《计算机系统结构》课程直播

2020. 5.19

听不到声音请及时调试声音设备，可以下课后补签到

请将ZOOM名称改为“姓名”；

本节内容

- 数据级并行性
  - GPU存储模型
  - GPU的性能
- 线程级并行性
  - 高速缓存一致性

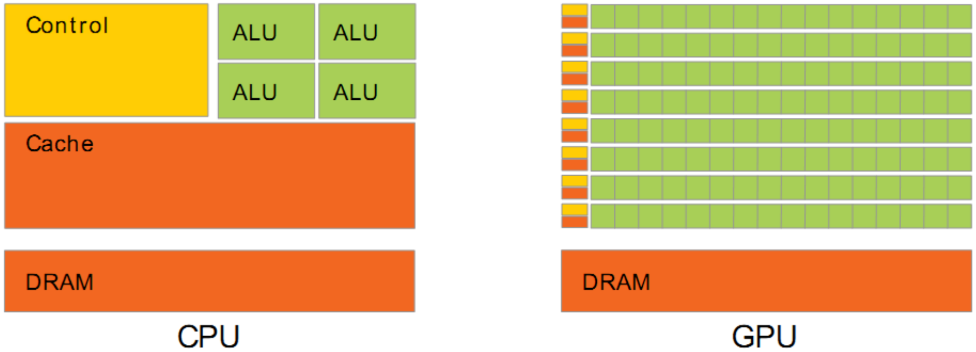
From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, （5th edition)

Graphics Processing Units

GPU

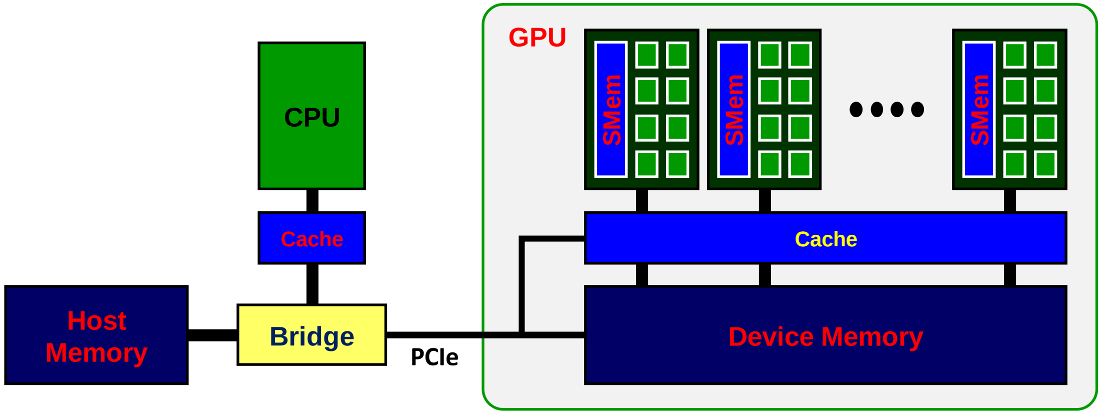
Using CPU+GPU Architecture

- CPU+GPU异构多核系统
  - 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，访问存储器延时低
- GPU 适合执行大量并行线程
  - 可扩放的并行执行
  - 高带宽的并行存取



强控制、弱计算

弱控制、强计算



Heterogeneous Computing

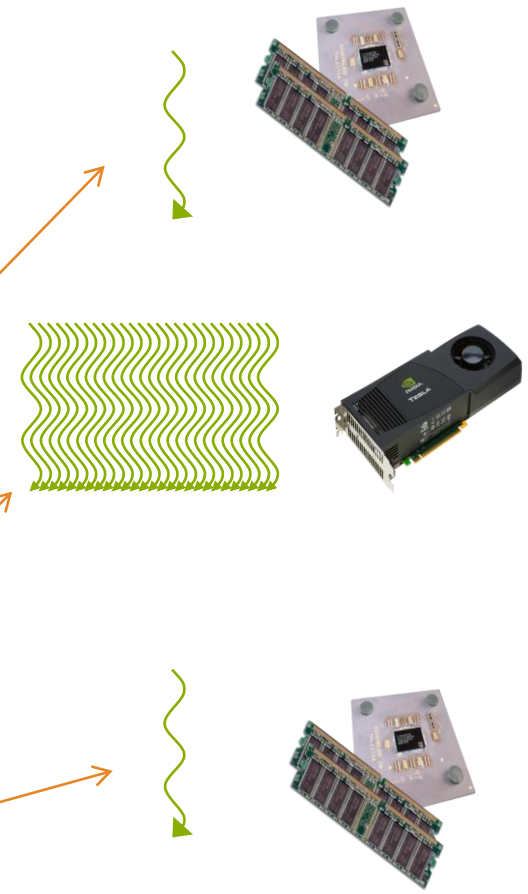
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N       1024
#define RADIUS    3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in, size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

serial code

parallel code

serial code



GPU: a multithreaded coprocessor

SP: scalar processor
'CUDA core'

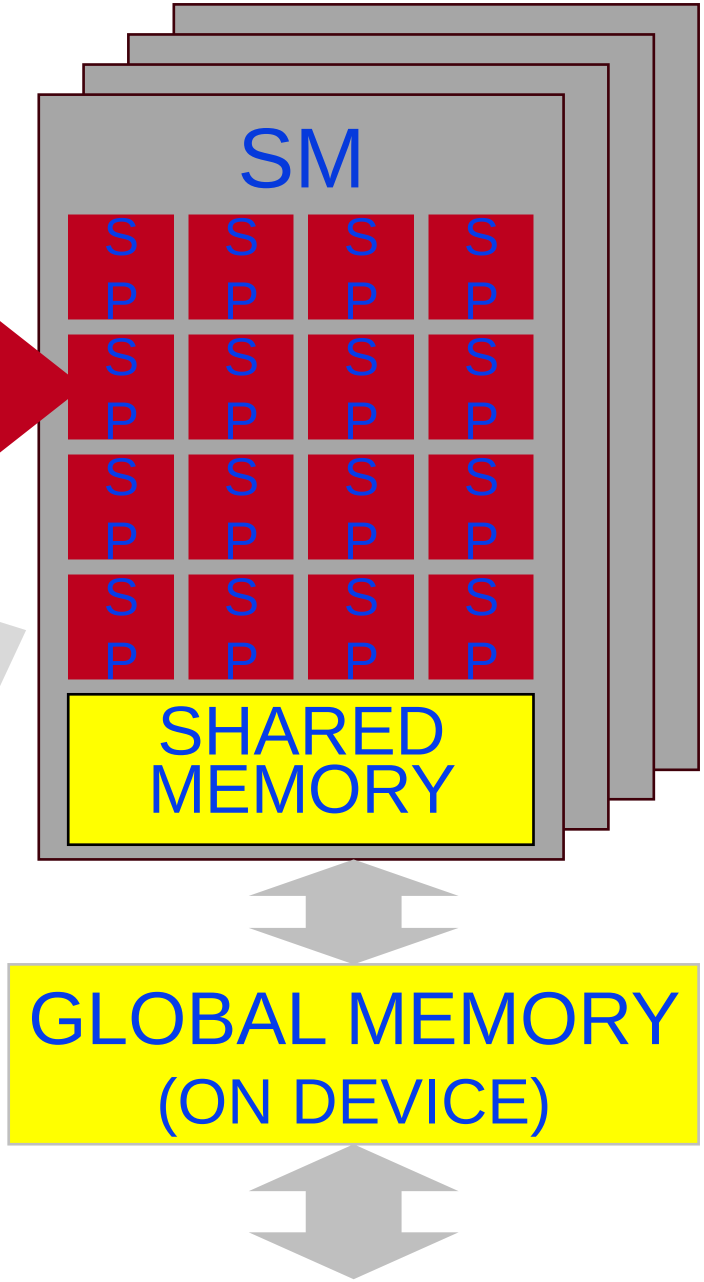
Executes one thread

SM

streaming multiprocessor

32xSP (or 16, 48 or more)

Fast local '**shared memory**'
(shared between SPs)
16 KiB (or 64 KiB)



总结：GPU的三个key idea

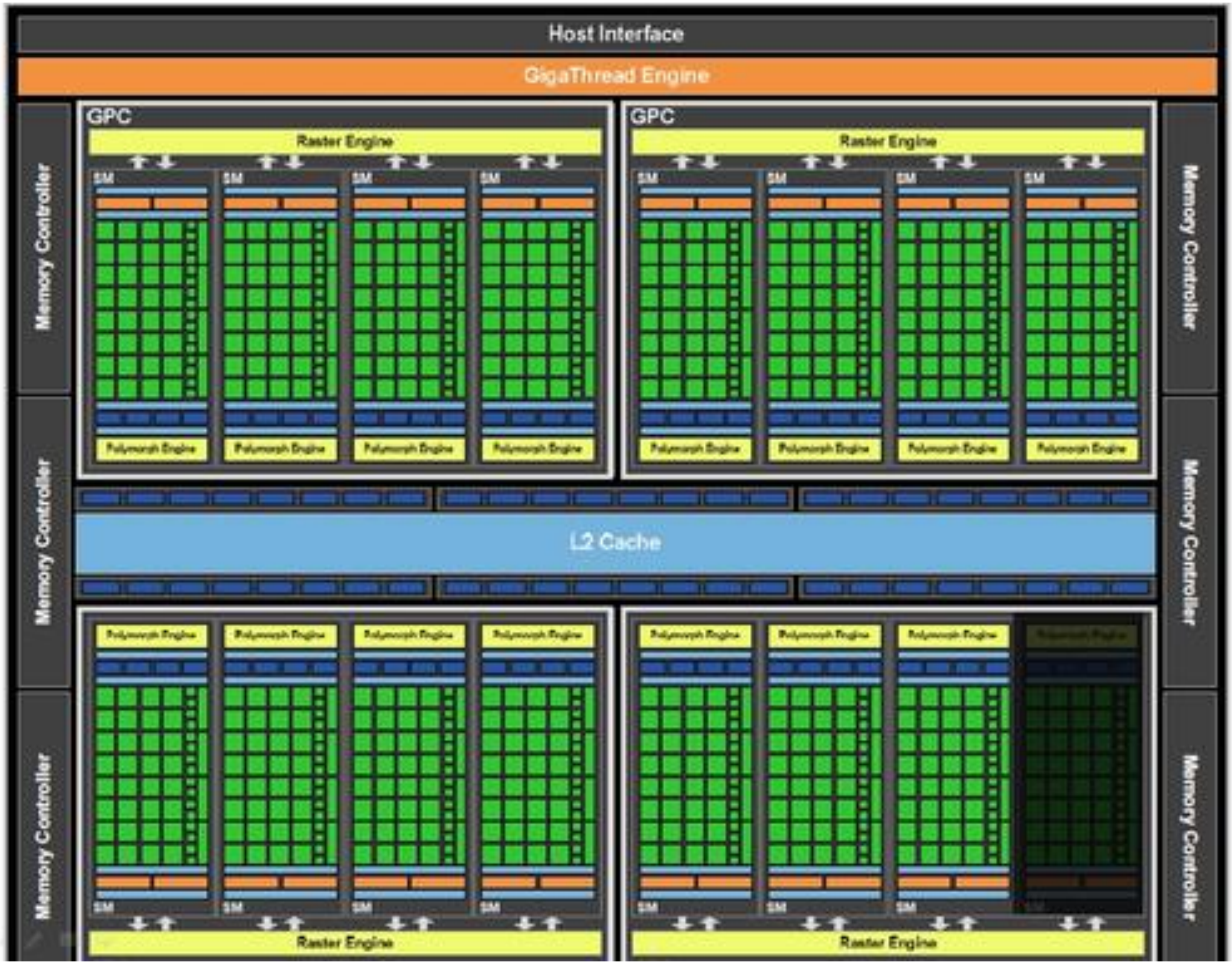
- Employ multiple processing cores
  - 简单的核 (比起ILP， 更注重TLP：thread-level parallelism)
- Pack cores full of ALU
  - 开发数据级的并行性 (SIMD)
  - 一条指令执行时，多个ALU同步处理不同的数据
  - 芯片面积额外增加的开销不大，但大大增强了计算能力
- 利用多线程提高系统的处理能力（吞吐量）
  - 轮流交替执行不同线程的代码段以隐藏延迟

Nvidia 通用图形加速单元体系结构

- 2008 Tesla
- 2010 Fermi
- 2012 Kepler
- 2014 Maxwell
- 2016 Pascal

- 每个SM包含的SP（GPU core）数量依据GPU架构而不同，Fermi架构GF100是32个，GF10X是48个，Kepler架构都是192个，Maxwell都是128个，Pascal64个。

