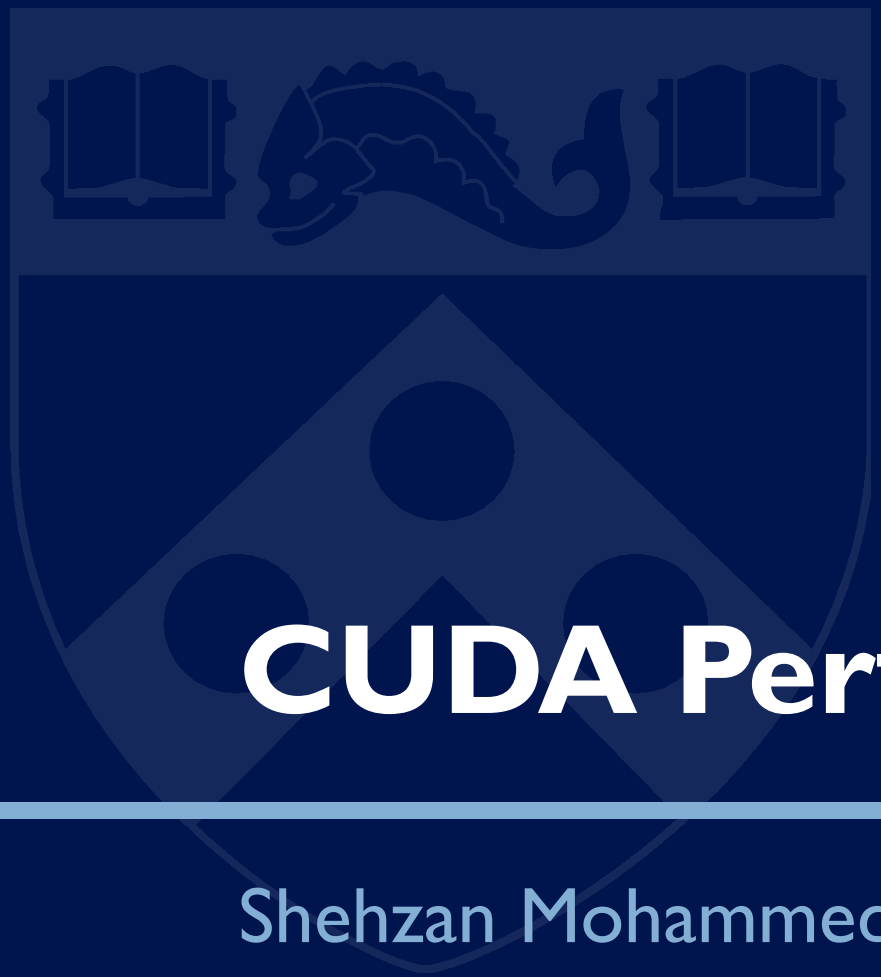
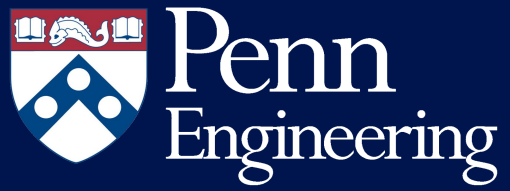


CUDA Performance

Shehzan Mohammed
CIS 5650 - Fall 2024



Acknowledgements

- Some slides from Varun Sampath

Agenda

- **Maximize Utilization**
  - Parallel Reduction Revisited
  - Warp Partitioning
- **Maximize Memory Throughput**
  - Memory Coalescing
  - Bank Conflicts
  - Dynamic Partitioning of SM Resources
- **Maximize Instruction Throughput**
  - Data Prefetching
  - Instruction Mix
  - Loop Unrolling
  - Thread Granularity

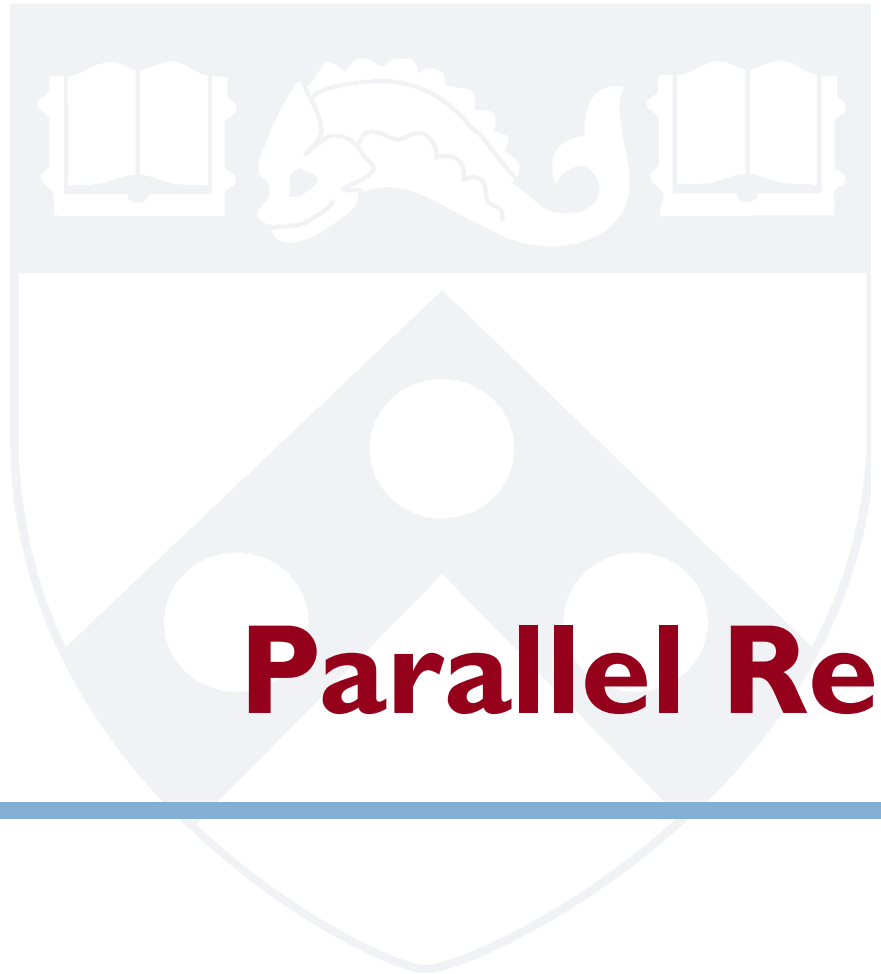
Efficient data-parallel algorithms

+

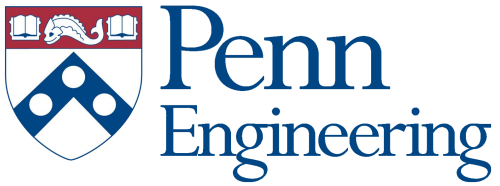
Optimizations based on GPU Architecture

=

Maximum Performance

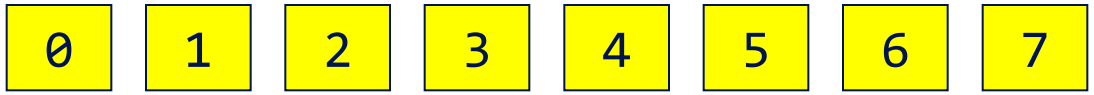


Parallel Reduction Revisited

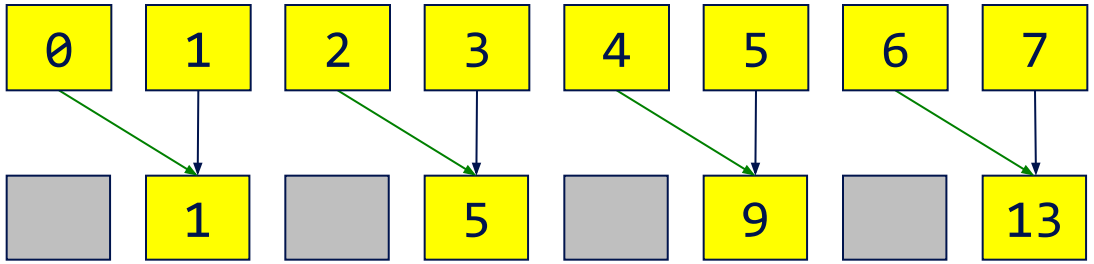


Parallel Reduction

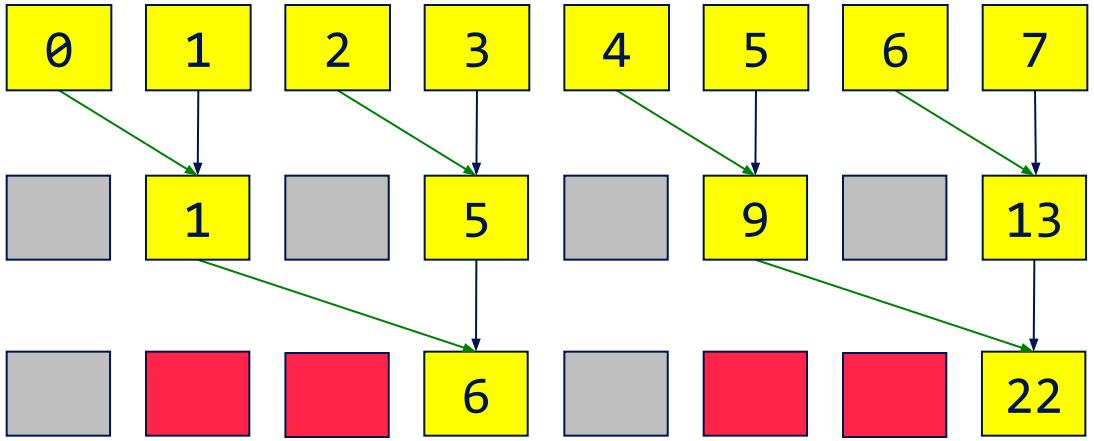
- Recall Parallel Reduction (sum)



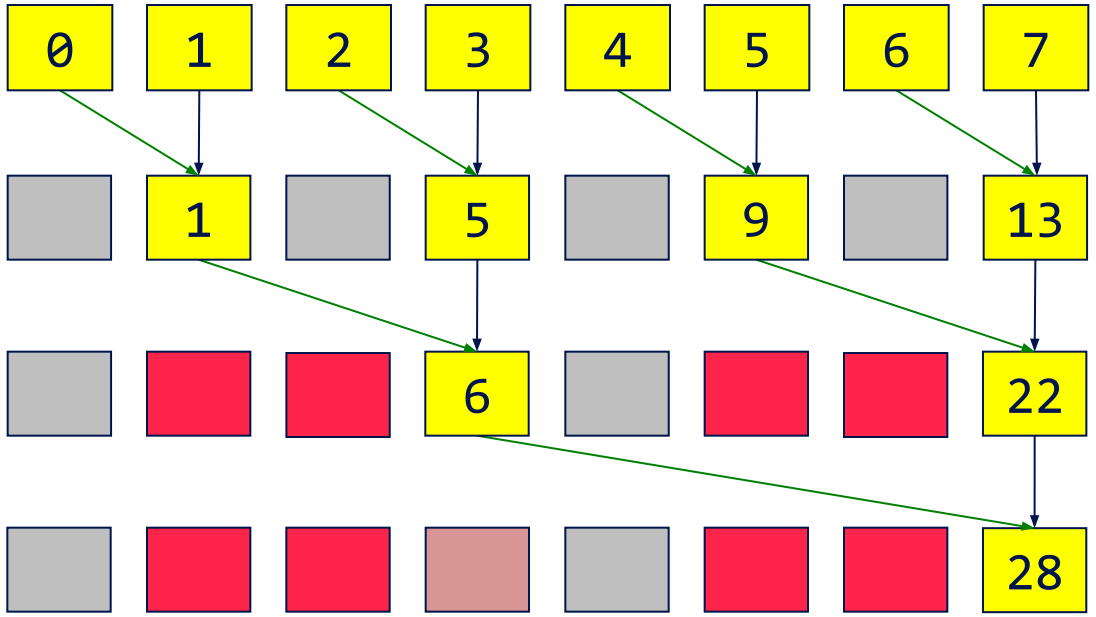
Parallel Reduction



Parallel Reduction

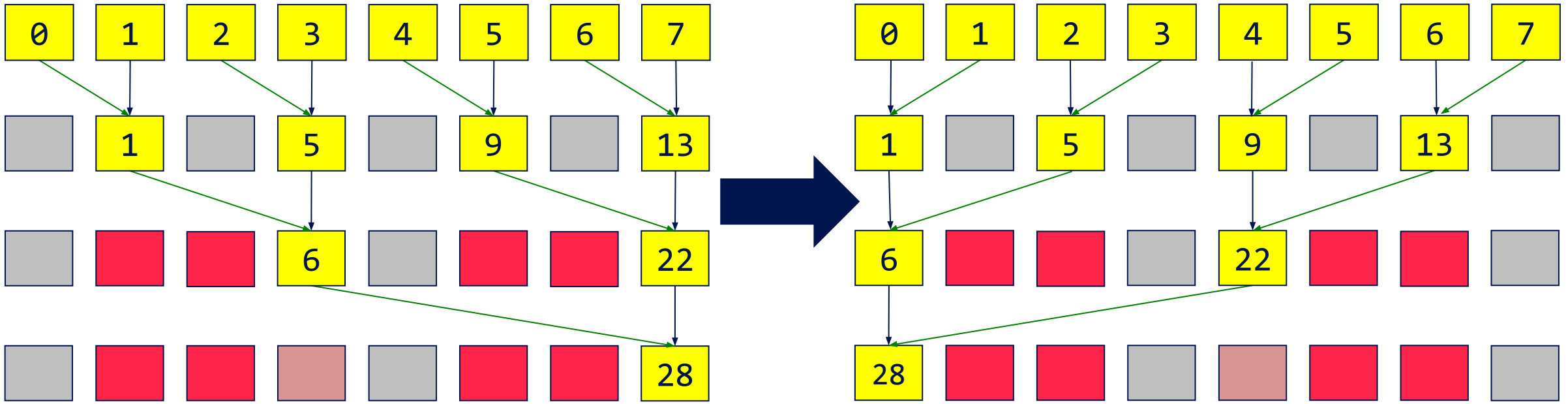


Parallel Reduction

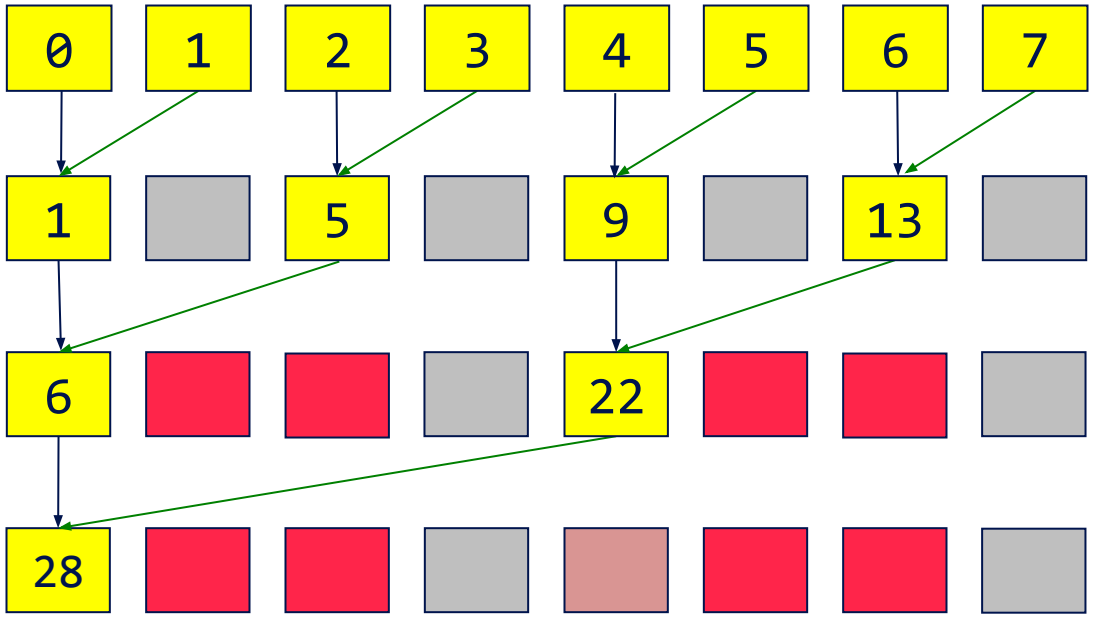


Parallel Reduction

- Mirror indexing



Parallel Reduction



Parallel Reduction – Kernel (partial code)

```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (int stride = 1; stride < blockDim.x; stride *= 2) {
  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] += partialSum[t + stride];
}
```


Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

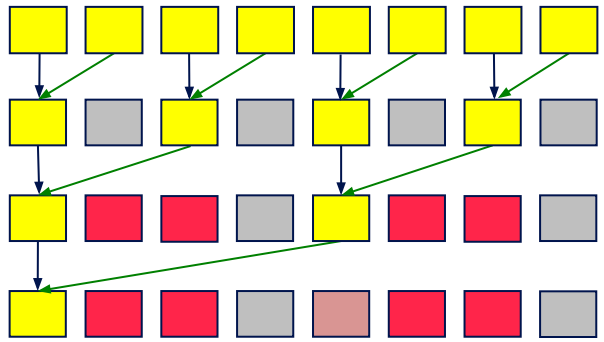
Parallel Reduction – Kernel (partial code)

```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (int stride = 1; stride < blockDim.x; stride *= 2) {
  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] += partialSum[t + stride];
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Parallel Reduction – Kernel (partial)

```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (int stride = 1; stride < blockDim.x; stride *= 2) {
  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] += partialSum[t + stride];
}
```



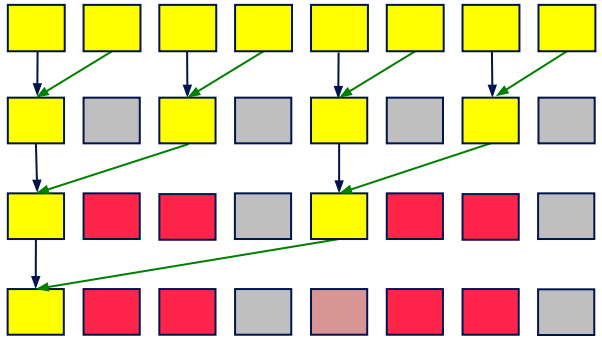
stride: 1, 2, 4…

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Parallel Reduction – Kernel (partial)

```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (int stride = 1; stride < blockDim.x; stride *= 2) {
  __syncthreads();
   if (t % (2 * stride) == 0)
      partialSum[t] += partialSum[t + stride];
}
```

Why?



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

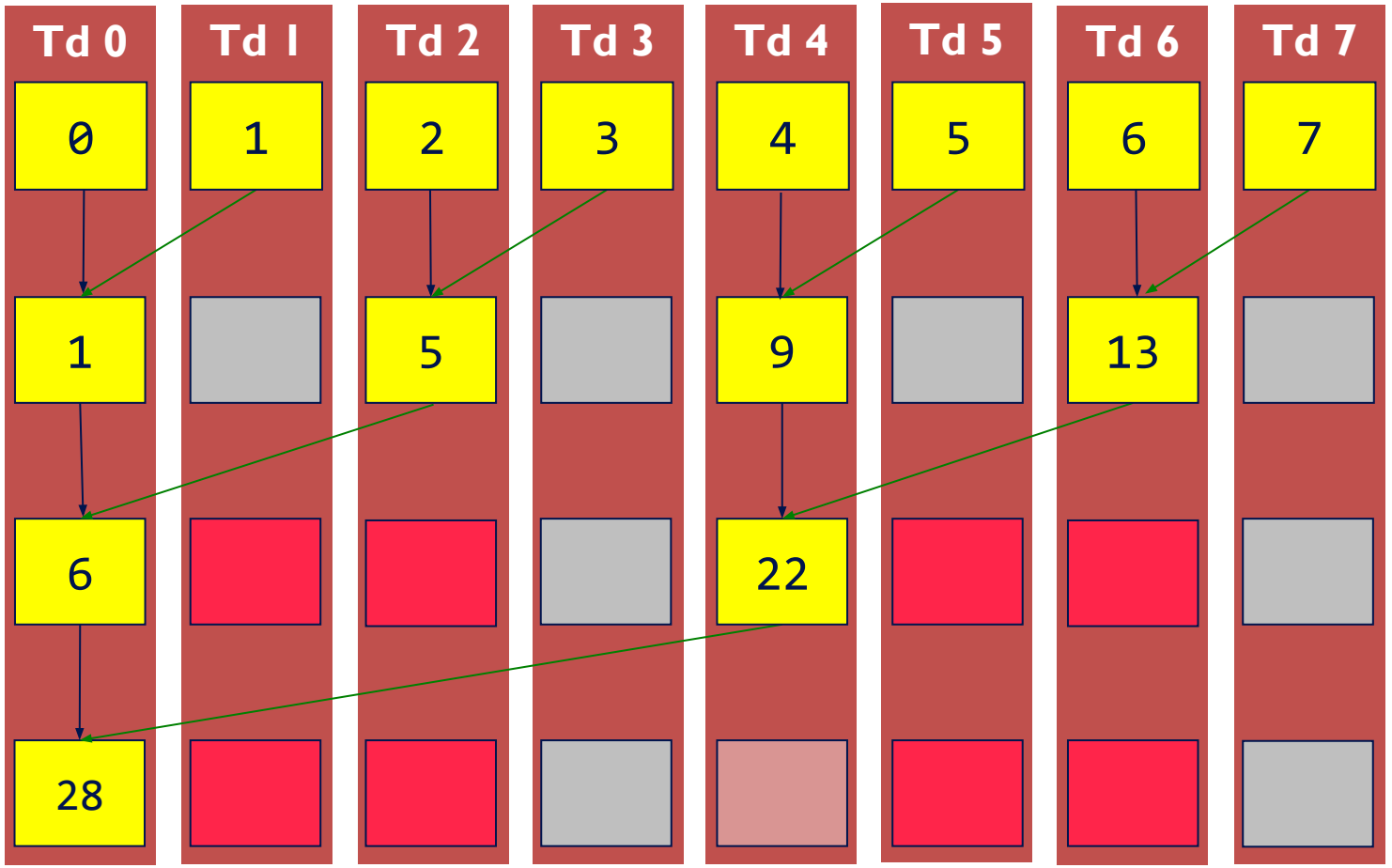
Parallel Reduction – Kernel (partial)

