PMPP 2015/16

CUDA Performance

(Preliminary) Course Schedule

you are here

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Introduction Final Projects, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A)
02.11.2015	Lecture, Final Projects assigned, Ex. 1 due, Ex. 2 assigned
03.11.2015	Questions and Answers (Q&A)
09.11.2015	Lecture, Exercise 2 due
10.11.2015	Lecture
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)

(Preliminary) Course Schedule

30.11.2015	
01.12.2015	
07.12.2014	
08.12.2014	
14.12.2014	
15.12.2014	
	Christmas Break
11.01.2015	2nd Presentation Status Final Projects (1)
12.01.2015	2nd Presentation Status Final Projects (2)
…	
01.02.2015	
02.02.2015	
08.02.2015	final Presentation Status Final Projects (1)
09.02.2015	final Presentation Status Final Projects (2)

Registering in TuCAN/Moodle

- in TuCAN, you can register for a module and for a course within the module
- only if you did both, you are actually registered for the course and automatically receive a Moodle account!

- If you still haven't signed the HHLR account request, please do so *now*

Today's Topics

- refresher from last lecture: extension of simple matrix multiplication example to incorporate shared memory
- fast memory access
  - bank conflicts in shared memory
  - memory coalescing for global memory
- control flow
  - avoiding divergence
- example: vector reduction

Tiled Multiply Using Thread Blocks

- one block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- one thread computes one element of P_{sub}
- assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape

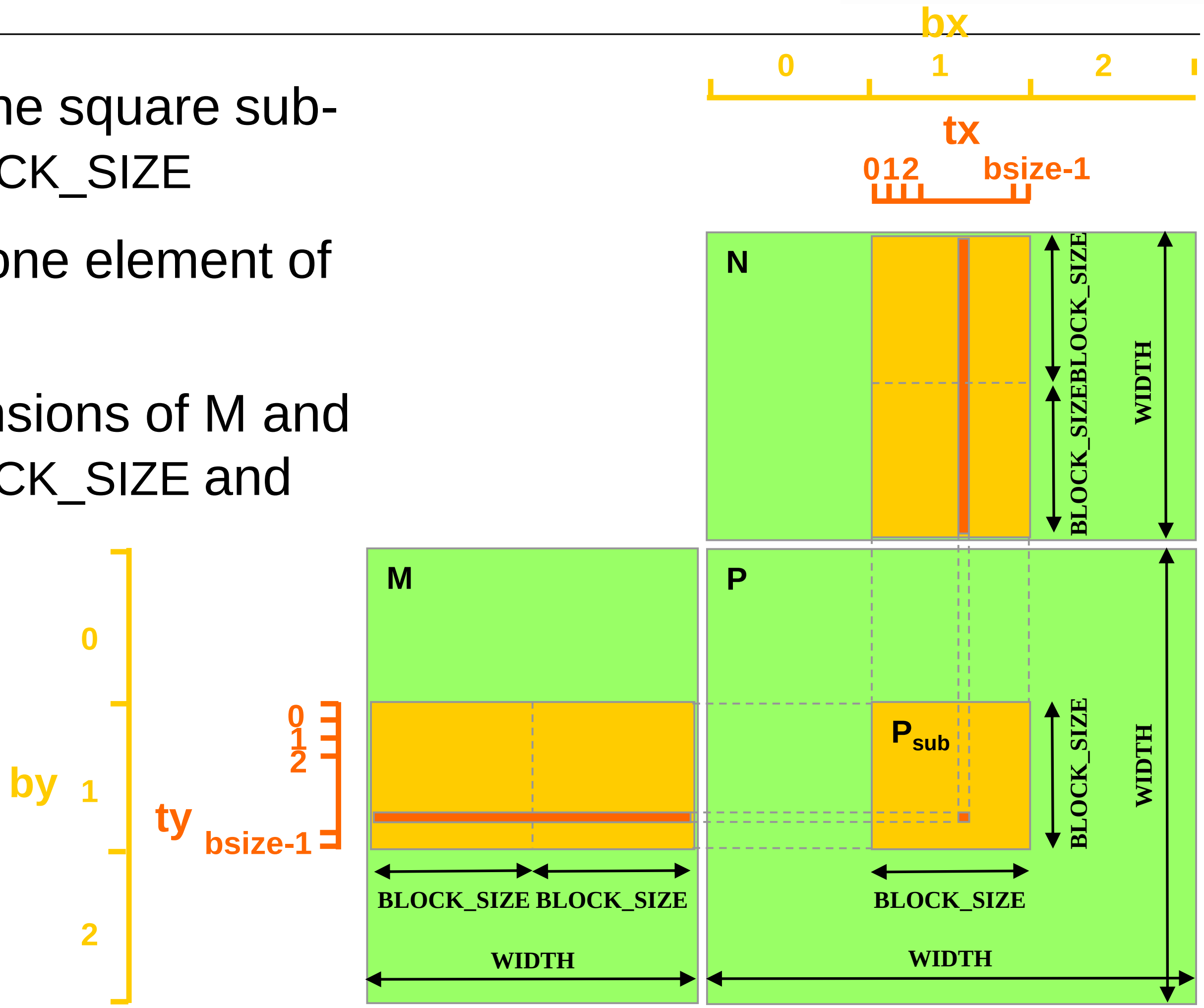


image source: NVIDIA

Shared Memory Bank Conflicts

- threads in the same warp may have bank conflict for Nsub accesses
- this should be minimal since the warp likely spans the horizontal direction, resulting in broadcast of Msub accesses and no/little conflict for N accesses