Floorplan of Fermi GTX480

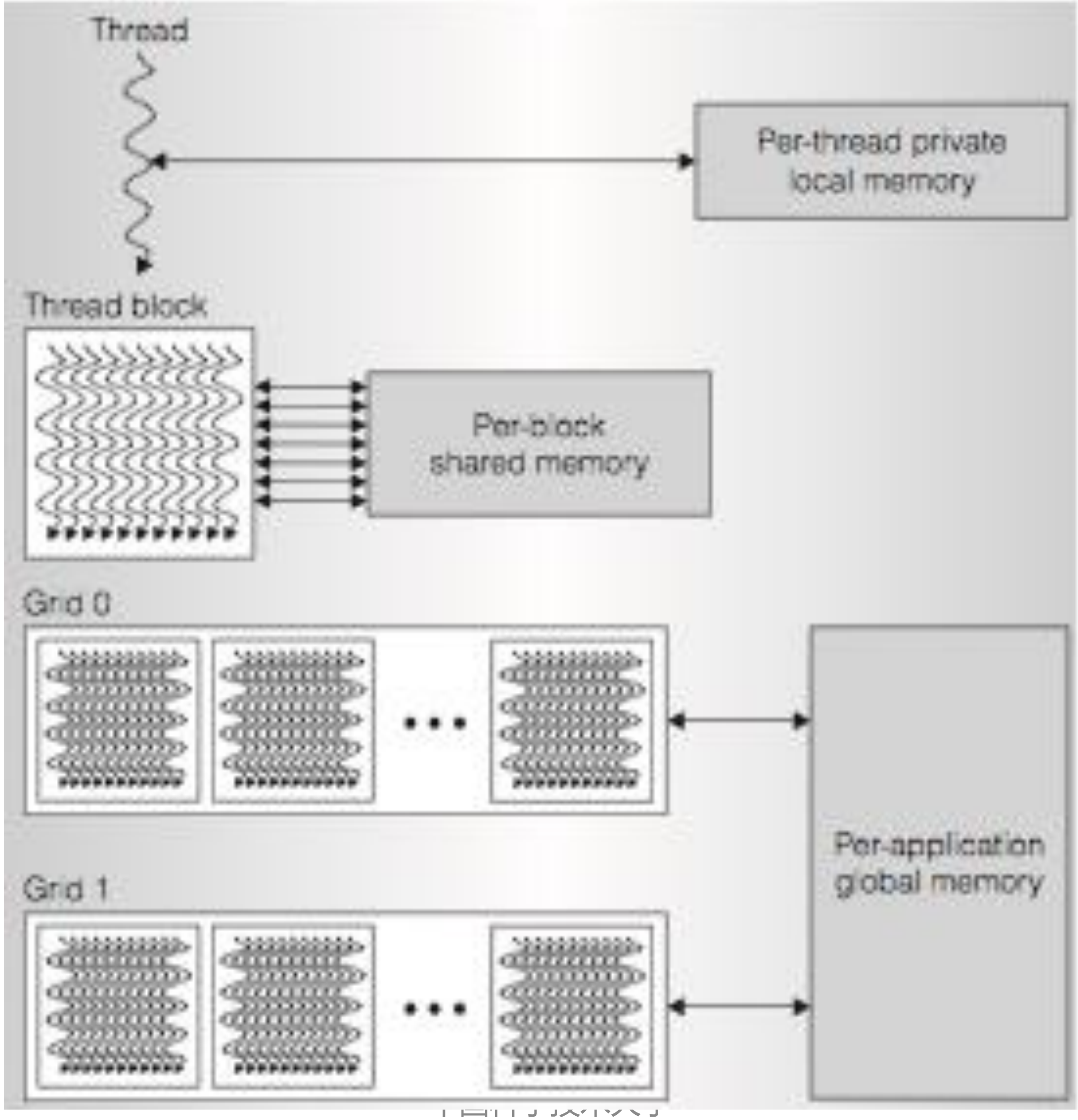


Pascal 架构(Tesla P100 2016)

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs[1]	5040	6840	10600
Peak FP64 GFLOPs[1]	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2
Manufacturing Process	28 nm	28 nm	16 nm FinFET

MEMORY MODEL OF CUDA AND COOPERATING THREADS

GPU Memory Hierarchy



[Nvidia, 2010]

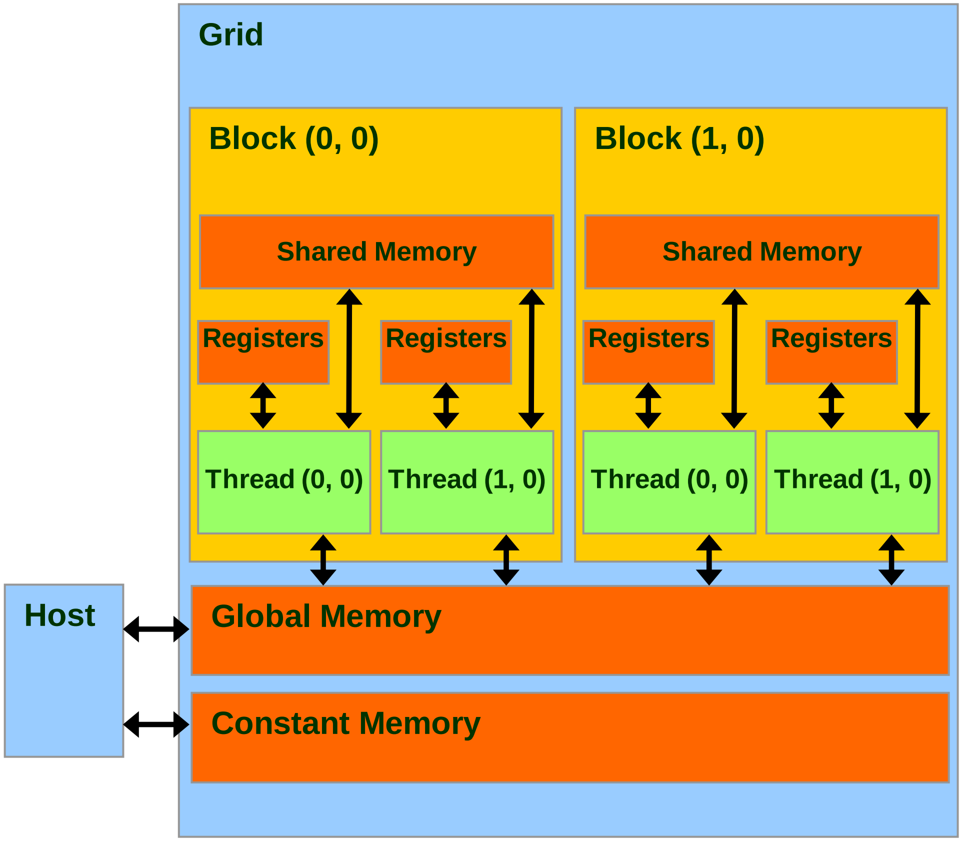
CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
`__device__ __local__`	`int LocalVar;`	local	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

G80 Implementation of CUDA Memories

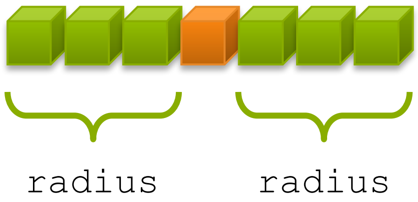
- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread local memory
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**



1D Stencil

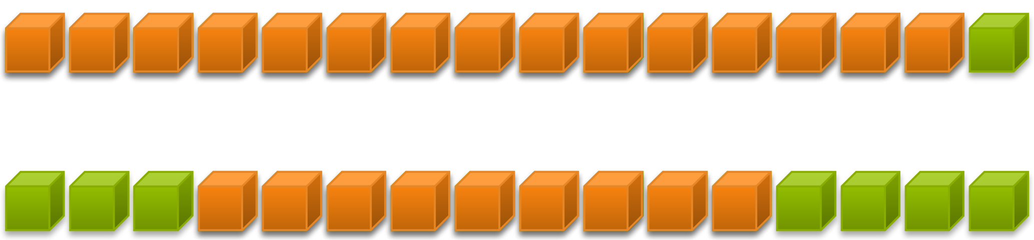
- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times



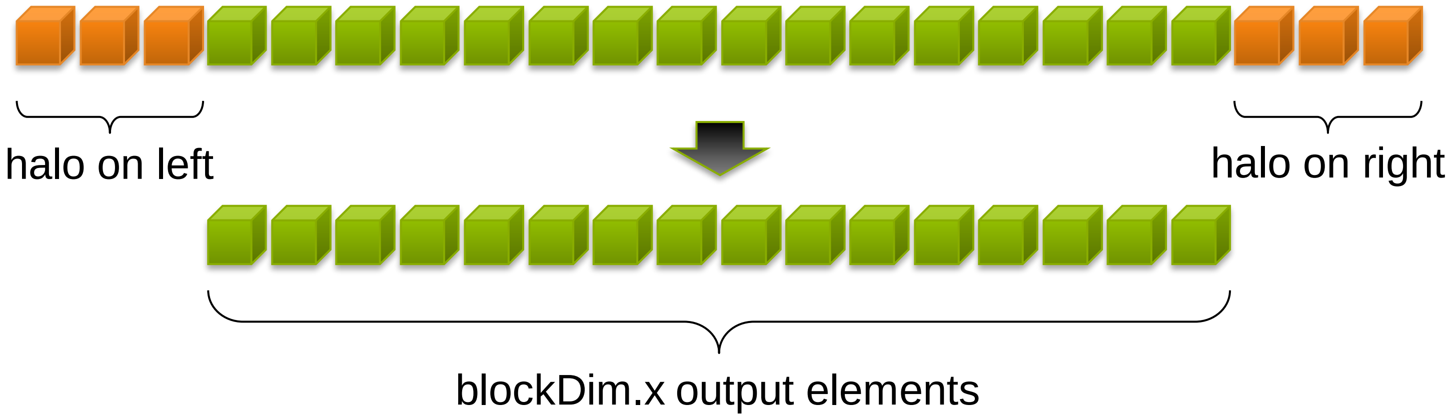
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

  - Each block needs a halo of radius elements at each boundary



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```

Stencil Kernel

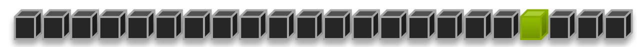
```
  // Apply the stencil
      result = 0;
  for (    offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```

Data Race!

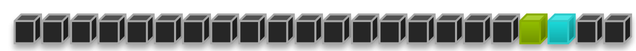
- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18] 

Skipped, threadIdx > RADIUS

Load from temp[19] 

__syncthreads()

- `__syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
                  int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
      synchthreads();
```

Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Review (1 of 2)

- Launching parallel threads
  - Launch N blocks with M threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

CUDA PERFORMANCE

Efficient data-parallel algorithms

+

Optimizations based on GPU Architecture

=

Maximum Performance

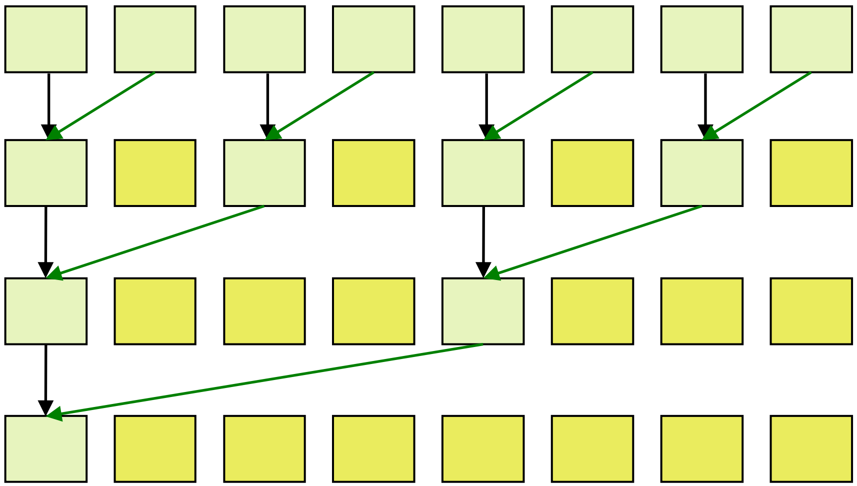
Parallel Reduction

- Recall *Parallel Reduction* (sum)

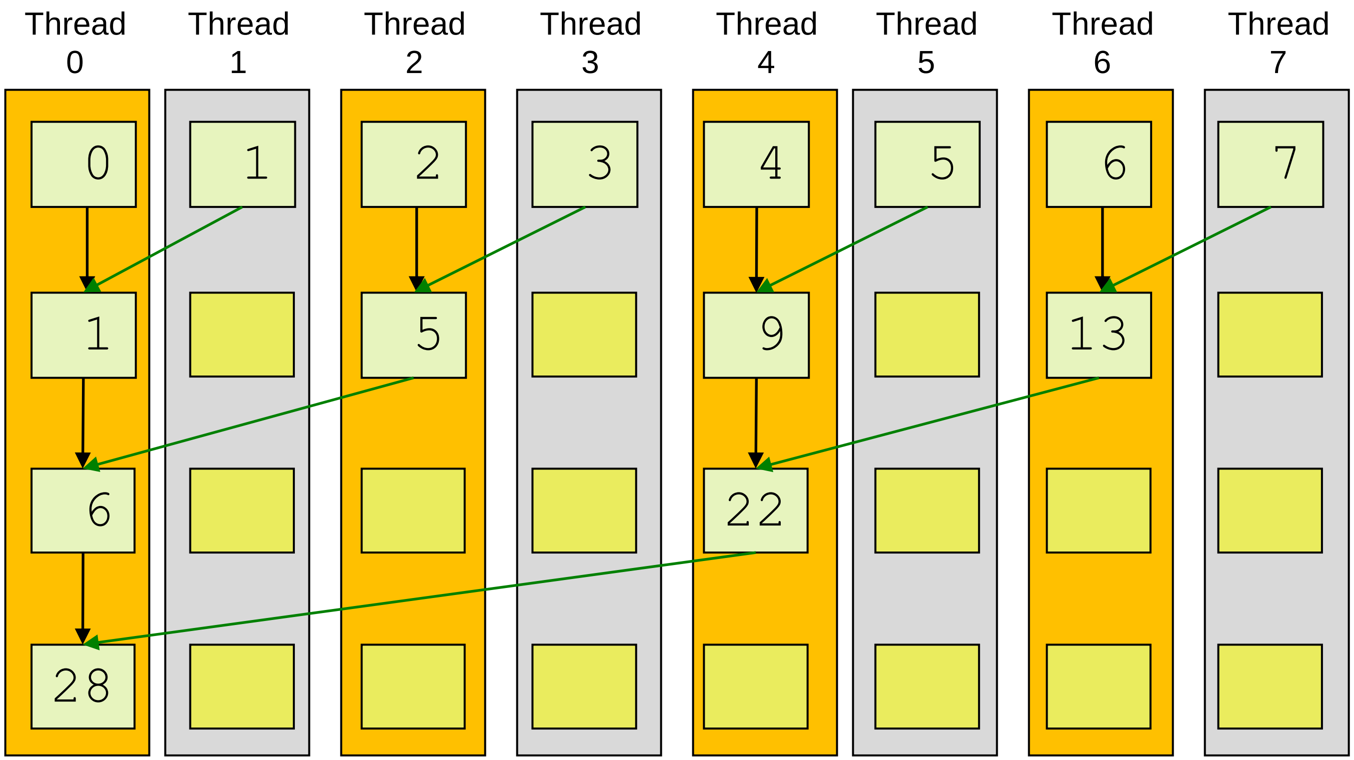
0	1	2	3	4	5	6	7

Parallel Reduction

- log(n) passes for n elements
- How would you implement this in CUDA?

