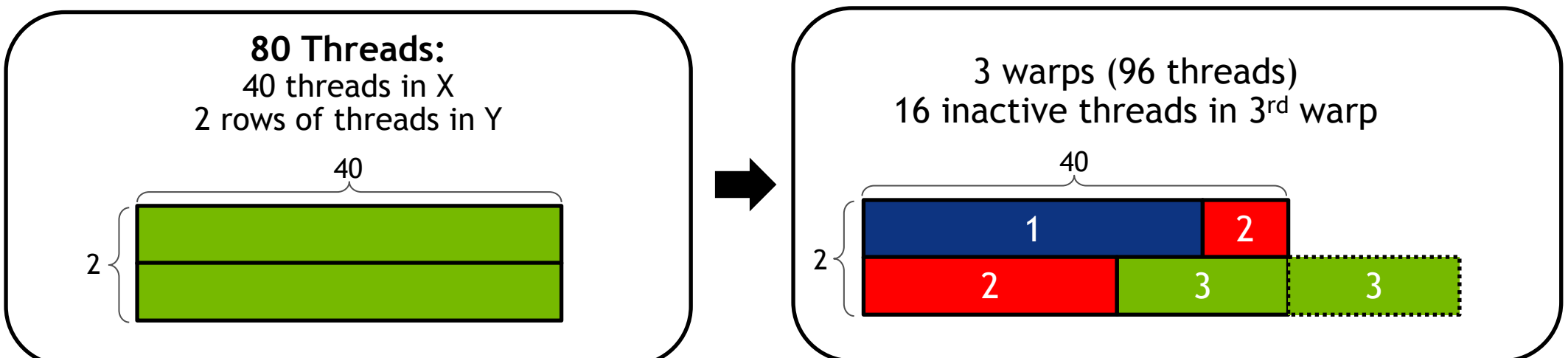
- **Thread blocks can be 1D, 2D, 3D**
  Only for convenience. HW “looks” at threads in 1D
- **Consecutive 32 threads** belong to the same **warp**



Control Flow

Mapping threads

- **Thread blocks can be 1D, 2D, 3D**
  Only for convenience. HW “looks” at threads in 1D
- **Consecutive 32 threads** belong to the same **warp**

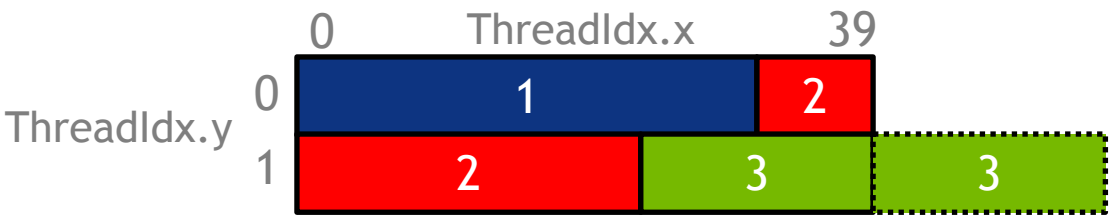


Control Flow

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 don't participate are masked out,
  but the whole warp executes both sides of the branch