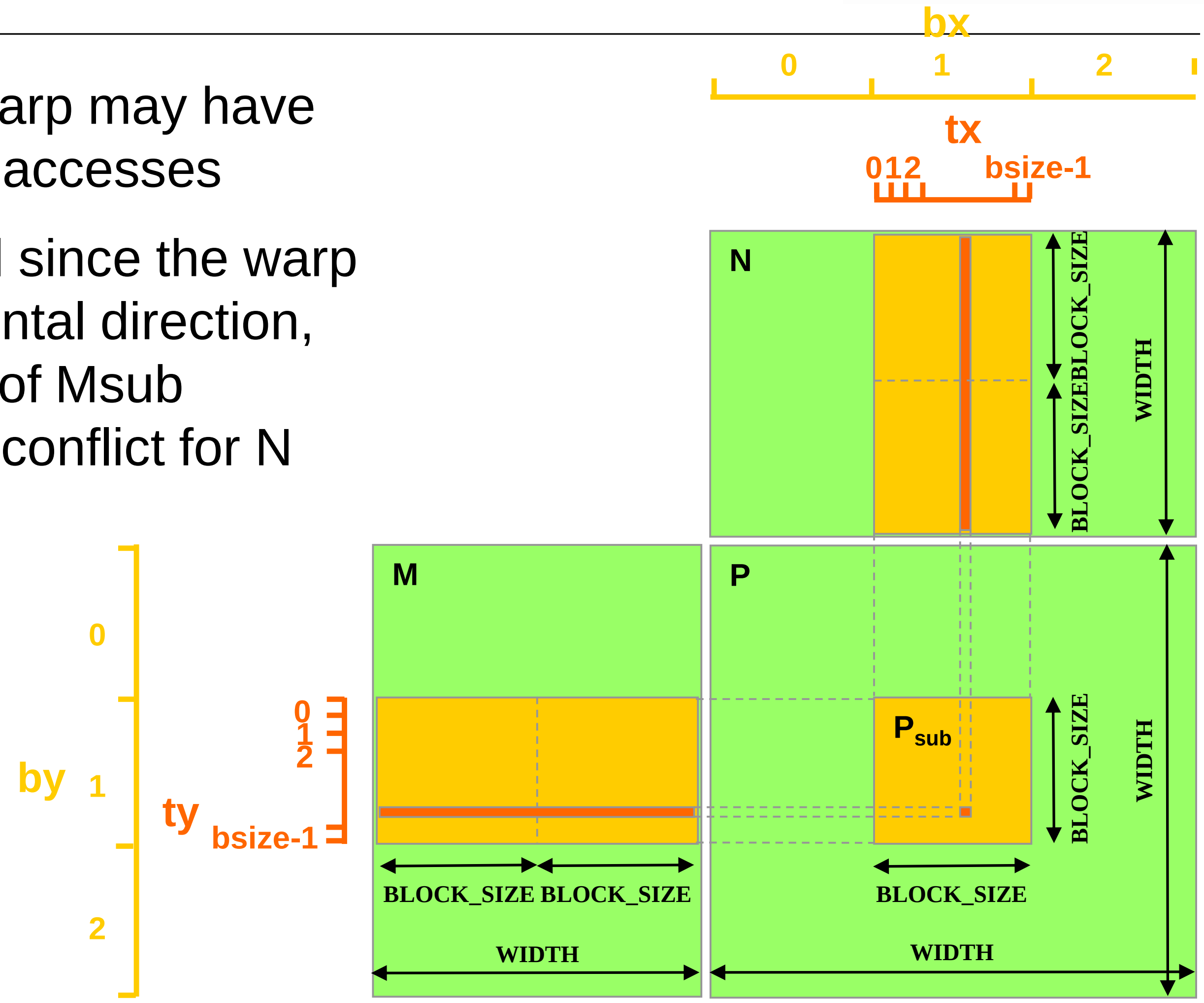


image source: NVIDIA

CUDA Compute Capabilities

- GPUs can be categorized into different classes according to their capabilities
- compute capability
  - major distinction between hardware architectures
  - 1.x (1.0, 1.1, 1.2, 1.3) NOT SUPPORTED ANYMORE!
  - 2.x (2.0, 2.1) for Fermi architecture
  - 3.x (3.0, 3.5, 3.7) for Kepler architecture
  - 5.x (5.0, 5.2, 5.3) for Maxwell architecture
- see details on the next slides and in the CUDA C Programming Guide
- requires different optimizations for different GPU generations

CUDA Compute Capabilities

<table>
<tr><th>Feature Support</th><th colspan="5">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>2.x</th><th>3.0</th><th>3.2</th><th>3.5, 3.7, 5.0, 5.2</th><th>5.3</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td rowspan="14" colspan="5">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td></tr>
<tr><td>Warp vote functions (Warp Vote Functions)</td></tr>
<tr><td>Double-precision floating-point numbers</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td></tr>
<tr><td>__ballot() (Warp Vote Functions)</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
<tr><td>Unified Memory Programming</td><td>No</td><td colspan="4">Yes</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td colspan="2">No</td><td colspan="3">Yes</td></tr>
<tr><td>Dynamic Parallelism</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td colspan="4">No</td><td>Yes</td></tr>
</table>

image source: NVIDIA

CUDA Compute Capabilities

<table>
<tr><th></th><th colspan="8">Compute Capability</th></tr>
<tr><th>Technical Specifications</th><th>2.x</th><th>3.0</th><th>3.2</th><th>3.5</th><th>3.7</th><th>5.0</th><th>5.2</th><th>5.3</th></tr>
<tr><td>Maximum number of resident grids per device (Concurrent Kernel Execution)</td><td colspan="2">16</td><td>4</td><td colspan="4">32</td><td>16</td></tr>
<tr><td>Maximum dimensionality of grid of thread blocks</td><td colspan="8">3</td></tr>
<tr><td>Maximum x-dimension of a grid of thread blocks</td><td>65535</td><td colspan="7">2^{31}-1</td></tr>
<tr><td>Maximum y- or z-dimension of a grid of thread blocks</td><td colspan="8">65535</td></tr>
<tr><td>Maximum dimensionality of thread block</td><td colspan="8">3</td></tr>
<tr><td>Maximum x- or y-dimension of a block</td><td colspan="8">1024</td></tr>
<tr><td>Maximum z-dimension of a block</td><td colspan="8">64</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="8">1024</td></tr>
<tr><td>Warp size</td><td colspan="8">32</td></tr>
<tr><td>Maximum number of resident blocks per multiprocessor</td><td>8</td><td colspan="4">16</td><td colspan="3">32</td></tr>
<tr><td>Maximum number of resident warps per multiprocessor</td><td>48</td><td colspan="7">64</td></tr>
<tr><td>Maximum number of resident threads per multiprocessor</td><td>1536</td><td colspan="7">2048</td></tr>
<tr><td>Number of 32-bit registers per multiprocessor</td><td>32 K</td><td colspan="3">64 K</td><td>128 K</td><td colspan="3">64 K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread block</td><td>32 K</td><td colspan="6">64 K</td><td>32 K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread</td><td colspan="2">63</td><td colspan="6">255</td></tr>
<tr><td>Maximum amount of shared memory per multiprocessor</td><td colspan="4">48 KB</td><td>112 KB</td><td>64 KB</td><td>96 KB</td><td>64 KB</td></tr>
<tr><td>Maximum amount of shared memory per thread block</td><td colspan="8">48 KB</td></tr>
<tr><td>Number of shared memory banks</td><td colspan="8">32</td></tr>
<tr><td>Amount of local memory per thread</td><td colspan="8">512 KB</td></tr>
<tr><td>Constant memory size</td><td colspan="8">64 KB</td></tr>
<tr><td>Cache working set per multiprocessor for constant memory</td><td colspan="5">8 KB</td><td colspan="3">10 KB</td></tr>
<tr><td>Cache working set per multiprocessor for texture memory</td><td>12 KB</td><td colspan="7">Between 12 KB and 48 KB</td></tr>
<tr><td>Maximum width for a 1D texture reference bound to a CUDA array</td><td colspan="8">65536</td></tr>
<tr><td>Maximum width for a 1D texture reference bound to linear</td><td colspan="8">2^{27}</td></tr>
</table>