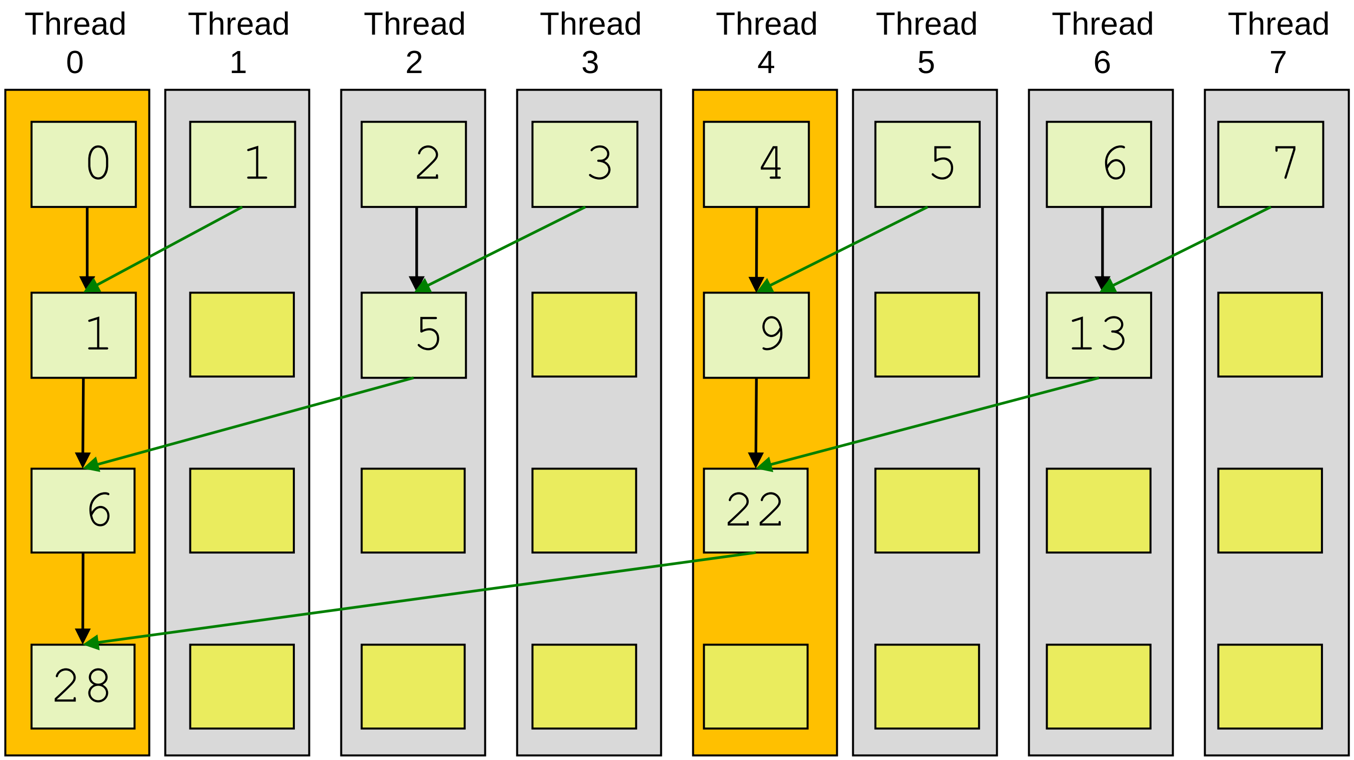


Parallel Reduction



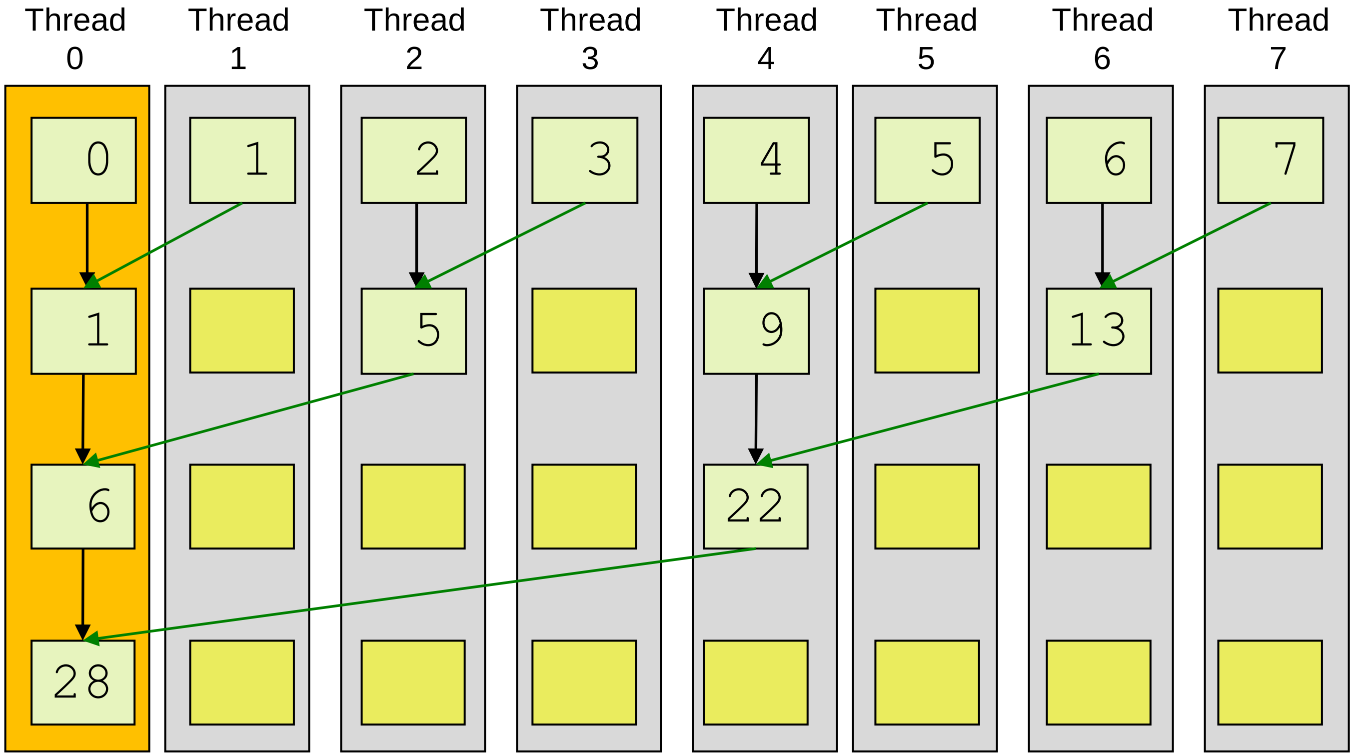
- 1st pass: threads 1, 3, 5, and 7 don't do anything
  - Really only need `n/2` threads for `n` elements

Parallel Reduction



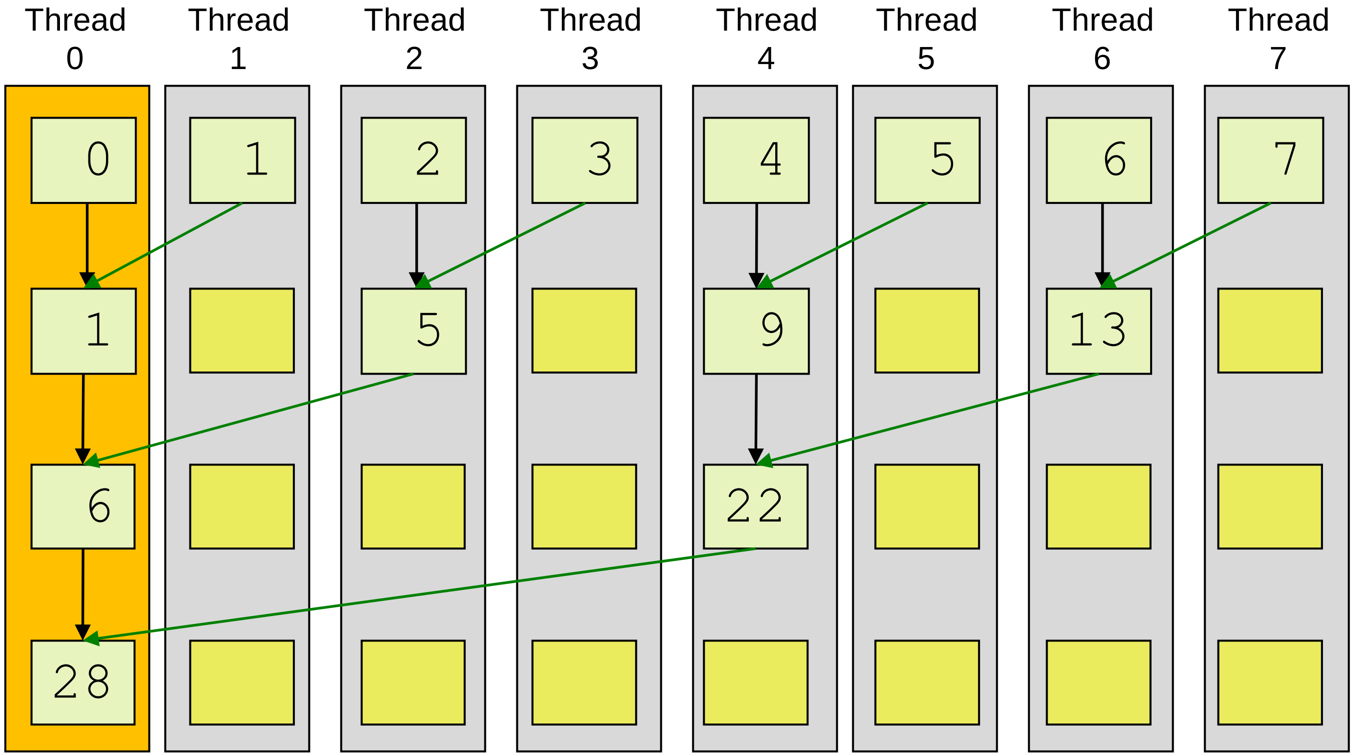
- 2nd pass: threads 2 and 6 also don’t do anything