Control Flow



Instructions, time

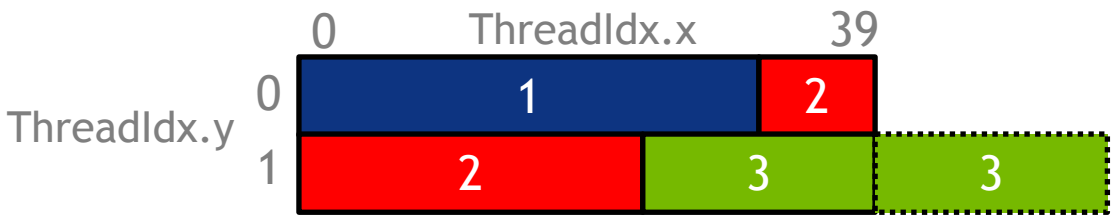
```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```

Warp 1: 0 … 31

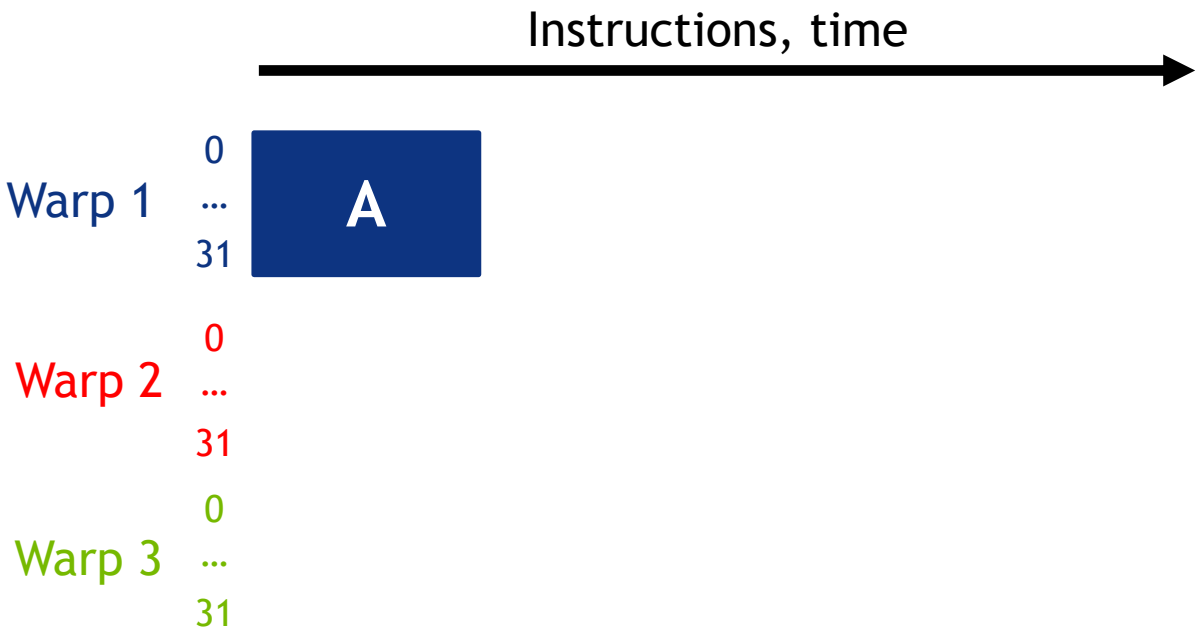
Warp 2: 0 … 31

Warp 3: 0 … 31

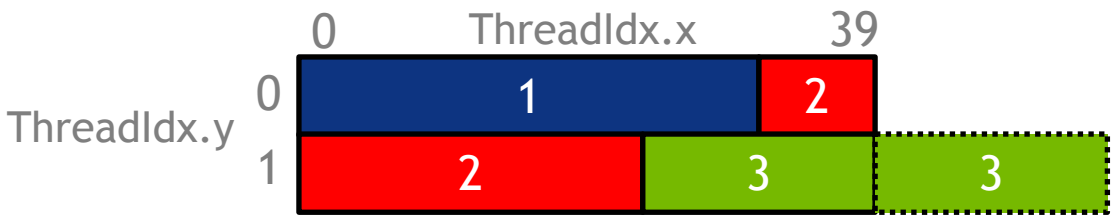
Control Flow



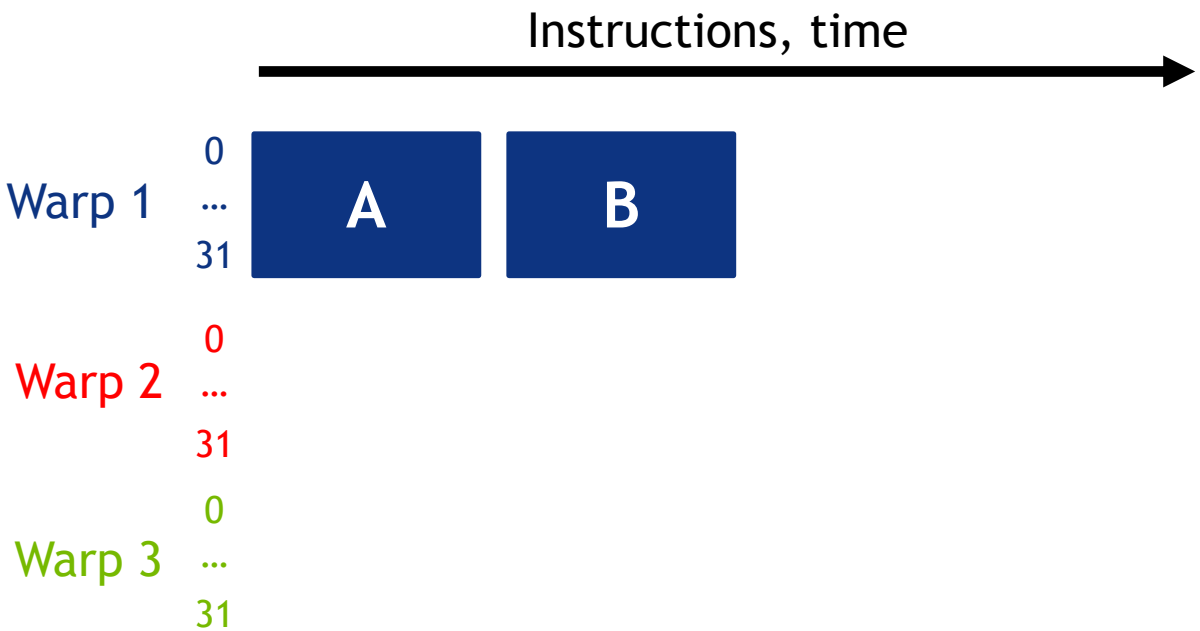

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



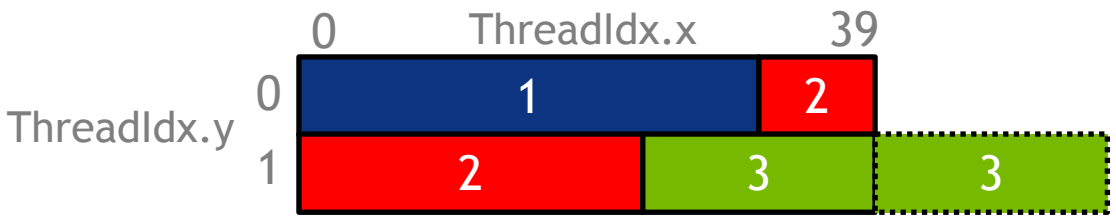
Control Flow



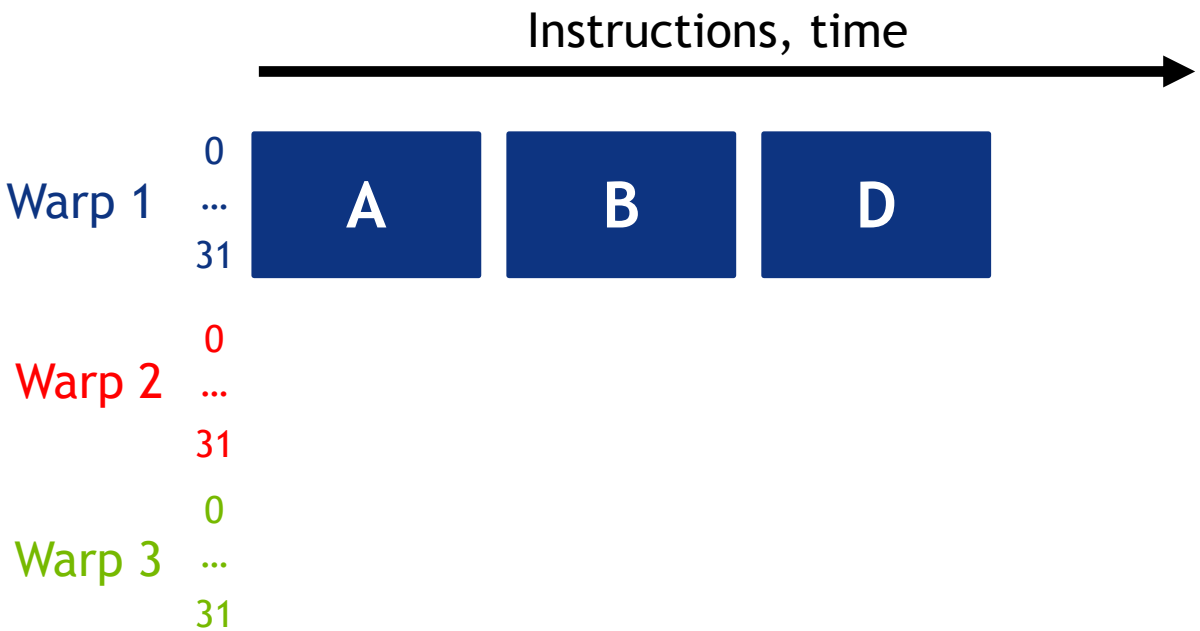

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



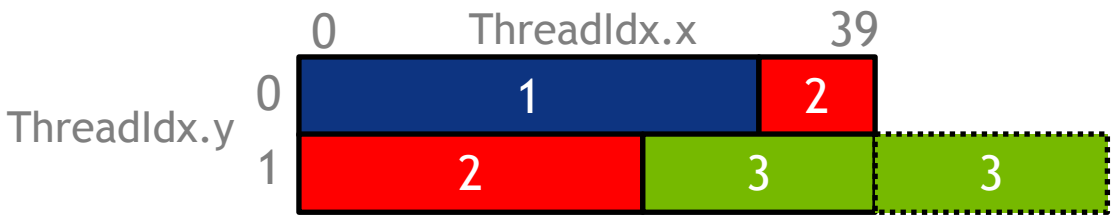
Control Flow



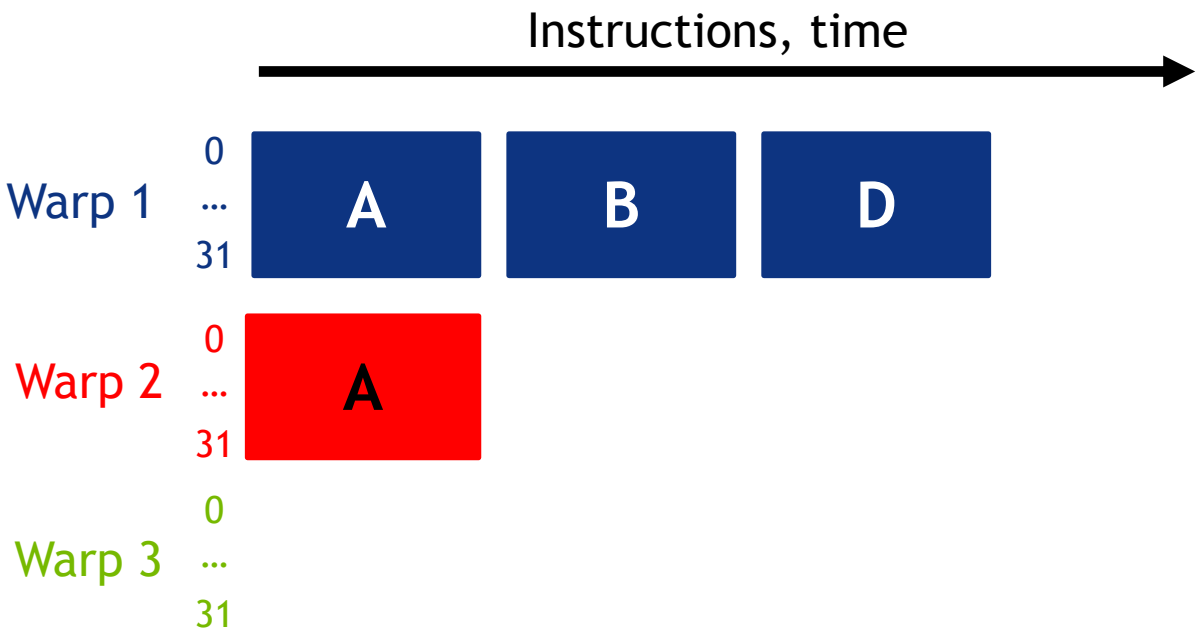

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



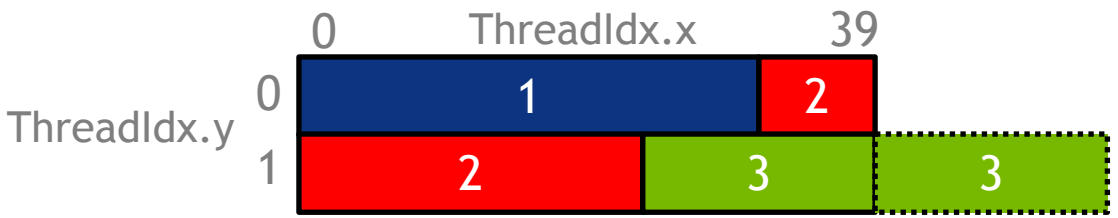
Control Flow



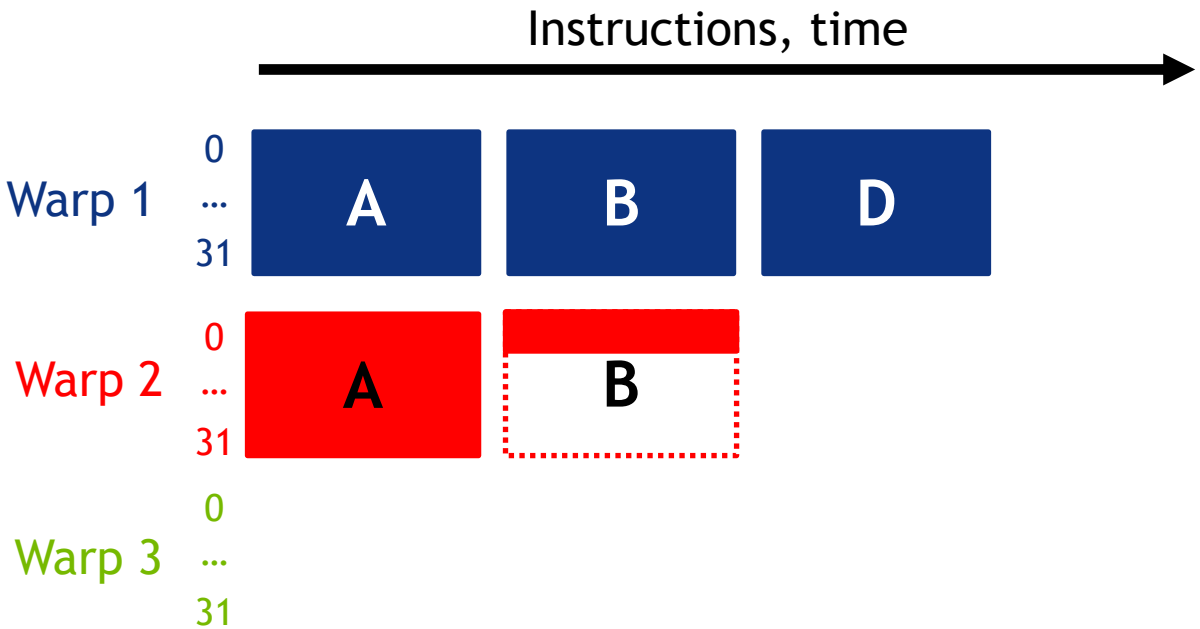

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



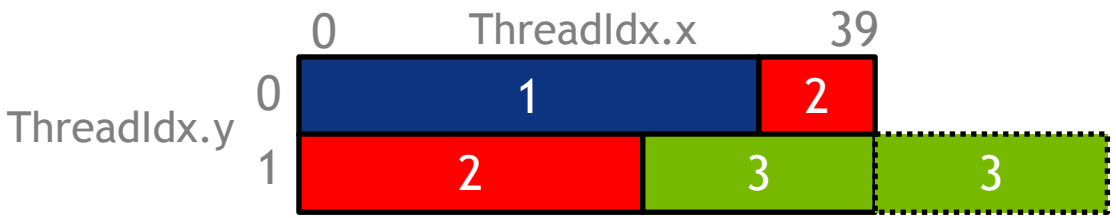
Control Flow



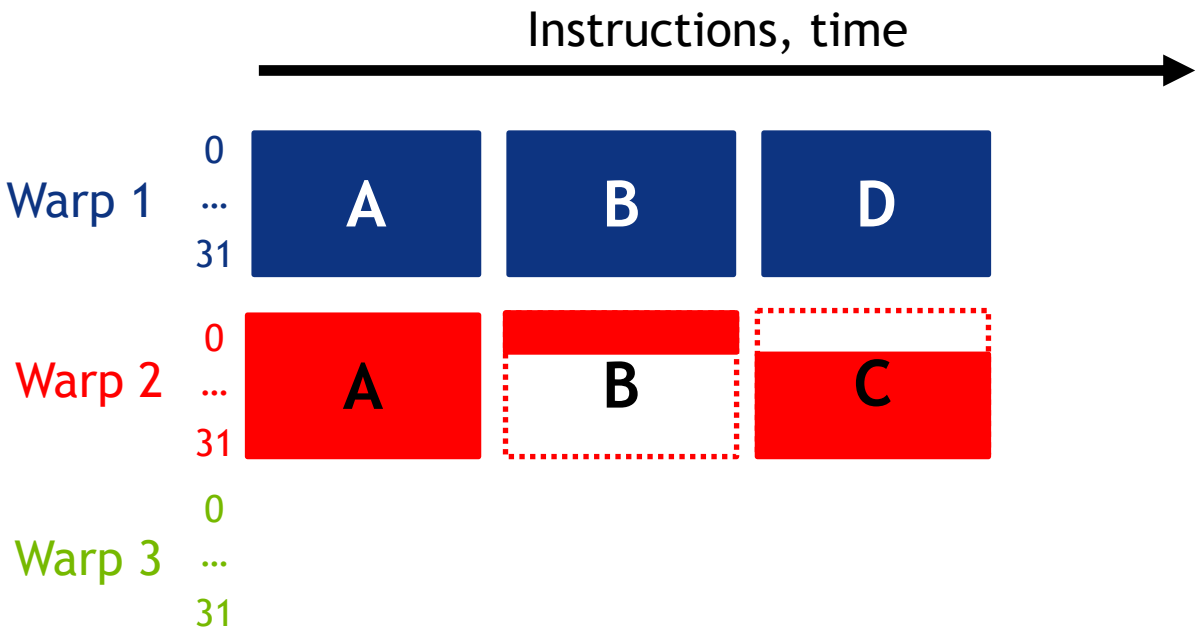

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



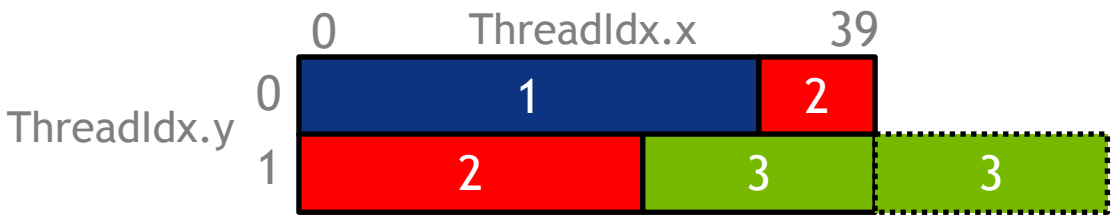
Control Flow



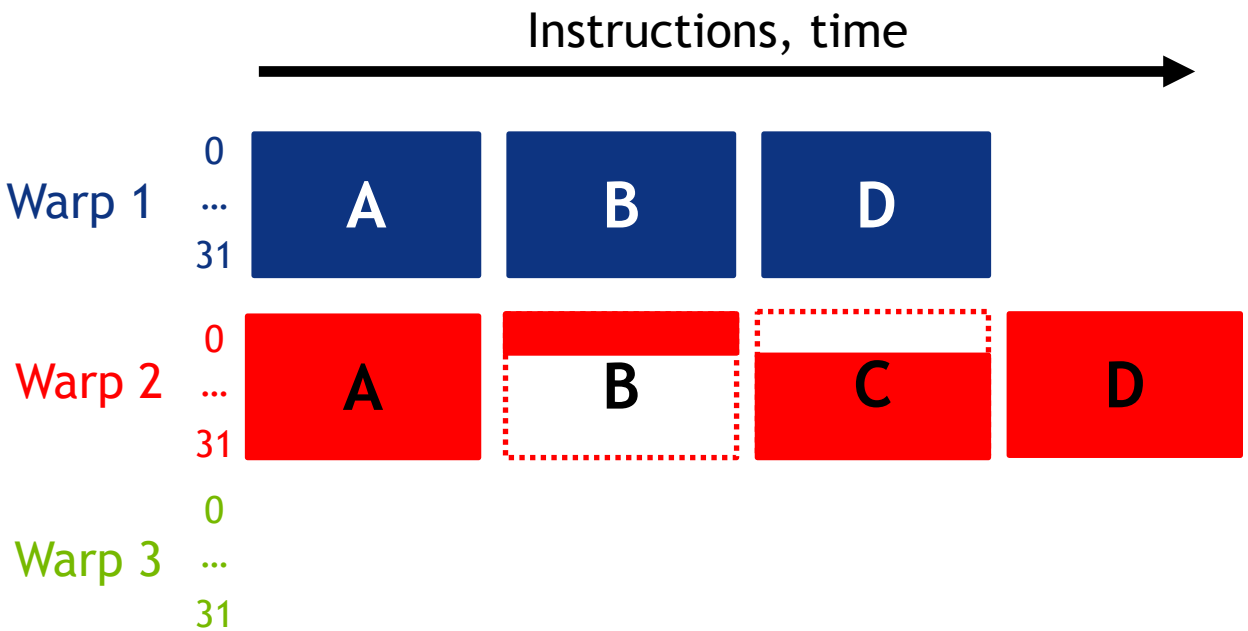

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



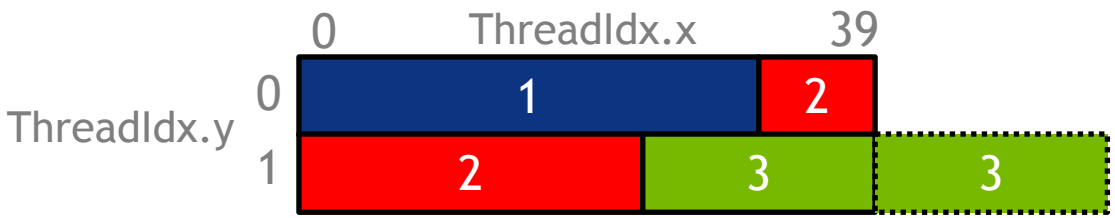
Control Flow



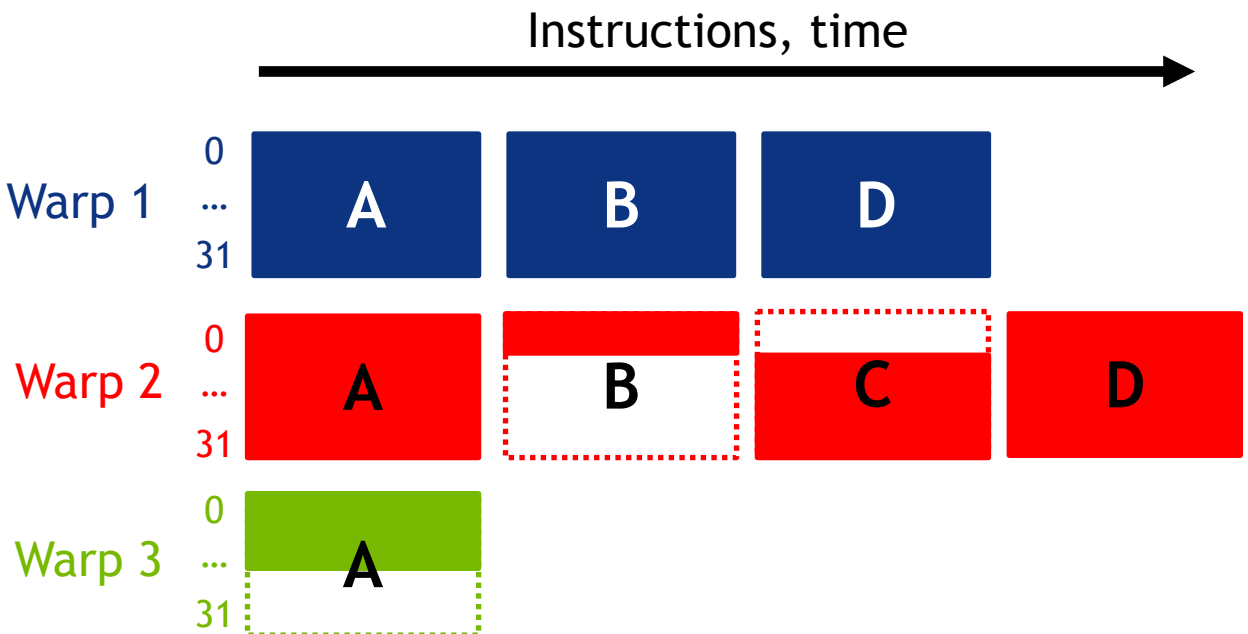

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



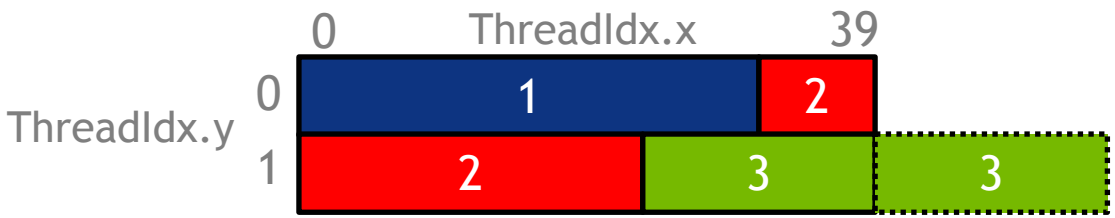
Control Flow



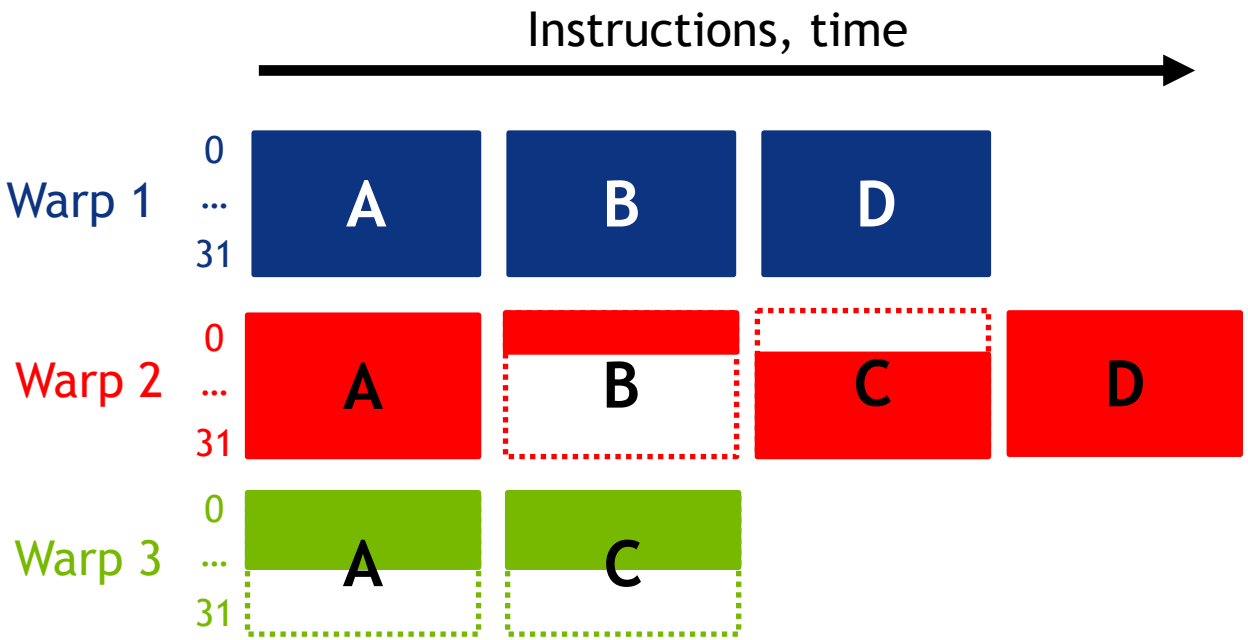

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



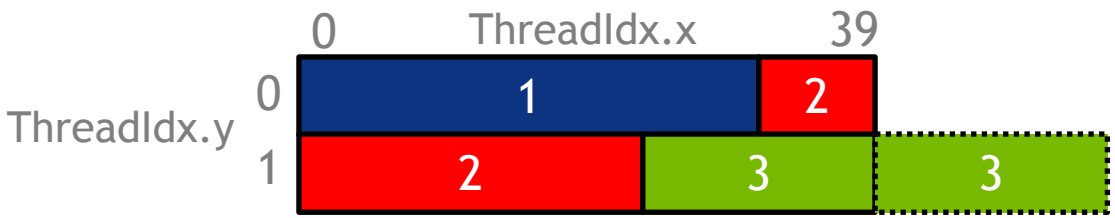
Control Flow



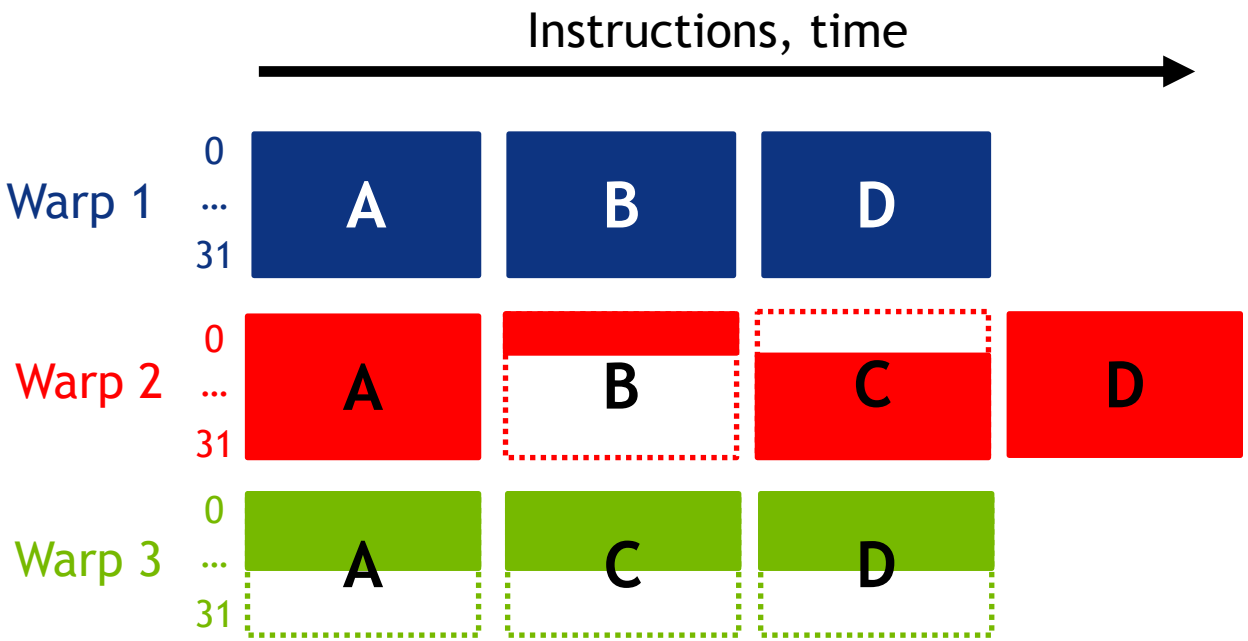

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



Control Flow




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



Control Flow

Takeaways

- 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())

Increasing In-Flight instructions

2 Ways to improve parallelism:

- **Improve occupancy**
  More threads -> more instructions
- **Improve instruction parallelism (ILP)**
  More **independent instructions** per thread
- **Using vectorized instructions**

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

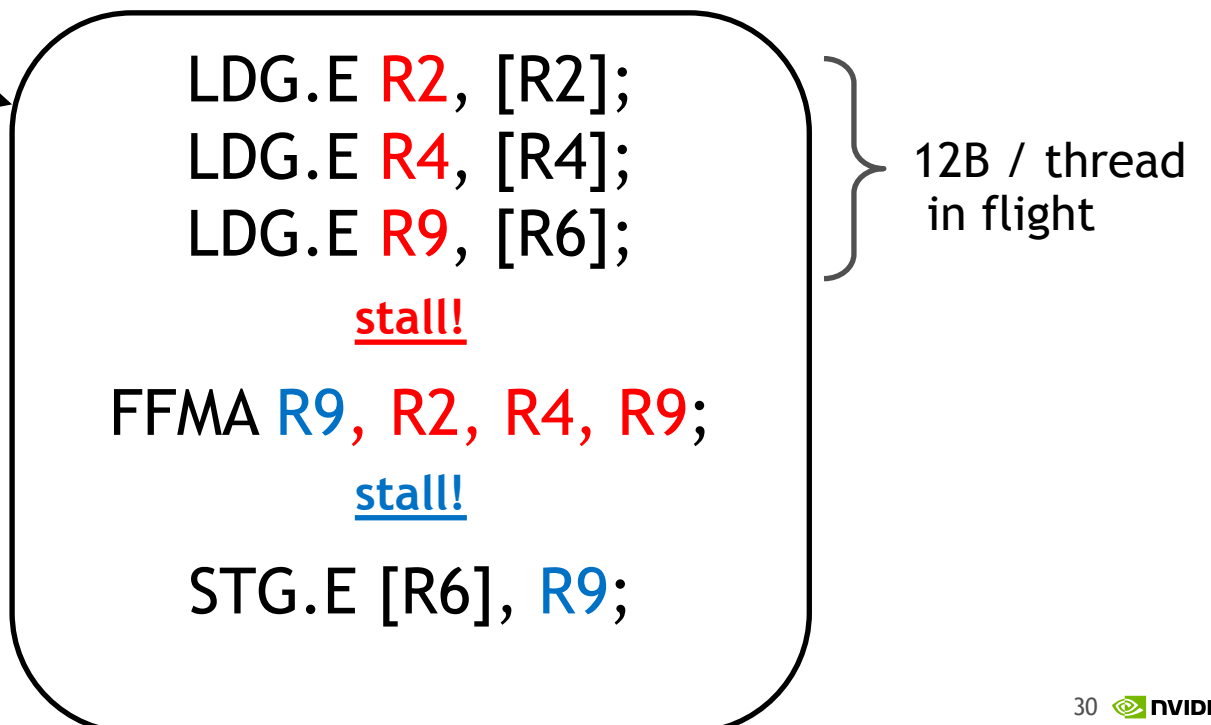
  Some pipelines need multiple cycles to issue a warp

- All the **arguments are ready**

  Argument isn't ready if a previous instruction hasn't yet produced it

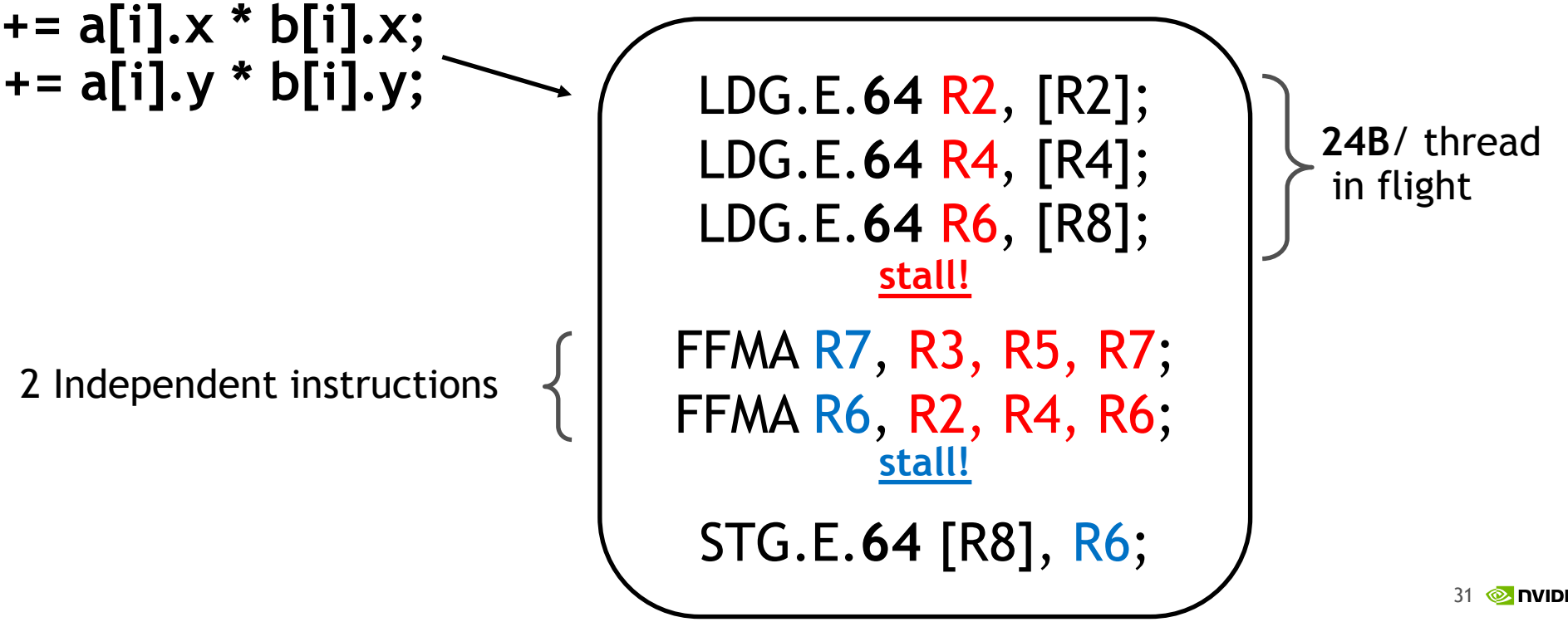
Instruction Issue Example

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



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;
}
```