```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (int stride = 1; stride < blockDim.x; stride *= 2) {
  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] += partialSum[t + stride];
}
```

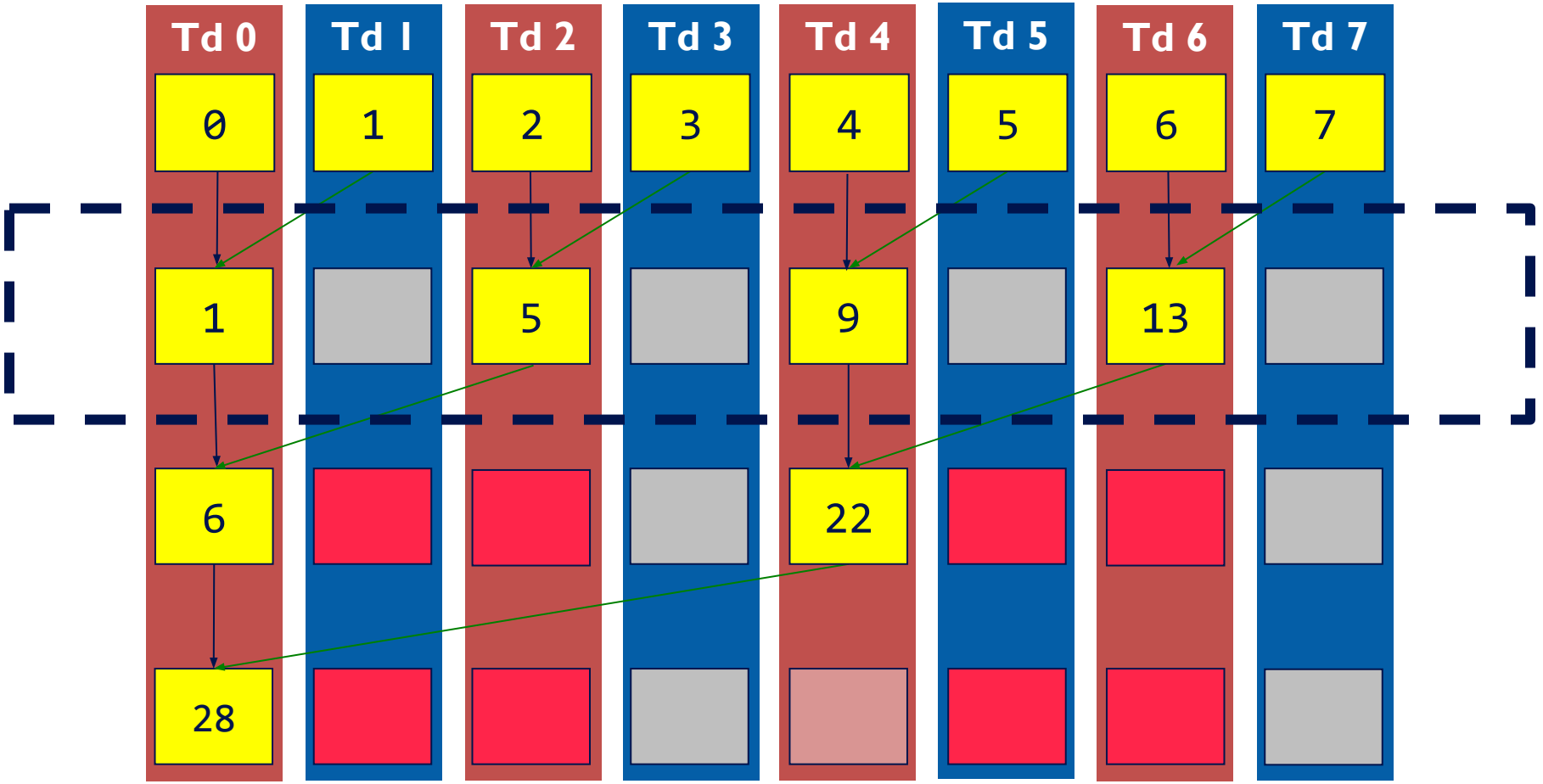
- Compute sum in same shared memory
- As stride increases, what do more threads do?

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Parallel Reduction

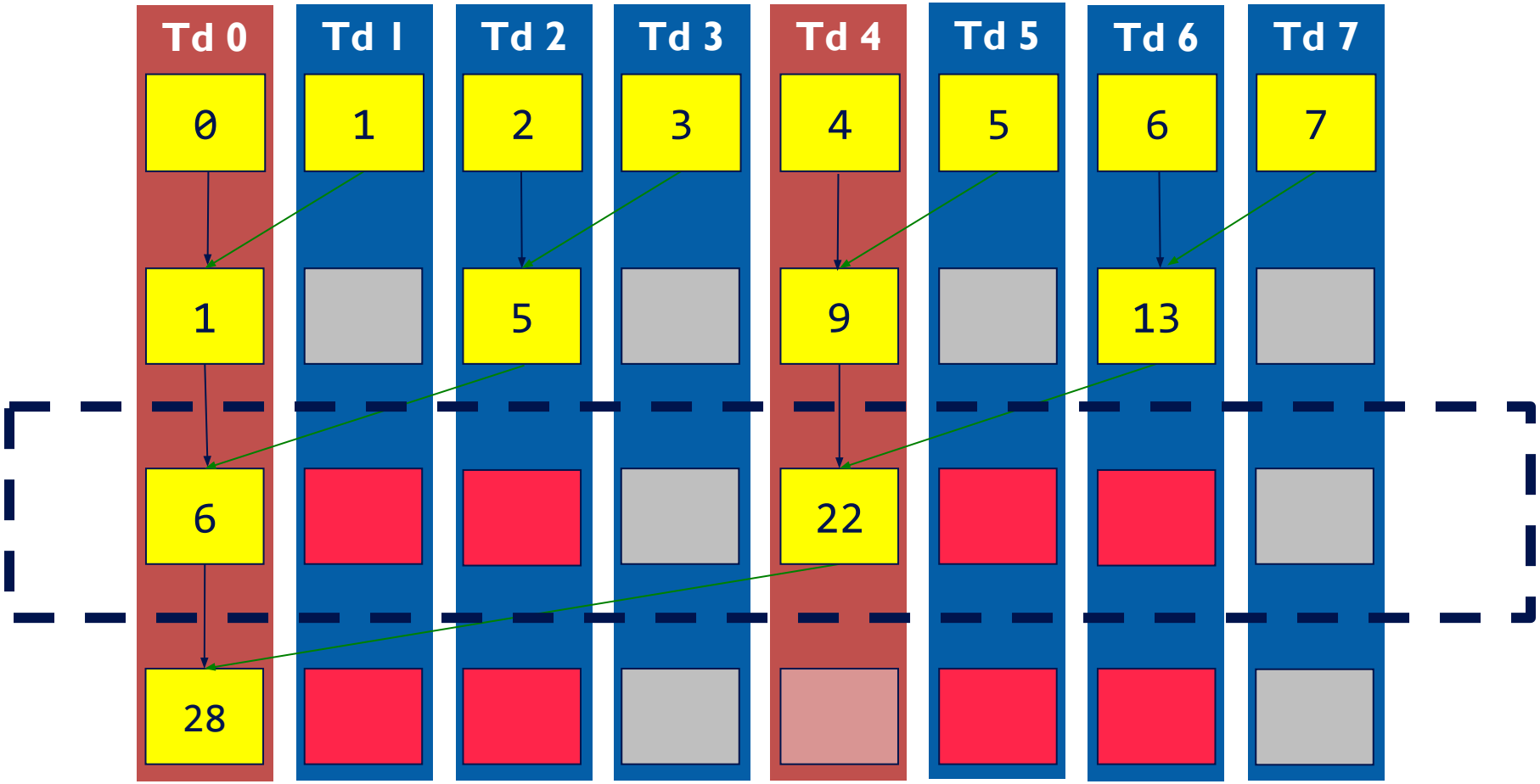


Parallel Reduction



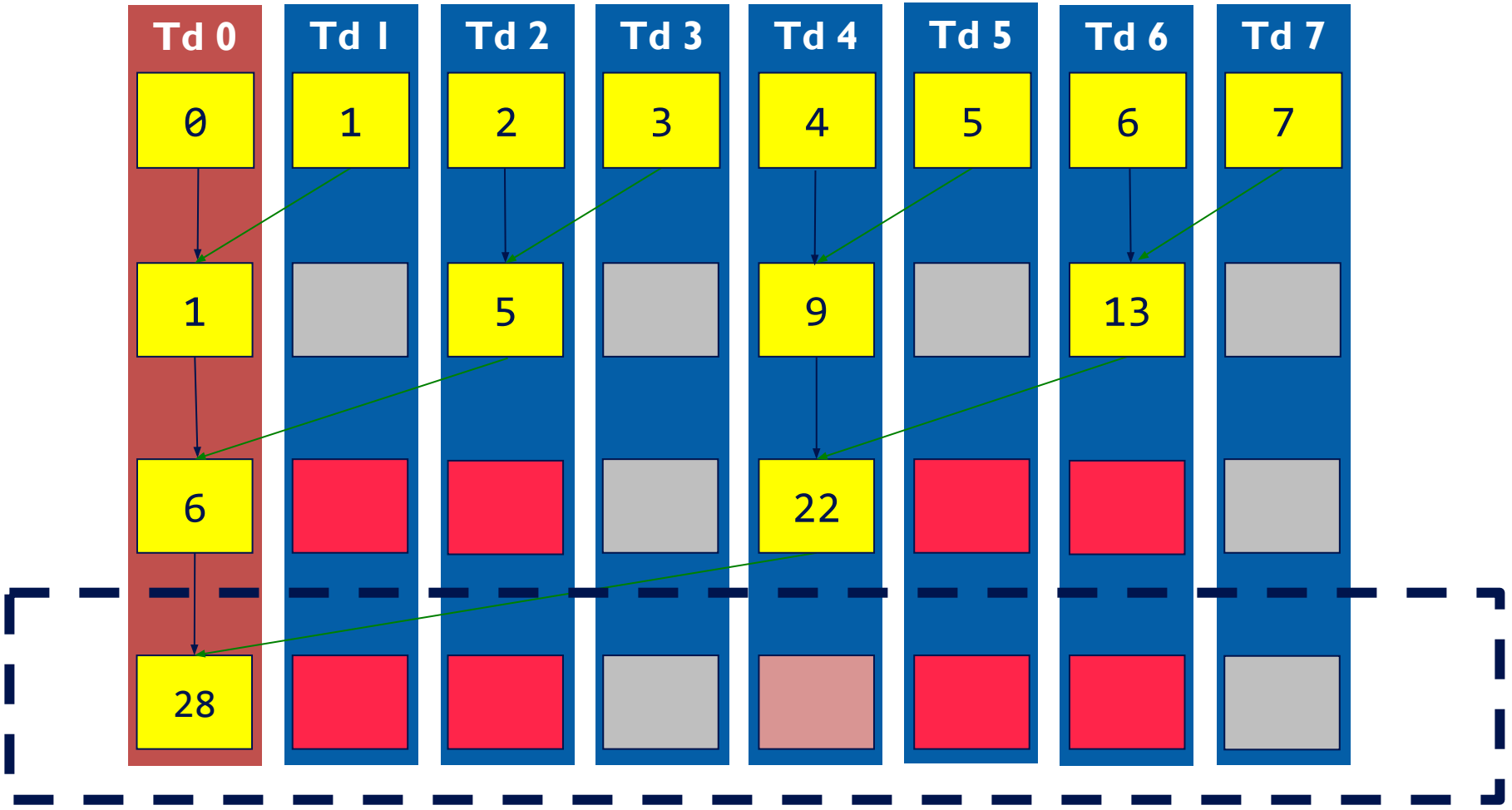
- **1st pass**: threads 1, 3, 5, and 7 don't do anything
- Really only need **n/2** threads for **n** elements

Parallel Reduction



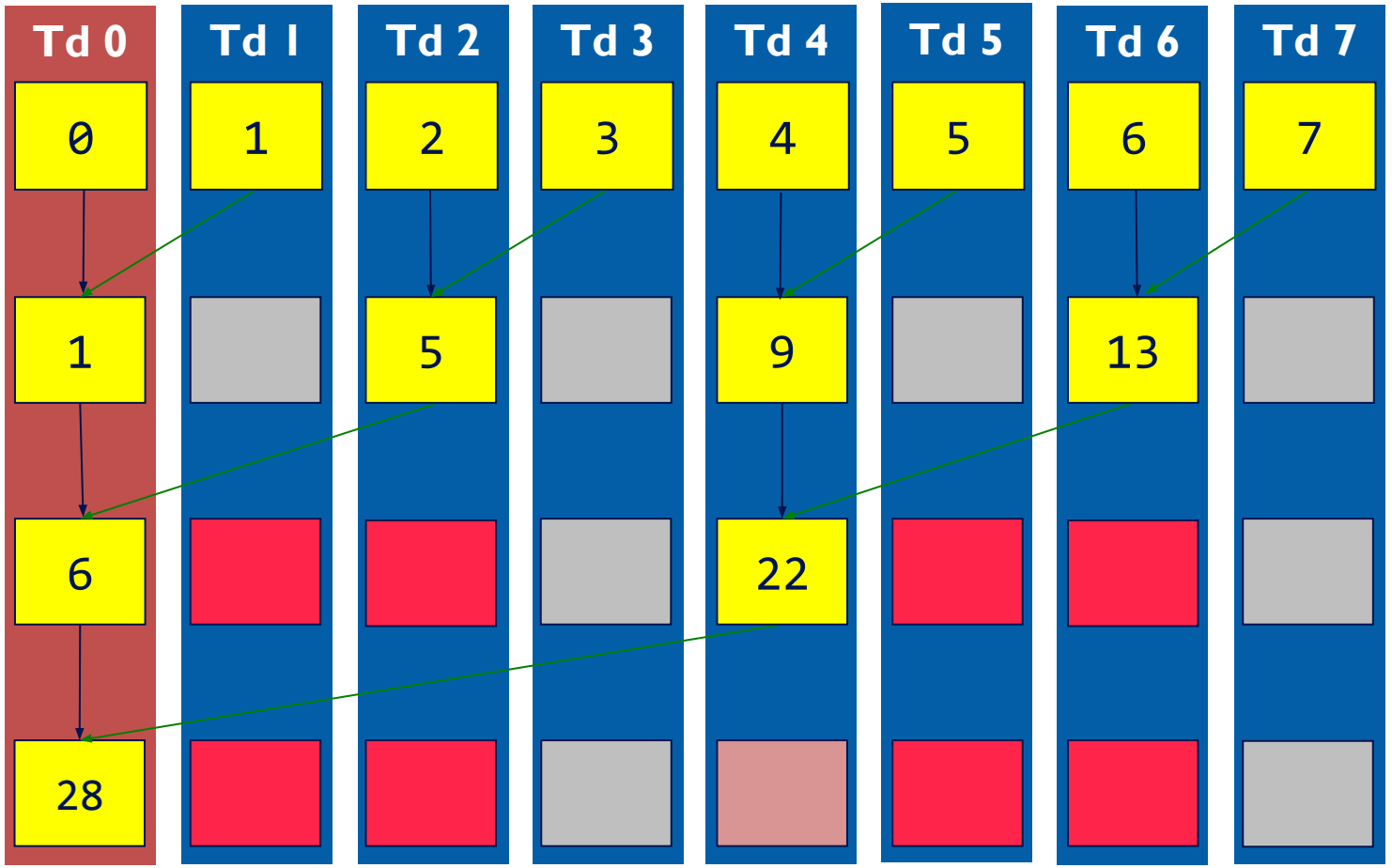
- **2nd pass**: threads 2, and 6 also don't do anything

Parallel Reduction



- **3rd pass**: threads 4 now also doesn't do anything

Parallel Reduction

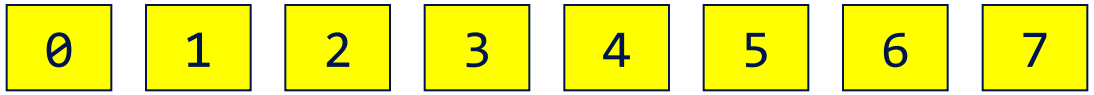


- **In general, number of threads required is halved at each pass**

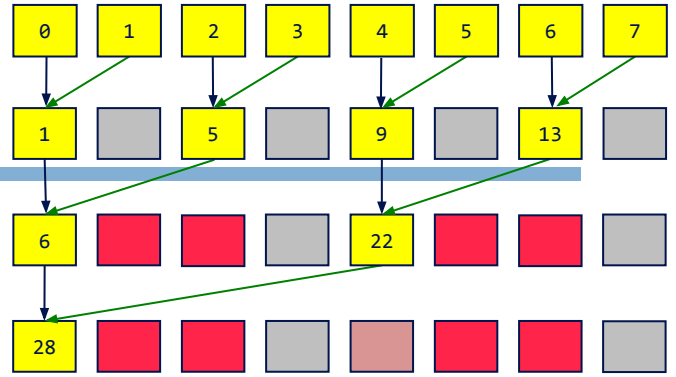
Parallel Reduction

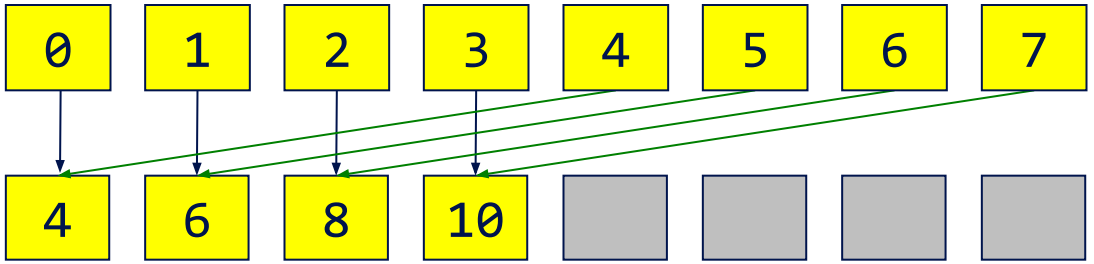
- What if we ***tweaked*** the implementation?

Parallel Reduction

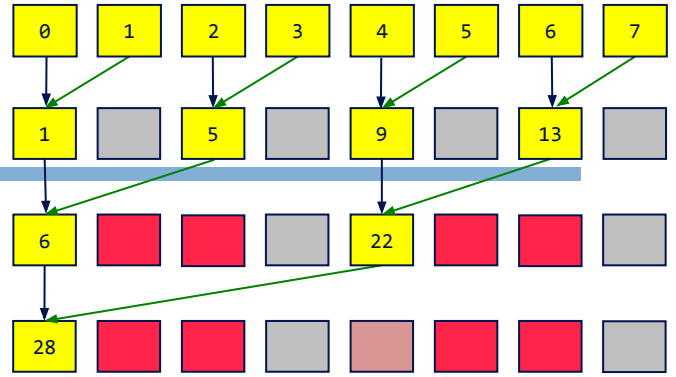


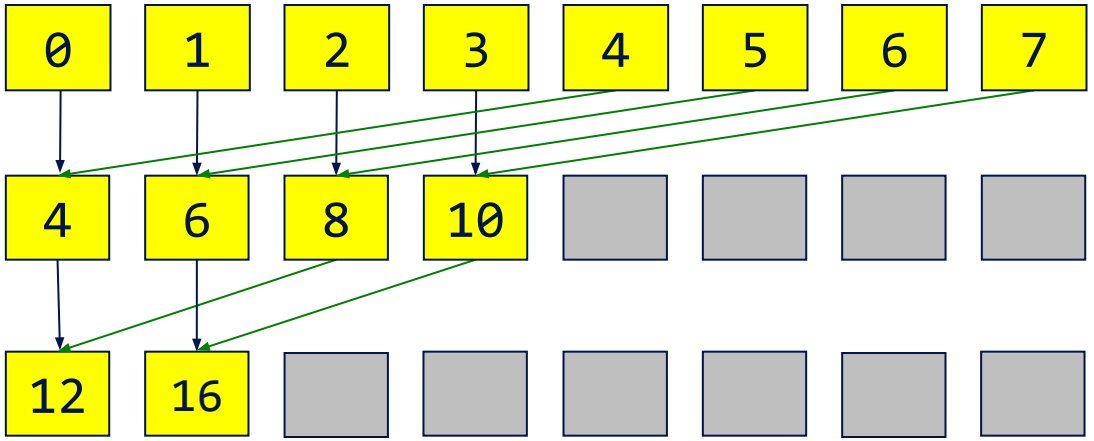
Parallel Reduction



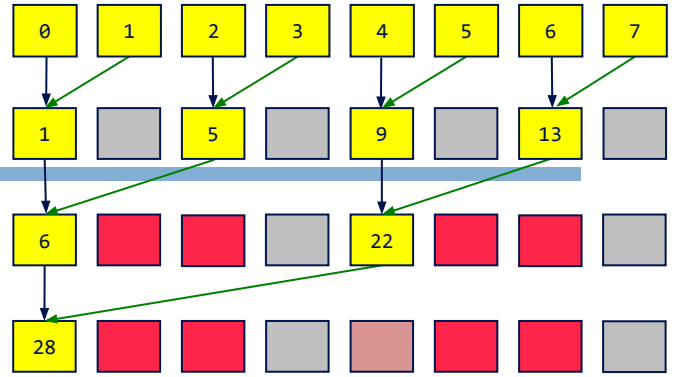


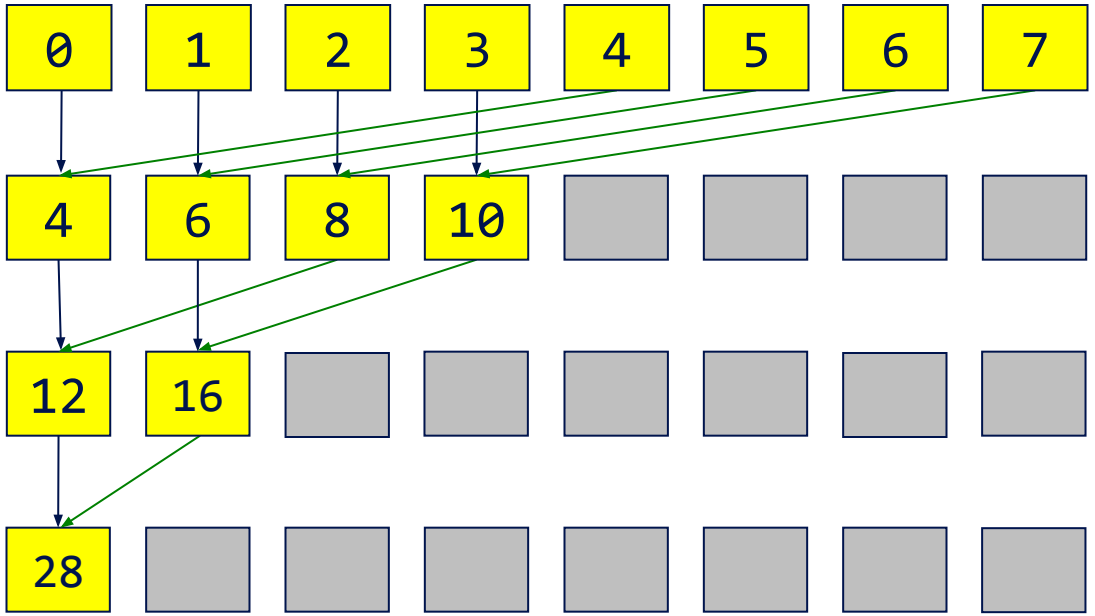
Parallel Reduction





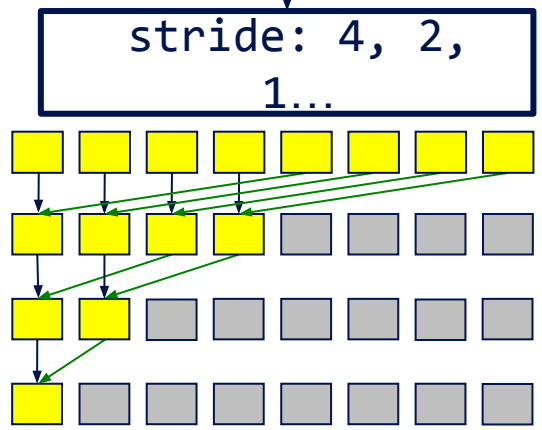
Parallel Reduction





Parallel Reduction – Kernel (partial)