Parallel Reduction



- 3rd pass: thread 4 also doesn't do anything

Parallel Reduction

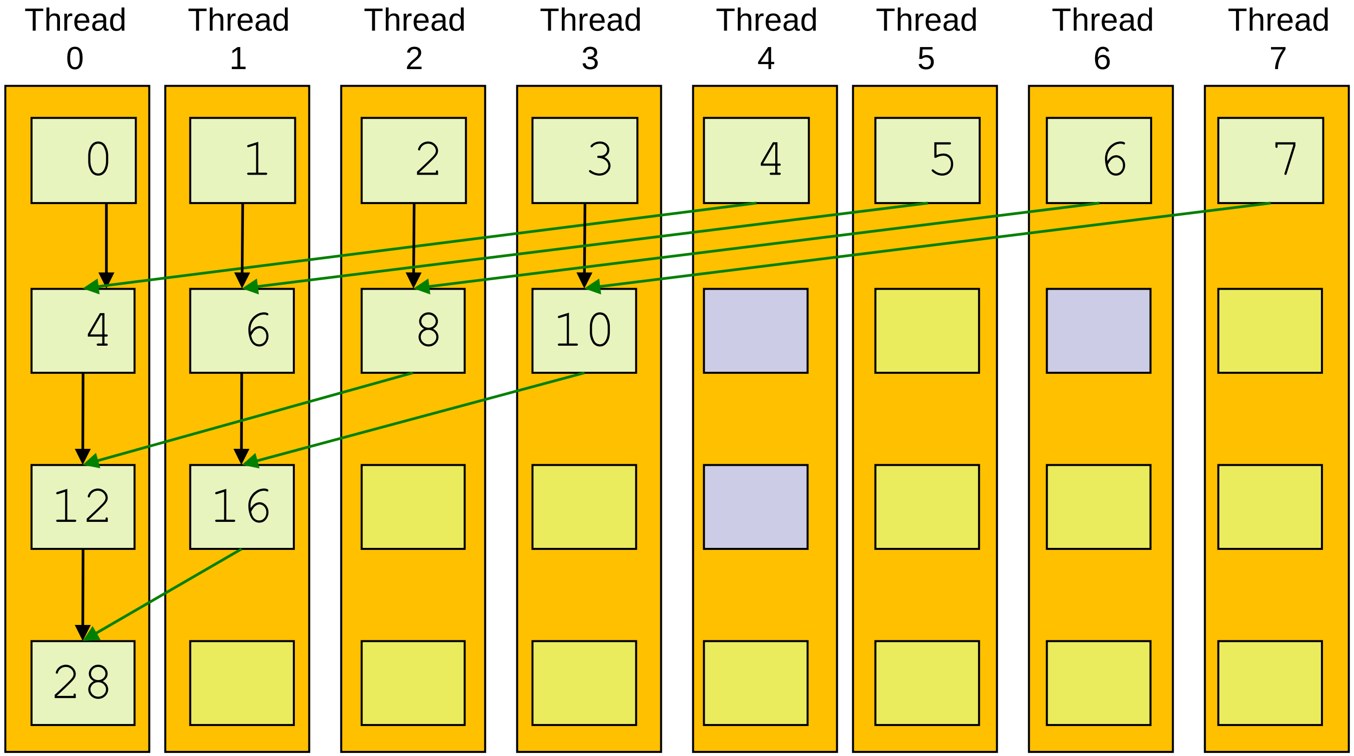


- In general, number of required threads cuts in half after each pass

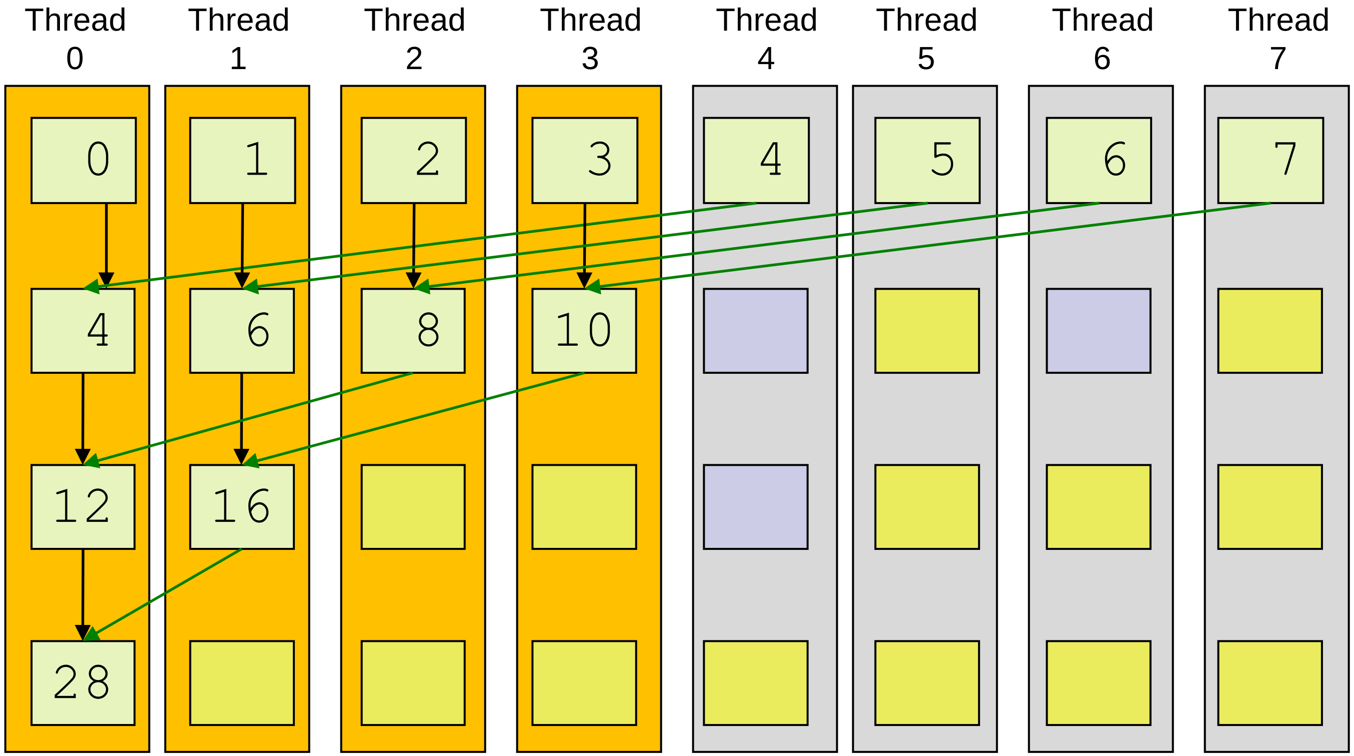
Parallel Reduction

- What if we *tweaked* the implementation?

Parallel Reduction

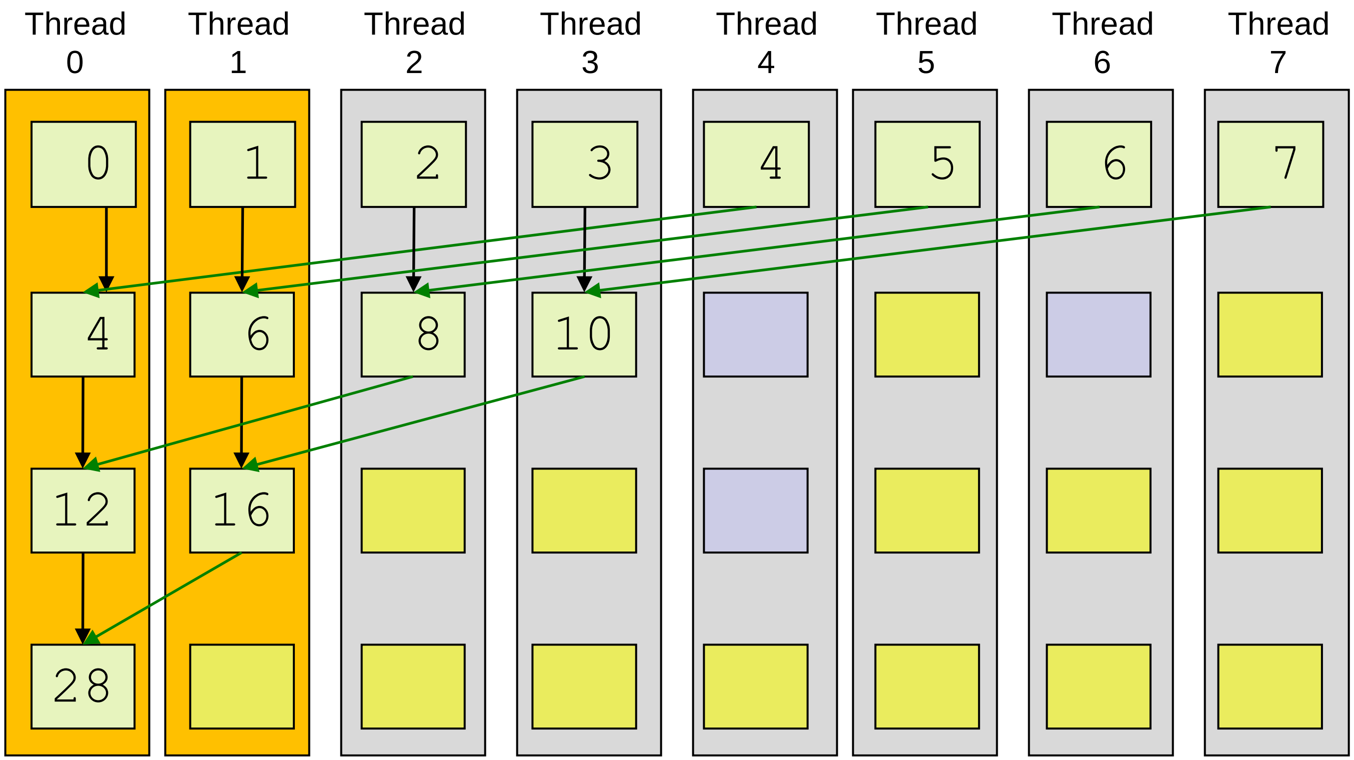


Parallel Reduction



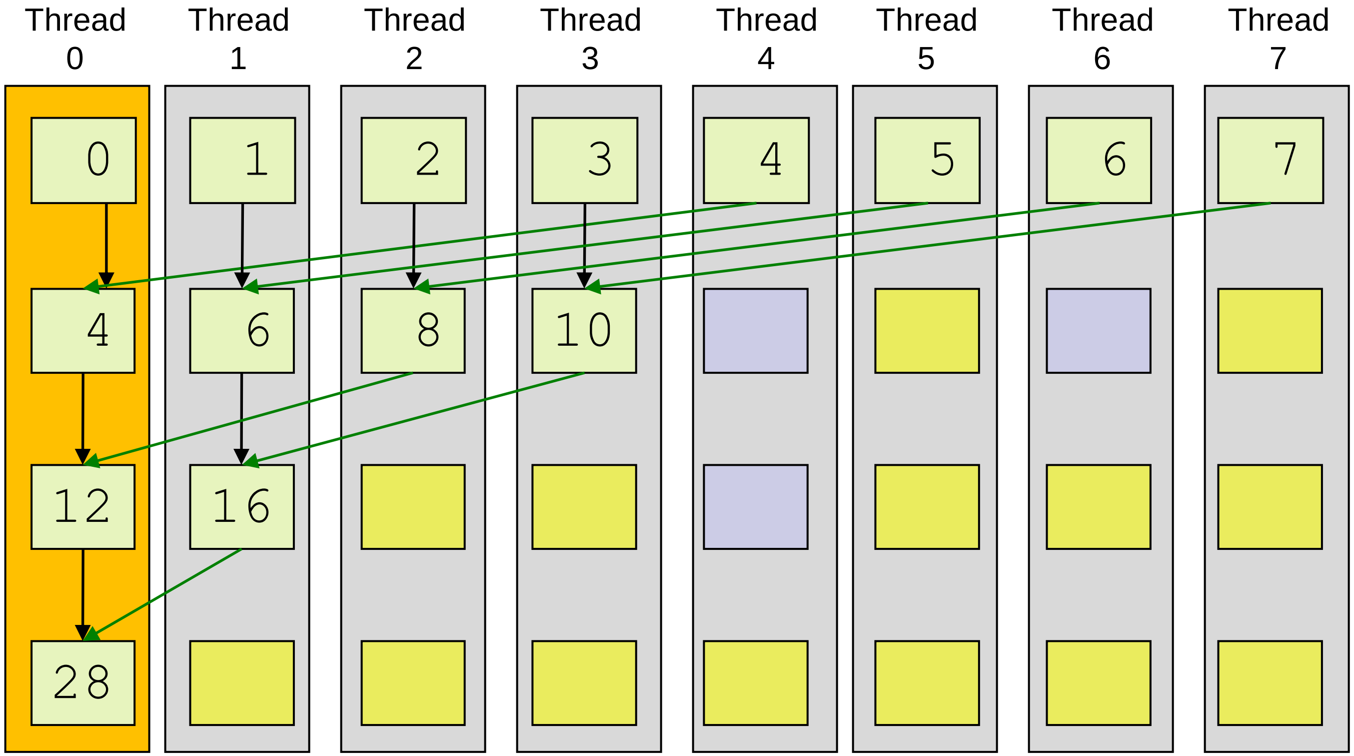
- 1st pass: threads 4, 5, 6, and 7 don't do anything
  - Really only need `n/2` threads for `n` elements

Parallel Reduction



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

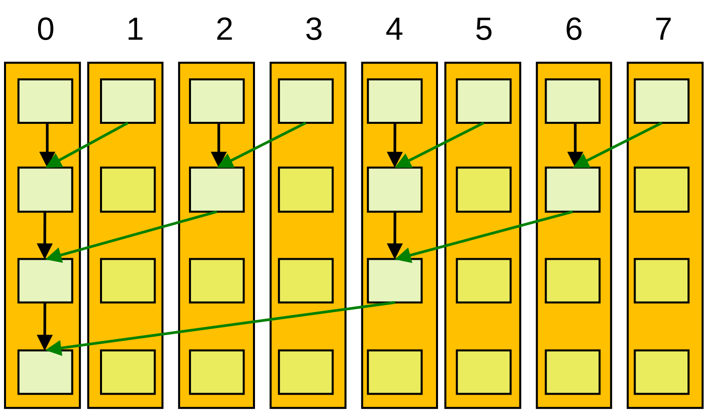
Parallel Reduction



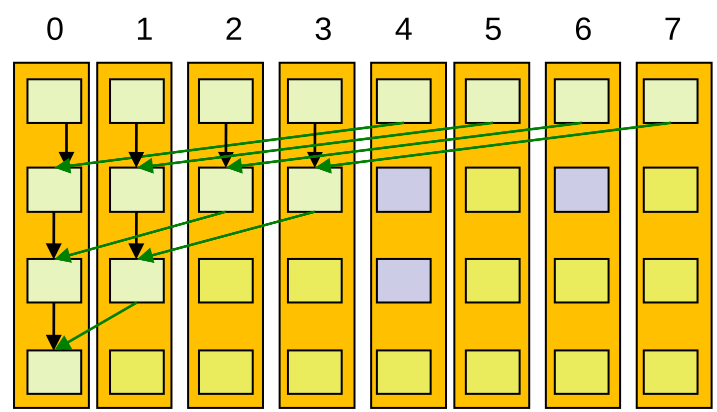
- 3rd pass: thread 1 also doesn't do anything