■ ■ ■

image source: NVIDIA

CUDA Compute Capabilities

- GPU to Capability mapping
  - see also https://developer.nvidia.com/cuda-gpus

GeForce Desktop Products		GeForce Notebook Products	
GPU	Compute Capability	GPU	Compute Capability
GeForce GTX TITAN Z	3.5	GeForce GTX 980M	5.2
GeForce GTX TITAN Black	3.5	GeForce GTX 970M	5.2
GeForce GTX TITAN	3.5	GeForce GTX 880M	3.0
GeForce GTX 980	5.2	GeForce GTX 870M	3.0
GeForce GTX 970	5.2	GeForce GTX 860M	3.0/5.0(**)
GeForce GTX 780	3.5	GeForce GTX 850M	5.0
GeForce GTX 770	3.0	GeForce 840M	5.0
GeForce GTX 760	3.0	GeForce 830M	5.0
GeForce GTX 750 Ti	5.0	GeForce 820M	2.1
GeForce GTX 750	5.0	GeForce GTX 780M	3.0
GeForce GTX 690	3.0	GeForce GTX 770M	3.0
GeForce GTX 680	3.0	GeForce GTX 765M	3.0
GeForce GTX 670	3.0	GeForce GTX 760M	3.0
GeForce GTX 660 Ti	3.0	GeForce GTX 680MX	3.0

image source: NVIDIA

CUDA Hardware

- GPUs also have different numbers of multiprocessors
  - see also https://developer.nvidia.com/cuda-gpus

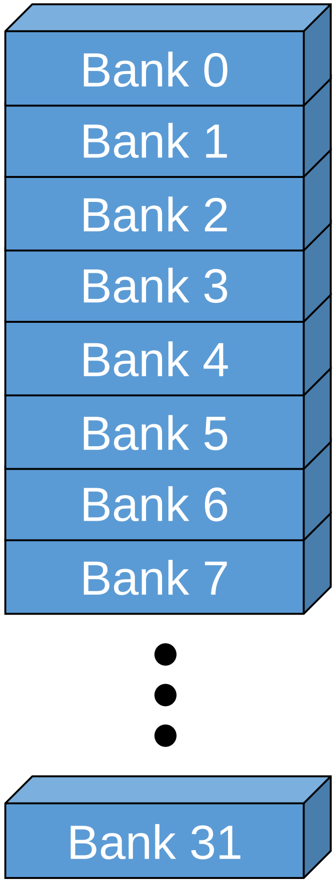
Desktop			Laptop		
GPU	MPs	Cores	GPU	MPs	Cores
Geforce GTX Titan X	24	3072	Geforce GTX 980	16	2048
Geforce GTX Titan Z	2x15	5760	Geforce GTX 980M	12	1536
Geforce GTX Titan Black	15	2880	Geforce GTX 970M	10	1280
Geforce GTX 980	16	2048	Geforce GTX 880M	12	1536
Geforce GTX 970	13	1664	Geforce GTX 870M	11	1344
Geforce GTX 780	12	2304	Geforce GTX 860M	6/5	1152/640*
Tesla K20X	14	2688	Geforce GTX 850M	5	640

* 1152 for compute 3.0 (Kepler) or 640 for compute 5.0 (Maxwell) depending on the actual chip in the card (re-branding!)

image source: NVIDIA

Parallel Memory Access

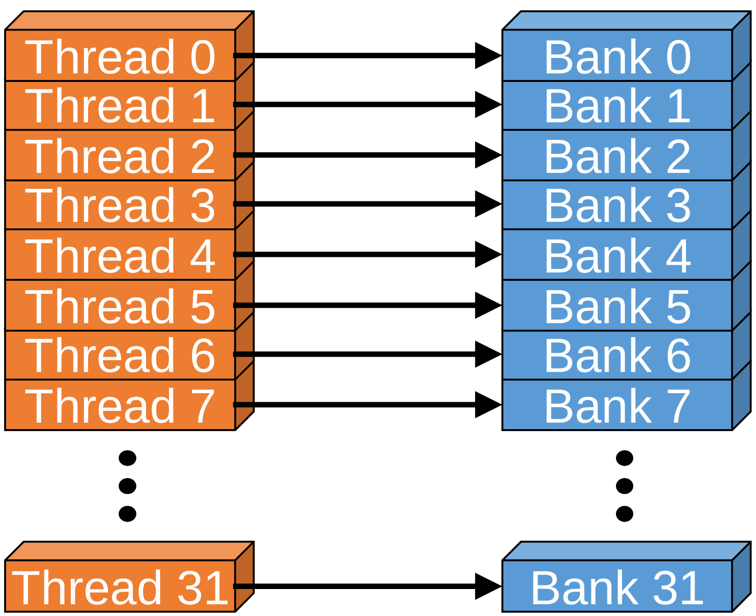
- in a parallel machine, many threads access memory
  - therefore, memory is divided into banks
  - essential to achieve high bandwidth
- each bank can service one address per cycle
  - a memory can service as many simultaneous accesses as it has banks
- multiple simultaneous accesses to a bank result in a bank conflict
  - conflicting accesses are serialized