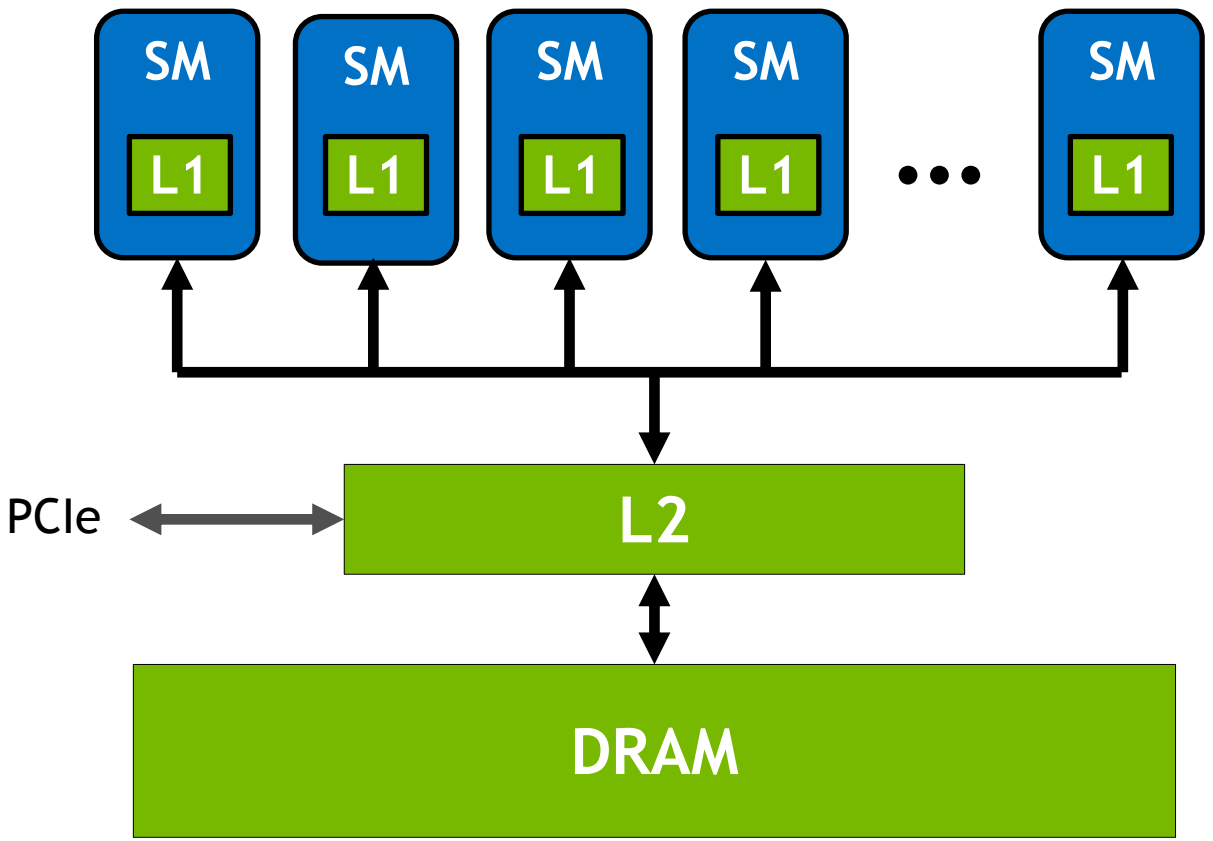
Fast Math intrinsics

Fast but less accurate math intrinsics are available.
2 ways to use the intrinsics:

- Whole file: compile with **--fast-math**
- Individual calls
  E.g. **__sinf(x), __logf(x), __fdivide(x,y)**

Volta's Memory System

V100



80 Streaming Multiprocessors
256KB register file (20 MB)

Unified Shared Mem / L1 Cache
128KB, Variable split
(~10MB Total, 14 TB/s)

6 MB L2 Cache
(2.5TB/s Read, 1.6TB/s Write)

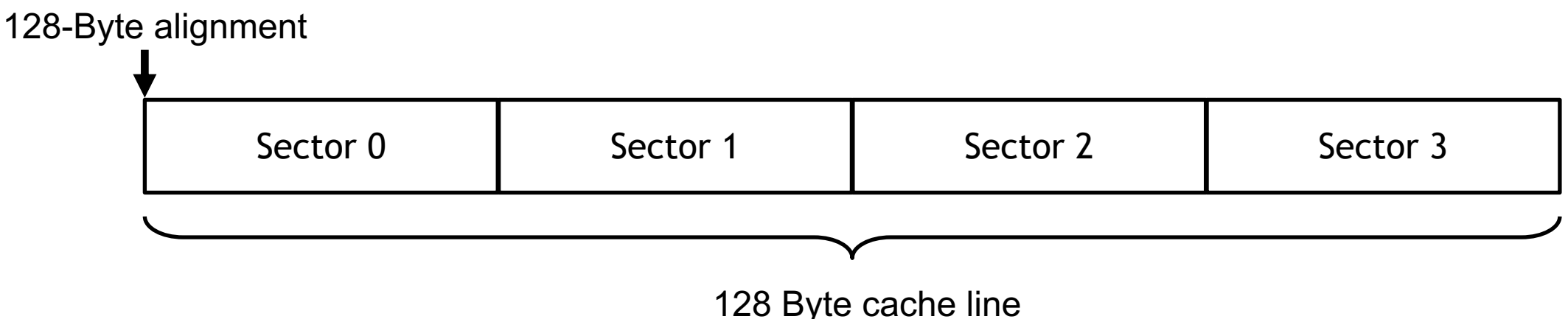
16/32 GB HBM2 (900 GB/s)
"Free" ECC.