Parallel Reduction

- What is 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];
```

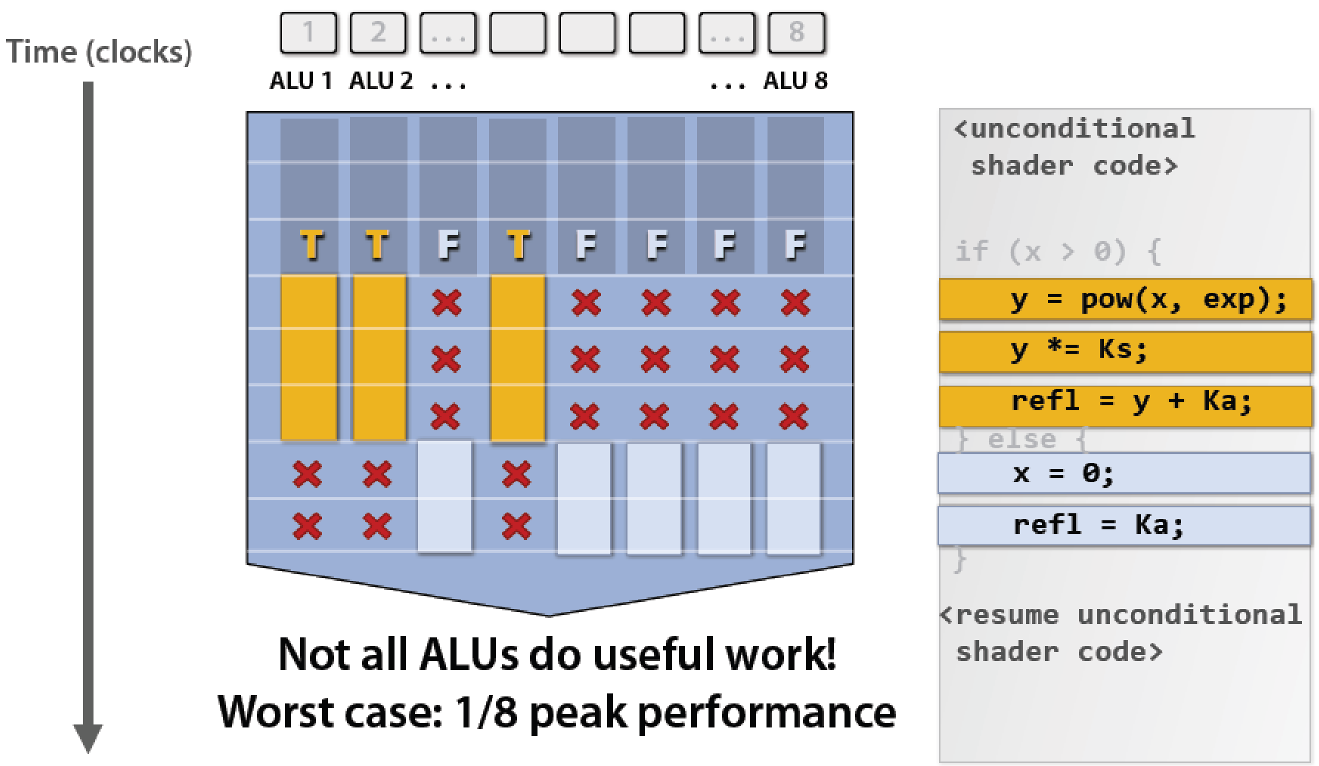
`stride` = 4, 2, 1, …

Warp Partitioning

- *Warp Partitioning*: how threads from a block are divided into warps
- Knowledge of warp partitioning can be used to:
  - Minimize divergent branches
  - Retire warps early

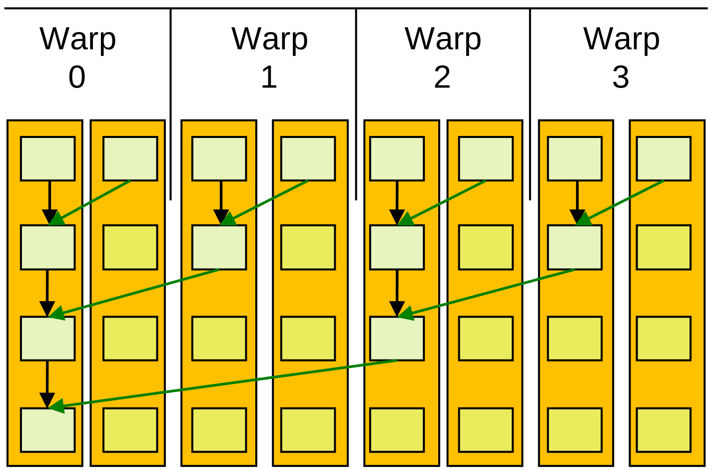
Warp Partitioning

Divergent branches are within a warp!

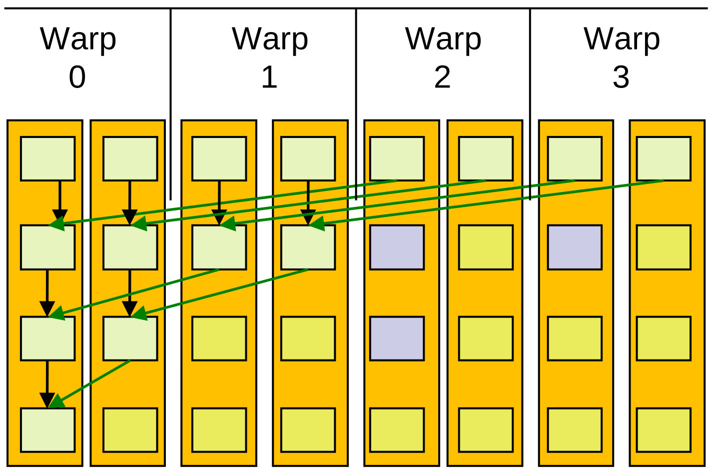


Warp Partitioning

- Pretend `warpSize == 2`



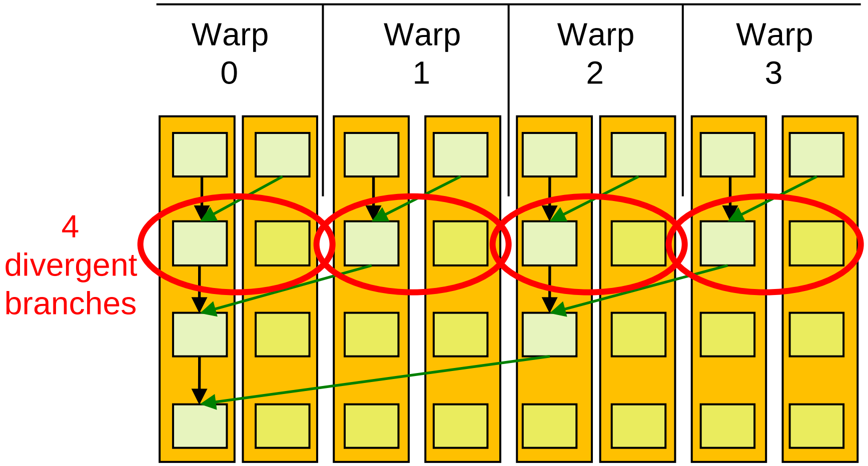
`stride` = 1, 2, 4, …



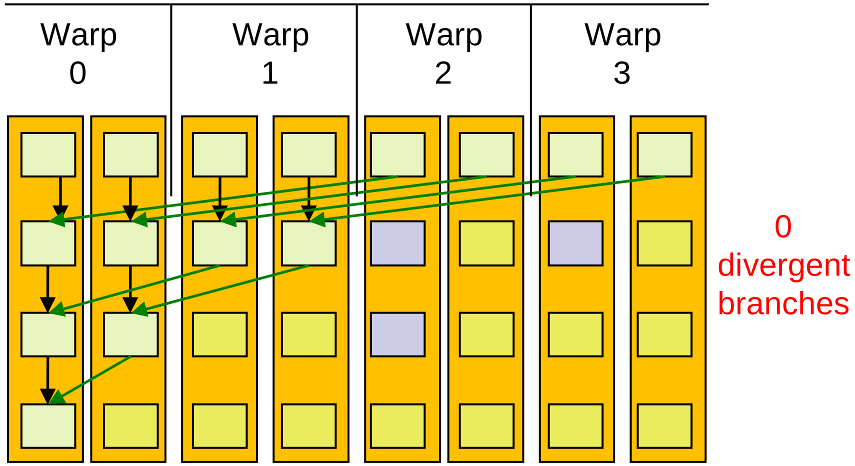
`stride` = 4, 2, 1, …

Warp Partitioning

- 1^{st} Pass



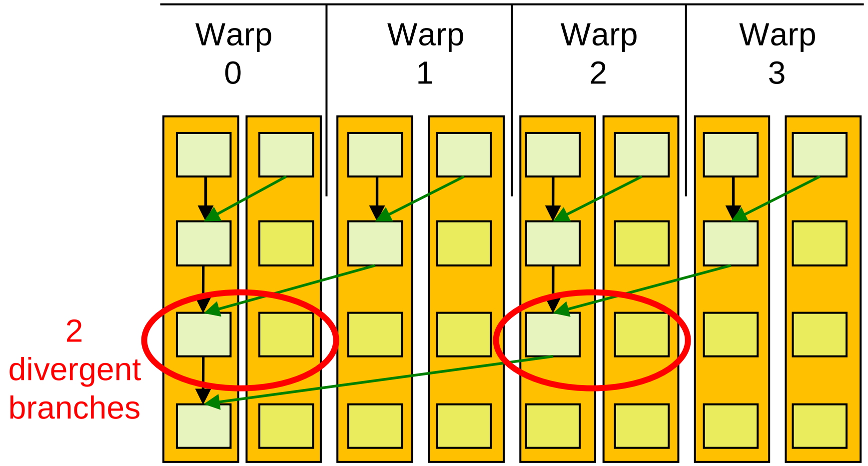
`stride` = 1, 2, 4, …



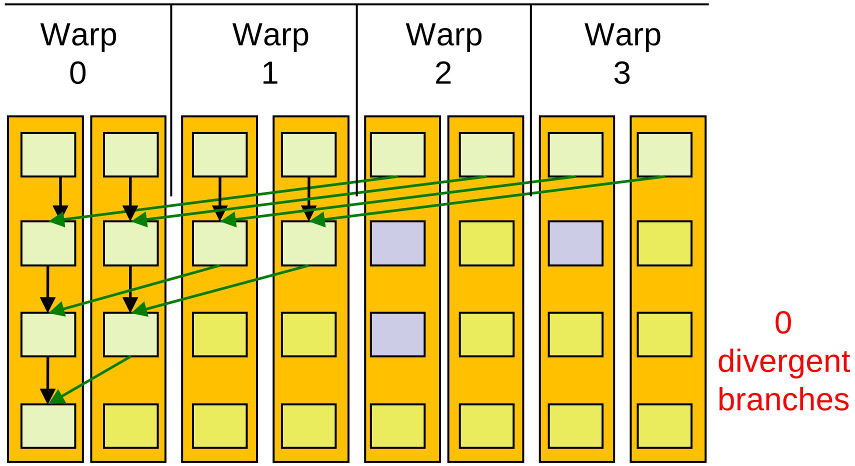
`stride` = 4, 2, 1, …

Warp Partitioning

- 2nd Pass



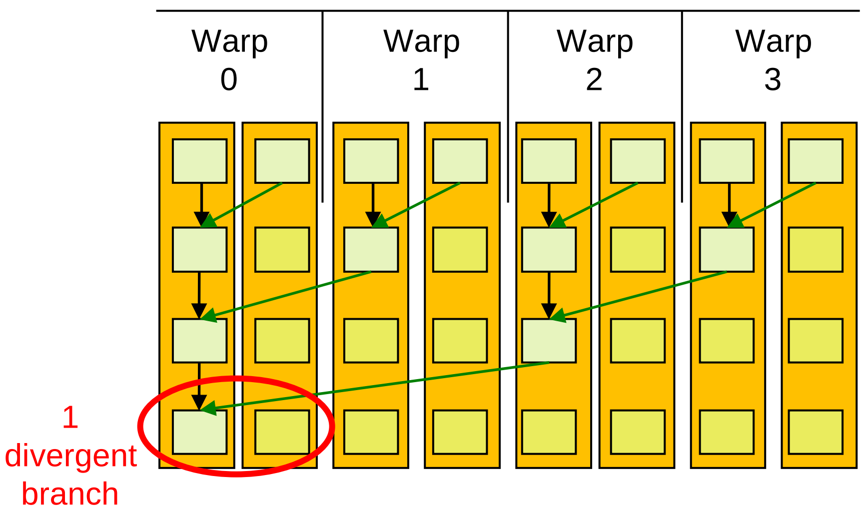
`stride` = 1, 2, 4, …



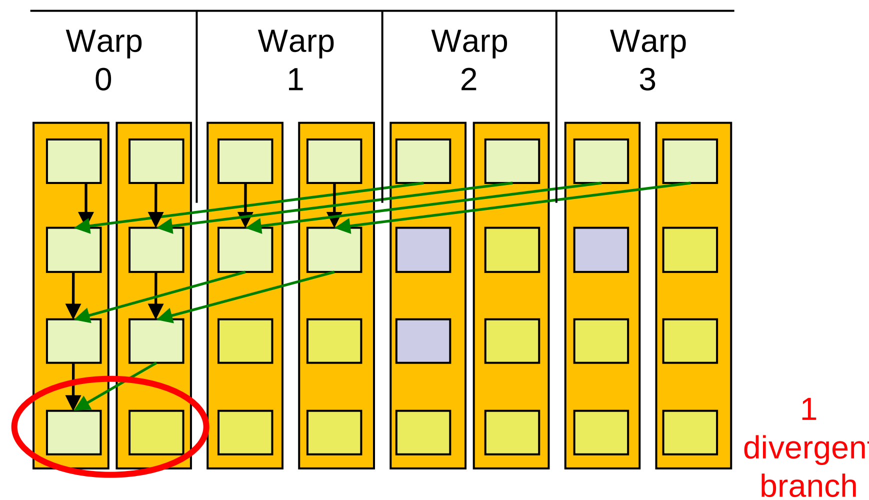
`stride` = 4, 2, 1, …

Warp Partitioning

- 2nd Pass



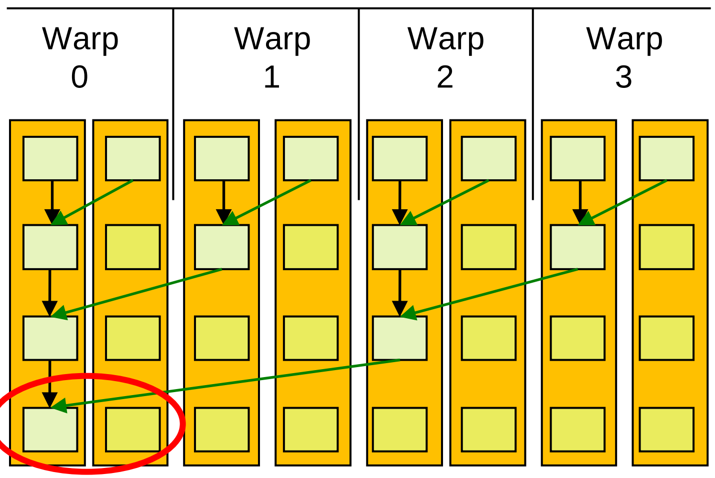
`stride` = 1, 2, 4, …



`stride` = 4, 2, 1, …

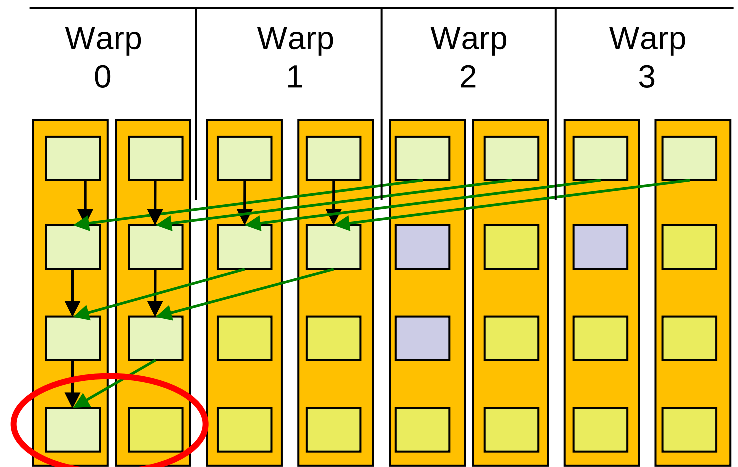
Warp Partitioning

- 2nd Pass



1 divergent branch

`stride` = 1, 2, 4, …