```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t + stride];
}
```



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

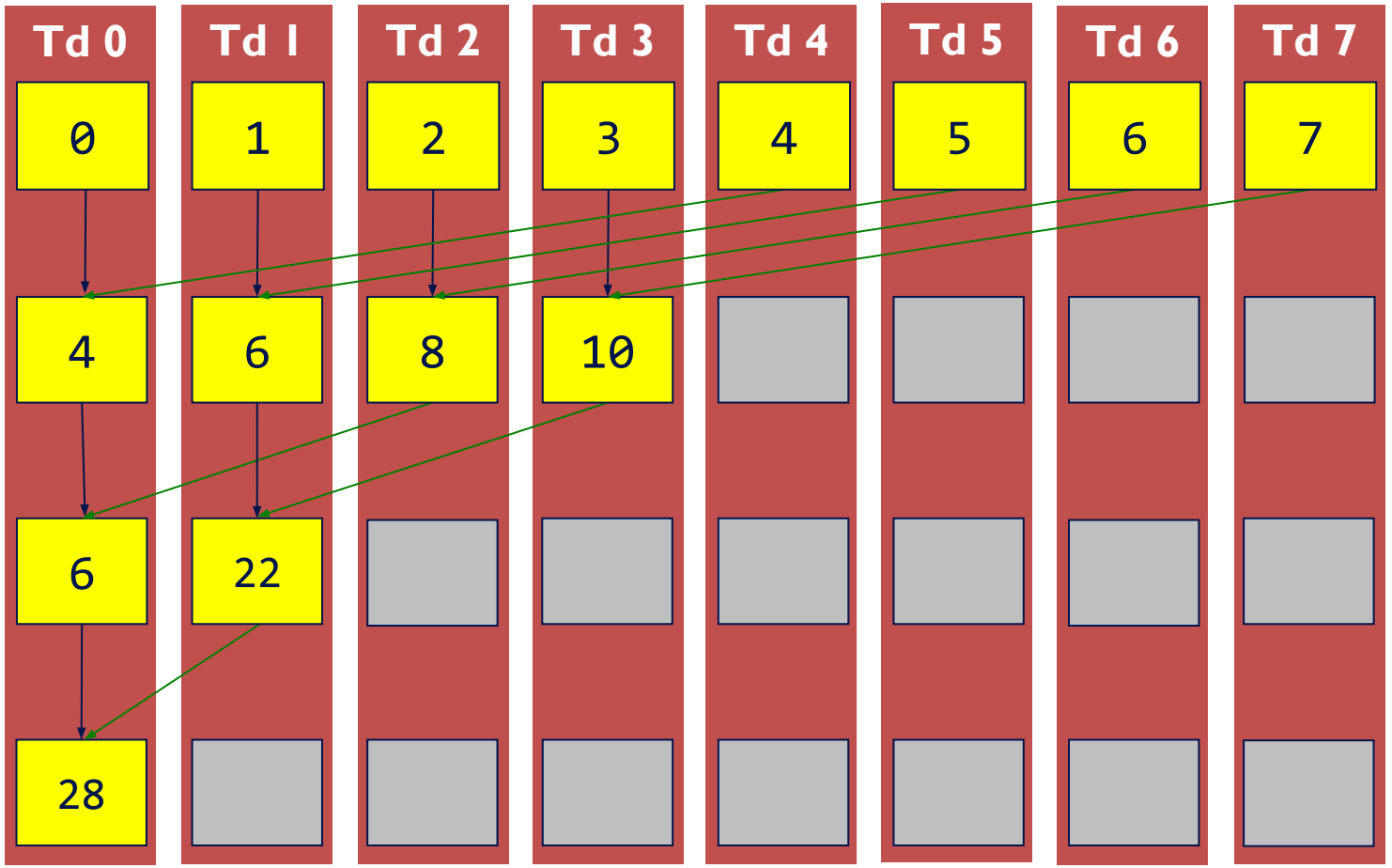
Parallel Reduction – Kernel (partial)

```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t + stride];
}
```

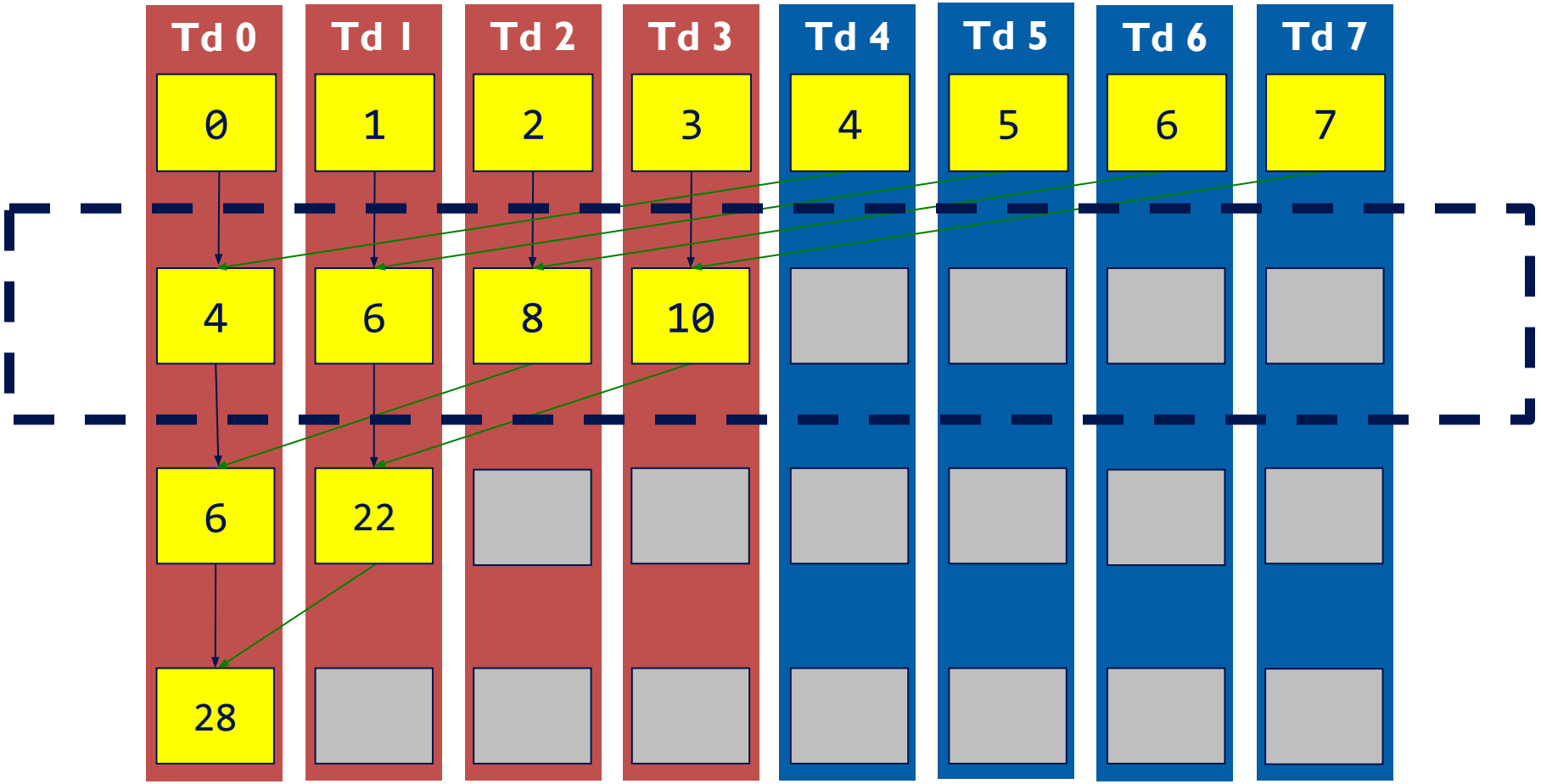
- **Condition changes**
- Operation remains same

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Parallel Reduction

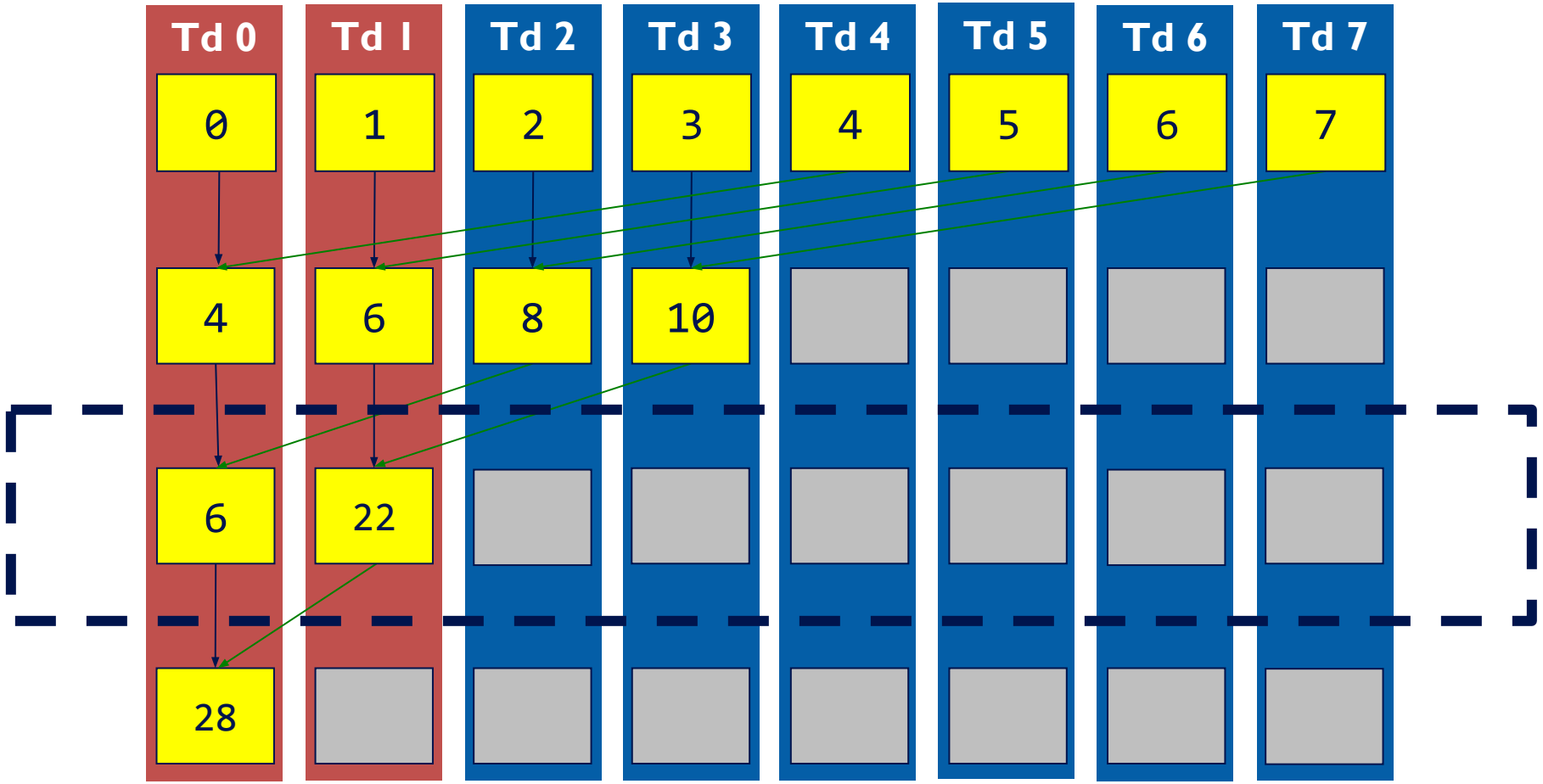


Parallel Reduction



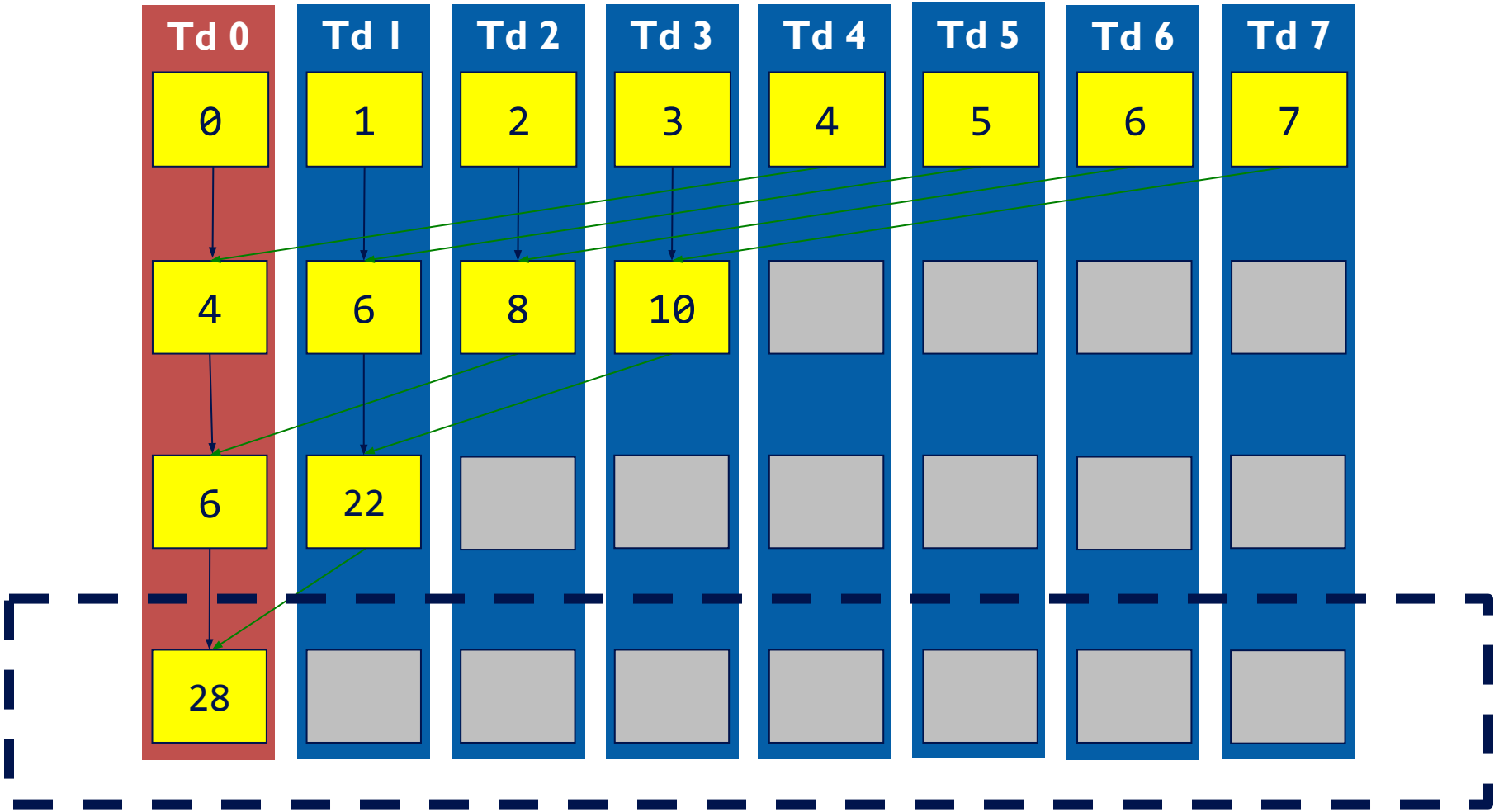
- **1st pass**: threads 4, 5, 6, and 7 don't do anything
- Still only need **n/2** threads for **n** elements, but can now launch half the threads/blocks

Parallel Reduction



- **2nd pass**: threads 2 and 3 don't do anything

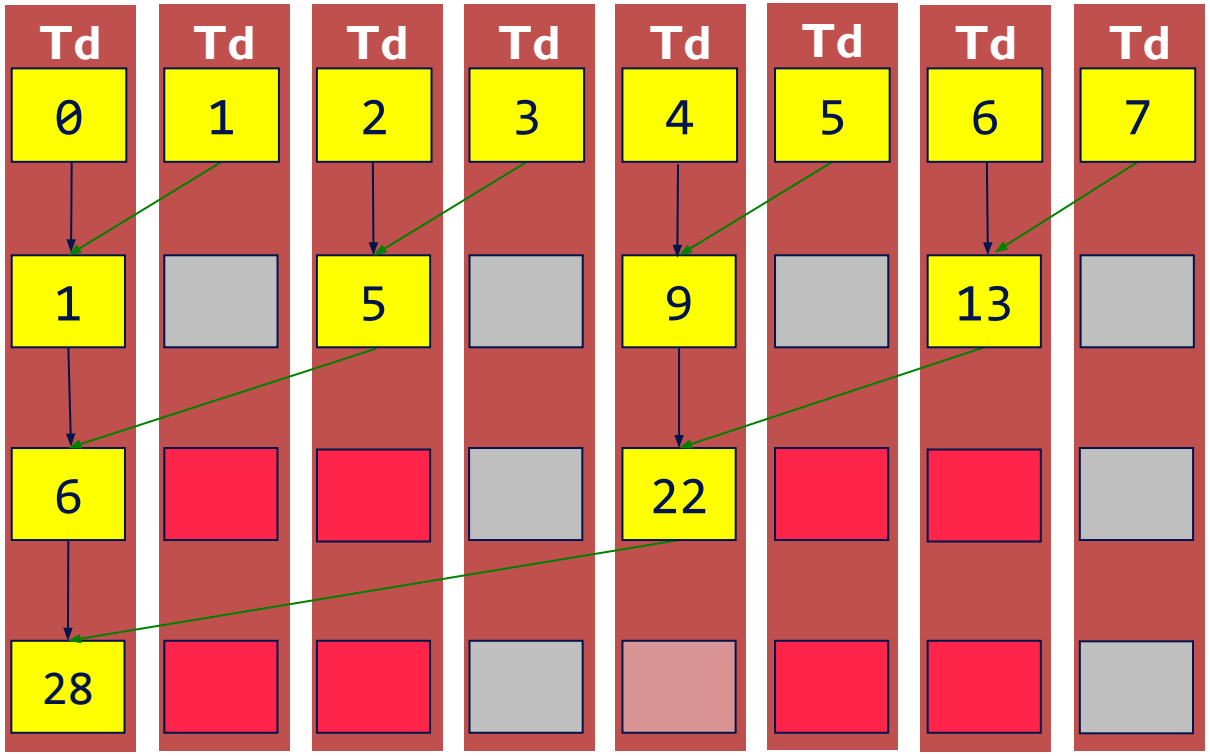
Parallel Reduction



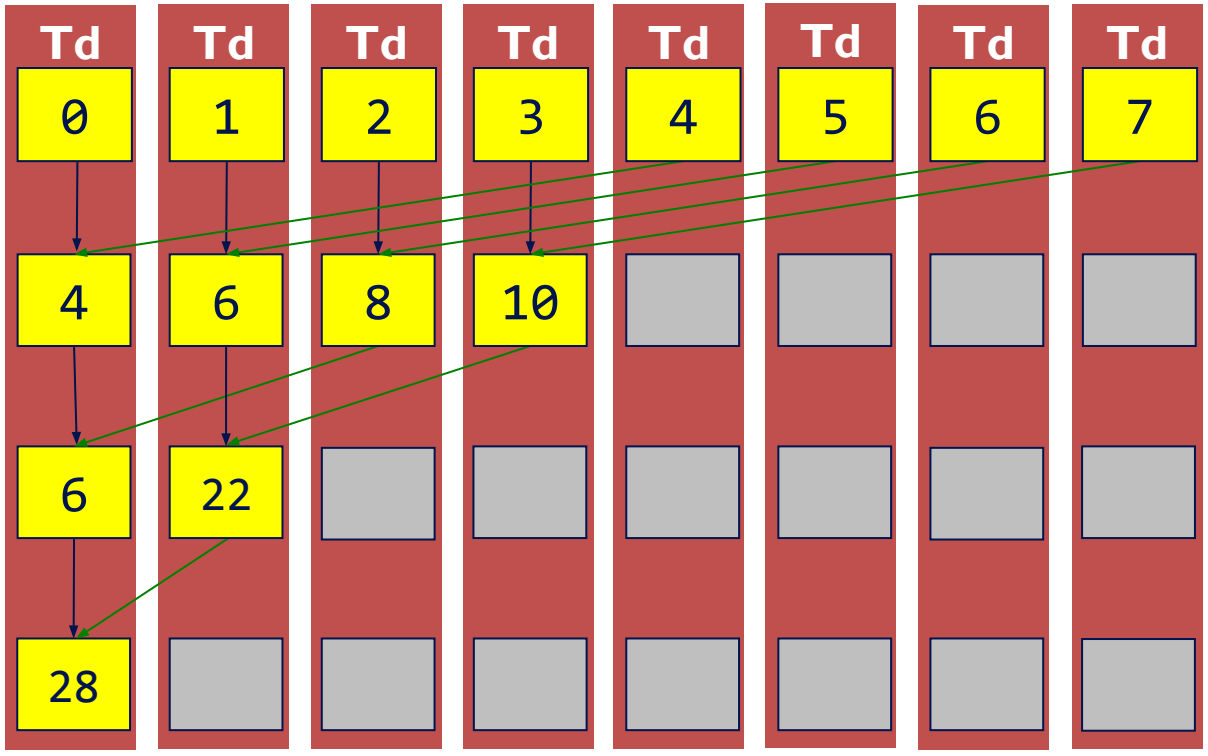
- **3rd pass**: thread 1 doesn't do anything

Parallel Reduction

- What's the difference?



stride = 1, 2, 4, …



stride = … 4, 2, 1

Parallel Reduction

- What is the difference?

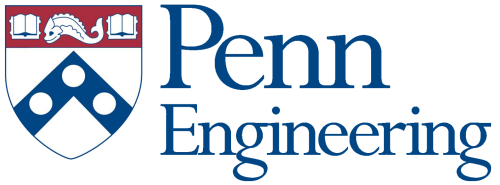
```
if (t % (2 * stride) == 0)
  partialSum[t] += partialSum[t + stride];
```

stride = 1, 2, 4, …