Bank Addressing Examples

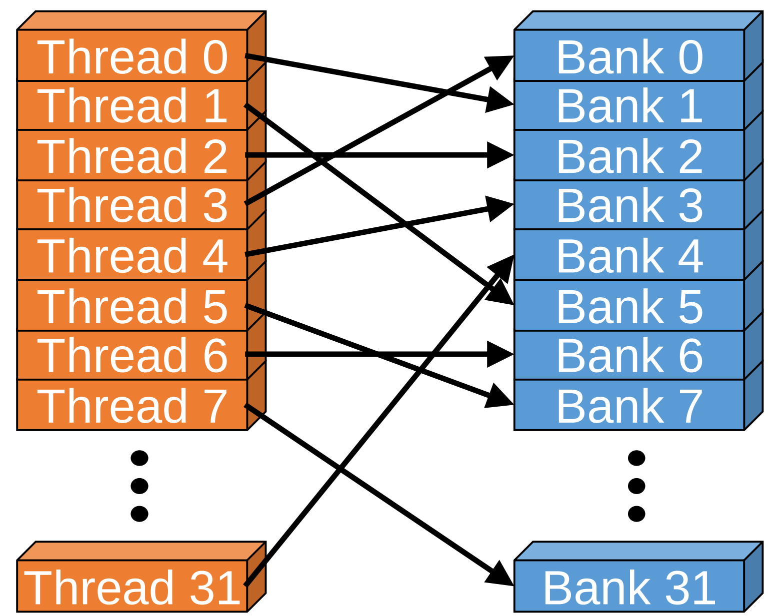
- linear addressing with stride == 1

➔ no bank conflicts



- random 1:1 permutation
- each thread accesses unique bank

➔ no bank conflicts



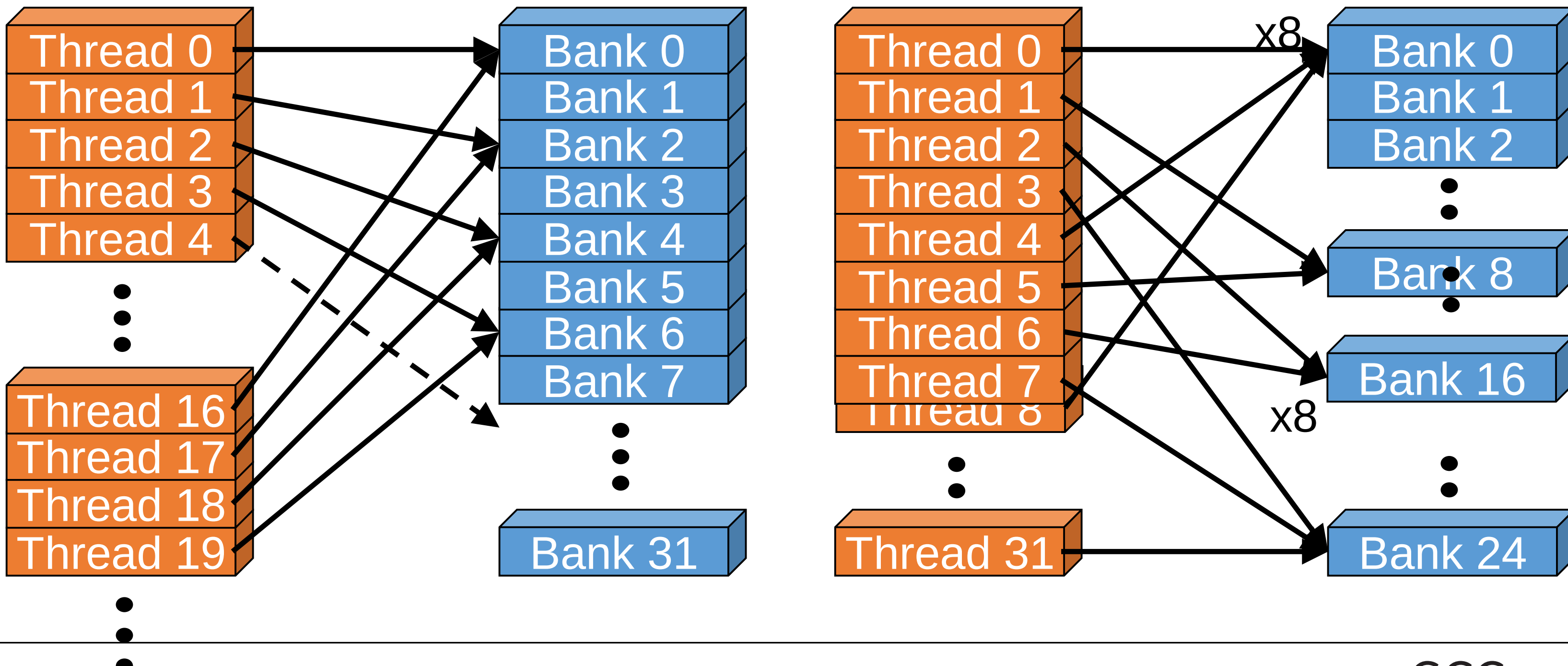
Bank Addressing Examples

- linear addressing with stride == 2

➔ 2 way bank conflicts

- linear addressing with stride == 8

➔ 8 way bank conflicts



How Addresses Map to Banks

- each bank has a bandwidth of 32 bits per clock cycle
  - e.g. 1 float
- successive 32 bit words are assigned to successive banks
  - additional 64 bit mode$_{3.x}$
- current hardware has 32 banks (compute capability 2.x and up)
  - so bank = address (in 32 bits) % 32
  - same as the size of a warp

Shared Memory Bank Conflicts

- the fast cases
  - if all threads of a warp access different banks, there is no bank conflict
  - if subgroups of warp threads access the identical bank AND address, there is no bank conflict (multicast)
- the slow case
  - multiple threads in the same warp access the same bank but different addresses

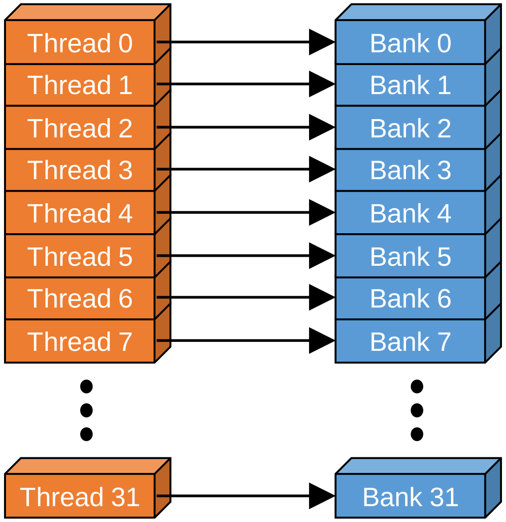
  ➔ bank conflict
  - must serialize the accesses
  - cost = max # of simultaneous accesses to a single bank
- shared memory is said to be as fast as registers if there are no bank conflicts

Linear Addressing

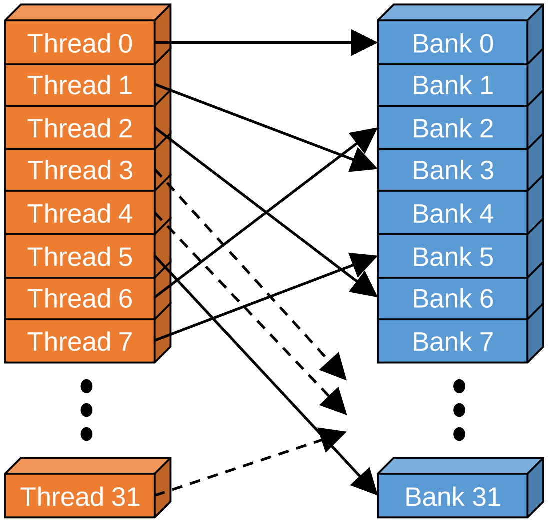
```
__shared__ float shared[256];
float foo = shared[baseIndex + s * threadIdx.x];
```

- this is only bank-conflict-free if `s` shares no common factors with the number of banks
  - 32 banks on current architectures
  - ➔ `s` must be odd

s=1



s=3



Data Types and Bank Conflicts

- this has no conflicts if type of shared is 32-bits

```
__shared__ float shared[256];
float foo = shared[bi+threadIdx.x];
```

- smaller data types led to bank conflicts with older hardware

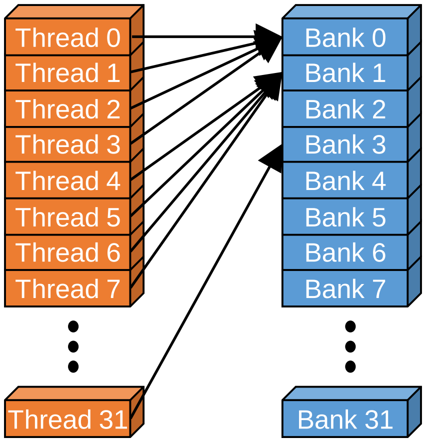
```
__shared__ char shared[256];
char foo = shared[bi+threadIdx.x];
```