1 divergent branch

`stride` = 4, 2, 1, …

Still diverge when number of elements

Warp Partitioning

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

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

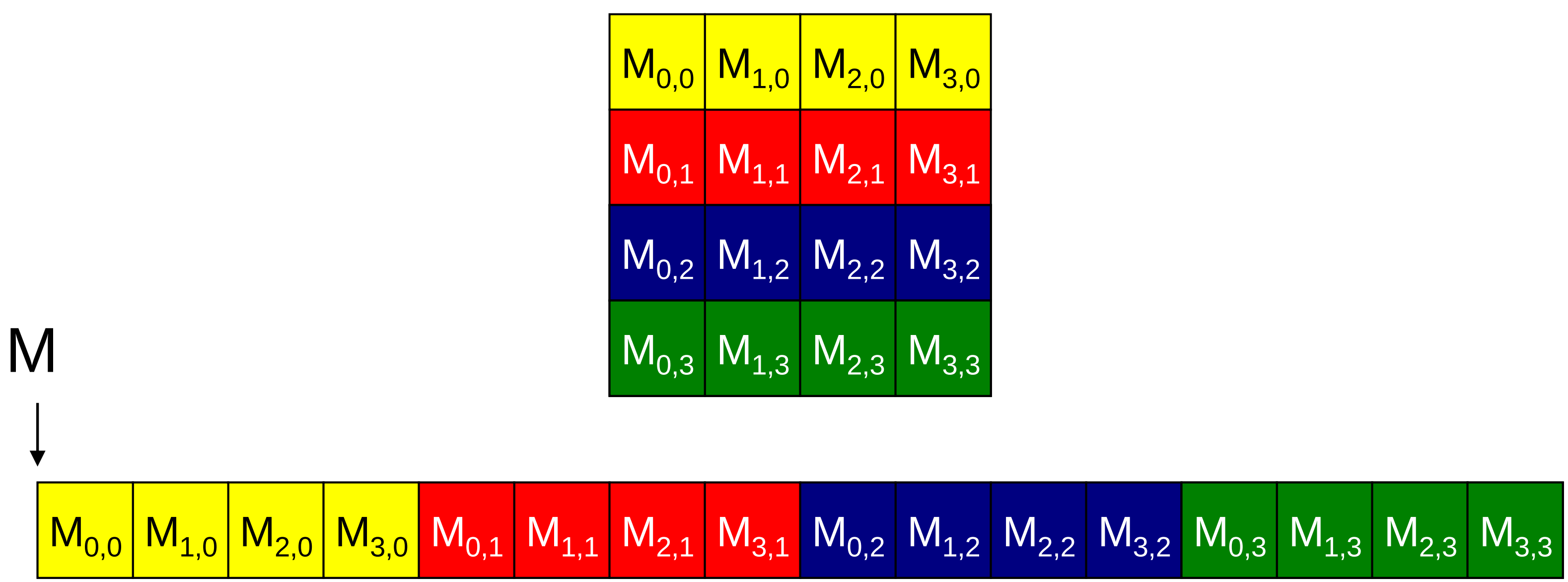
`stride` = 1, 2, 4, …

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

`stride` = 4, 2, 1, …

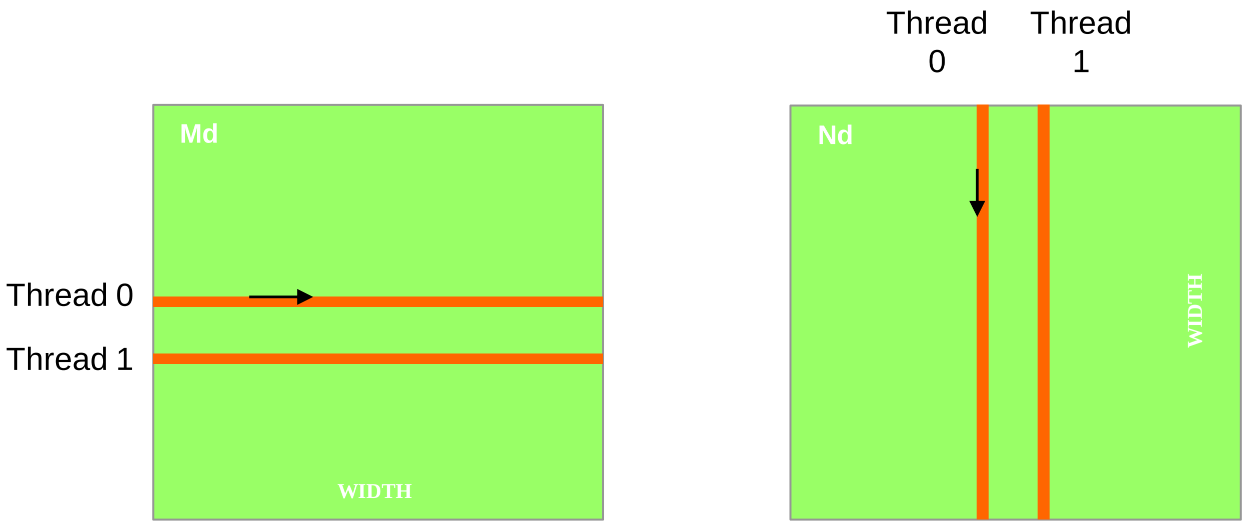
Memory Coalescing（合并访存）

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



Memory Coalescing

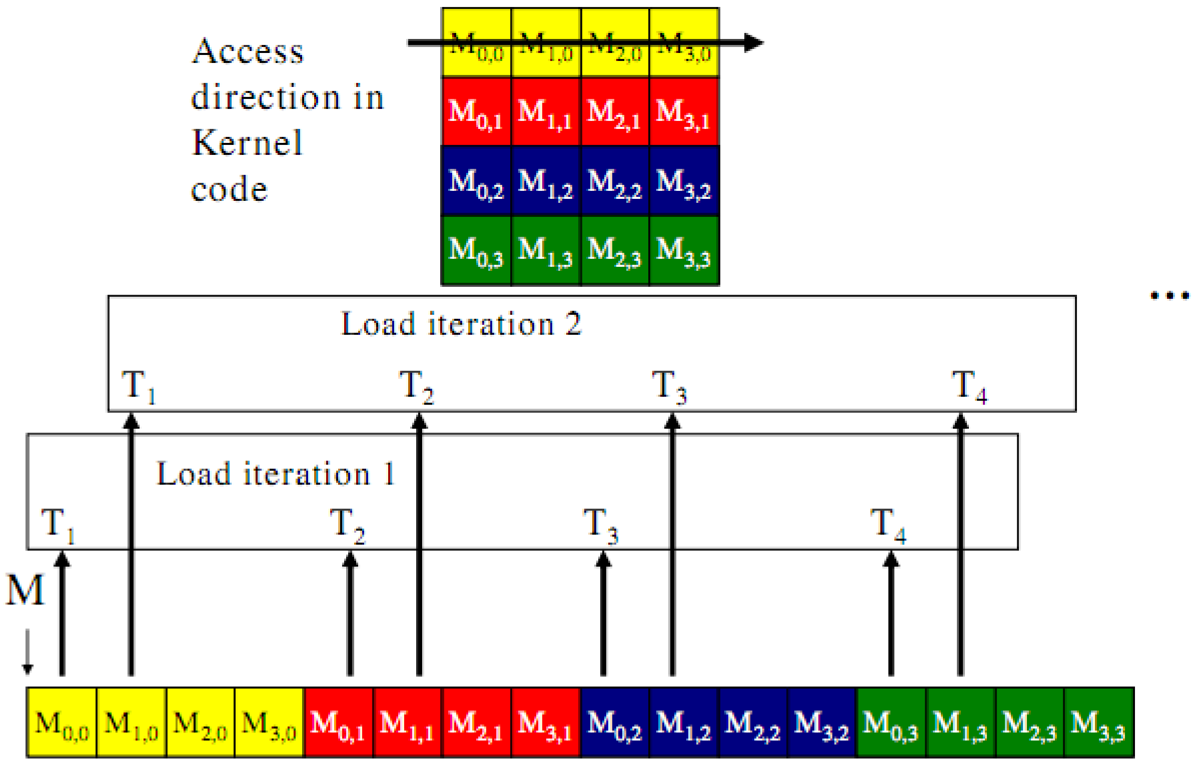
- Given a matrix stored *row-major* in *global memory*, what is a *thread*'s desirable access pattern?



a) column after column? b) row after row?

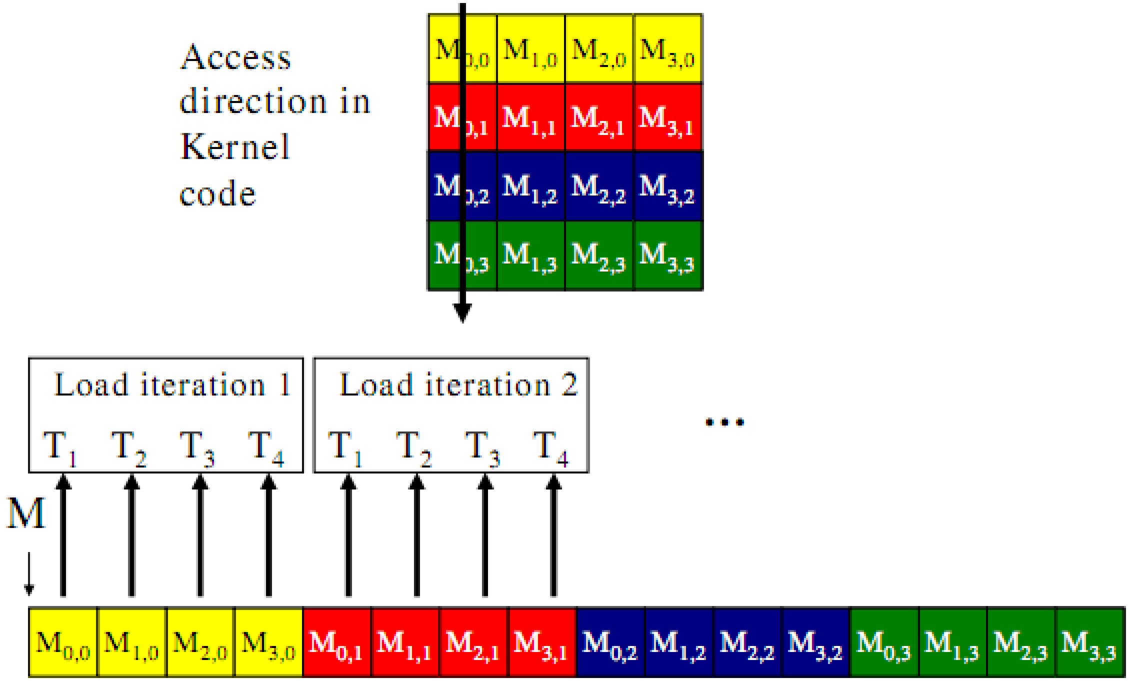
Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Memory Coalescing



a) column after column

Memory Coalescing



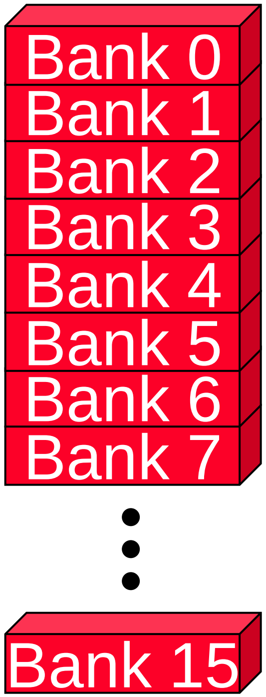
b) row after row

Memory Coalescing（合并访存）

- The GPU coalesce consecutive reads in a *half-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*…

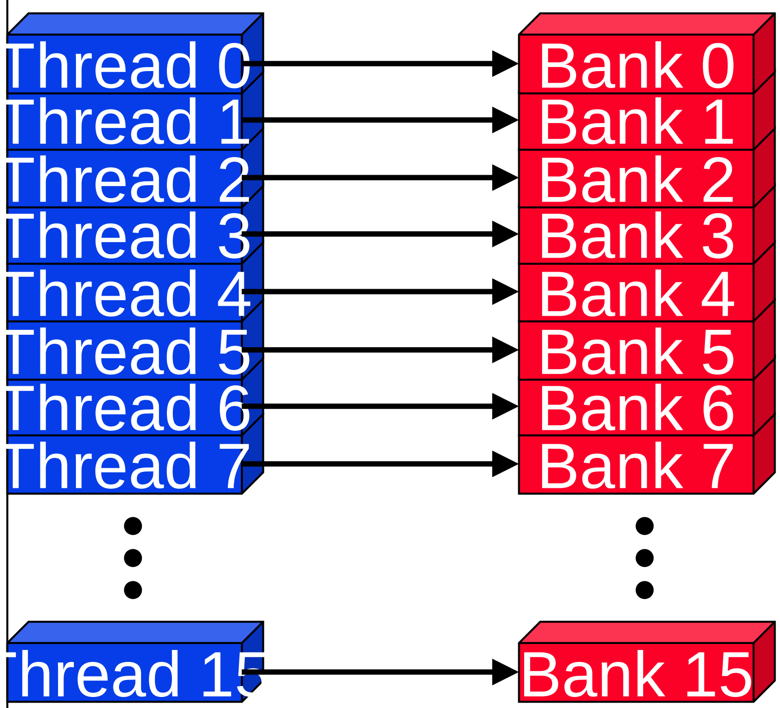
Bank Conflicts

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

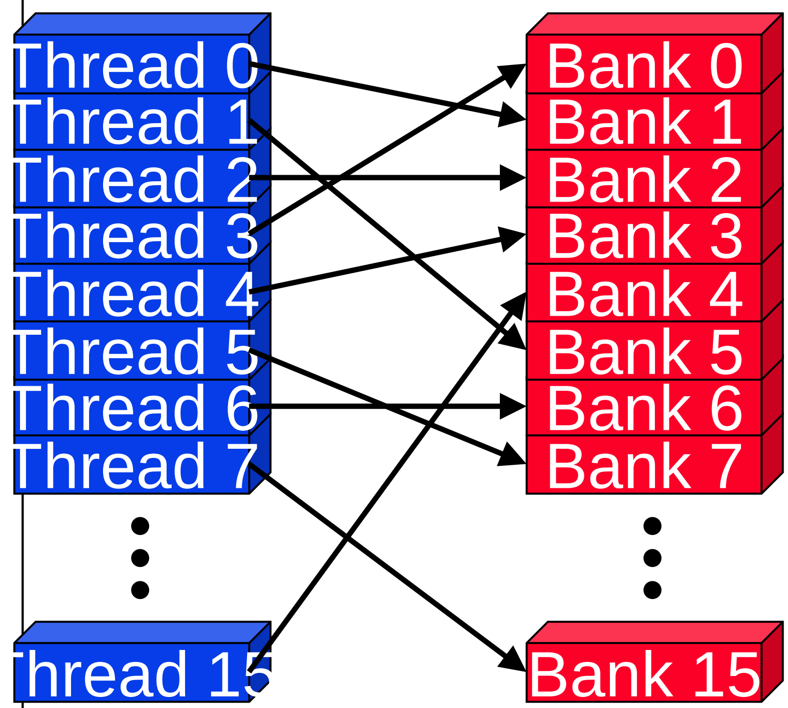


Bank Conflicts

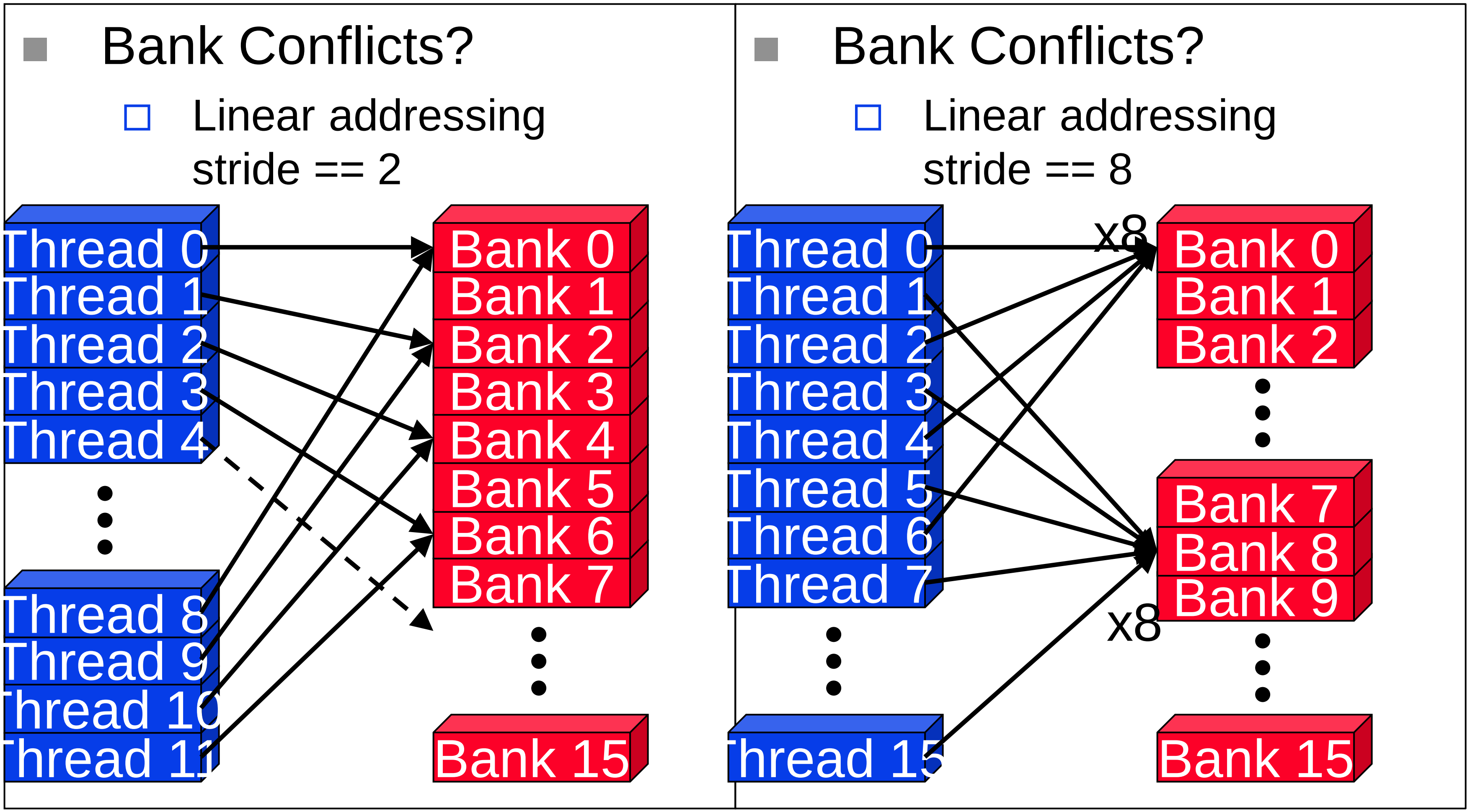