```
if (t < stride)
  partialSum[t] += partialSum[t + stride];
```

stride = … 4, 2, 1

Warp Partitioning



Warp Partitioning

- **Warp Partitioning**: how threads from a block are divided into warps
- Knowledge of warp partitioning can be used to:
  - Minimize divergent branches
  - Retire warps early
- **Partition based on consecutive increasing `threadIdx`**

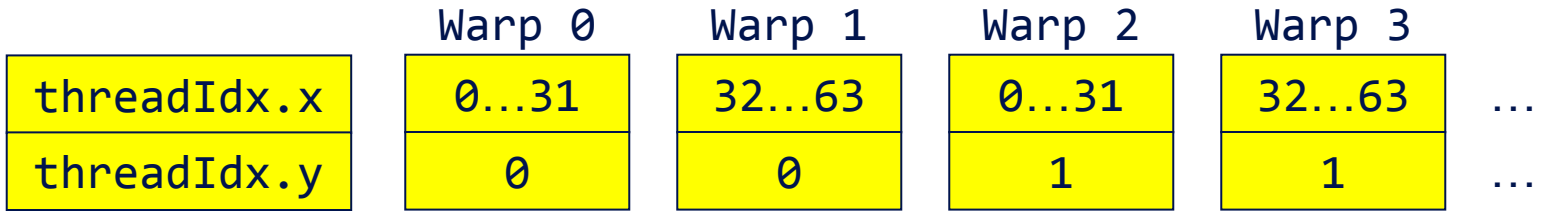
Warp Partitioning

- 1D Block
  - `threadIdx.x` between 0-1023 (Fermi & newer)
  - Warp `n`
    - Starts with thread `32n`
    - Ends with thread `32(n + 1) - 1`
  - Last warp is padded if block size is not a multiple of 32

Warp Partitioning

- 2D Block
  - Increasing `threadIdx` means
    - Increasing `threadIdx.x`
    - Starting with row `threadIdx.y` == 0

- Example for 64x8 block



Warp Partitioning

- 2D Block

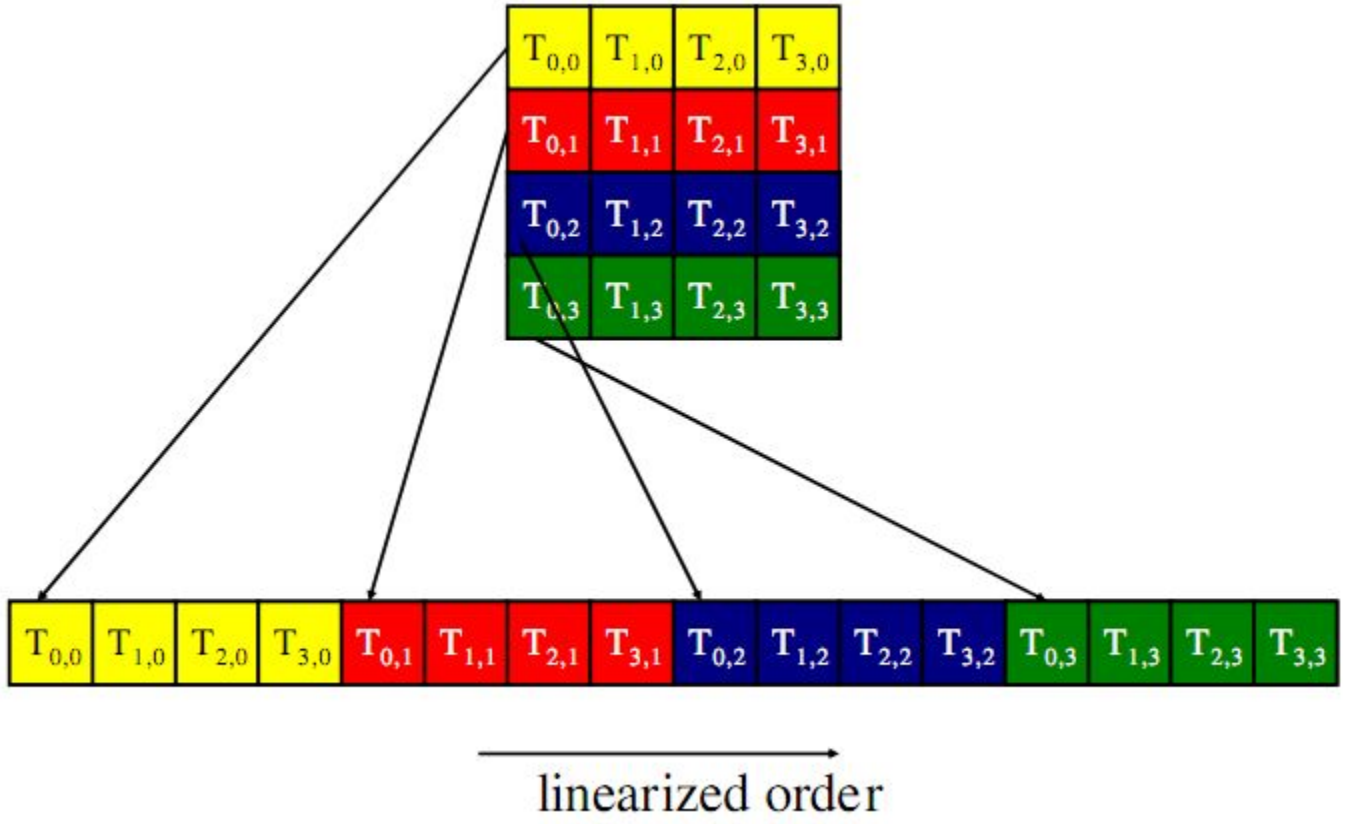


Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Warp Partitioning

- 3D Block
  - Start with `threadIdx.z` == 0
  - Partition as a 2D block
  - Increase `threadIdx.z` and repeat

Warp Partitioning

- **Divergent branches are within a warp!**

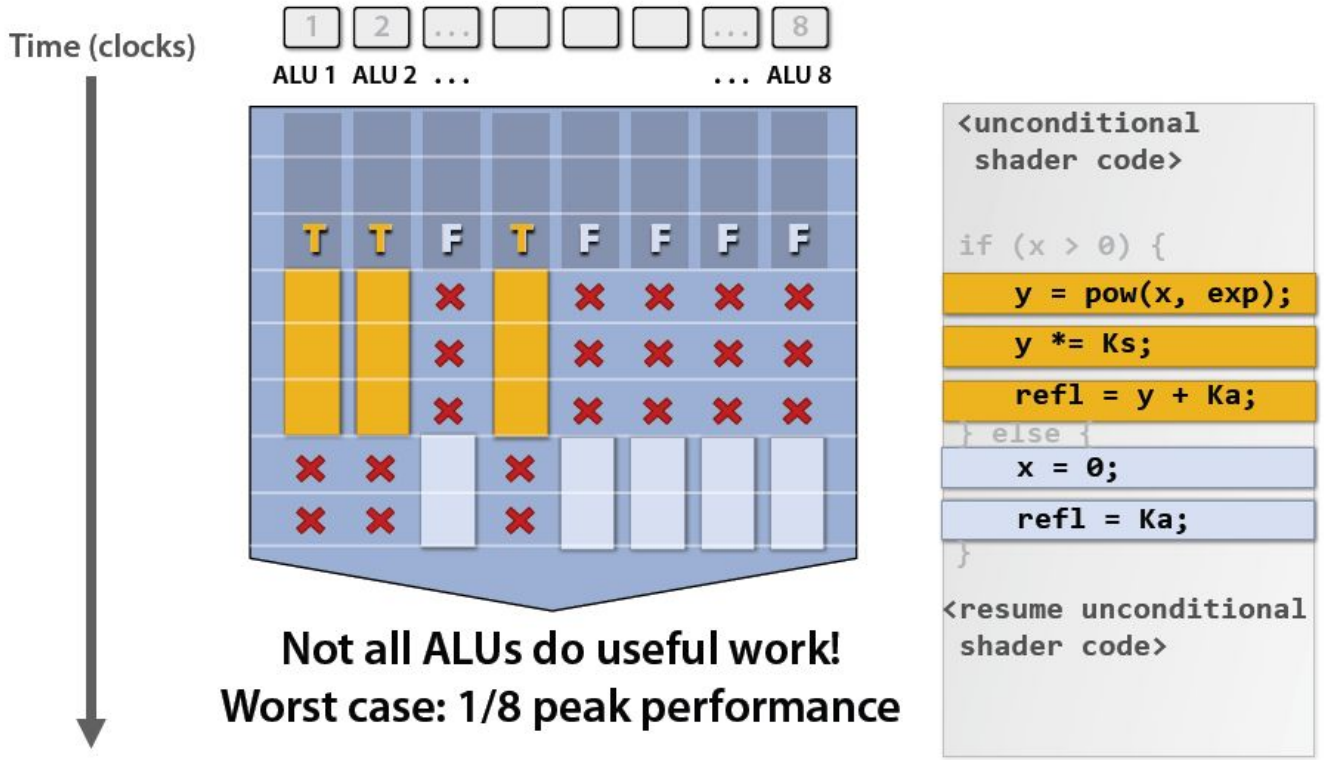


Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

Warp Partitioning

- For `warpSize == 32`, does any warp have a divergent branch with this code:

```
if (threadIdx.x > 15)
{
  // ...
}
```

Warp Partitioning

- For any `warpSize > 1`, does any warp have a divergent branch with this code:

```
if (threadIdx.x > warpSize - 1)
{
  // ...
}
```

Warp Partitioning

- Given knowledge of warp partitioning, which parallel reduction is better?

```
if (t % (2 * stride) == 0)
  partialSum[t] += partialSum[t + stride];
```

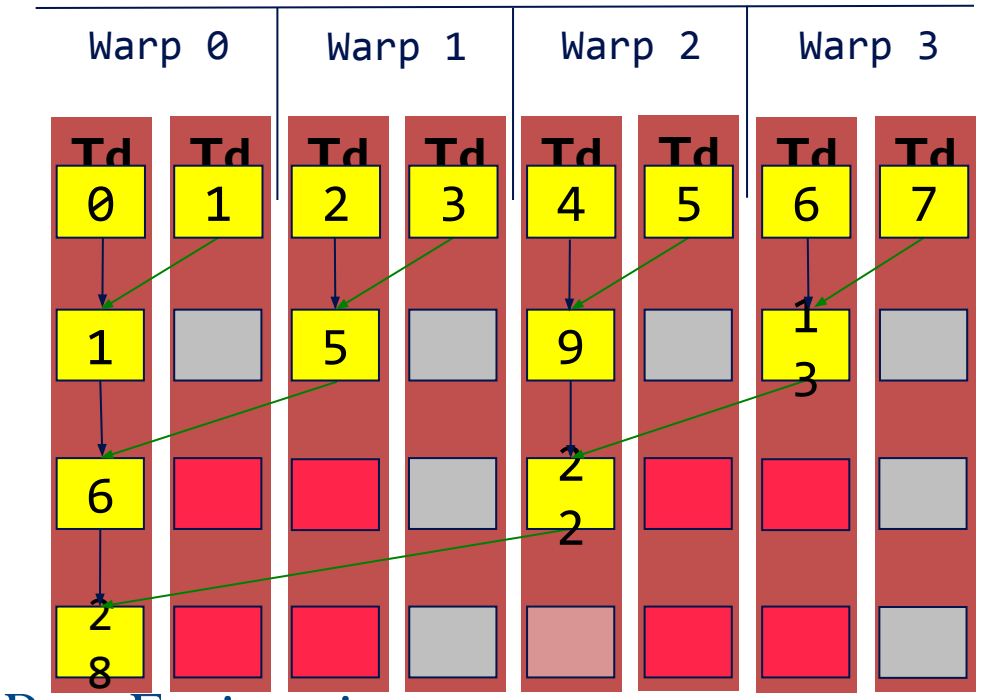
stride = 1, 2, 4, …

```
if (t < stride)
  partialSum[t] += partialSum[t + stride];
```

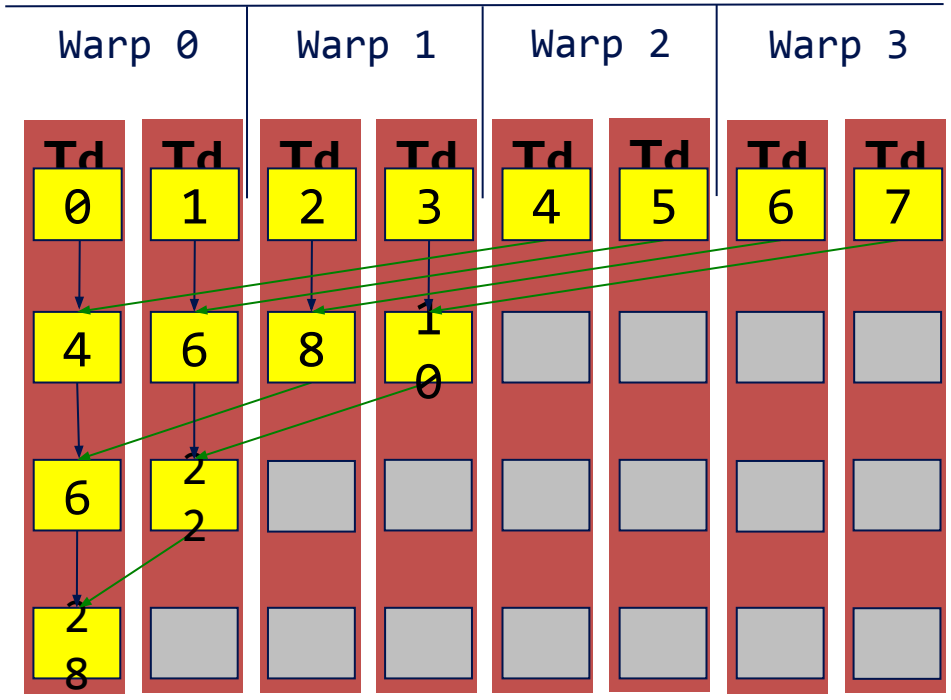
stride = … 4, 2, 1

Warp Partitioning

- Pretend `warpSize == 2`



stride = 1, 2, 4, …

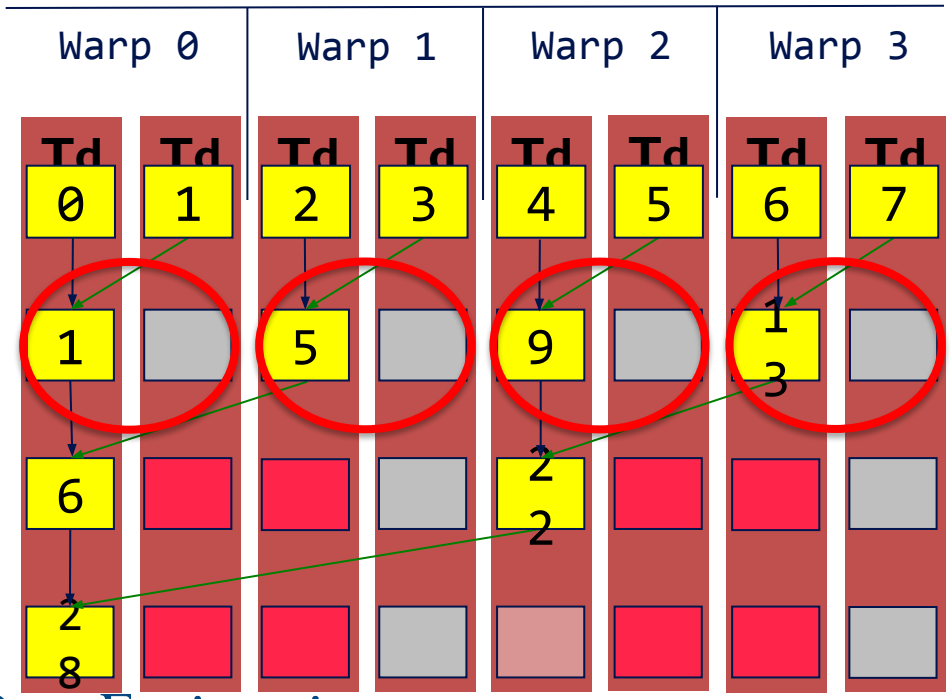


stride = … 4, 2, 1

Warp Partitioning

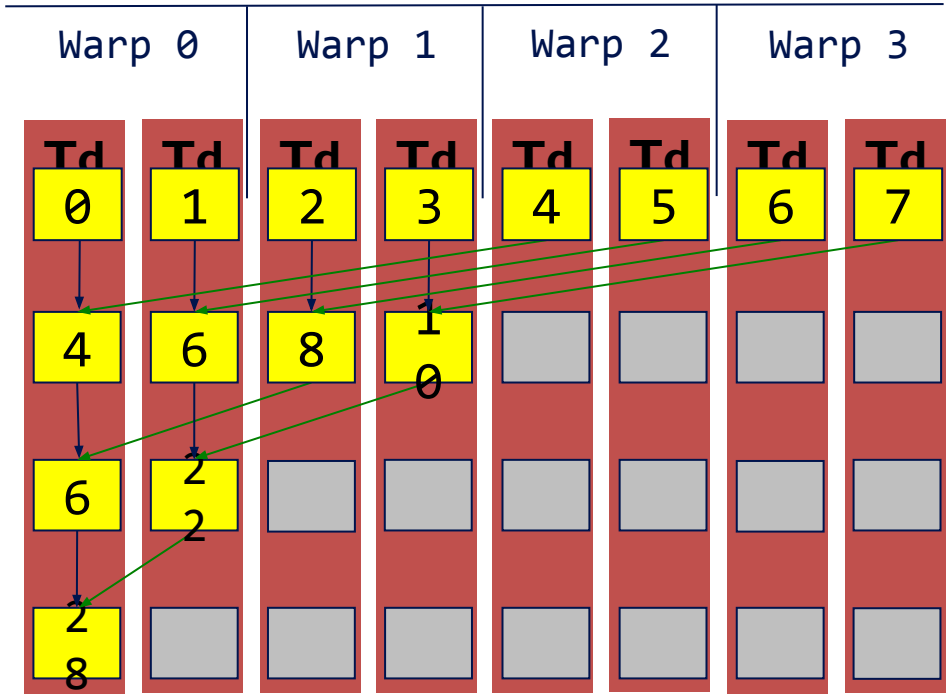
- 1st pass

4 divergent warps



stride = 1, 2, 4, …

0 divergent warps

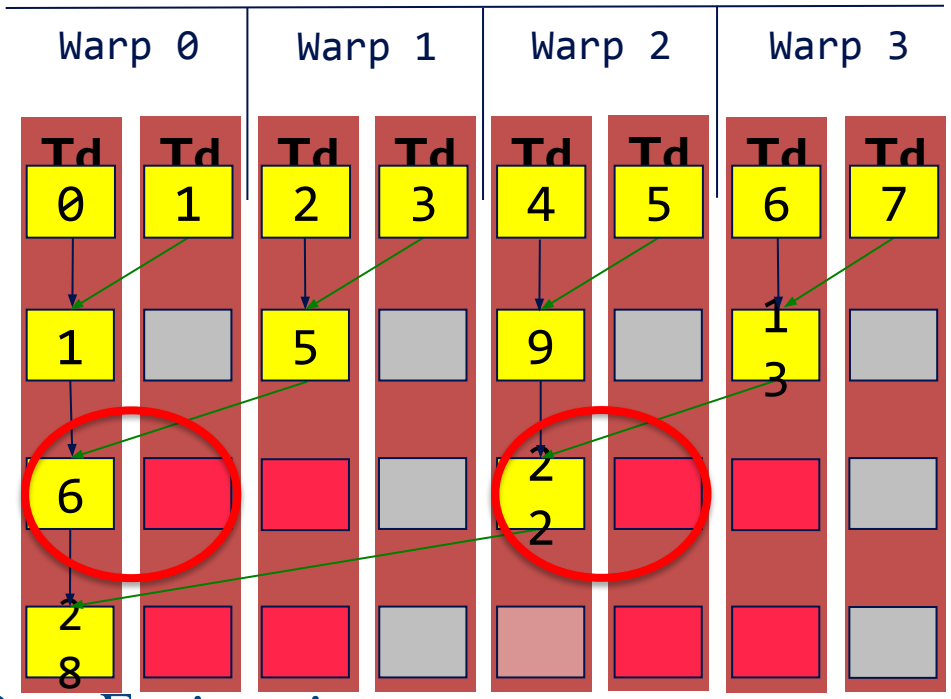


stride = … 4, 2, 1

Warp Partitioning

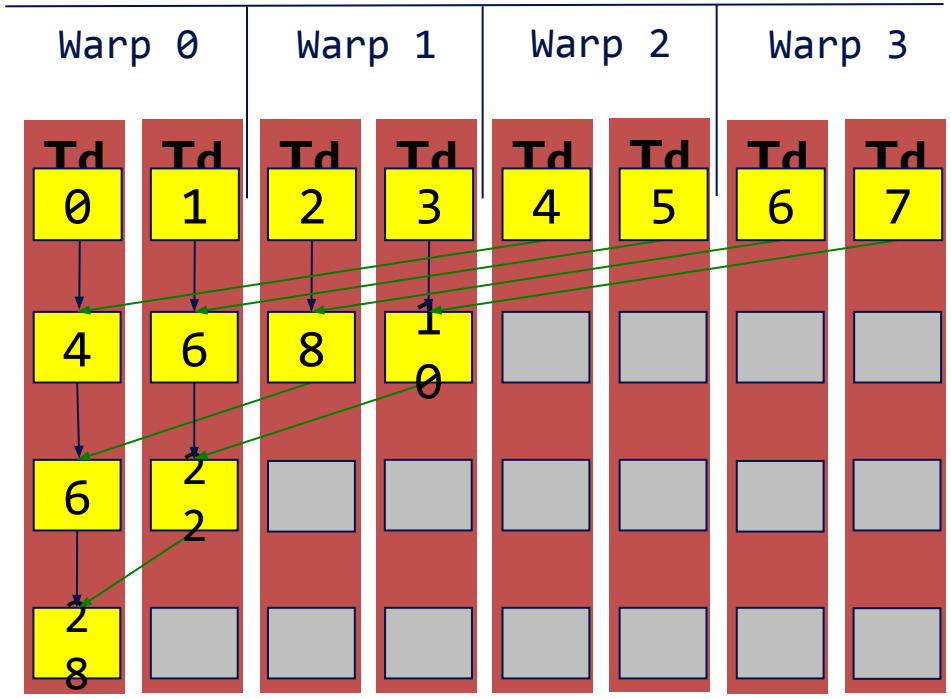
- 2nd pass

2 divergent warps



stride = 1, 2, 4, …

0 divergent warps

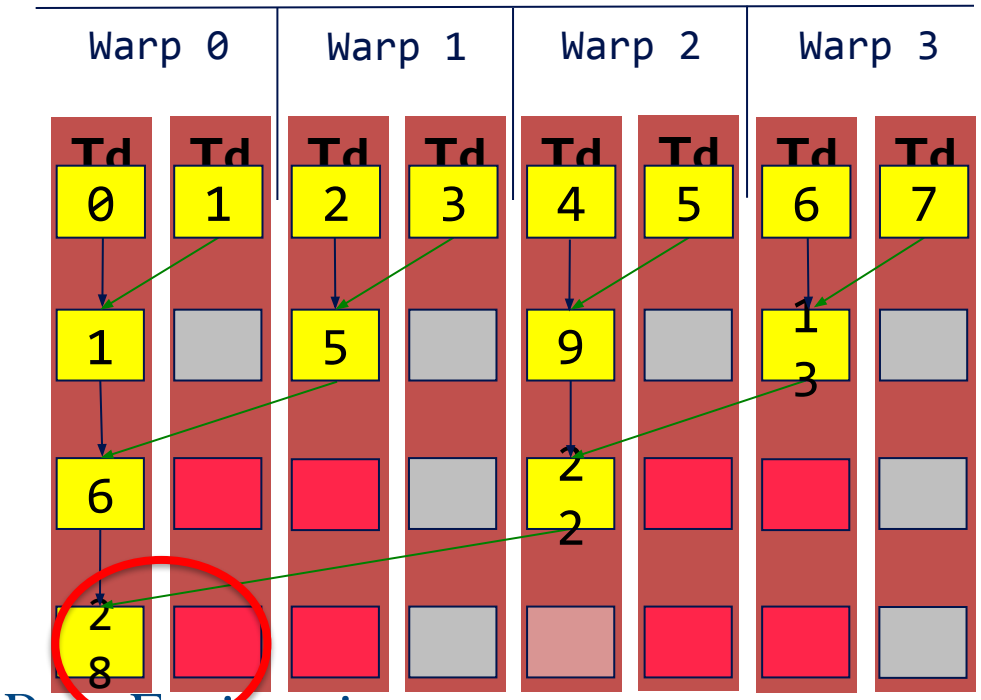


stride = … 4, 2, 1

Warp Partitioning

- 3rd pass

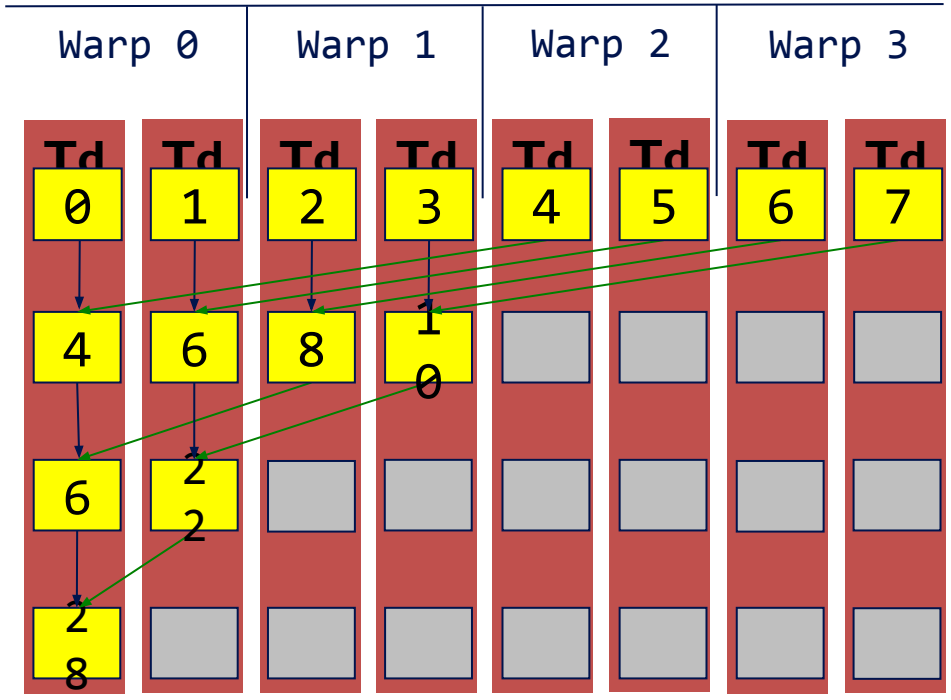
1 divergent warp



stride = 1, 2, 4, …

Optimized algorithm still diverges when `elements < warpSize`

1 divergent warp



stride = … 4, 2, 1

Warp Partitioning

- Good partitioning also allows warps to be retired early.
  - Better hardware utilization