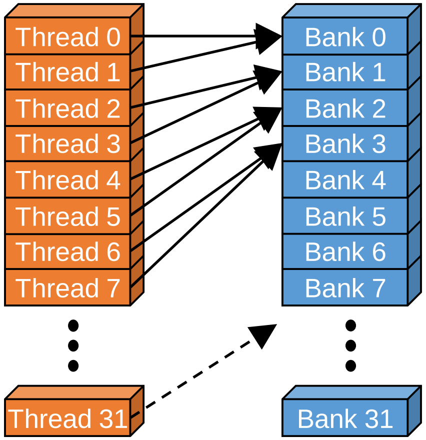
➔ 4 way bank conflict

```
__shared__ short shared[256];
short foo = shared[bi+threadIdx.x];
```

➔ 2 way bank conflict

- solved by multicast feature on 2.x cards

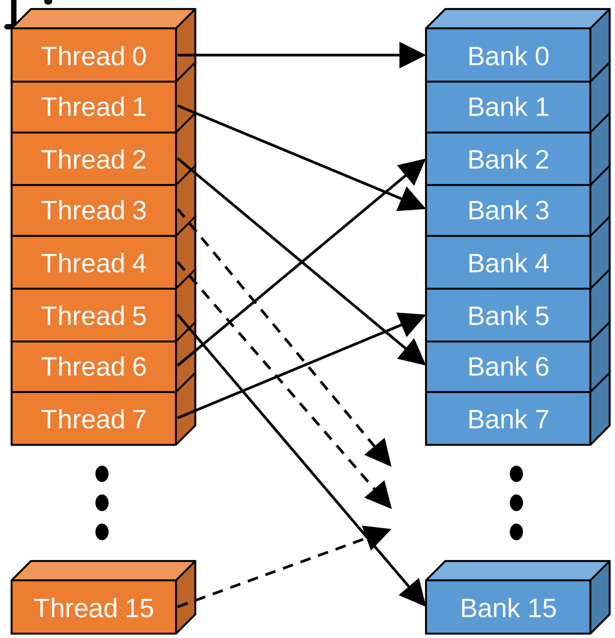




Structs and Bank Conflicts

- struct assignments compile into as many memory accesses as there are struct members

```
struct vector { float x, y, z; };
__shared__ struct vector vectors[64];
struct vector v =
vectors[bi+threadIdx.x];
```



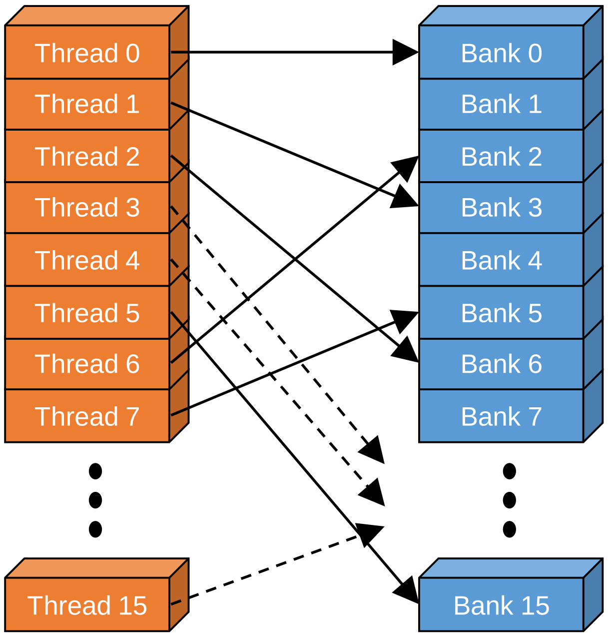
  - this has no bank conflicts for vector
  - struct size is 3 words
  - 3 accesses per thread,
    implicit stride of 3
    (no common factor with 32)

Structs and Bank Conflicts

- struct assignments compile into as many memory accesses as there are struct members

```
struct myType { float f; char c; };
  __shared__ struct myType myTypes[64];
struct myType m =
myTypes[bi+threadIdx.x];
```

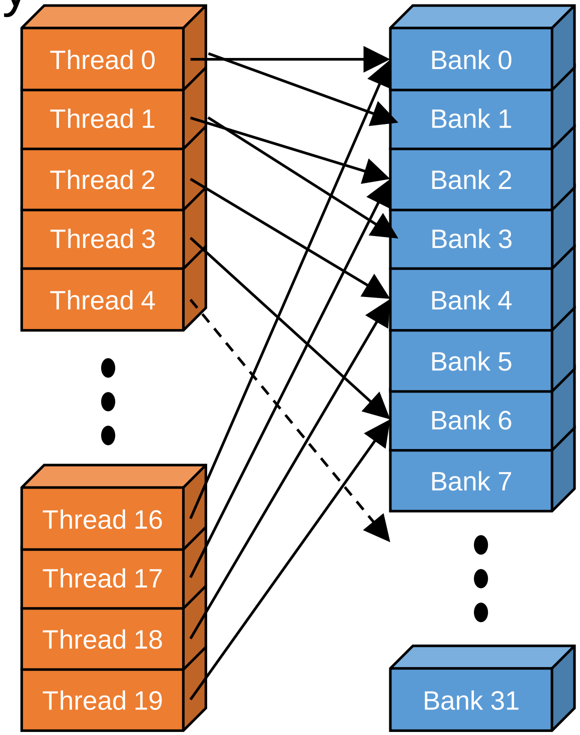
  - this has bank conflicts for **`myType`**
  - 2 accesses per thread
  - stride of 5 bytes



Common Array Bank Conflict Patterns

- 1D array
- each thread loads 2 elements into shared memory

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

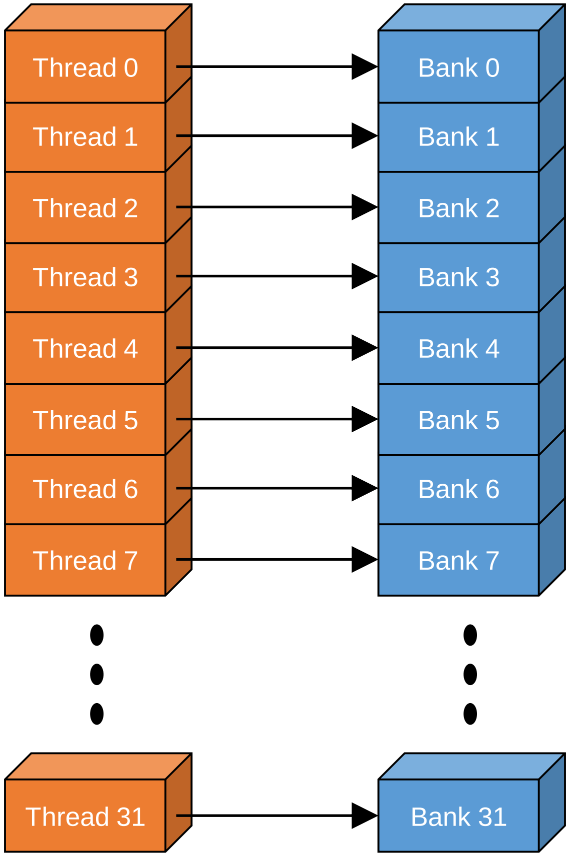


  - 2-way-interleaved loads result in 2-way bank conflicts
  - this makes sense for traditional CPU threads, locality in cache line usage, and reduced sharing traffic
  - not in shared memory usage where there is no cache line effects but banking effects