- Bank Conflicts?
  - Linear addressing stride == 1



- Bank Conflicts?
  - Random 1:1 Permutation



Bank Conflicts



Bank Conflicts

- Fast Path 1 (G80)
  - All threads in a half-warp access different banks

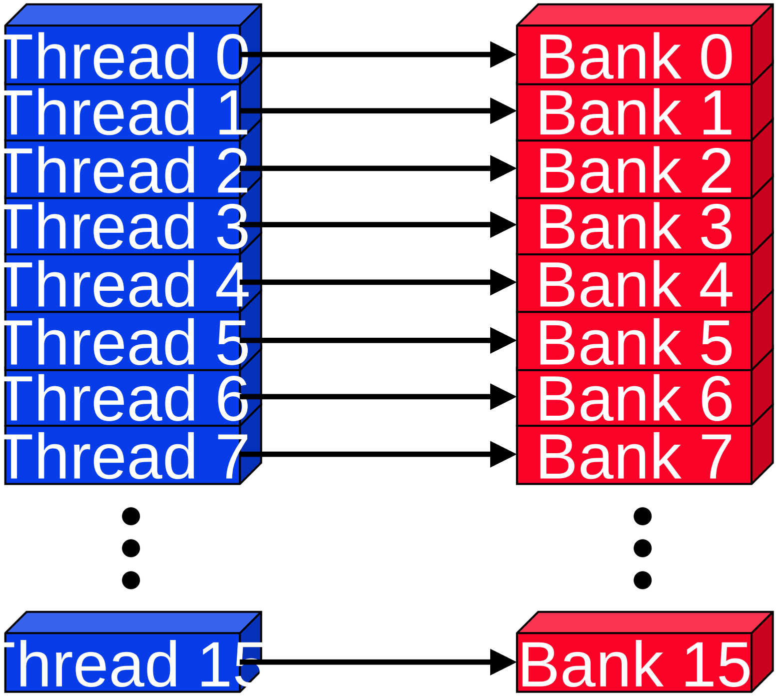


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Fast Path 2 (G80)
  - All threads in a half-warp access the same address

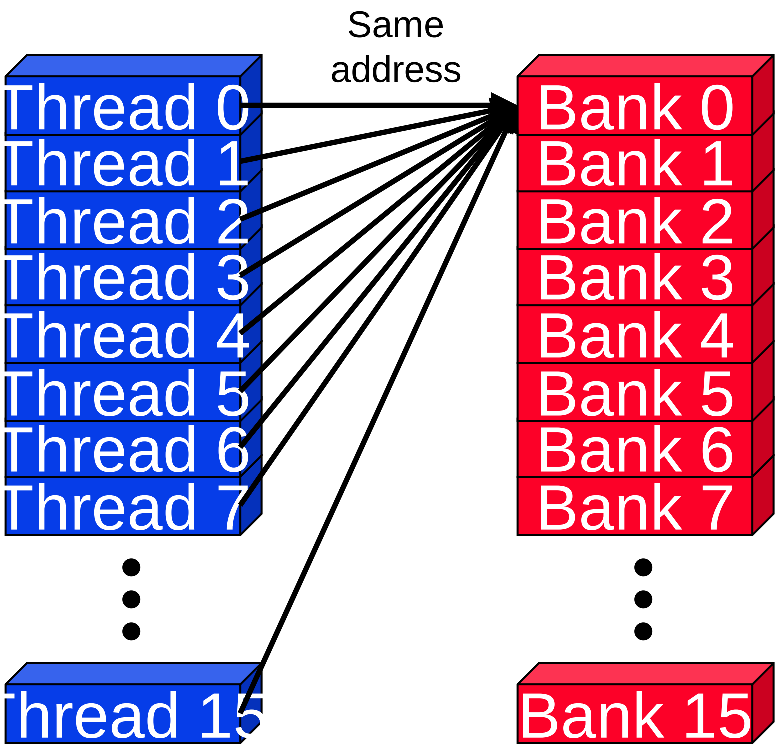


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Slow Path (G80)
  - Multiple threads in a half-warp access the same bank
  - Access is serialized
  - What is the cost?

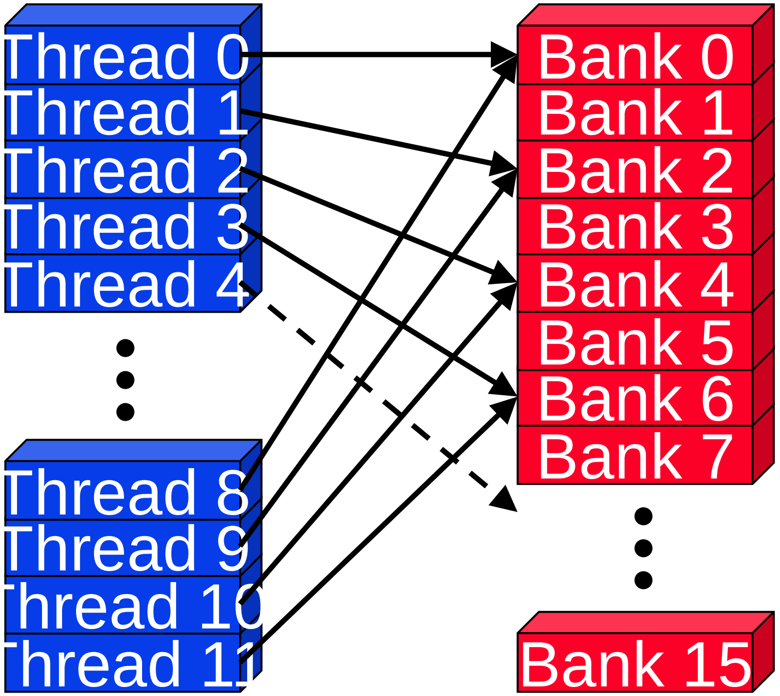


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

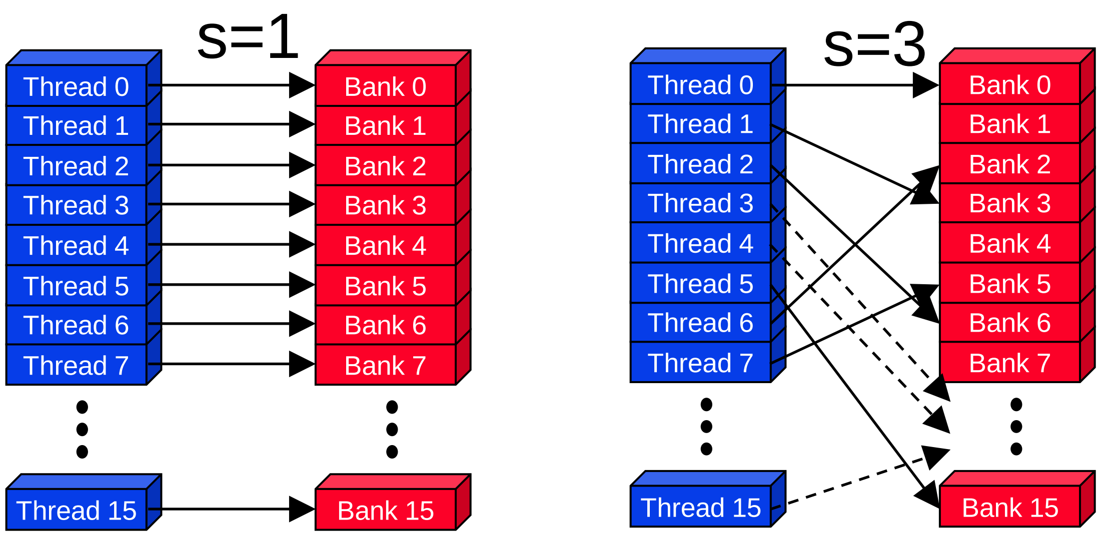
Bank Conflicts

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

- For what values of `s` is this conflict free?
  - Hint: The G80 has 16 banks

Bank Conflicts

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



no conflicts: stride 和 bank数目 没有公因子， S必须为奇数

下一节

- 线程级并行性（TLP）
  - 多核处理器中的关键问题：高速缓存一致性

From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, （5th edition)

高速缓存一致性与假共享：例题

- 如下代码在SMP（shared memory multiprocessors）环境下执行，sum和sum_local是全局变量，被NUM_THREADS个线程所共享：