```
if (t % (2 * stride) == 0)
  partialSum[t] += partialSum[t + stride];
```

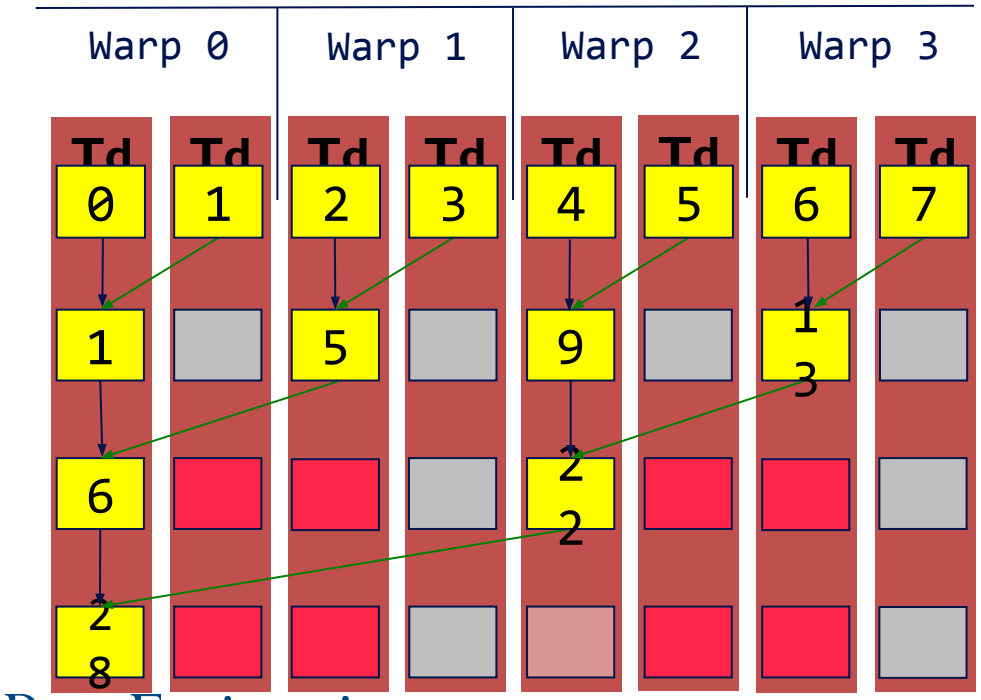
stride = 1, 2, 4, …