Better Array Access Pattern

- each thread loads one element in every consecutive group of blockDim elements.

```
shared[tid] = global[tid];
shared[tid + blockDim.x] =
      global[tid + blockDim.x];
```



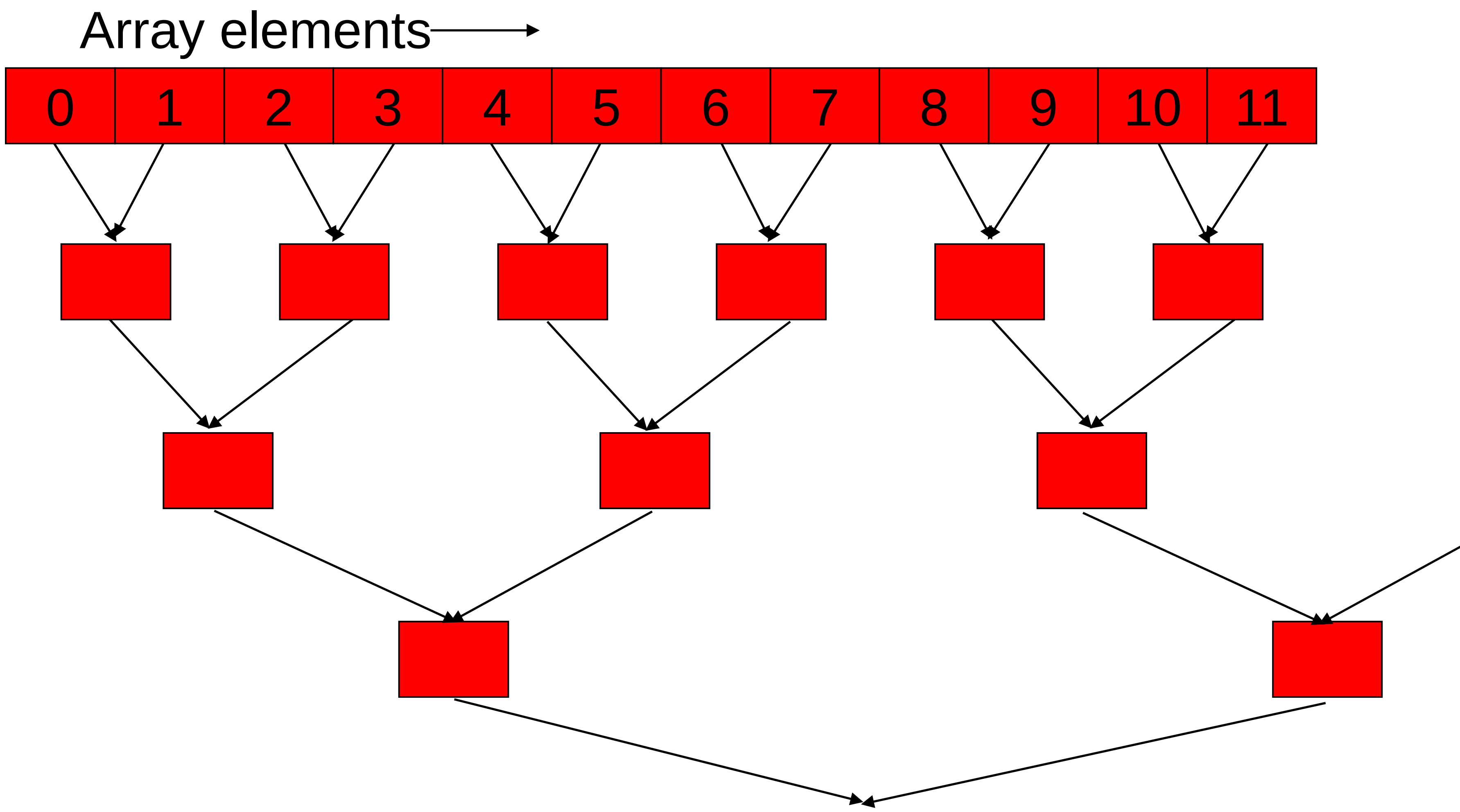
Example: Vector Reduction

- compute the sum, max, … of all elements in an array

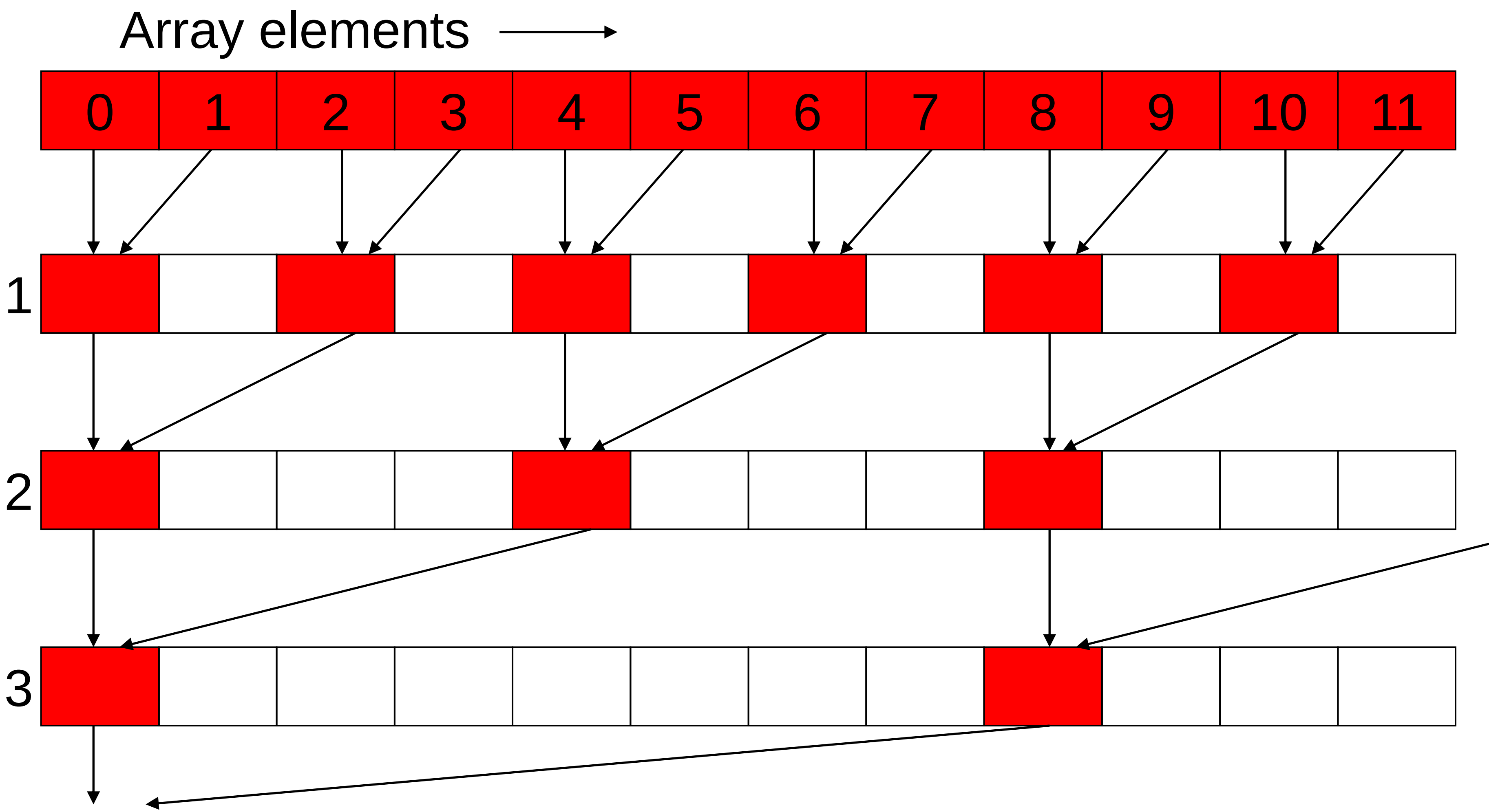
```
float sum=0;
float array[128];
for (int i = 0; i < 128; ++i) {
    sum += array[i];
}
```