double sum=0.0, **sum_local[NUM_THREADS]**;

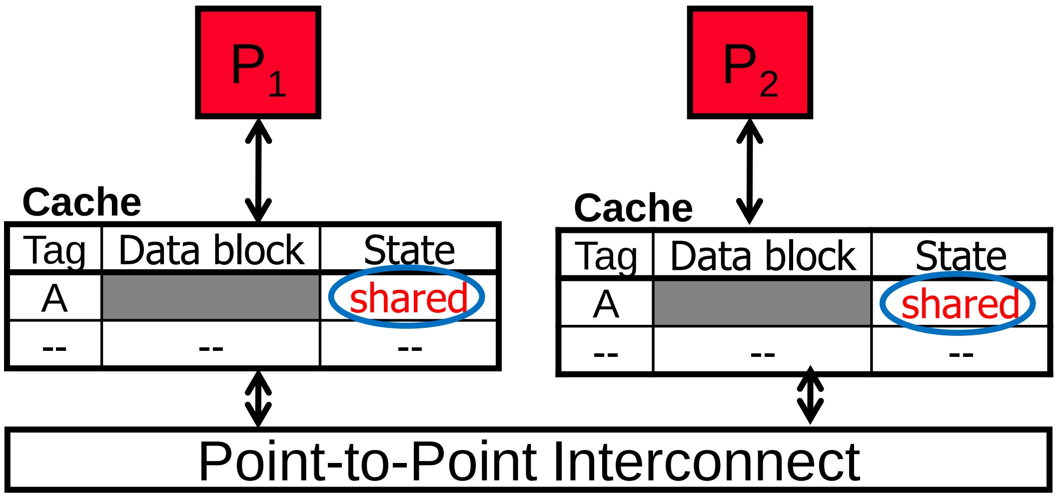
```
#pragma omp parallel num_threads(NUM_THREADS)
//由NUM_THREADS个线程执行以下相同的代码段
 {  int me = omp_get_thread_num();
    sum_local[me] = 0.0;
    #pragma omp for  //并行for语句，不同线程处理部分数据
    for (i = 0; i < N; i++)
       sum_local[me] += x[i] * y[i];  //将结果存入对应该线程的sum_local元素中
   #pragma omp atomic        //并行原子操作，
    sum += sum_local[me];  //求总和
 }
```

高速缓存一致性和假共享

假设:

- P1写一个数据块内的第i个字
- P2写同一块内的第k个字

会发生什么？



初始时，P1和P2共享一个数据块，

私有cache中的状态都是shared

高速缓存一致性和假共享

- **高速缓存一致性协议以数据块为单位，而不是以字为单位**
- **一个高速缓存数据块包含的字数多于1**

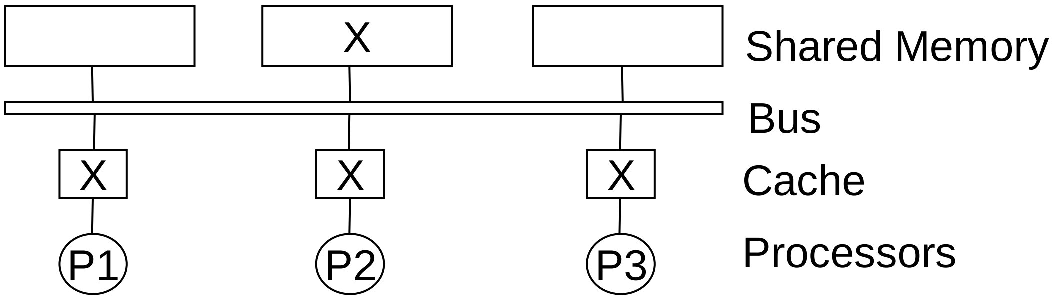
状态位	标记位	Word 0	Word 1	……	Word N

一个高速缓存数据块

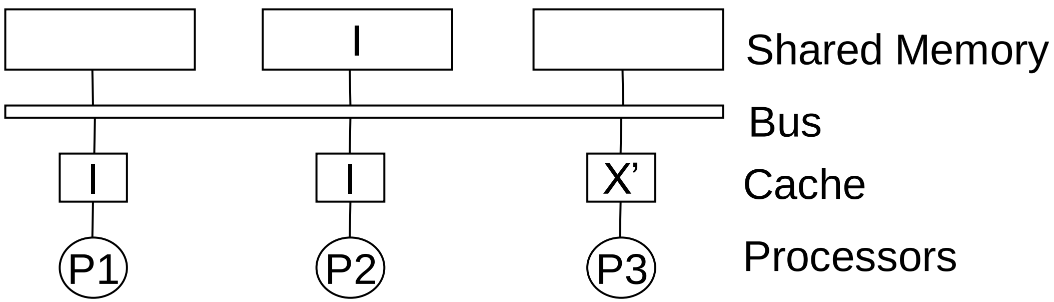
假共享：

- 当两个或更多处理器共享同一个数据块的不同部分时是假共享

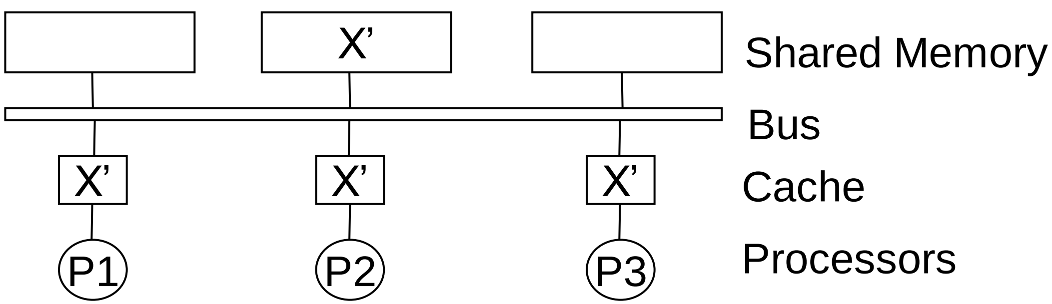
Bus Based Snooping Protocol



X is a shared variable that has a copy in all caches. Then a write occurred



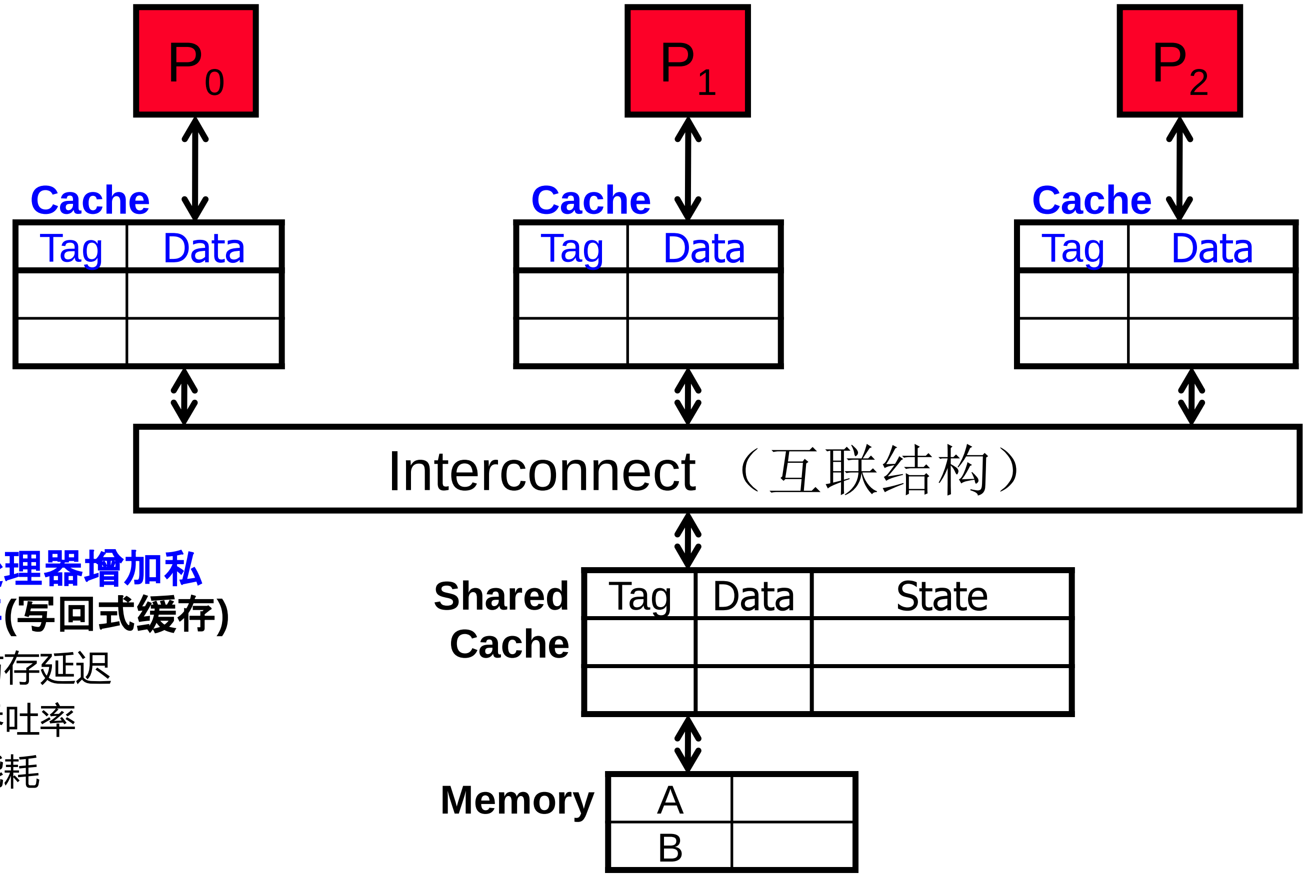
For Write Invalidate all the cache copies are marked as “invalid” except the most recent one



For **Write Update** all the cache copies are updated with the most recent value

Assume a write through cache protocol

增加私有高速缓存



- **为每一个处理器增加私有高速缓存(写回式缓存)**
  - 降低访存延迟
  - 增加吞吐率
  - 减少能耗

谢 谢！