Cache Lines & Sectors

Moving data between L1, L2, DRAM

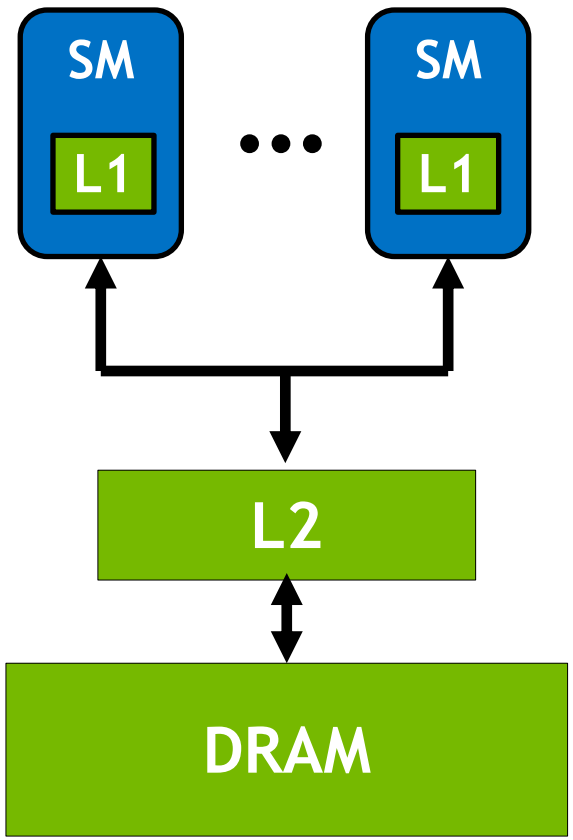
Memory access granularity = **32 Bytes = 1 sector**
(32B for Maxwell, Pascal, Volta. Kepler and before: variable, 32B or 128B, depending on architecture, access type, caching / non-caching options)

A **cache line is 128 Bytes**, made of **4 sectors.**
Cache ”management” granularity = 1 cache line



Memory Reads

Getting data from Global Memory

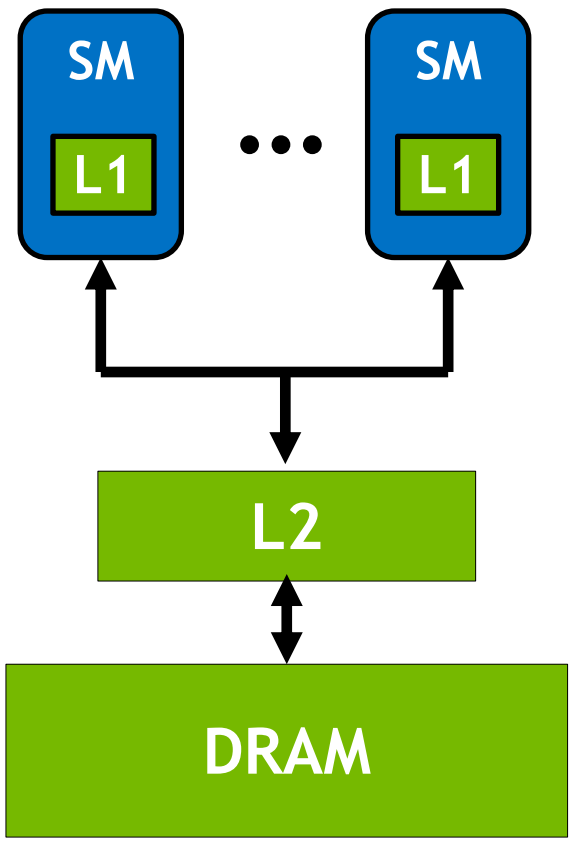


Checking if the data is in L1 (if not, check L2)

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).

Instruction modifiers can influence cache behavior (inline PTX only)

L1, L2 Caches

Why do 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 do GPU have caches?

Caches on GPUs are useful for:

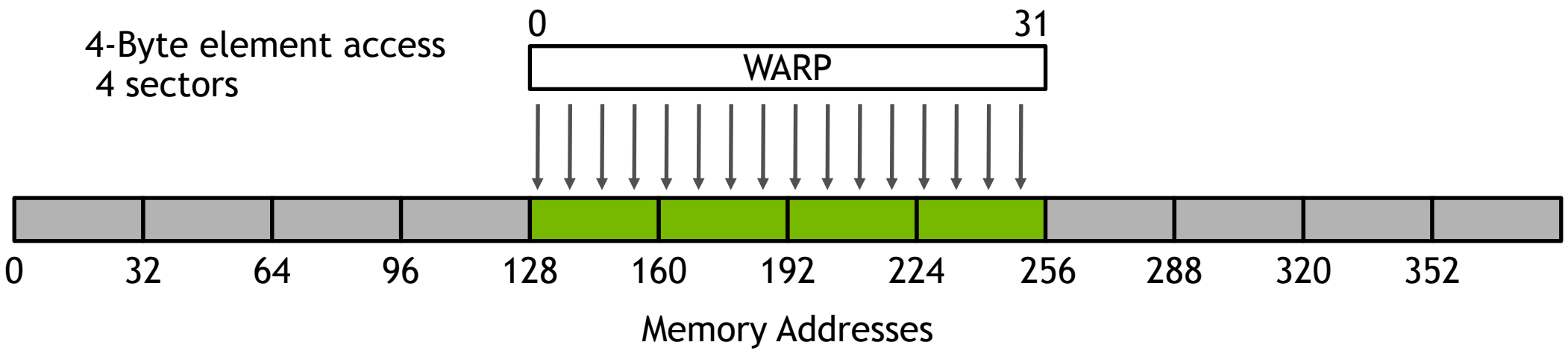
- “Smoothing” irregular, unaligned access patterns
- Caching common data accessed by many threads
- Faster register spills, local memory
- Fast atomics
- Codes that don’t use shared memory (naïve code, OpenACC, …)

Access Patterns

Warps and Sectors

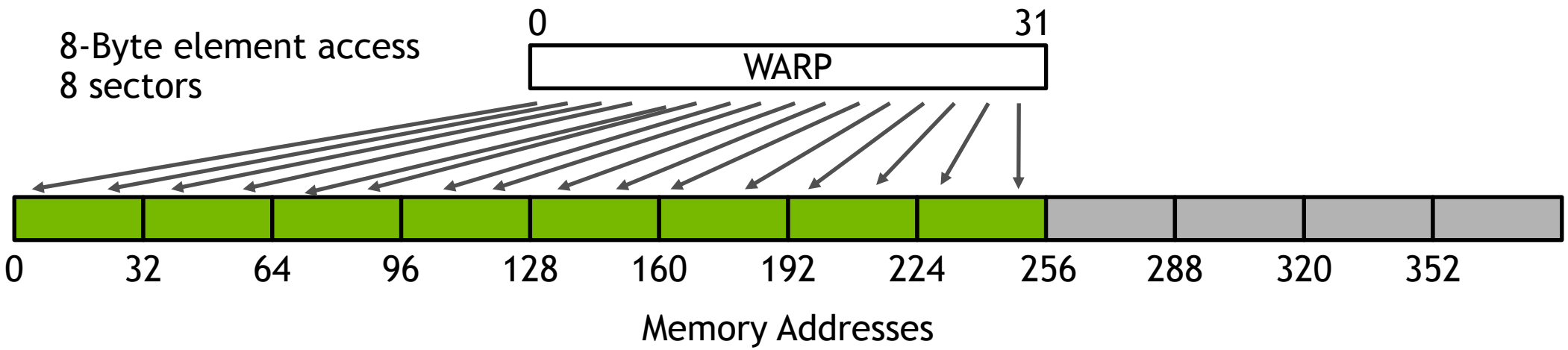
For each warp: How many **sectors** needed?

Depends on addresses, active threads, access size.
Natural element sizes = 1B, 2B, 4B, 8B, 16B.



Access Patterns

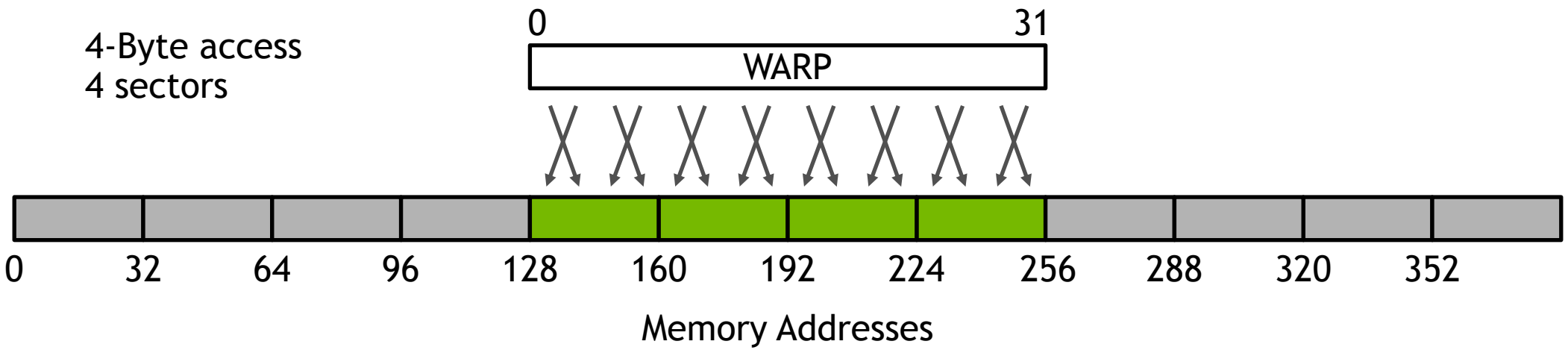
Warps and Sectors



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

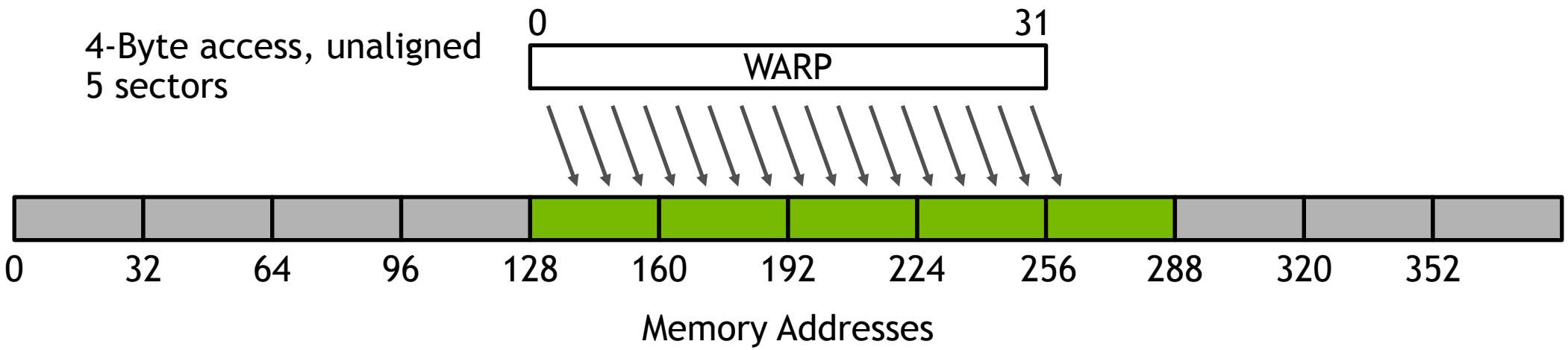
Access Patterns

Warps and Sectors



Access Patterns

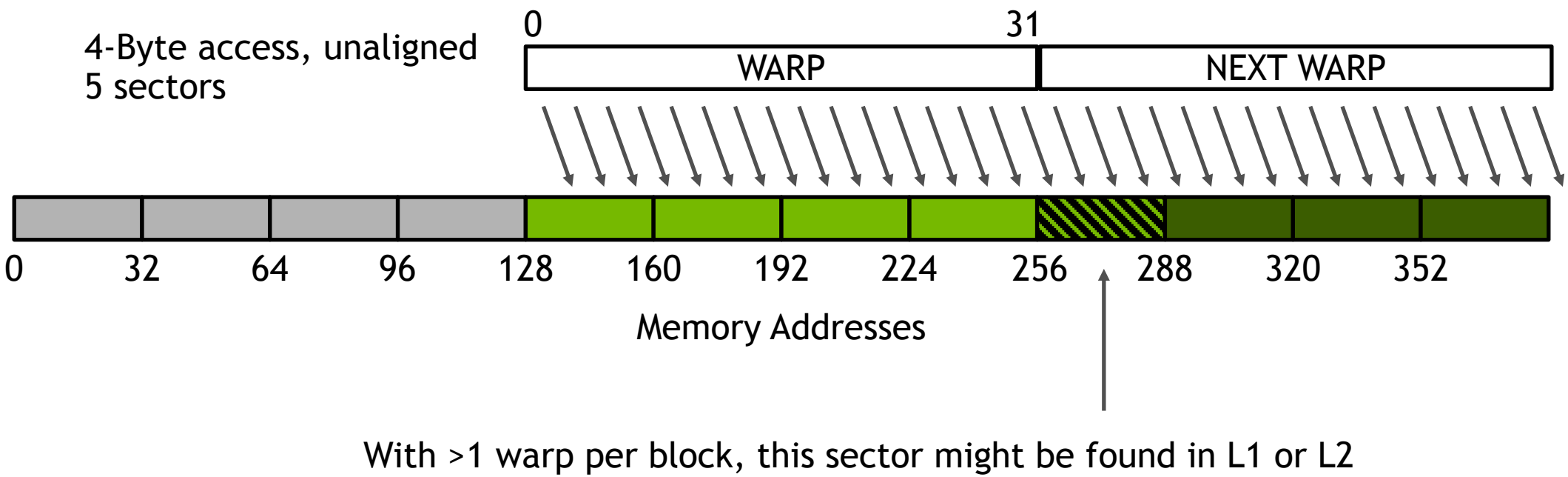
Warps and Sectors



128 bytes requested, 160 bytes read (80% efficiency)