- parallel implementation with logarithmic execution time

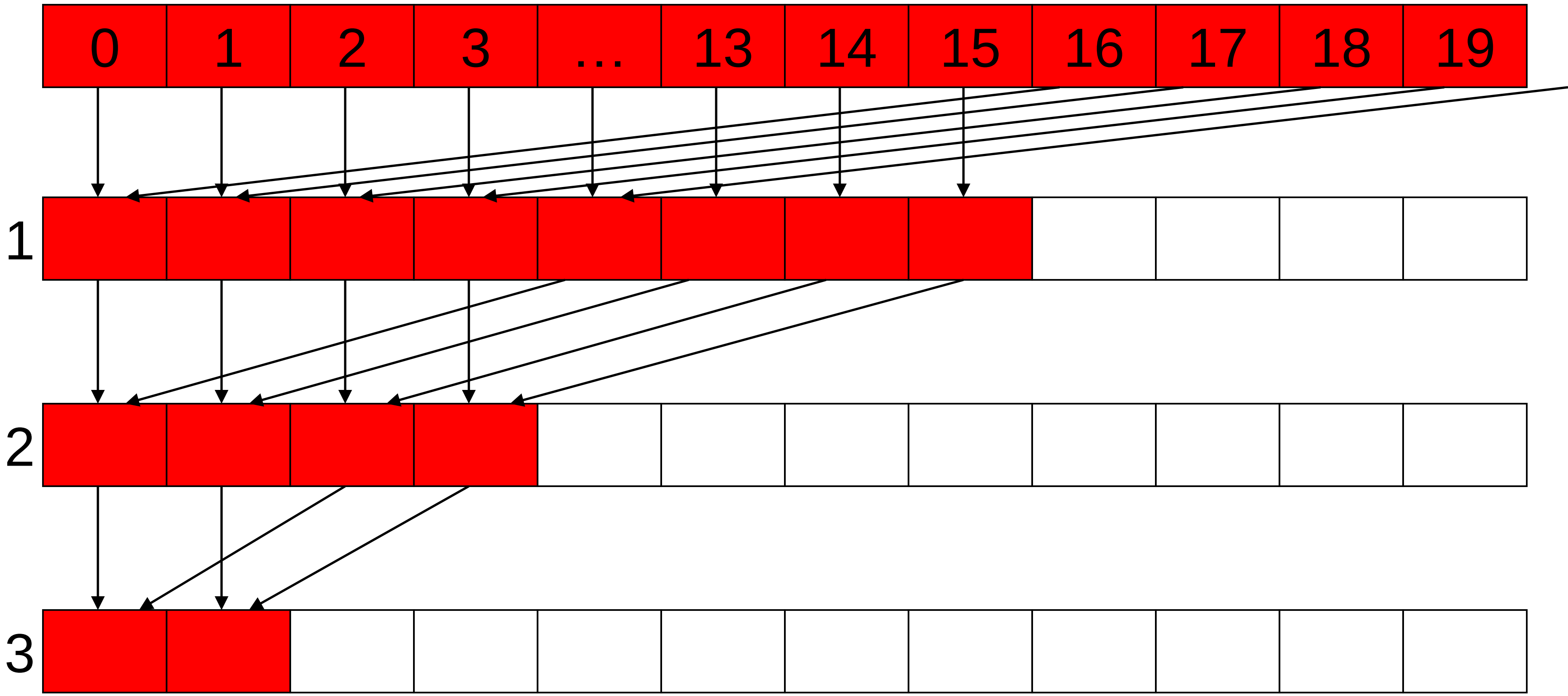
Vector Reduction



Vector Reduction with Bank Conflicts

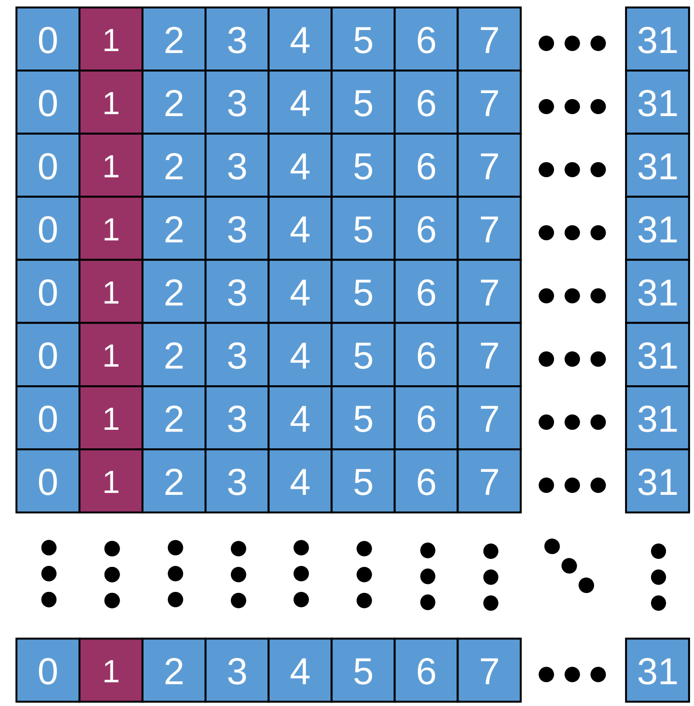


No Bank Conflicts



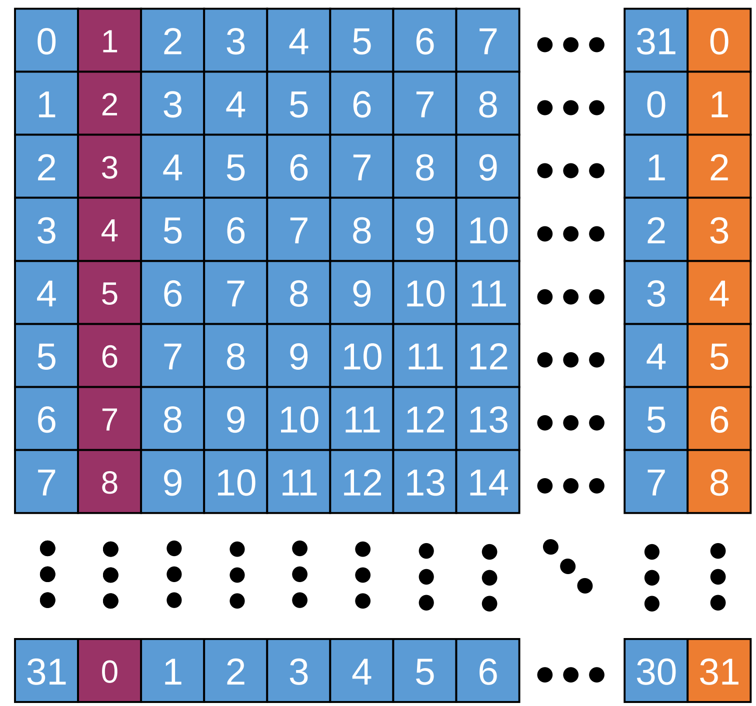
Common Bank Conflict Patterns (2D)

- operating on 2D array of floats in shared memory
  - e.g. image processing
- example:
  32x32 submatrix (subimage)
  - each thread processes a row
  - threads in a block access the elements in each column simultaneously
  - example: row 1 in purple
  - ➔ 32-way bank conflicts
    rows all start at bank 0



Common Bank Conflict Patterns (2D)

- solution 1
  - pad the rows
  - add one float to the end of each row
  - stride becomes odd
- solution 2
  - transpose before processing
  - suffer bank conflicts during transpose
  - but possibly save them later
  - usually less writes than reads

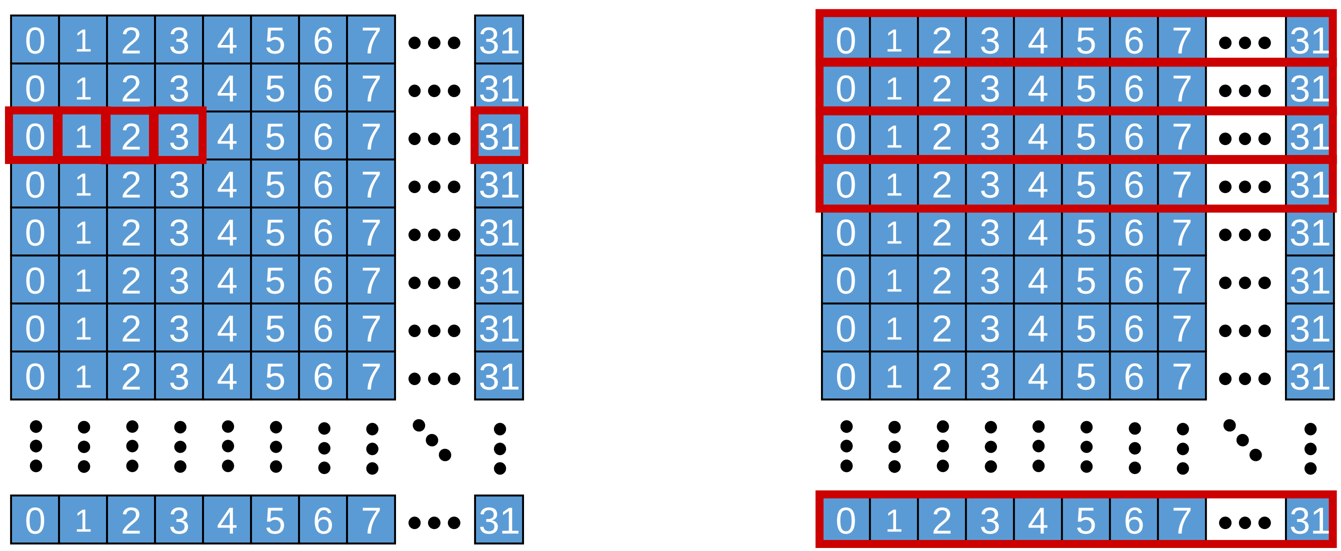


What about Matrix Multiplication?

- all warps in a block access the same row of M

➔ broadcast!

- all warps in a block access neighboring elements in a row as the access walks through neighboring columns!



Matrix Mult. Shared Memory Usage

- each block requires 2 * WIDTH * WIDTH * 4 bytes of shared memory
  - for WIDTH = 32, each block requires 8KB
  - depending on shared memory configuration 2 to 6 blocks can fit into the shared memory of an SMX
- but:
  - each SMX can only take 2048 threads (on K20X)
  - each SMX can only take 2 blocks of 1024 threads each