```
if (t < stride)
  partialSum[t] += partialSum[t + stride];
```

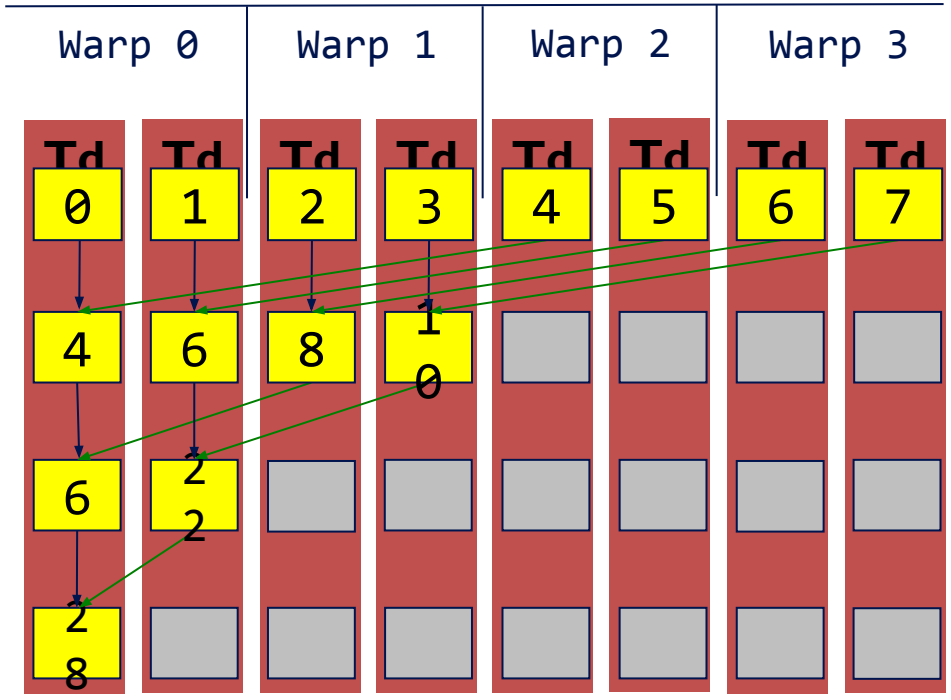
stride = … 4, 2, 1

Warp Partitioning

- Pretend `warpSize == 2`



stride = 1, 2, 4, …

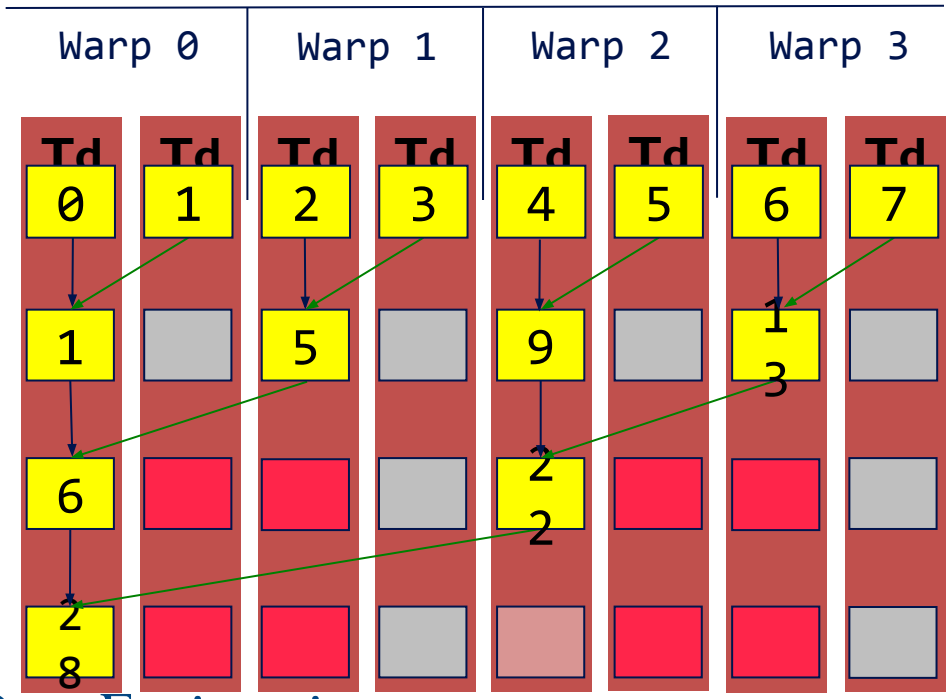


stride = … 4, 2, 1

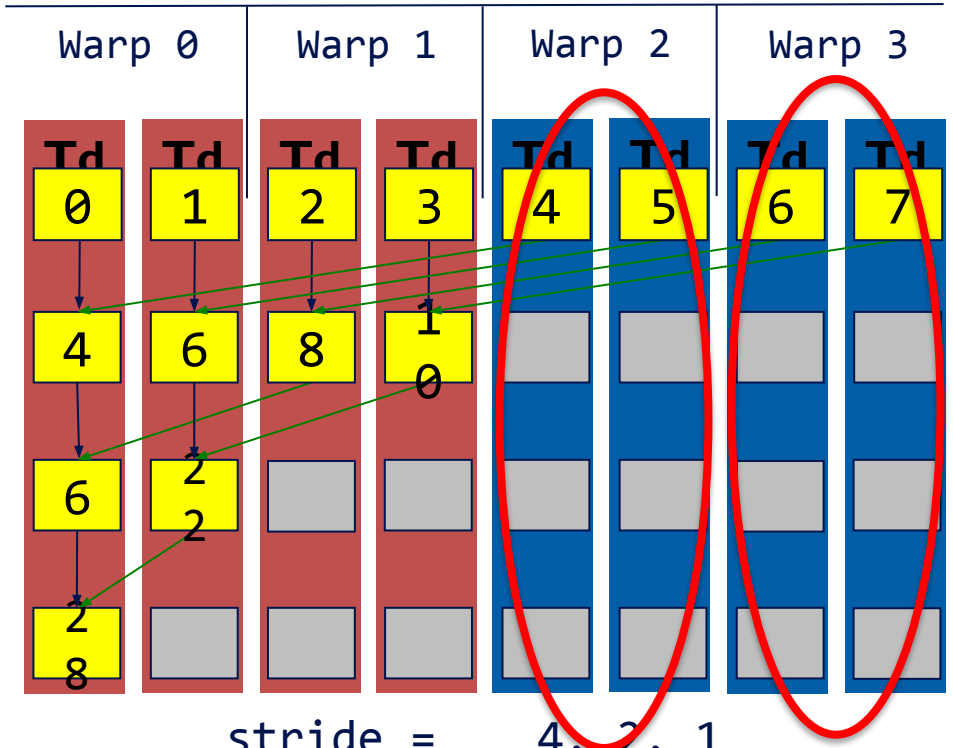
Warp Partitioning

- 1st pass

0 warps retired



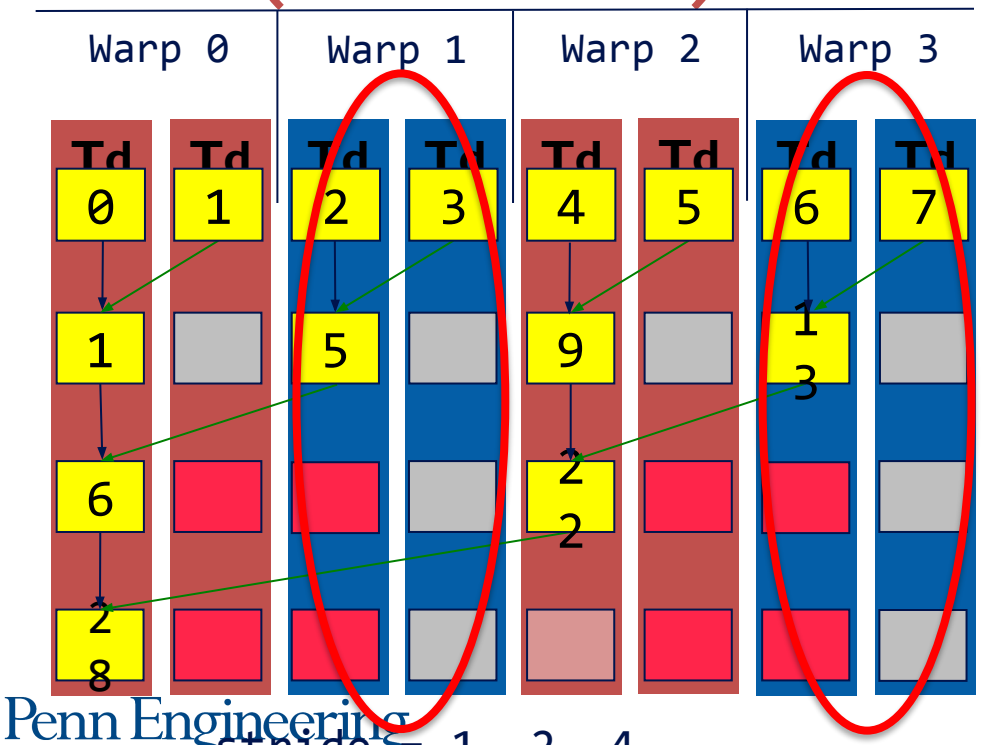
2 warps retired



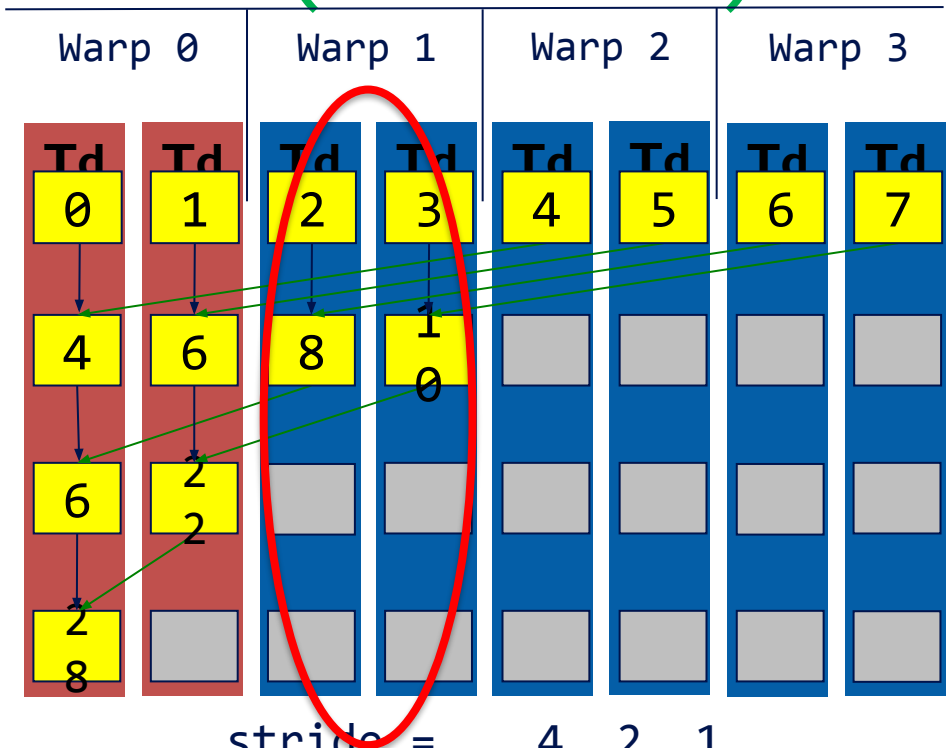
Warp Partitioning

- 2nd pass

2 warps retired (2 total)



1 warp retired (3 total)

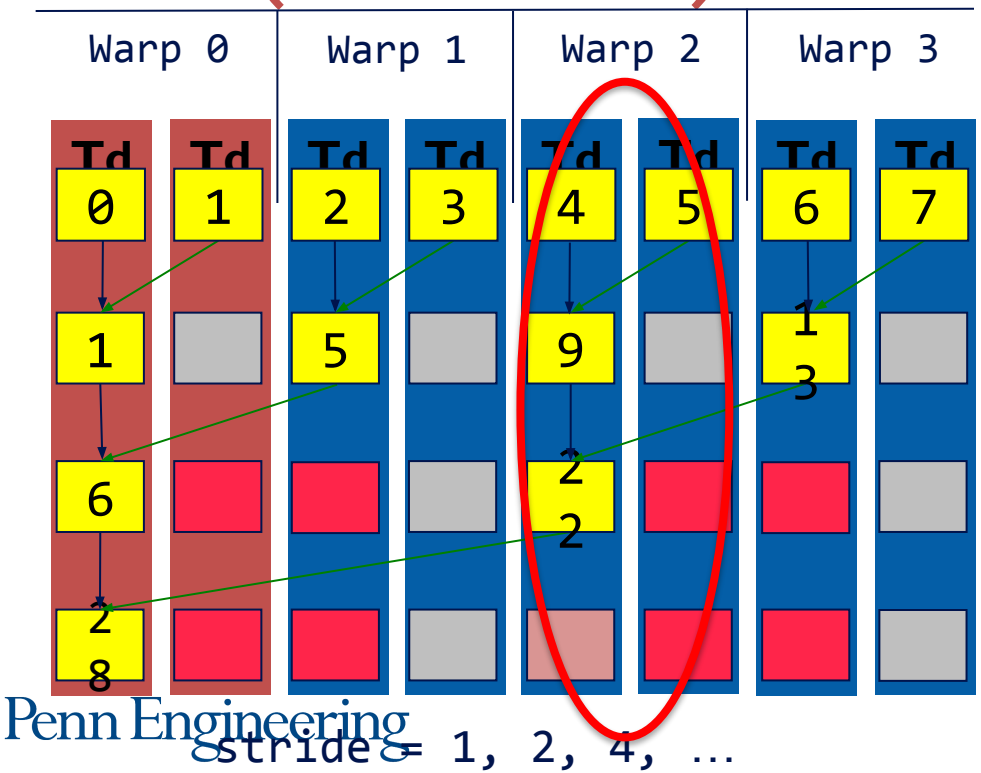


Warp Partitioning

- 3rd pass

1 warps retired

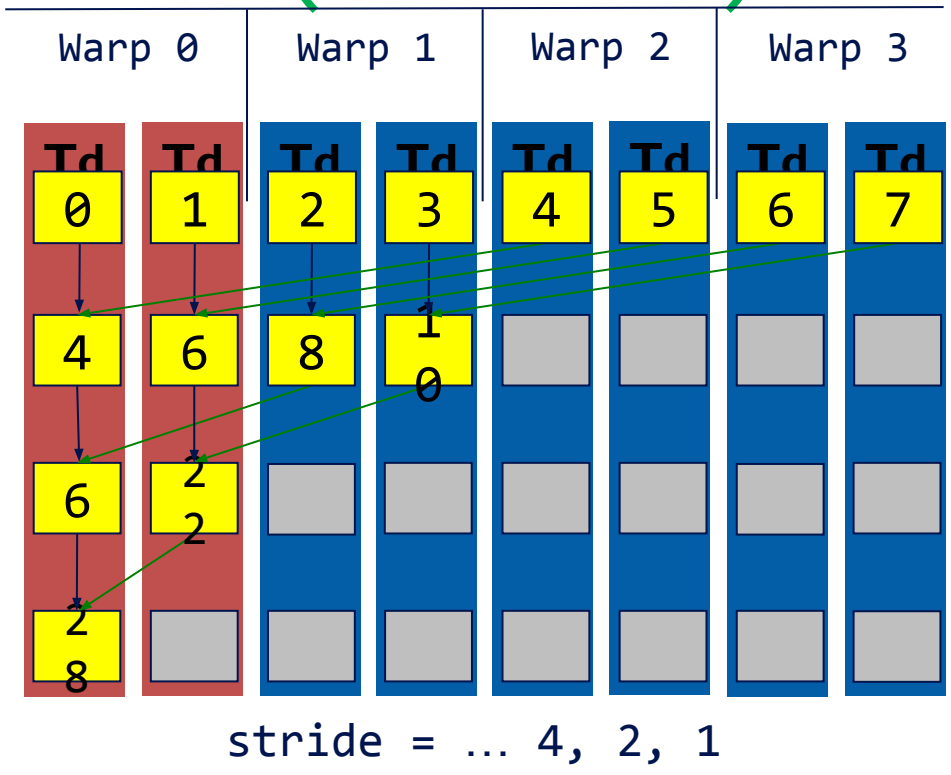
(3 total)



stride = 1, 2, 4, …

0 warps retired

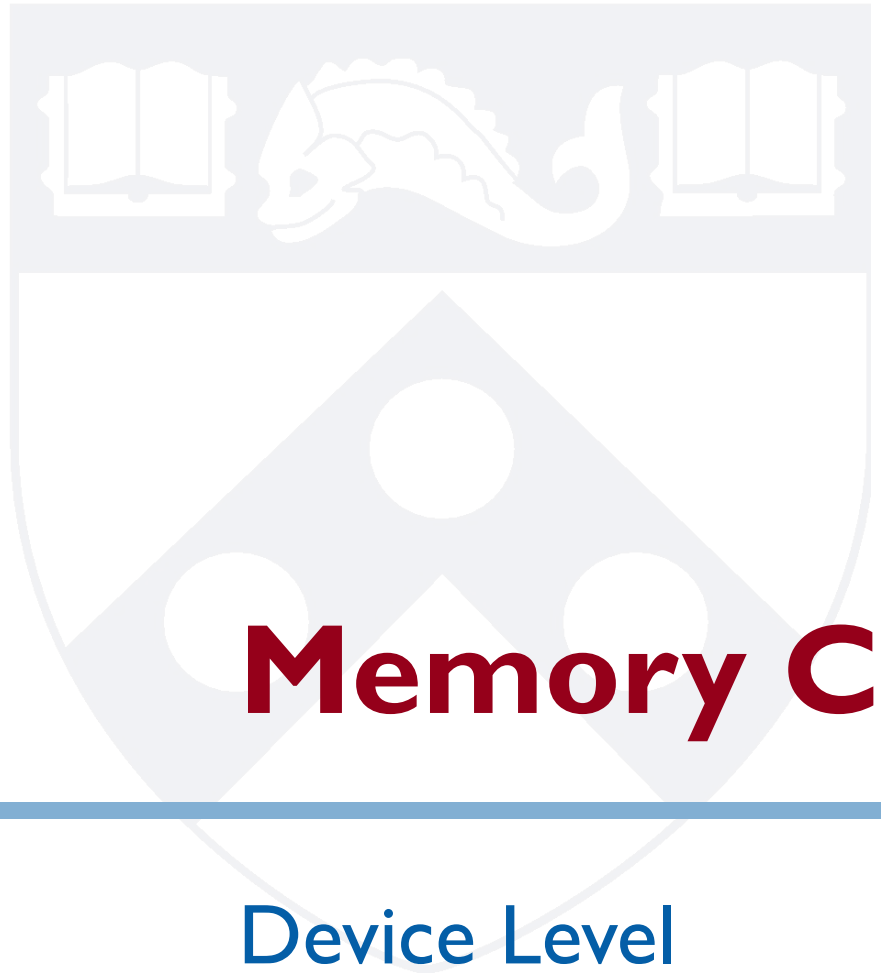
(3 total)



stride = … 4, 2, 1

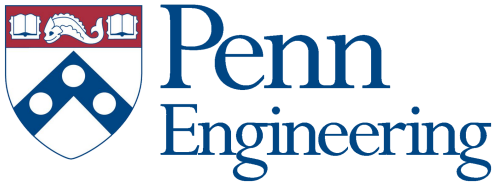
Warp Partitioning

- Construct/Rework the algorithm such that divergence within a warp is minimal
  - At best none, at worst – limited divergence
- Try to retire warps early – saves SM clock time
  - More warps available to be scheduled



Memory Coalescing

Device Level



Memory Coalescing

- Given a matrix stored **row-major** in **global memory**, what is a **thread**'s desirable access pattern?

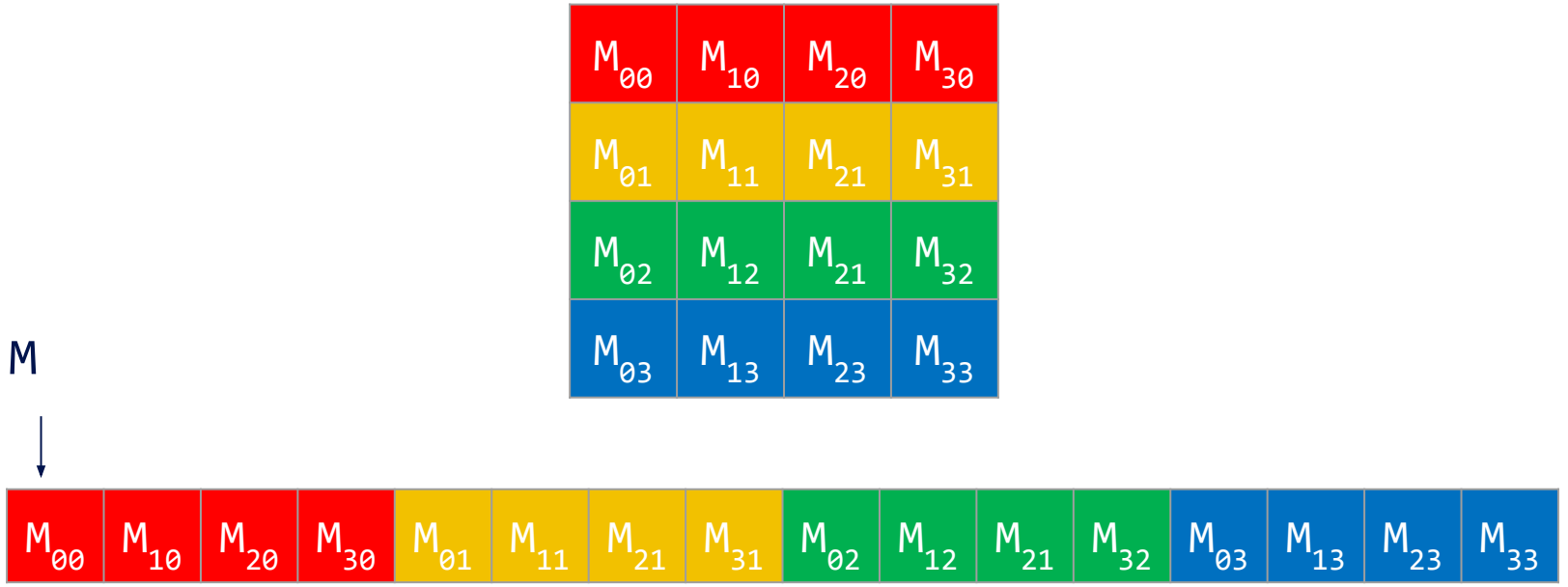
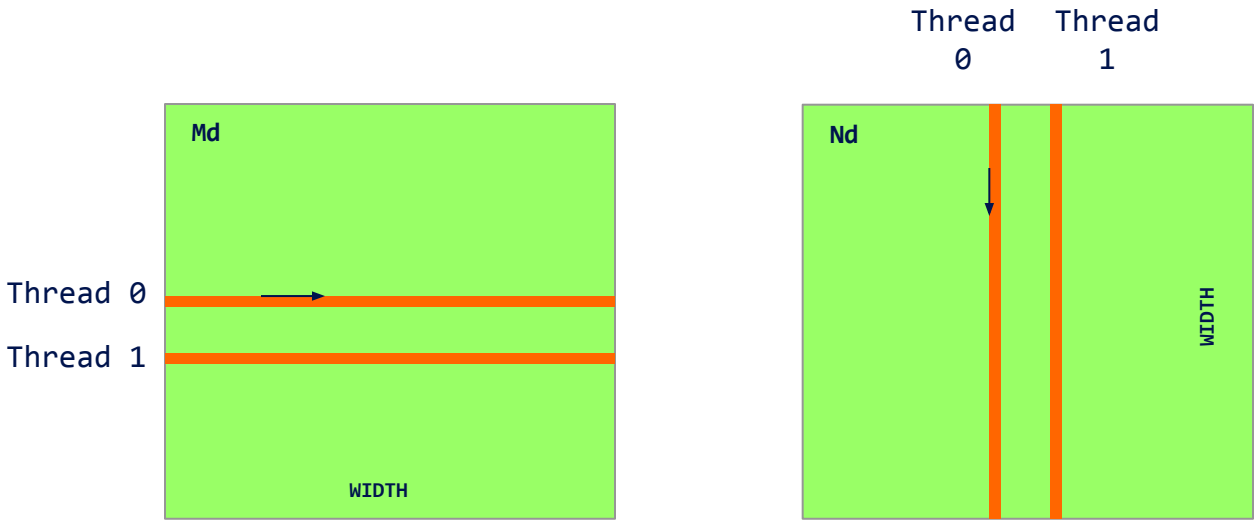


Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

Memory Coalescing

- Given a matrix stored **row-major** in **global memory**, what is a **thread**'s desirable access pattern?



a) column after column?

b) row after row?

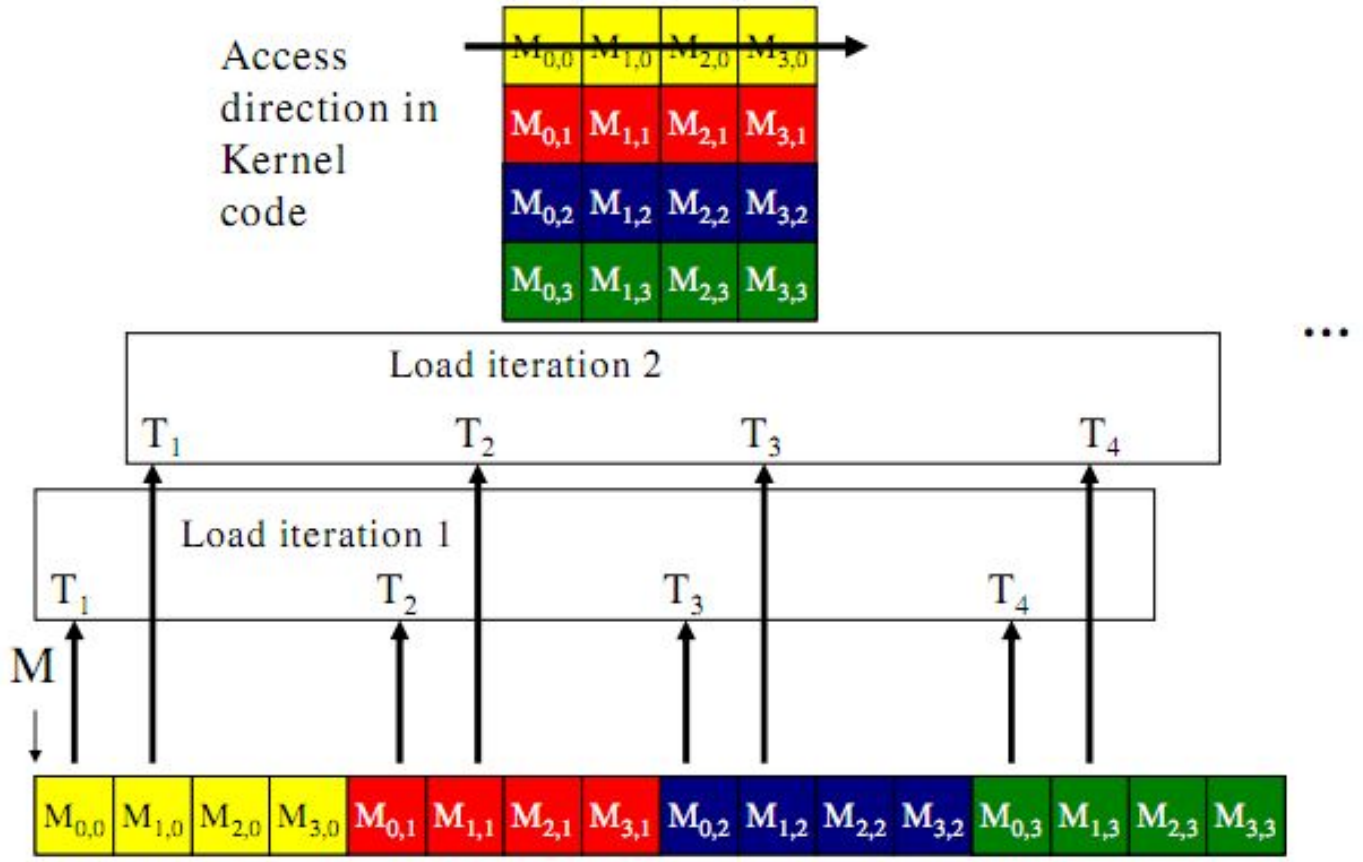
Penn Engineering

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Memory Coalescing

- Given a matrix stored **row-major** in **global memory**, what is a **thread**'s desirable access pattern?
  - a) column after column
    - Individual threads read increasing, **consecutive** memory address
  - b) row after row
    - Adjacent threads read increasing, **strided** memory addresses

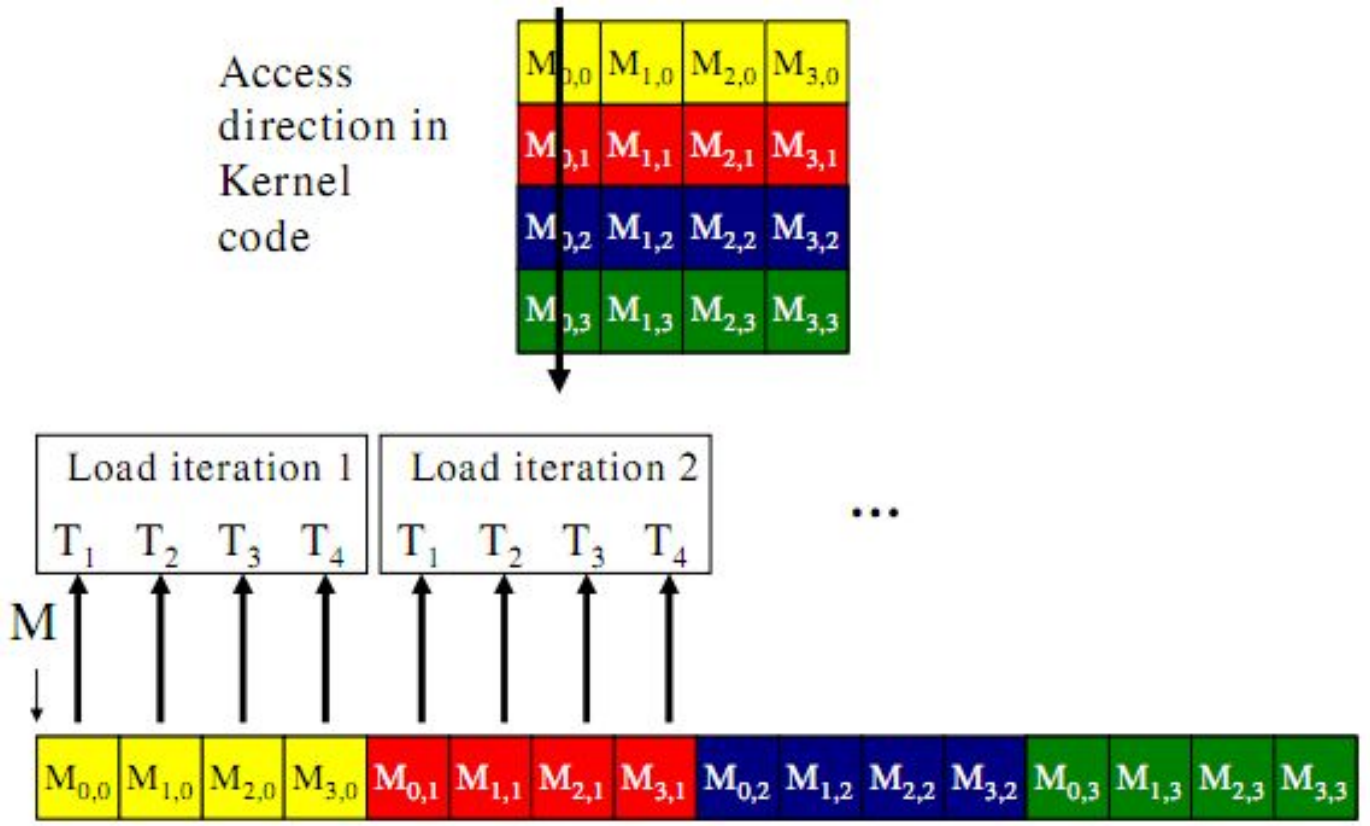
Memory Coalescing



a) column after column

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Memory Coalescing



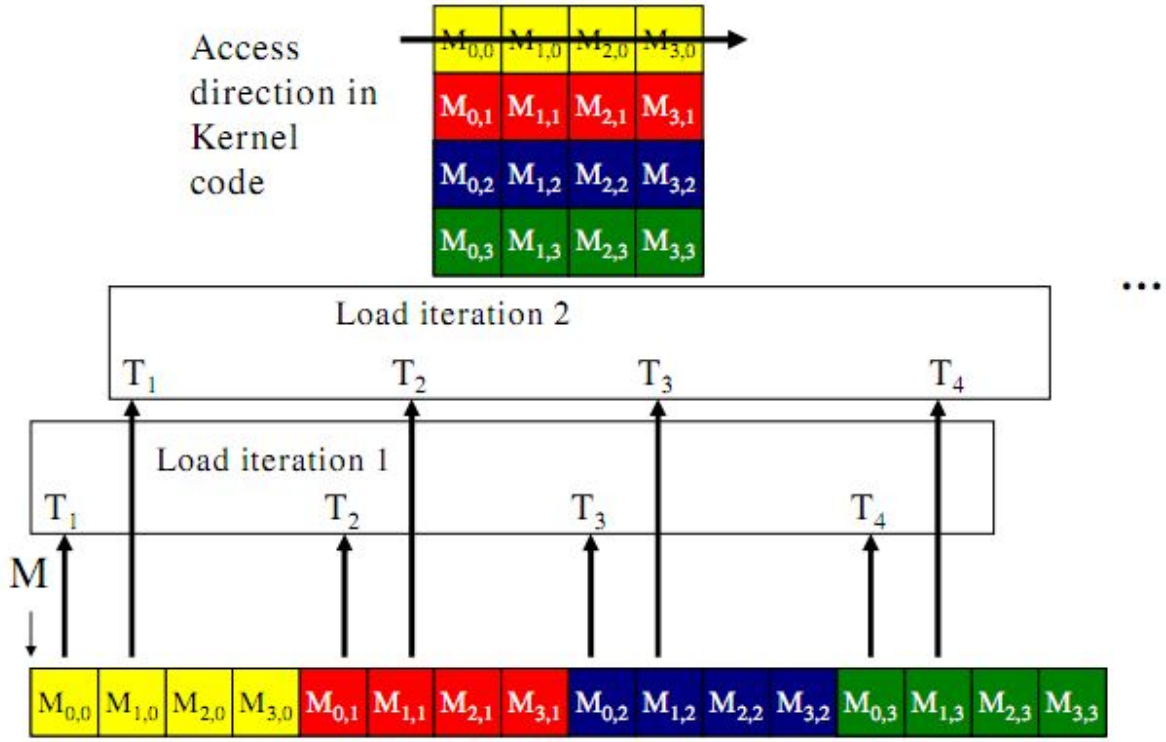
b) row after row

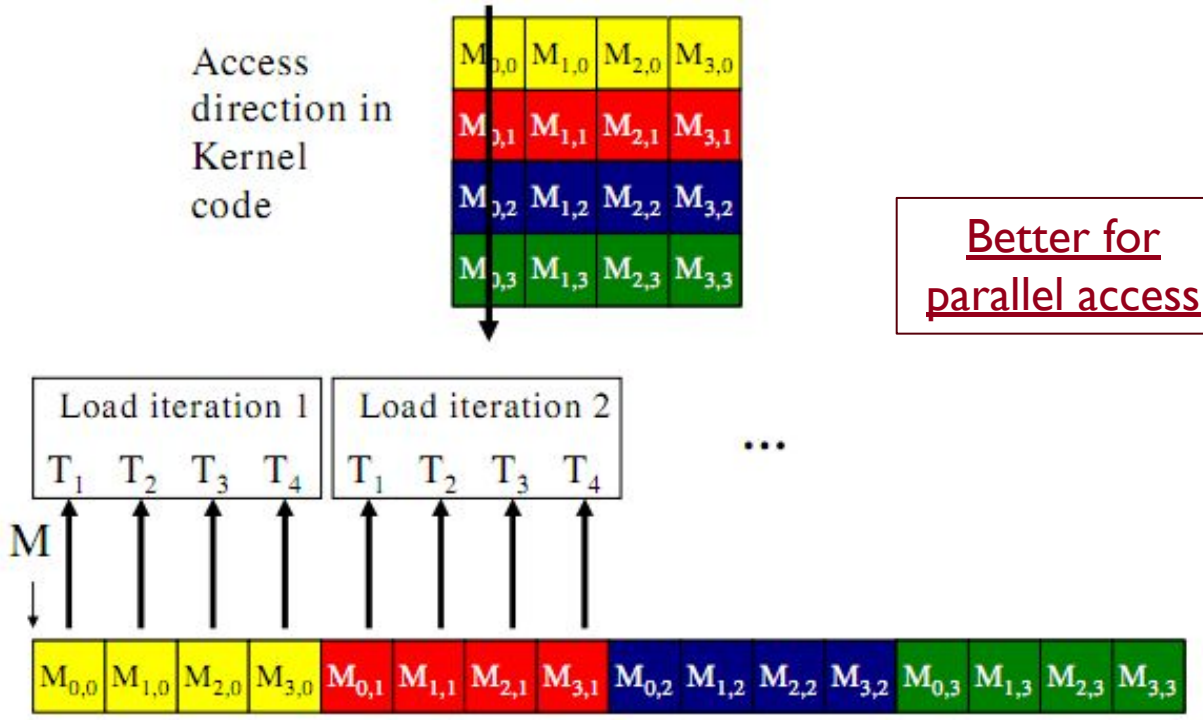
Penn Engineering

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Memory Coalescing

- Think in terms of **parallel threads access**
- Want **threads within an iteration** to be accessing continuous memory





Better for parallel access

Penn Engineering

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Memory Coalescing

- Global memory bandwidth (DRAM)
  - G80 – 86.4 GB/s, GT200 – 150 GB/s, Fermi – 177 GB/s
  - Kepler – 192 GB/s, Maxwell – 224 GB/s, Pascal – 320 GB/s
  - Turing – 448 GB/s, Ampere – 760 GB/s, Lovelace – 760 GB/s
- Achieve peak bandwidth by requesting large, **consecutive locations** from DRAM
  - Accessing random location results in much lower bandwidth

Memory Coalescing

- The GPU coalesces consecutive reads in a **full-warp** into a single read
- Strategy: read global memory in a coalesce-able fashion into shared memory
  - Then access shared memory randomly at maximum bandwidth
    - Ignoring bank conflicts…

See Appendix G in the NVIDIA CUDA C Programming Guide for coalescing alignment requirements

Memory Coalescing

- **Memory coalescing** – rearrange access patterns to improve performance
- Useful today but will be less useful with large on-chip caches

Bank Conflicts

Warp Level

Penn Engineering

Bank Conflicts

- Shared Memory
  - Sometimes called a **parallel data cache**
    - Multiple threads can access shared memory at the same time
  - Memory is divided into **banks**

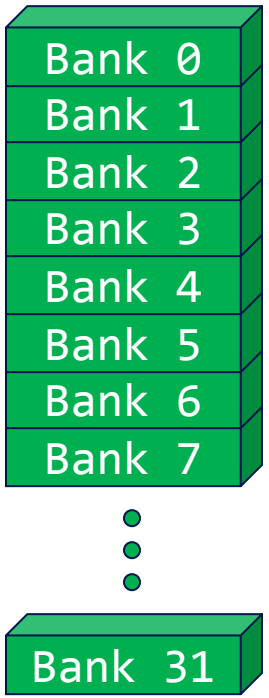


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- **Banks**
  - Each bank can service one **address** per cycle
  - Per-bank bandwidth: 32-bits per clock cycle
  - Successive 32-bit **words** are assigned to successive banks

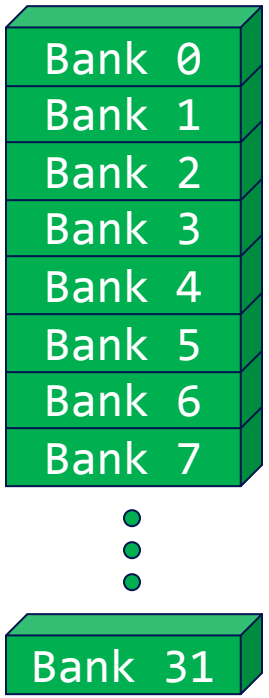


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- **Bank Conflict**: Two simultaneous accesses to the same bank, but not the same address
  - **Serialized !!!**
- G80-GT200: 16 banks, with 8 SPs concurrently executing
- Fermi & Newer: 32 banks, with 16 SPs concurrently executing

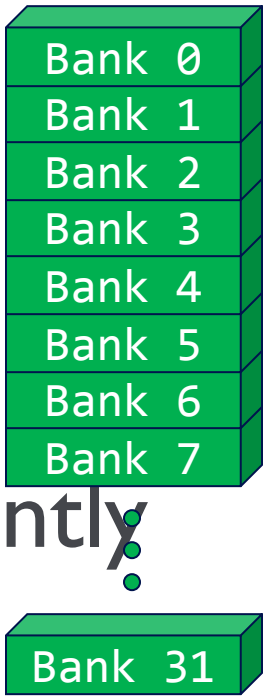
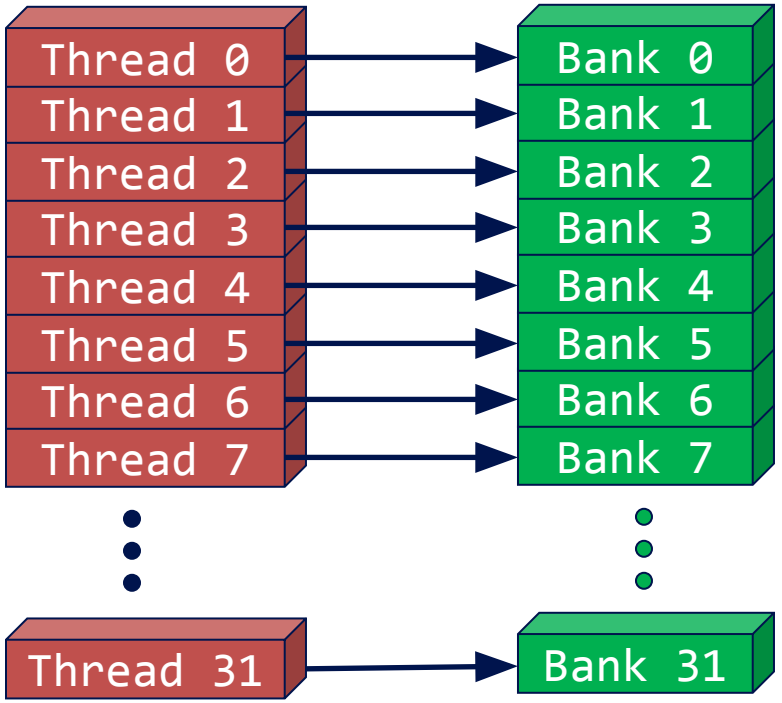


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Bank Conflicts?
  - Linear addressing `stride == 1`



- Bank Conflicts?
  - Random **1:1** Permutation

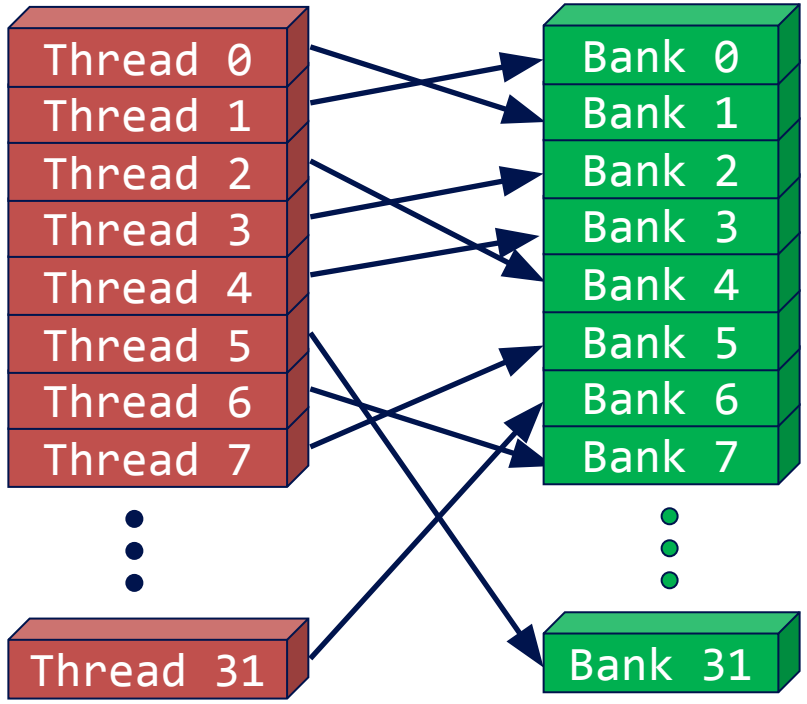
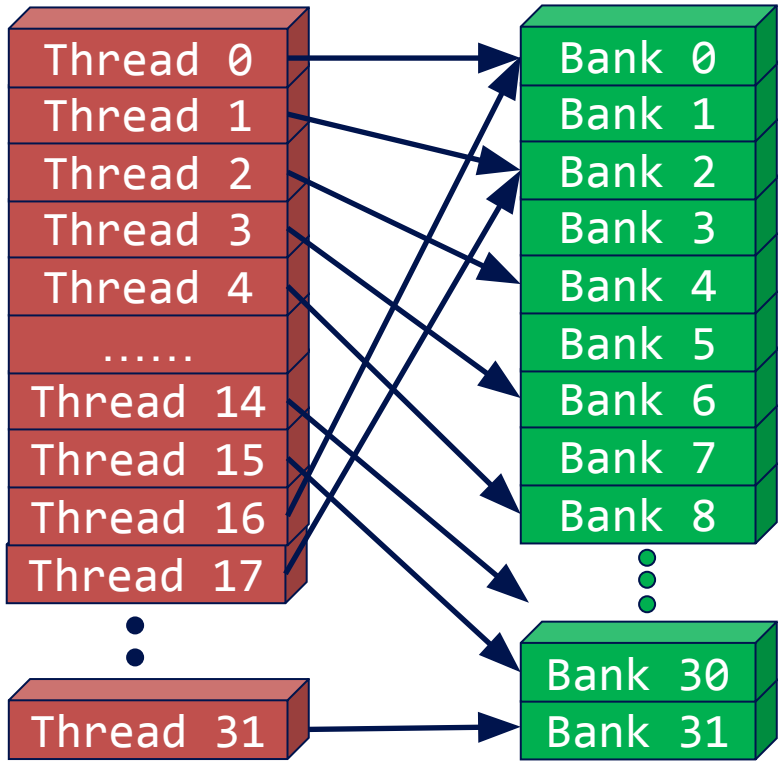


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Bank Conflicts?
  - Linear addressing `stride == 2`
  - `Bank = thread * stride % 32`



- Bank Conflicts?
  - Linear addressing `stride == 8`
  - `Bank = thread * stride % 32`

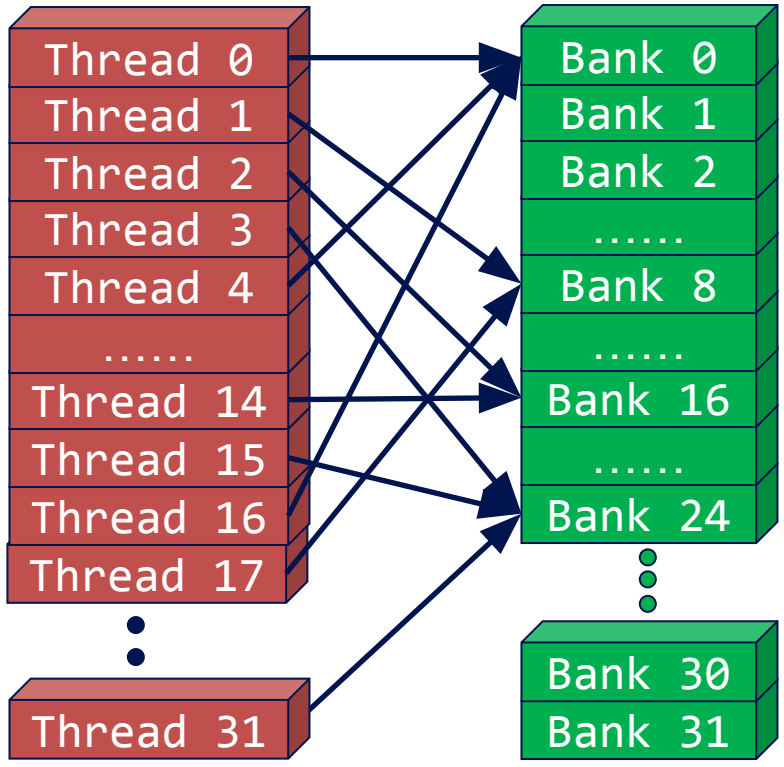


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Fast Path (Fermi & newer)
  - All threads in a warp access different banks

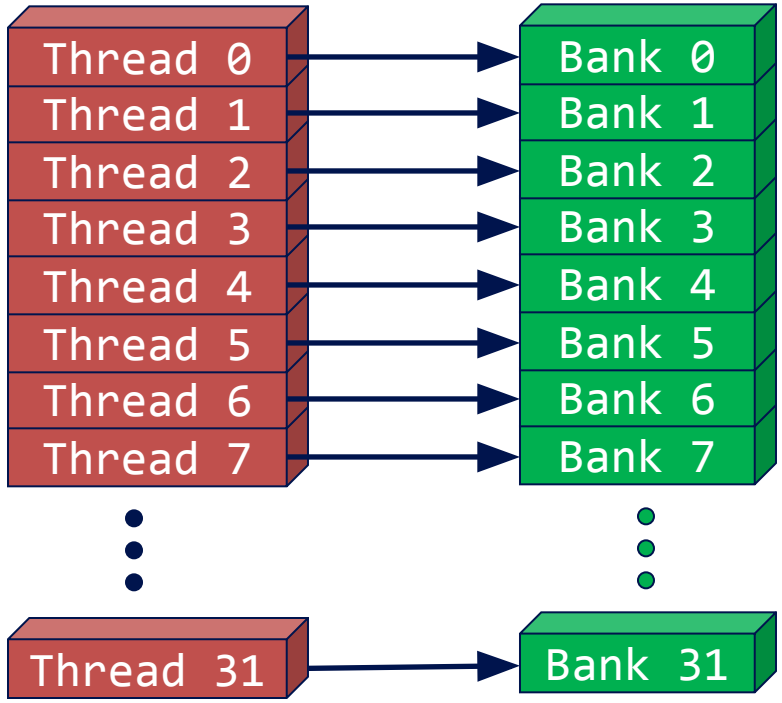


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Fast Path 2 (Fermi & newer)
  - Two or more threads in a warp access the **same address**
  - "**Broadcast**"

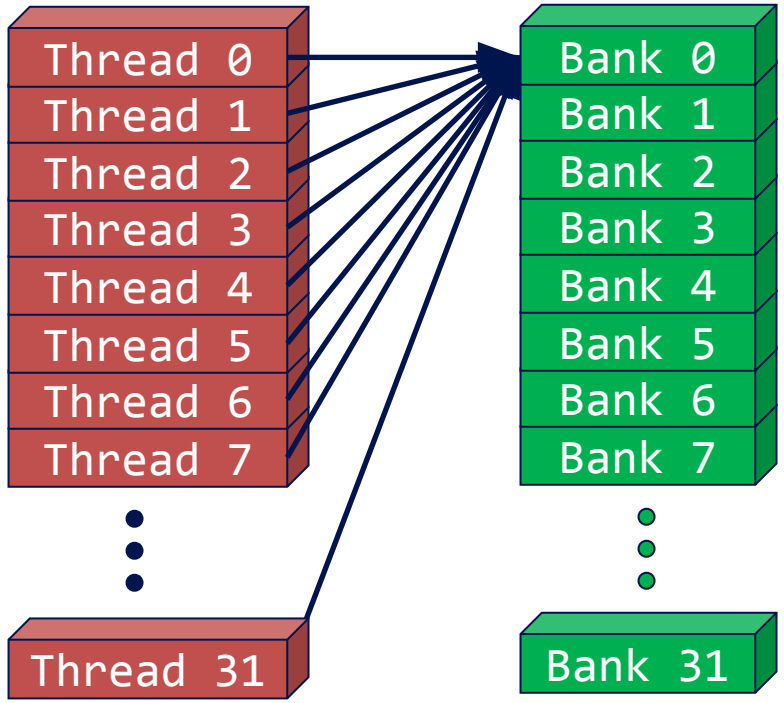


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- **Slow Path** (Fermi & newer)
  - Multiple threads in a warp access the same bank
  - Access is serialized
  - What is the cost?

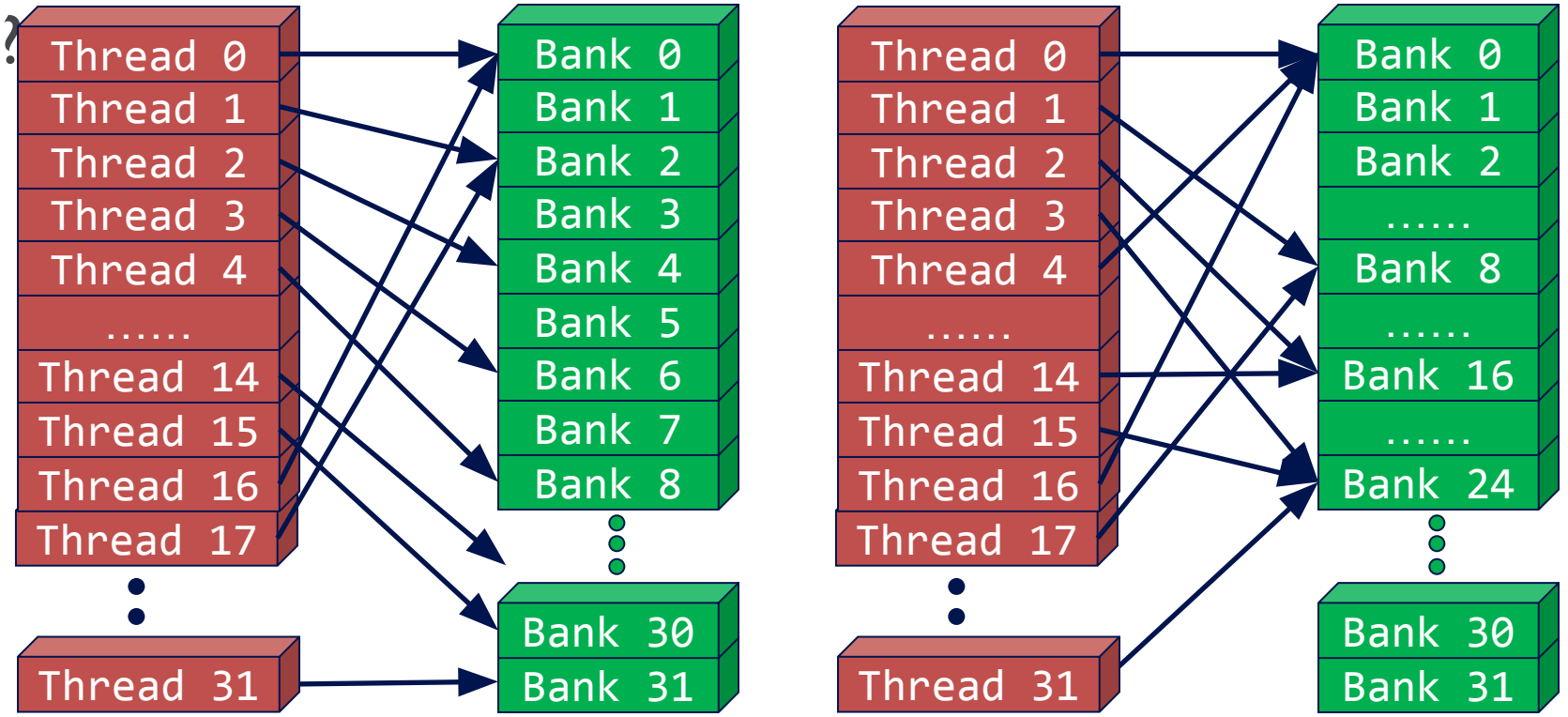


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- For what values of s is this conflict free?

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

Bank Conflicts

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

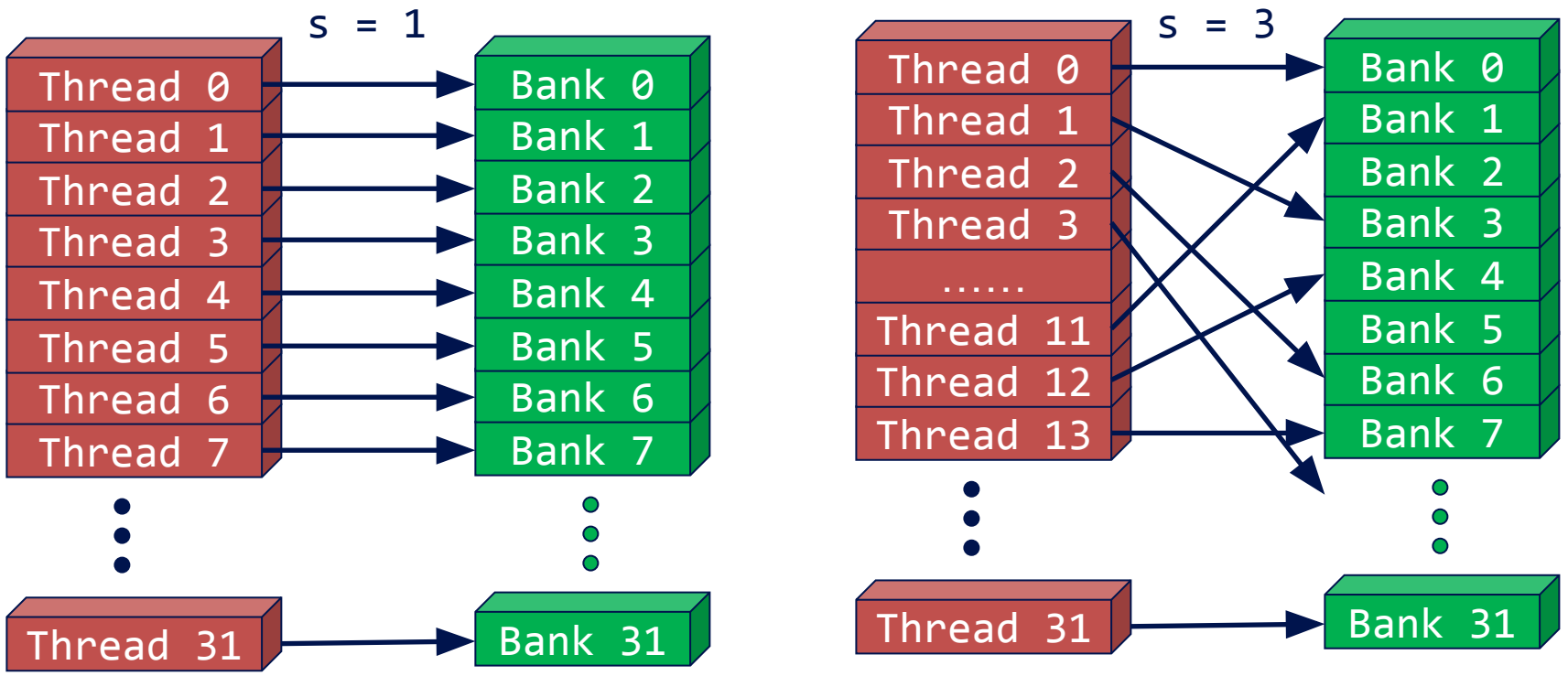


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

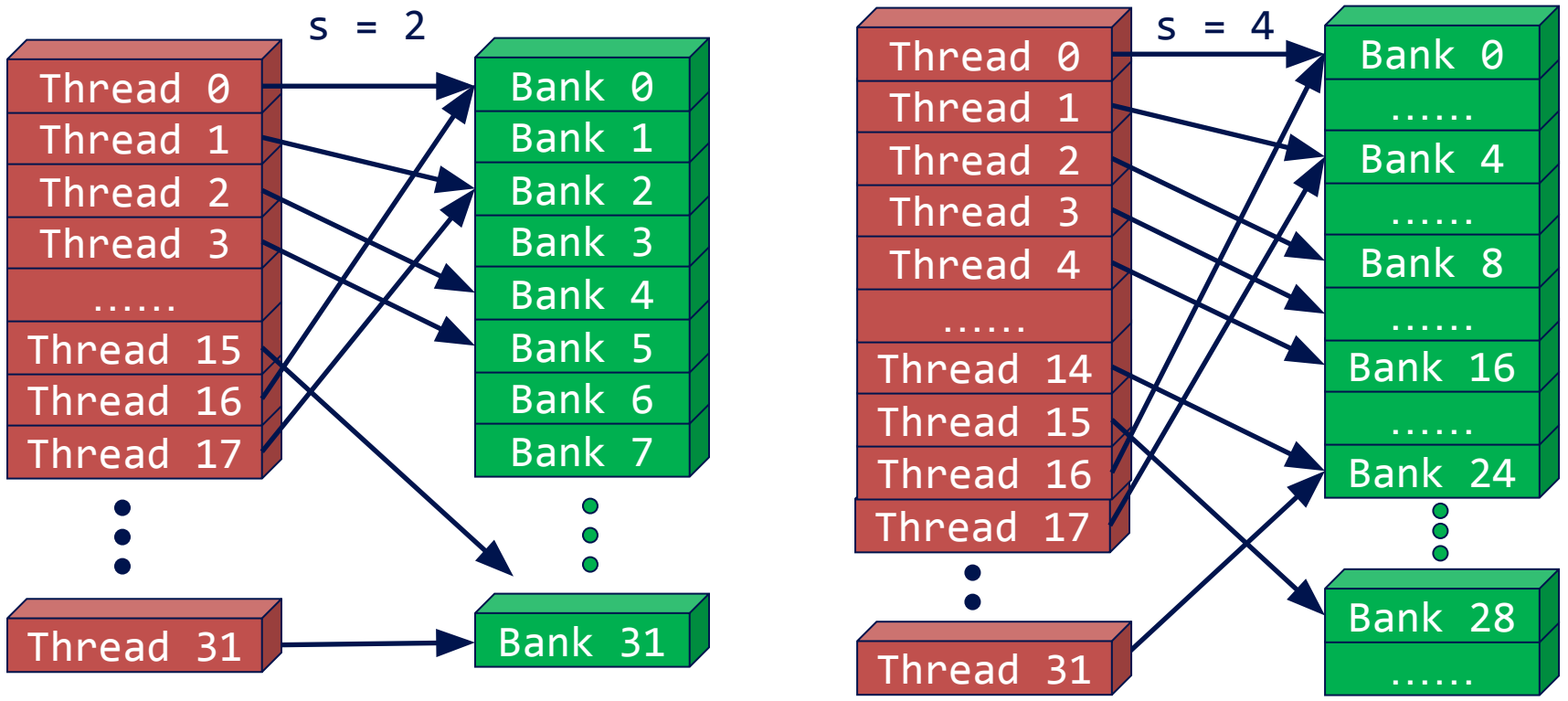
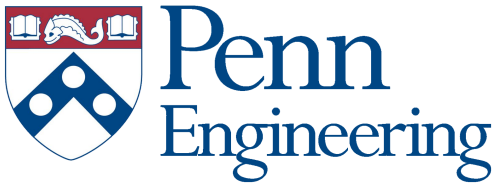


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

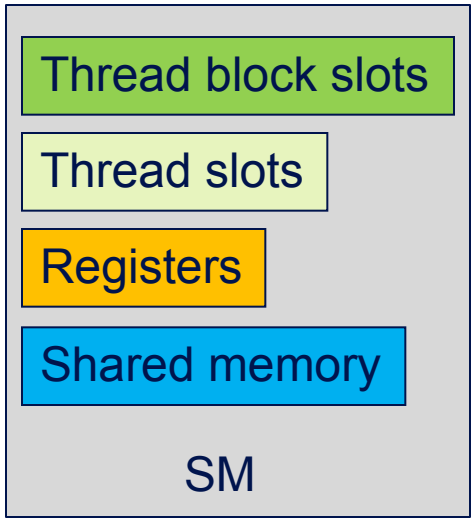
- Without using a profiler, how can we tell what kind of speedup we can expect by removing bank conflicts?
- What happens if more than one thread in a warp writes to the same shared memory address (non-atomic instruction)?

SM Resource Partitioning



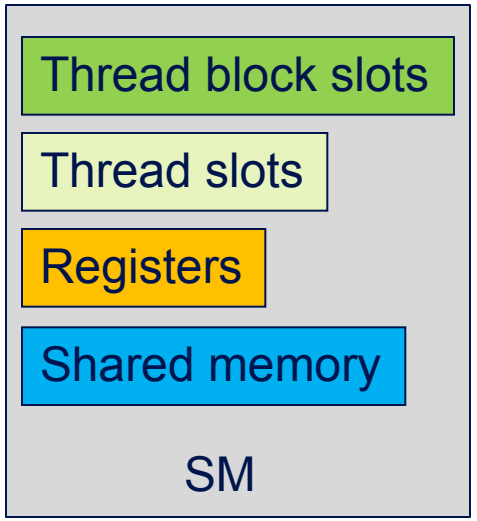
SM Resource Partitioning

- Recall a SM dynamically partitions resources:



SM Resource Partitioning

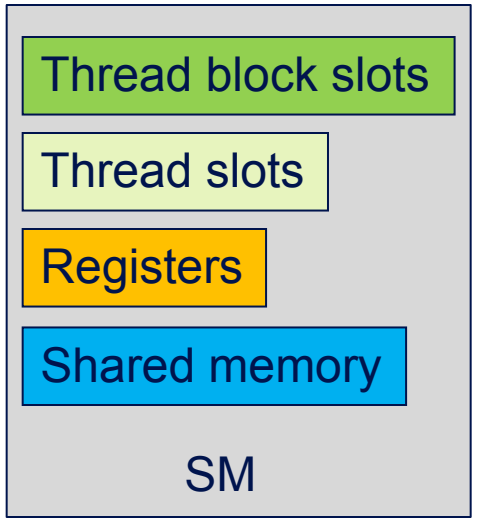
- Recall a SM dynamically partitions resources:



G80 Limits	Fermi Limits	Kepler Limits
8	8	16
768	1536	2048
32KB	32 KB	64KB
16KB	48KB	48KB

SM Resource Partitioning

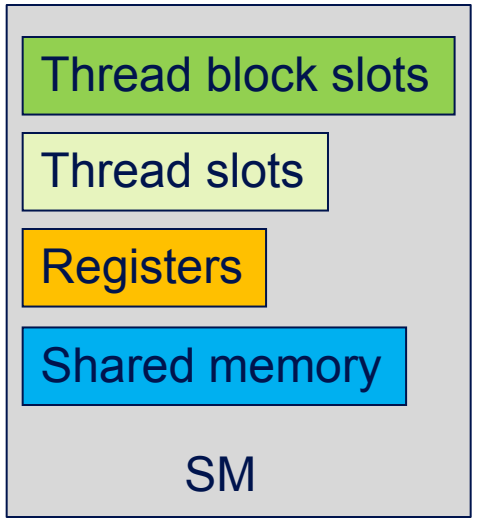
- We can have
  - 8 blocks of 96 threads
  - 4 blocks of 192 threads
  - But not 8 blocks of 192 threads



G80 Limits
8
768
32KB (8K registers)
16k

SM Resource Partitioning

- We can have (assuming 256 thread blocks)
  - 768 threads (3 blocks) using 10 registers each
  - 512 threads (2 blocks) using 15 registers each

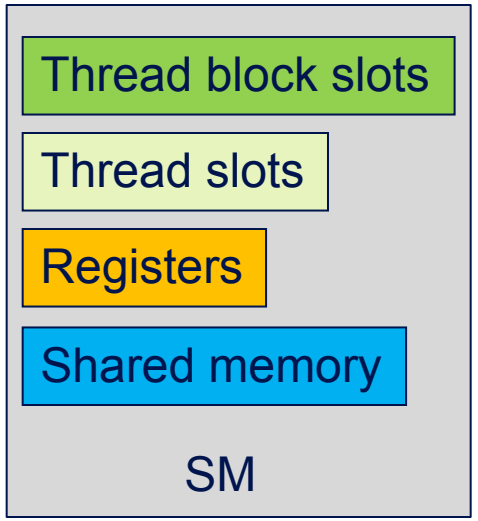


G80 Limits
8
768
32KB (8K registers)
16k

SM Resource Partitioning

- We can have (assuming 256 thread blocks)
  - 768 threads (3 blocks) using 10 registers each
  - 512 threads (2 blocks) using 15 registers each

- More registers decreases thread-level parallelism
  - Can it ever increase performance?



G80 Limits
8
768
32KB (8K registers)
16k

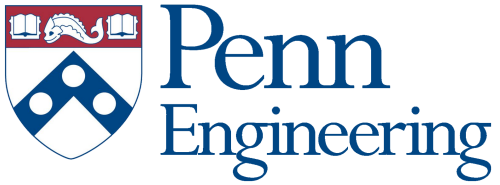
SM Resource Partitioning

- **Performance Cliff**: Increasing resource usage leads to a dramatic reduction in parallelism
  - For example, increasing the number of registers, unless doing so hides latency of global memory access
- **Occupancy**: The ratio of active warps and maximum possible warps.
  - Low occupancy = less latency hiding
  - High occupancy = better performance? (not always)

SM Resource Partitioning

- GPU programmers are always trying to balance between occupancy and resources used
- Useful CUDA functions to help you determine the block size given resources used
  - cudaOccupancyMaxActiveBlocksPerMultiprocessor
  - cudaOccupancyMaxPotentialBlockSize
  - cudaOccupancyMaxPotentialBlockSizeVariableSMem

Data Prefetching



Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Execute instruction that are not Dependent on memory read

Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Use global memory after the above line
from enough warps hide the memory latency

Data Prefetching

- **Prefetching** data from global memory can effectively increase the number of independent instructions between global memory read and use

Data Prefetching

- Recall tiled matrix multiply:

```
for (/* ... */)
{
  // Load current tile into shared memory
  __syncthreads();
  // Accumulate dot product
  __syncthreads();
}
```

Data Prefetching

- Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

Data Prefetching

- Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

Data Prefetching

- Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

Prefetch for next iteration of the loop

Data Prefetching

- Tiled matrix multiply with prefetch:

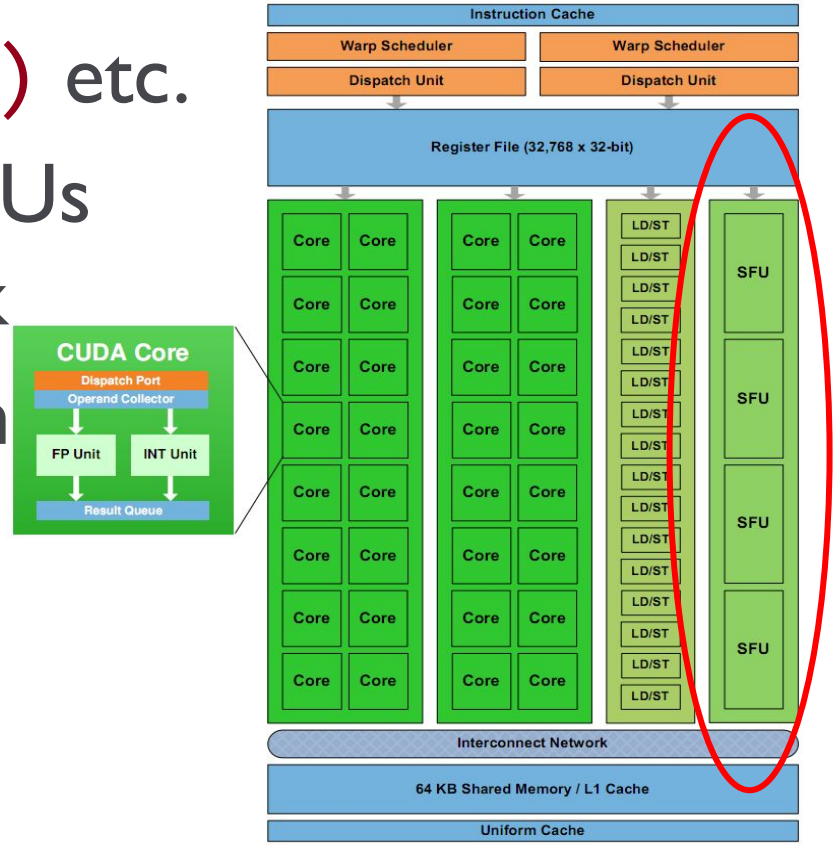
```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