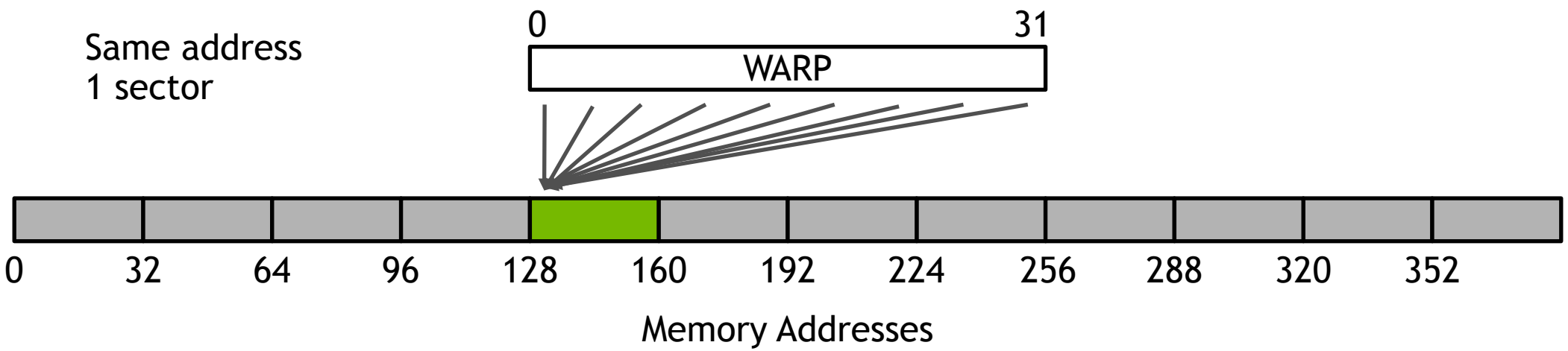
Access Patterns

Warps and Sectors



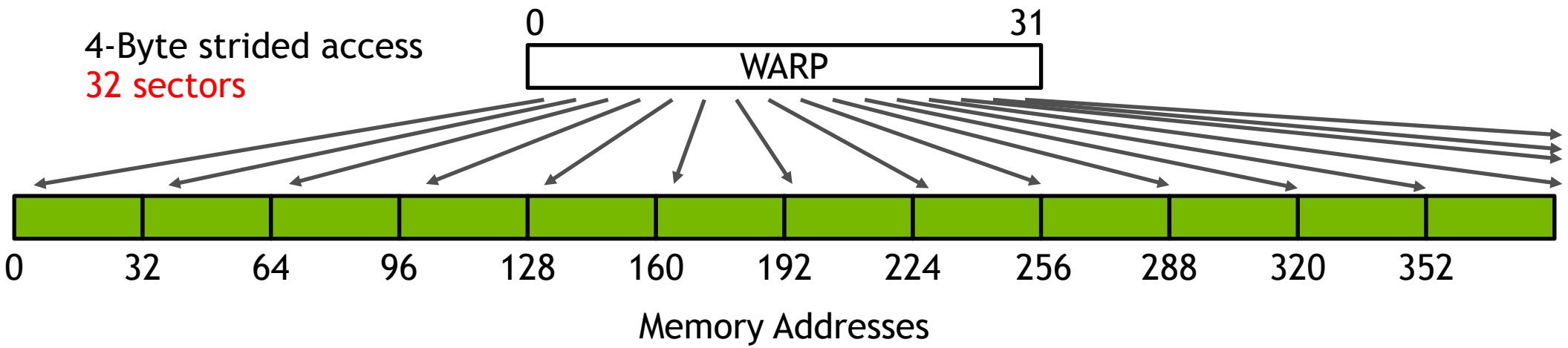
Access Patterns

Warps and Sectors



Access Patterns

Warps and Sectors



128 bytes requested, 1024 bytes transferred!
Using only a few bytes per sector. Wasting lots of BW!

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

Shared Memory

Scratch-pad memory on each SM

- **User-managed cache**, HW does not evict data
- Data written to SMEM stays there till user overwrites

Useful for:

- **Storing frequently-accessed data**, to reduce DRAM accesses
- **Communication** among threads of a threadblock

Performance benefits compared to DRAM:

- **20-40x lower latency**
- **~15x higher bandwidth**
- Accessed at 4-byte granularity
- GMEM granularity is 32B.

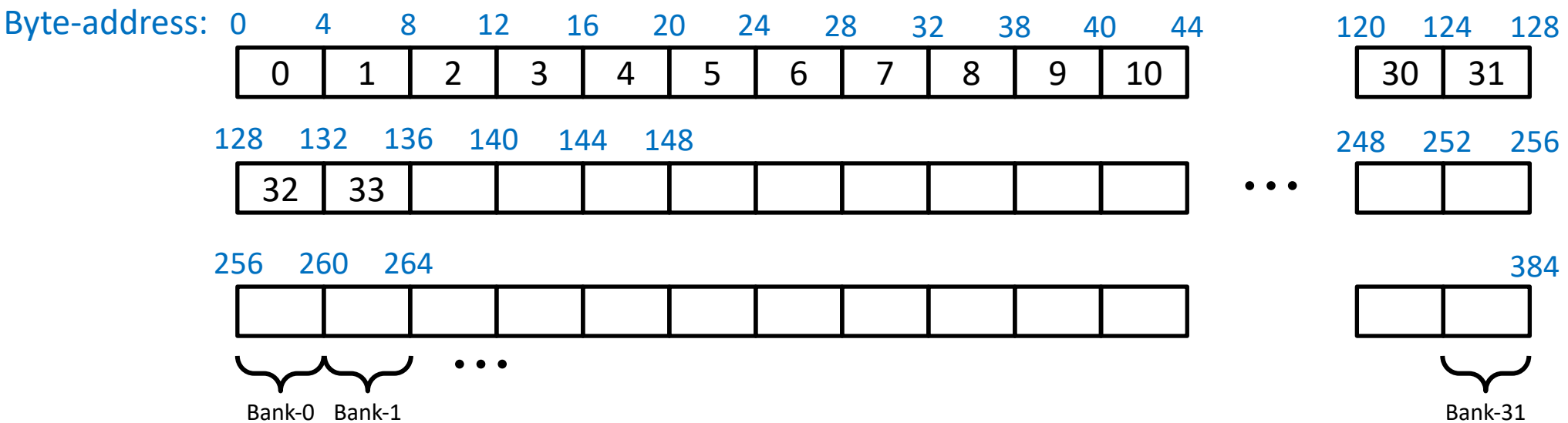
Volta Shared Memory

- **Default 48KB/threadblock, opt in to get 96KB**
- **32 banks, 4 bytes wide**
  - Bandwidth: 4 bytes per bank per clock per SM
    128 bytes per clk per SM
  - V100: ~14 TB/s aggregate across 80 SMs
- **Mapping addresses to banks:**
  - Successive 4-byte 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 SMEM Banks

With 4-Bytes data



Shared Memory Instruction Operation

Threads in a warp provide addresses

HW determines into which 4-byte words addresses fall

Reads (LDS):

Fetch the data, distribute the requested bytes among threads
Multi-cast capable

Writes (STS):

Multiple threads writing the same address: one “wins”

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

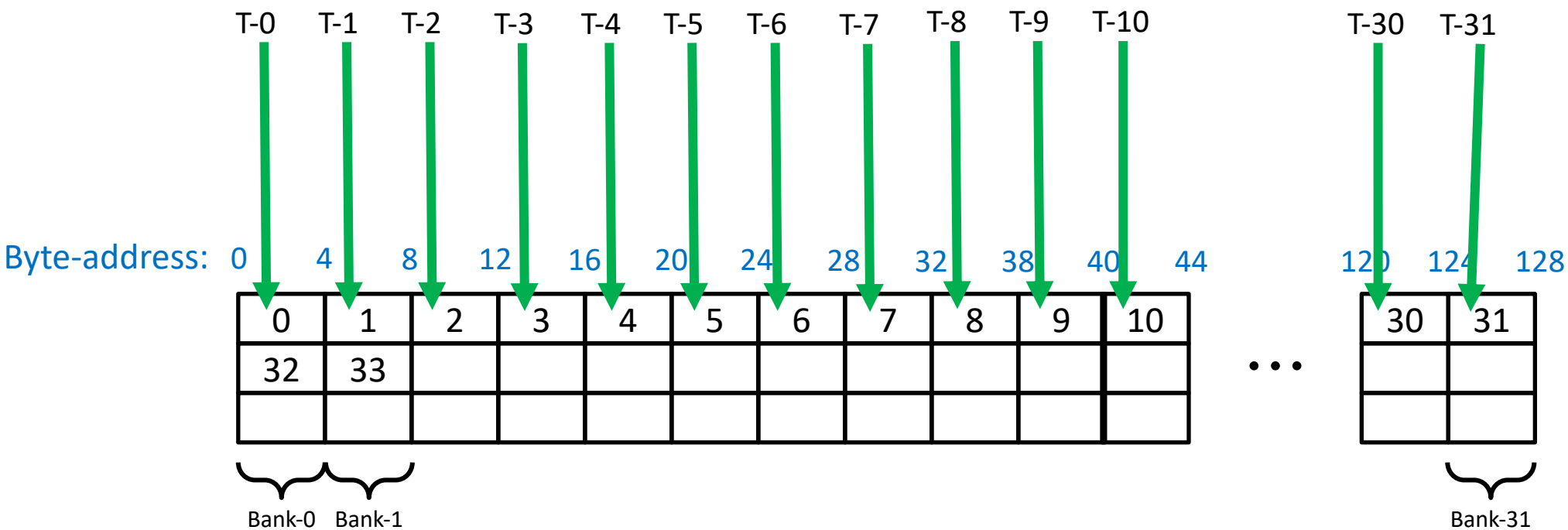
N-way bank conflict: ***N*** threads in a warp conflict

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

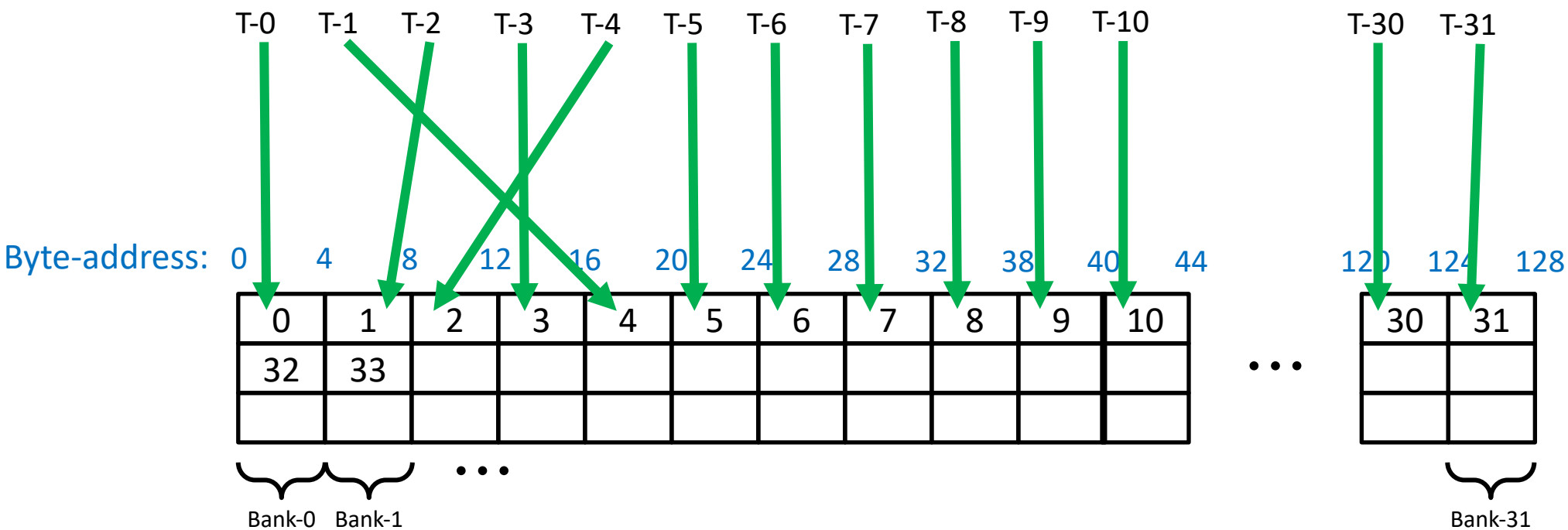
There is **no bank conflict** if:

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

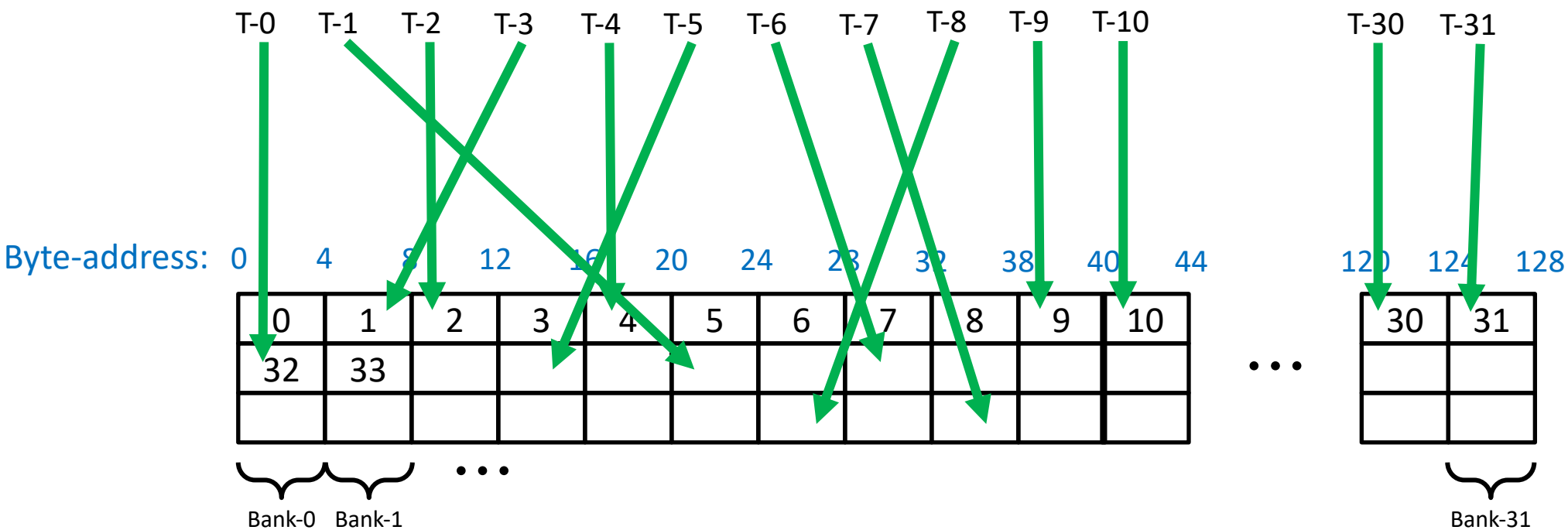
No Bank Conflicts



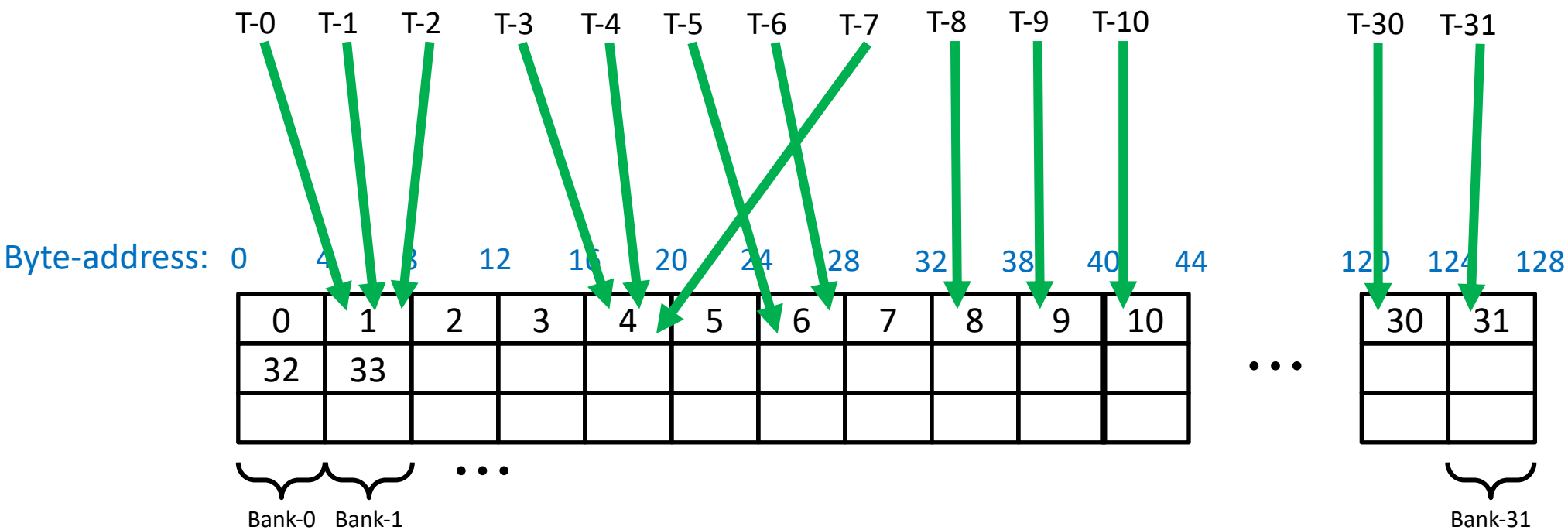
No Bank Conflicts



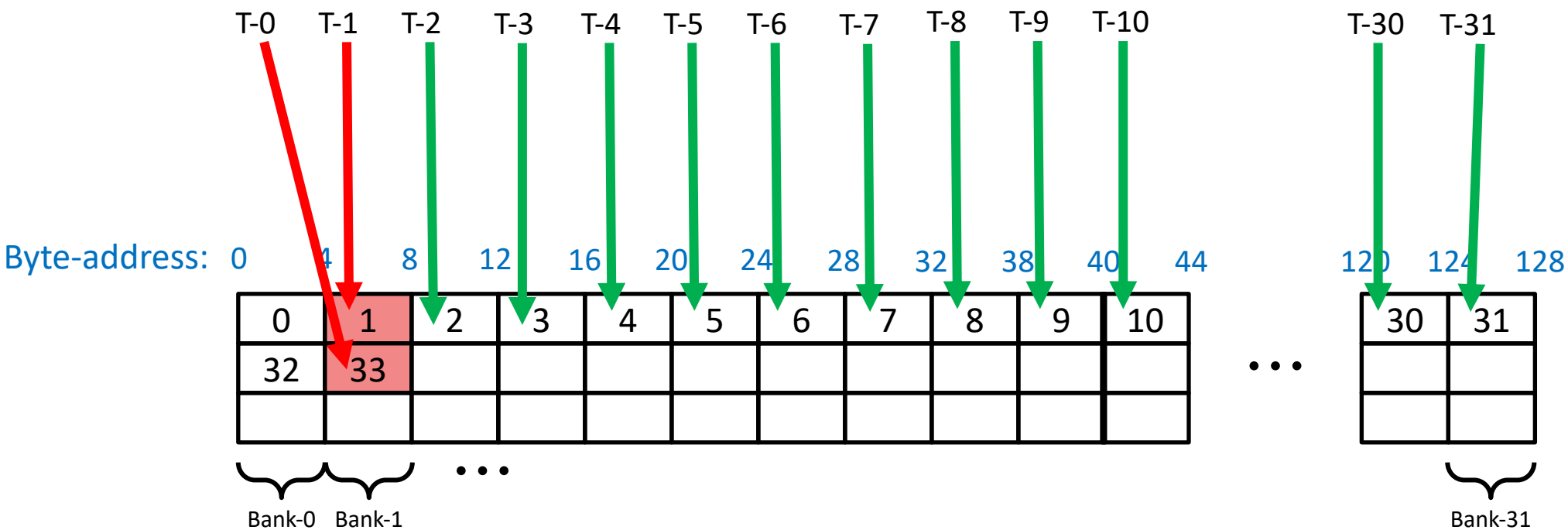
No Bank Conflicts



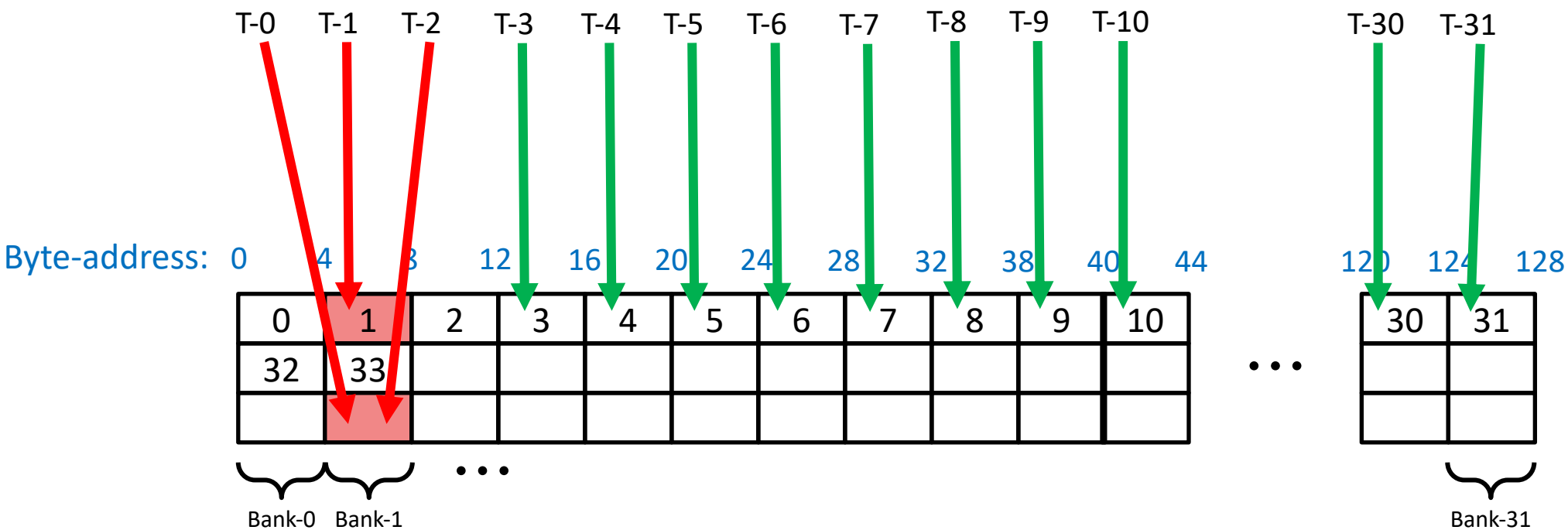
No Bank Conflicts



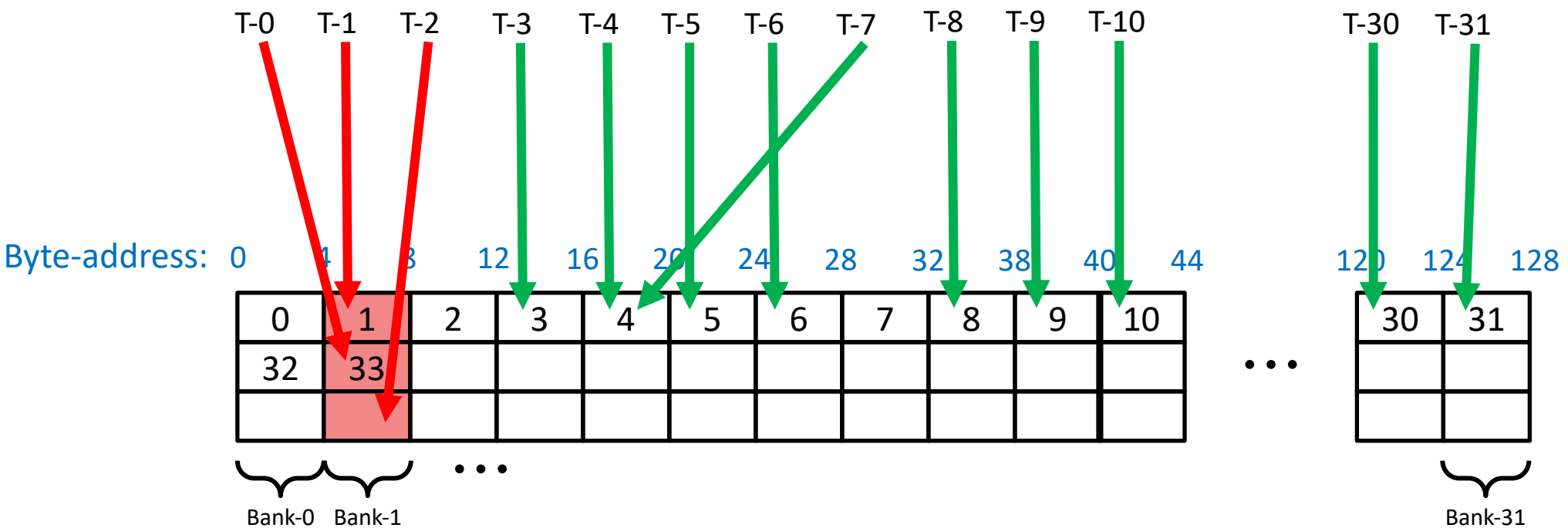
2-way Bank Conflict



2-way Bank Conflict



3-way Bank Conflict



Bank Conflict Resolution

4B or smaller words:

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

8B words are accessed in 2 phases:

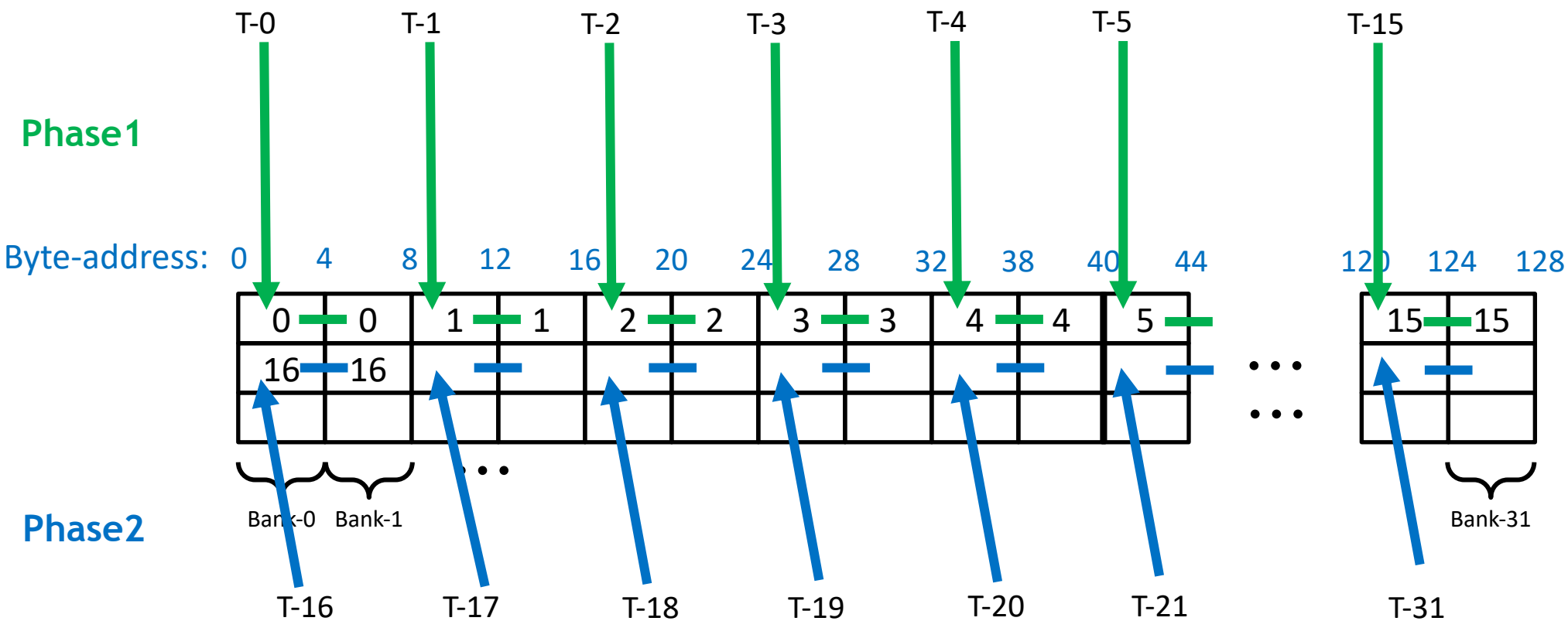
- Process addresses of the **first 16** threads in a warp
- Process addresses of the **second 16** threads in a warp

16B words are accessed in 4 phases:

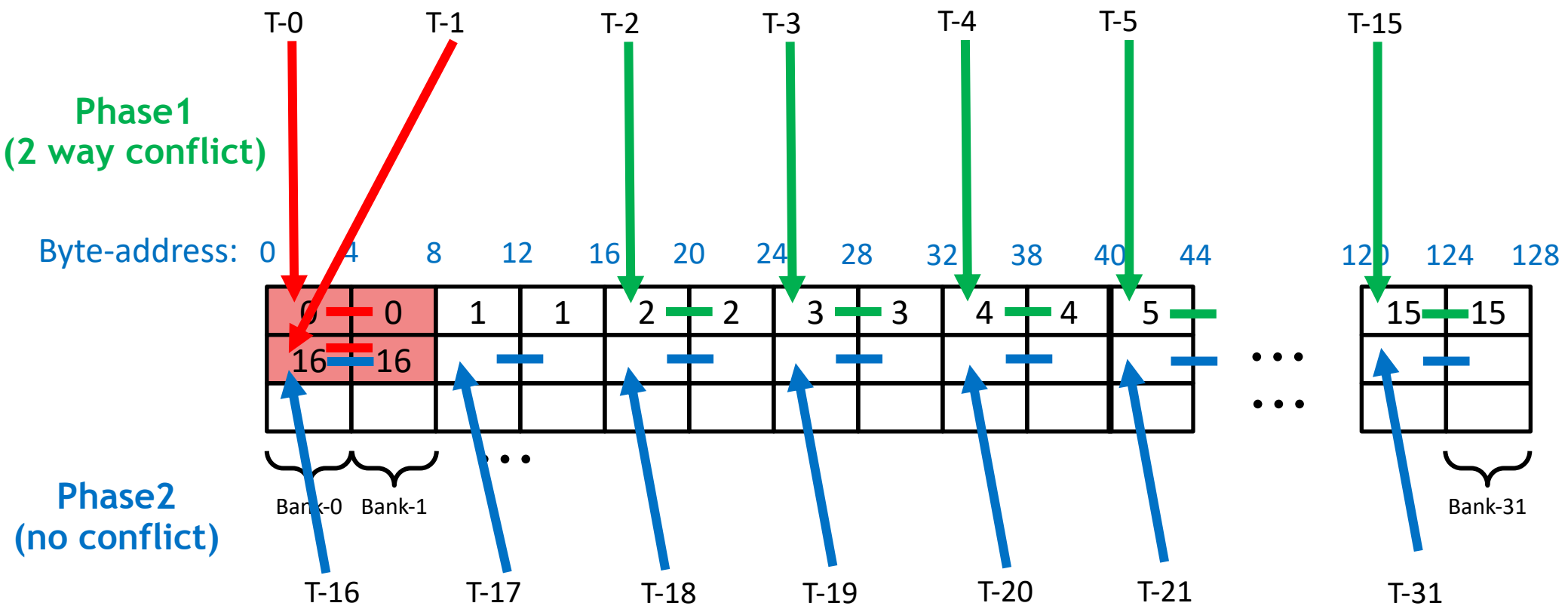
- Each phase processes a quarter of a warp

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 threadblock, 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

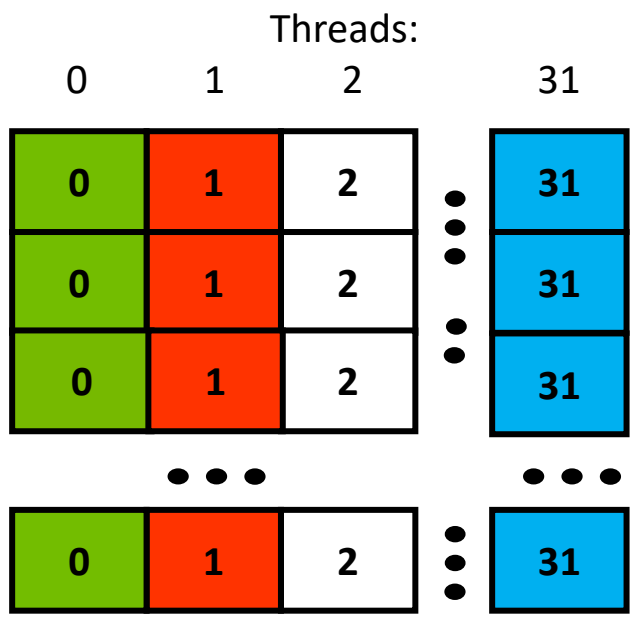
32x32 SMEM array (.e.g. __shared__ float sm[32][32])

Warp accesses a row : No conflict

Warp accesses a column : 32-way conflict

Number indentifies which warp is accessing data
Color indicates in which bank data resides

Bank 0
Bank 1
…
Bank 31



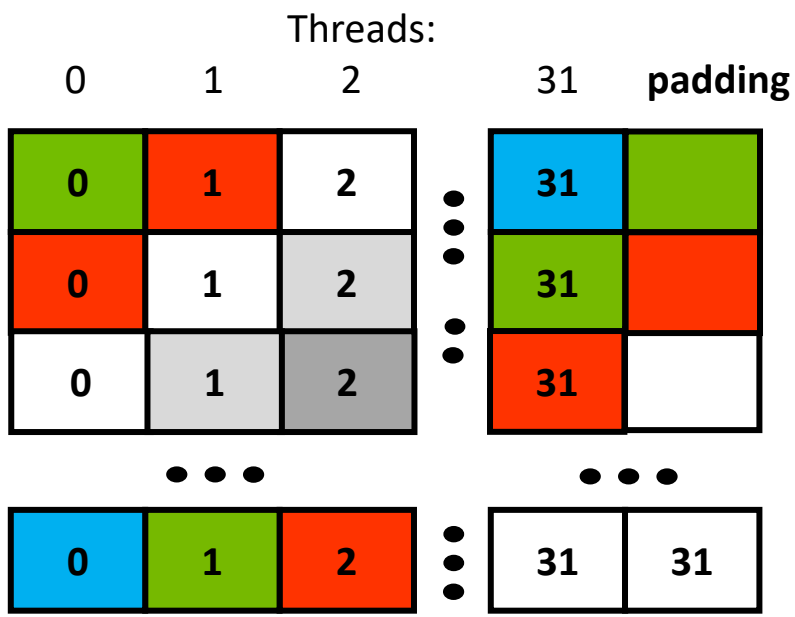
Case Study: Matrix Transpose

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

Warp accesses a row or a column: no conflict

Number indentifies 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 tremendous resource

- Very high bandwidth (14 TB/s), much lower latency than Global Memory
- Data is programmer-managed, no evictions by hardware
- Volta: up to 96KB 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

Constant Memory

- Globally-scoped arrays qualified with **__constant__**
- Total constant data size limited to **64 KB**
- Throughput = 4B per clock per SM (ideal if entire warp reads the same address)
- Can be used directly in arithmetic instructions (saving registers)
- Example use : Stencil coefficients

Running Faster

Solving the bottlenecks

A piece of code can be:

- **Compute bound** (saturating compute units)
  Solution: Reduce the number of instructions executed
  Using vector types, intrinsics, tensor cores, FMAs
- **Bandwidth bound** (saturating memory bandwidth)
  Solution: Reduce the amount of data transferred
  Optimal access patterns, using lower precision
- **Latency bound**
  Solution: Increase the number of instructions / mem accesses in flight

Running Faster

- Start with something simple. Simplicity and massive parallelism are key
- Profile first, optimize then. Do not over optimize (opportunity cost)
- Use libraries as building blocks whenever possible – e.g. CUB for parallel primitives (reduce/scan/sort/partition/..) at warp/block/device level
- Other libraries : CUBLAS, CUFFT, CUTLASS, CUSOLVER…

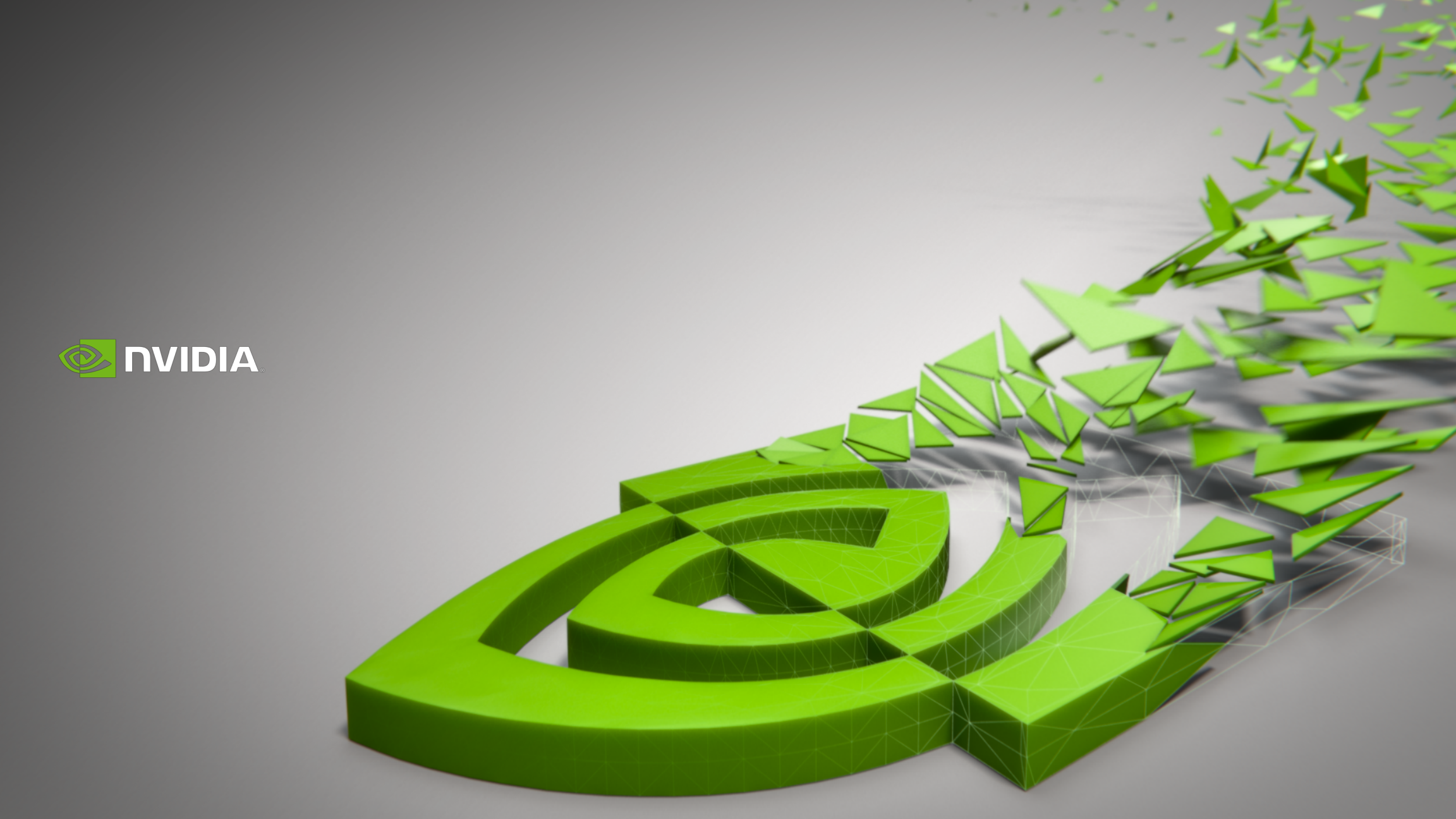
Resources

- CUDA zone : https://developer.nvidia.com/cuda-zone

- K80 details (compute capability 3.7) :

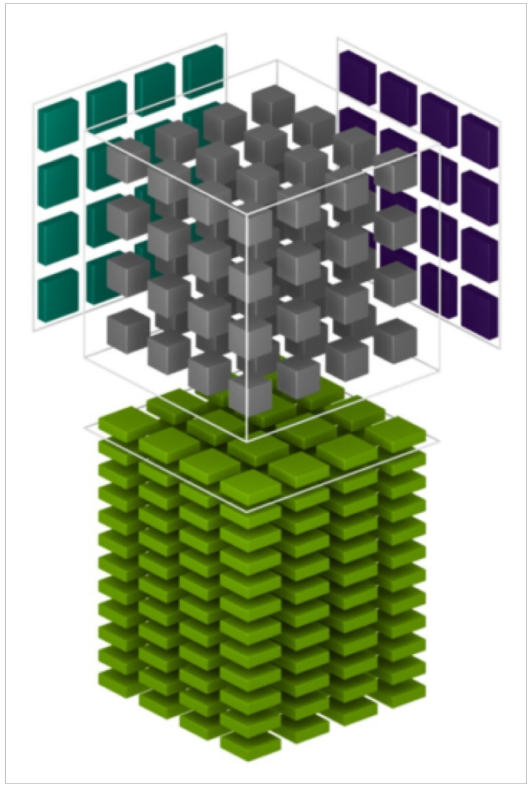
https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications

- CUB : http://nvlabs.github.io/cub/


NVIDIA

Tensor Cores

Volta/Turing



125 Tflops Peak

Matrix Multiplication Pipeline,
half precision inputs

Used in CUBLAS, CUDNN, CUTLASS

Optimized libraries can reach ~90% of peak

Exposed in CUDA

Tensor Cores

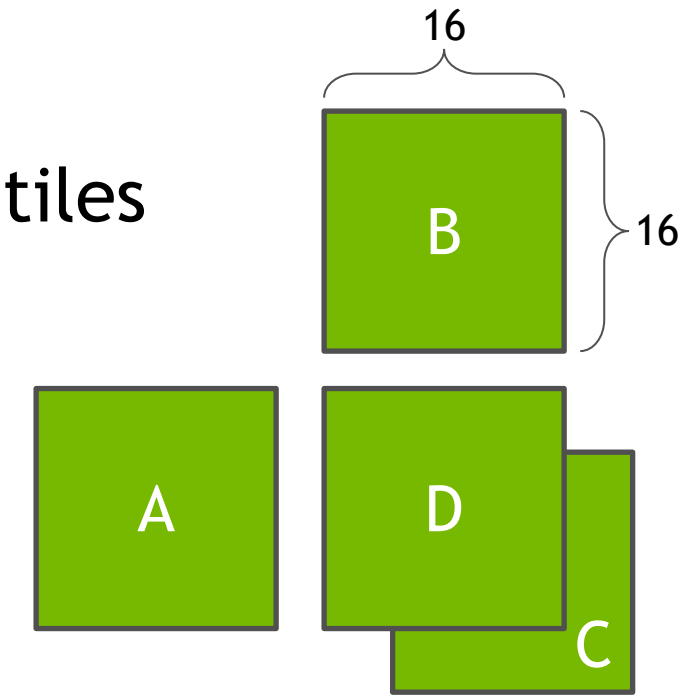
Using Tensor Cores in your CUDA code

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 x B + C

A and B : **FP16 only**
C and D : Same, either FP16 or FP32.