These instructions executed by enough threads will hide the memory latency of the prefetch

Instruction Mix

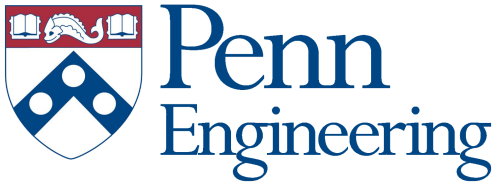
- Special Function Units (SFUs)
  - Single-precision floating-point
  - Use to compute `__sinf()`, `__expf()` etc.
  - Modern GPUs have either 16 or 32 SFUs each running at 1 instruction per clock
  - **Use when speed trumps precision**



Fermi Streaming Multiprocessor (SM)

Image: NVIDIA Fermi Whitepaper

Loop Unrolling



Loop Unrolling

- Instructions per iteration
  - One floating-point multiply
  - One floating-point add
  - What else?

```
for (int k = 0; k < BLOCK_SIZE; k++)
{
  Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Loop Unrolling

- Other instructions per iteration
  - Update loop counter

```
for (int k = 0; k < BLOCK_SIZE; k++)
{
  Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Loop Unrolling

- Other instructions per iteration
  - Update loop counter
  - Branch

```
for (int k = 0; k < BLOCK_SIZE; k++)
{
  Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Loop Unrolling

- Other instructions per iteration
  - Update loop counter
  - Branch
  - Address arithmetic

```
for (int k = 0; k < BLOCK_SIZE; k++)
{
  Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

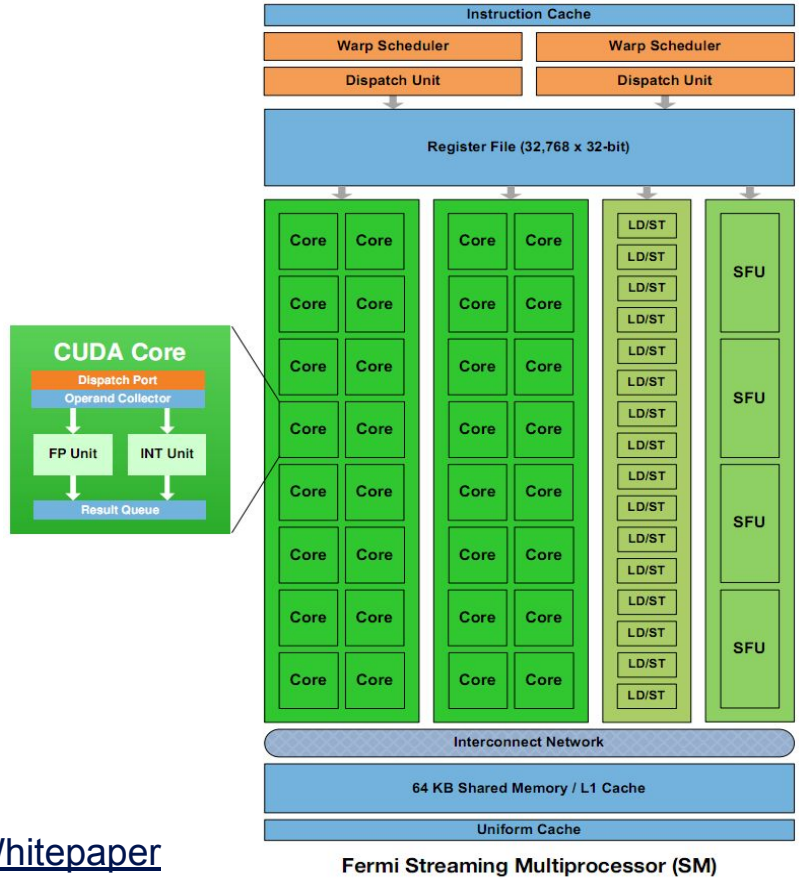
Loop Unrolling

- Instruction Mix
  - 2 floating-point arithmetic instructions
  - 1 loop branch instruction
  - 2 address arithmetic instructions
  - 1 loop counter increment instruction

```
for (int k = 0; k < BLOCK_SIZE; k++)
{
  Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Instruction Mix

- Only 1/3 are floating-point calculations
  - But want full theoretical TFLOPs
  - Consider **loop unrolling**



Fermi Streaming Multiprocessor (SM)

Image: NVIDIA Fermi Whitepaper

Loop Unrolling

- No more loop
  - No loop count update
  - No branch
  - Constant indices – no address arithmetic instructions

```
Pvalue +=
  Ms[ty][0] * Ns[0][tx] +
  Ms[ty][1] * Ns[1][tx] +
  ...
  Ms[ty][15] * Ns[15][tx]; // BLOCK_SIZE = 16
```

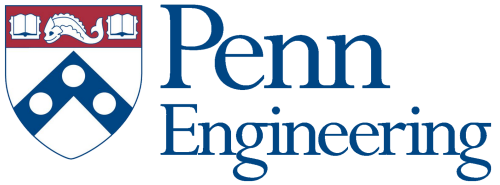
Loop Unrolling

- Automatically:

```
#pragma unroll BLOCK_SIZE
for (int k = 0; k < BLOCK_SIZE; k++)
{
  Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Disadvantages to unrolling?

Thread Granularity

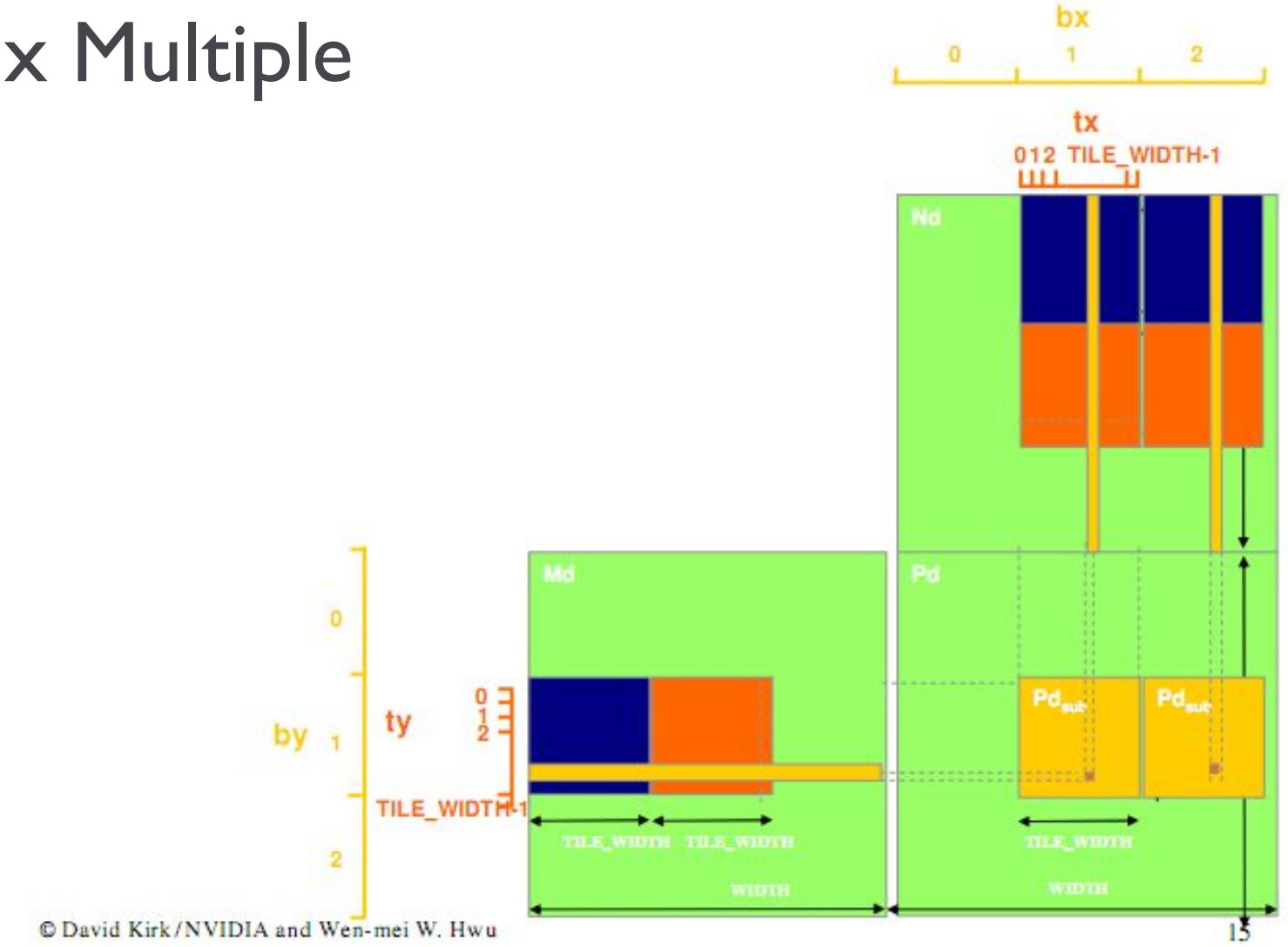


Thread Granularity

- How much work should one thread do?
  - Parallel Reduction
    - Reduce two elements?
  - Matrix multiply
    - Compute one element of Pd?

Thread Granularity

- Matrix Multiple

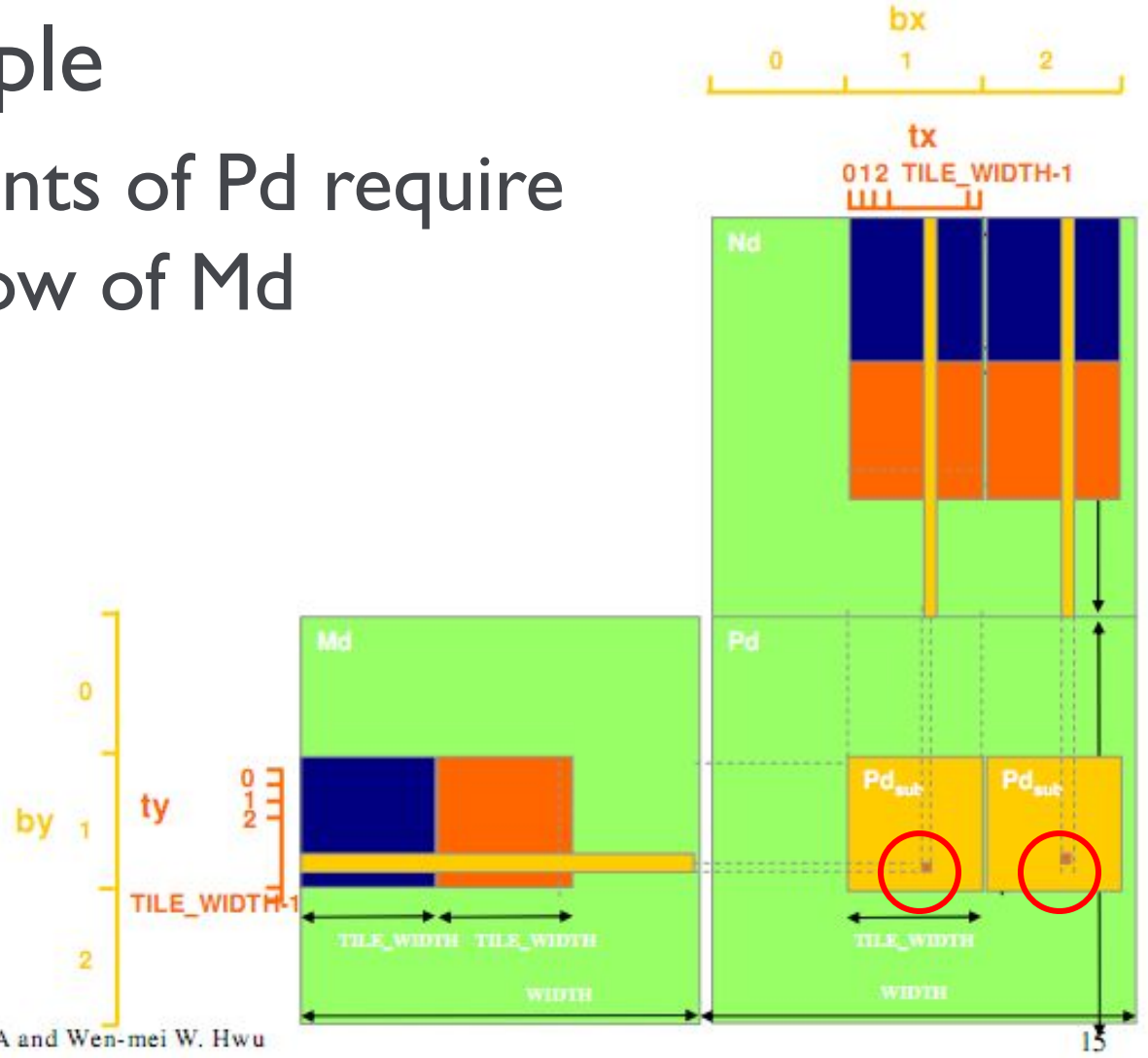


Penn Engineering

Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Thread Granularity

- Matrix Multiple
  - Both elements of Pd require the same row of Md



Penn Engineering

Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Thread Granularity

- Matrix Multiple
  - Compute both Pd elements in the same thread
    - Reduces global memory access by ¼
    - Increases number of independent instructions
      - What is the benefit?
    - New kernel uses more registers and shared memory
      - What does that imply?

Summary

- Most of GPU Programming is about optimization
- Maximize utilization, memory throughput and instruction throughput
- Need to find the balance between
  - The amount of compute per thread
  - The resources used per thread