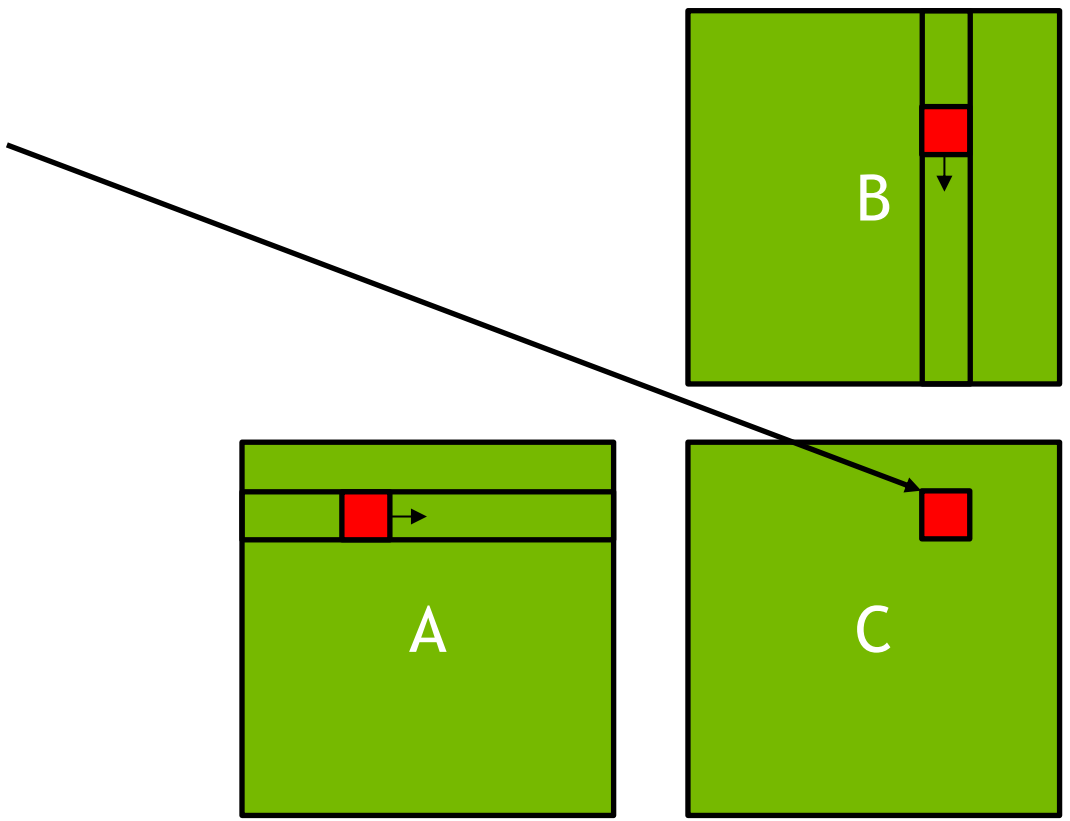


Tensor Cores

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



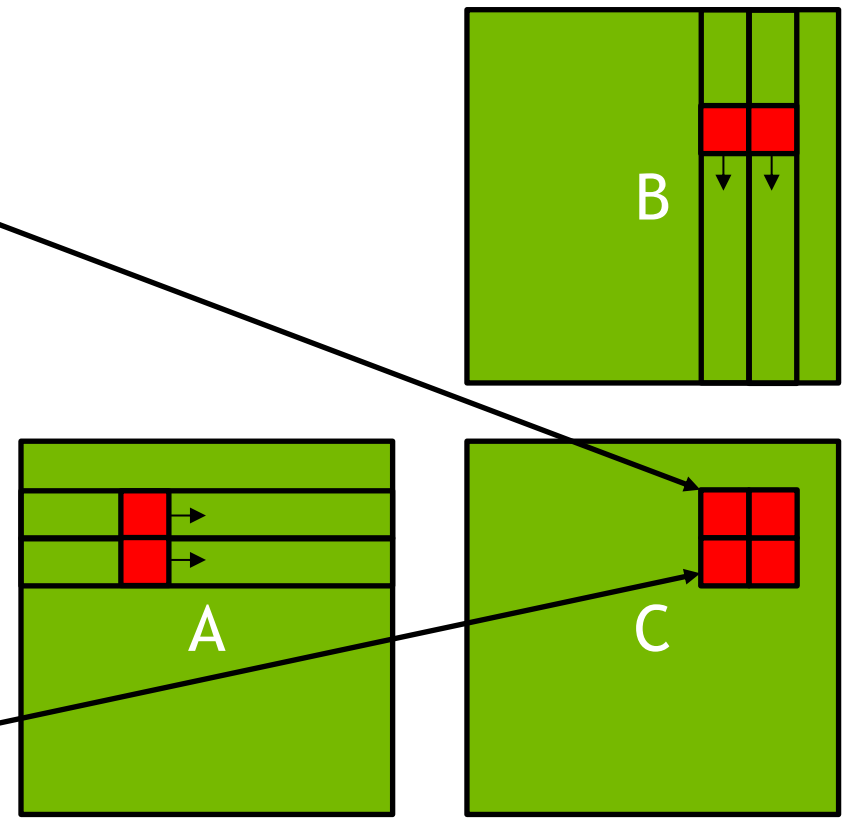
Tensor Cores

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 threadblock,
with inputs staged in shared memory



Volta's L1 Cache

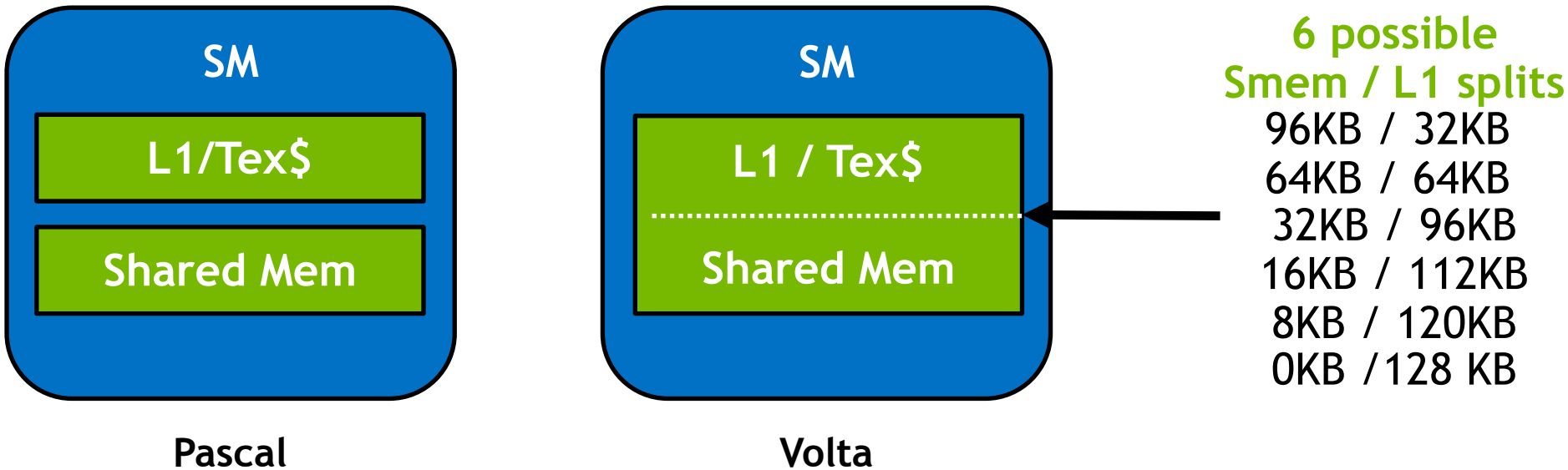
Pascal vs Volta

Pascal : **24KB**
Achievable BW = **2.6 TB/s**

Volta : Variable size **32 KB ~ 128KB**
Achievable BW = **14.4 TB/s**
Lower latency!

L1 caching: Global Mem, Texture, Local Mem (inc. register spills)

Volta's Unified L1



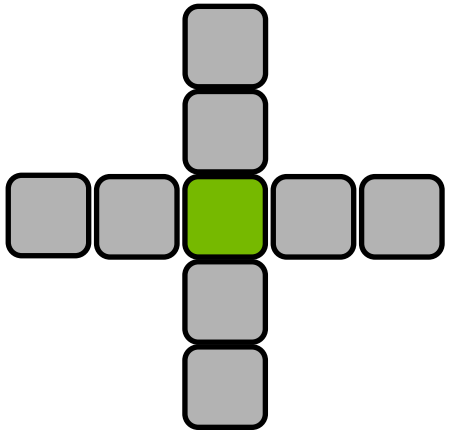
How to specify the L1 / Smem split on Volta:

cudaFuncSetAttribute (MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout, carveout);

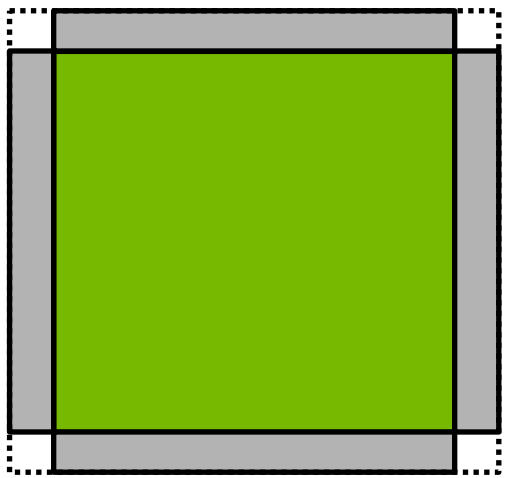
The driver usually does a pretty good job at choosing the right split.

2D Stencil Experiment

with and without Shared Memory



```
index = iy * nx + ix;
res = coef[0] * in[index];
for(i=1; i<=RADIUS; i++)
  res += coef[i] * (in[index-i] +
                    in[index+i] +
                    in[index-i*n1] +
                    in[index+i*n1]);
out[index] = res;
```

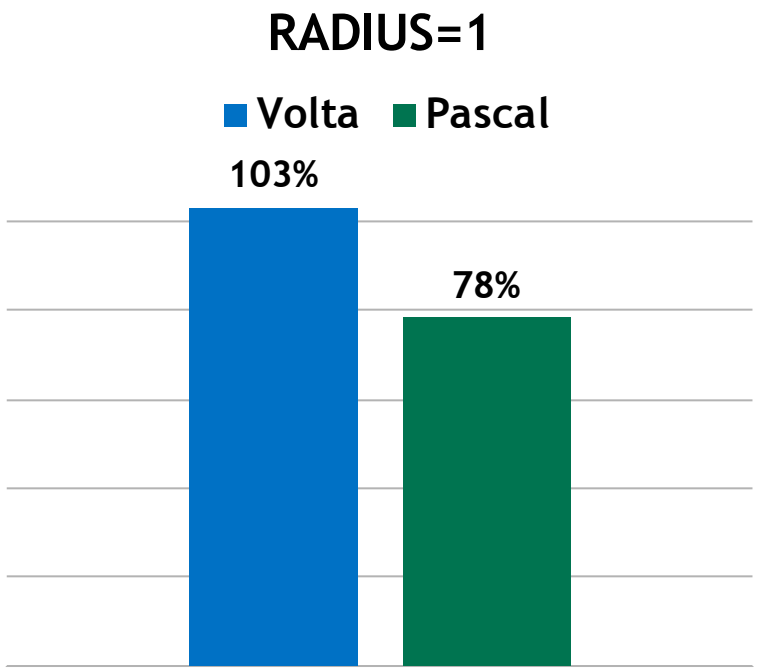


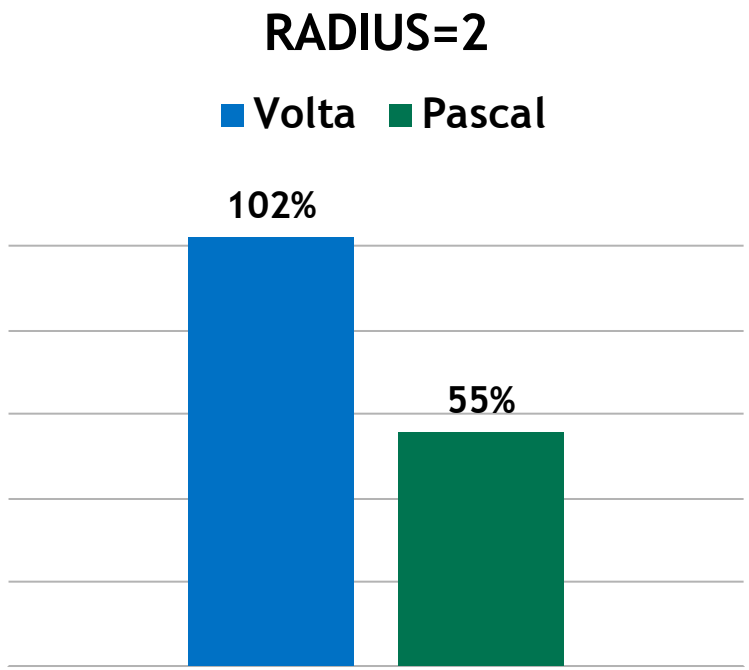
With Shared Memory:

- Load the input array and halos in shared memory
- __syncthreads()
- Compute the stencil from the shared memory

2D – Small stencils

Relative speed of L1 implementation versus Smem implementation

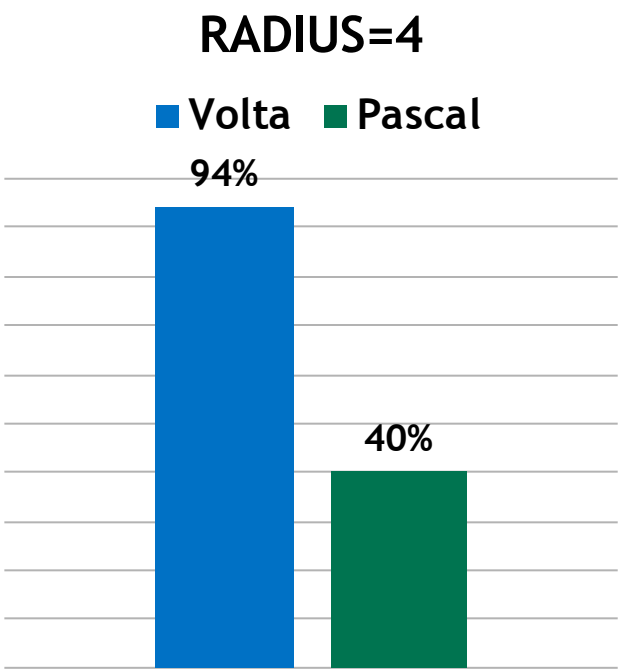


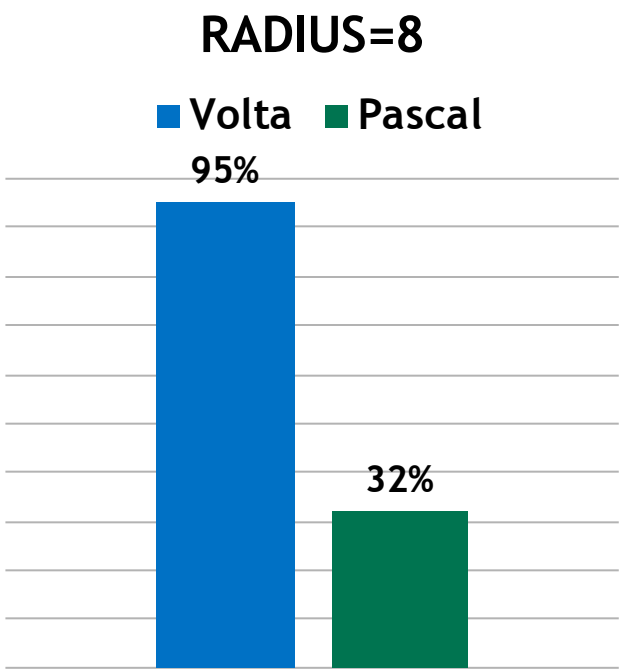


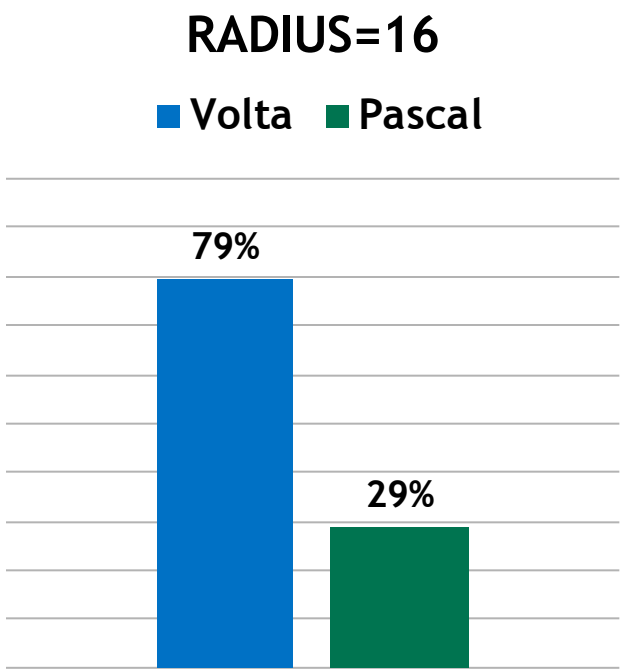
L1 implementation is faster than Shared Memory on Volta!

2D - Larger Stencils

Relative speed of L1 implementation versus Smem implementation







Shared Memory implementation is always faster for larger stencils