➔shared memory size is not a limitation for the Matrix Multiplication example

➔For current GPUs its still not an issue

Global Memory Access

- device can access 32 bit, 64 bit, or 128 bit at once

```
__device__ type device[32];
type data = device[tid];
```

- compiles to a single load instruction if
  - **`sizeof(type)`** is equal to 4, 8, or 16 and
  - variables of type **`type`** must be aligned to **`sizeof(type)`** bytes (that is, have their address be a multiple of **`sizeof(type)`**).

- alignment requirement is automatically fulfilled for built-in types like **`float2`** or **`float4`**

Global Memory Access

TECHNISCHE UNIVERSITÄT DARMSTADT

- for structures, the size and alignment requirements can be enforced by the compiler using the alignment specifiers `__align__(8)` or `__align__(16)`

```
struct __align__(8) { float a; float b; };
struct __align__(16) { float a; float b; float c;};
```

- for structures larger than 16 bytes, the compiler generates several load instructions
  - use `__align__(16)` to minimize number of loads

```
struct __align__(16) { float a; float b; float c;
                            float d; float e; };
```

  - compiles into two 128-bit load instructions

Coalesced Global Memory Access

- coalescing: combine loads from several threads in a warp into fewer transactions
- will only consider compute capability 2.0 and up
  - very strict rules on older hardware
- global memory partitioned into continuous segments of 128 bytes
- accessed via L1 and L2 cache which are aligned with these segments

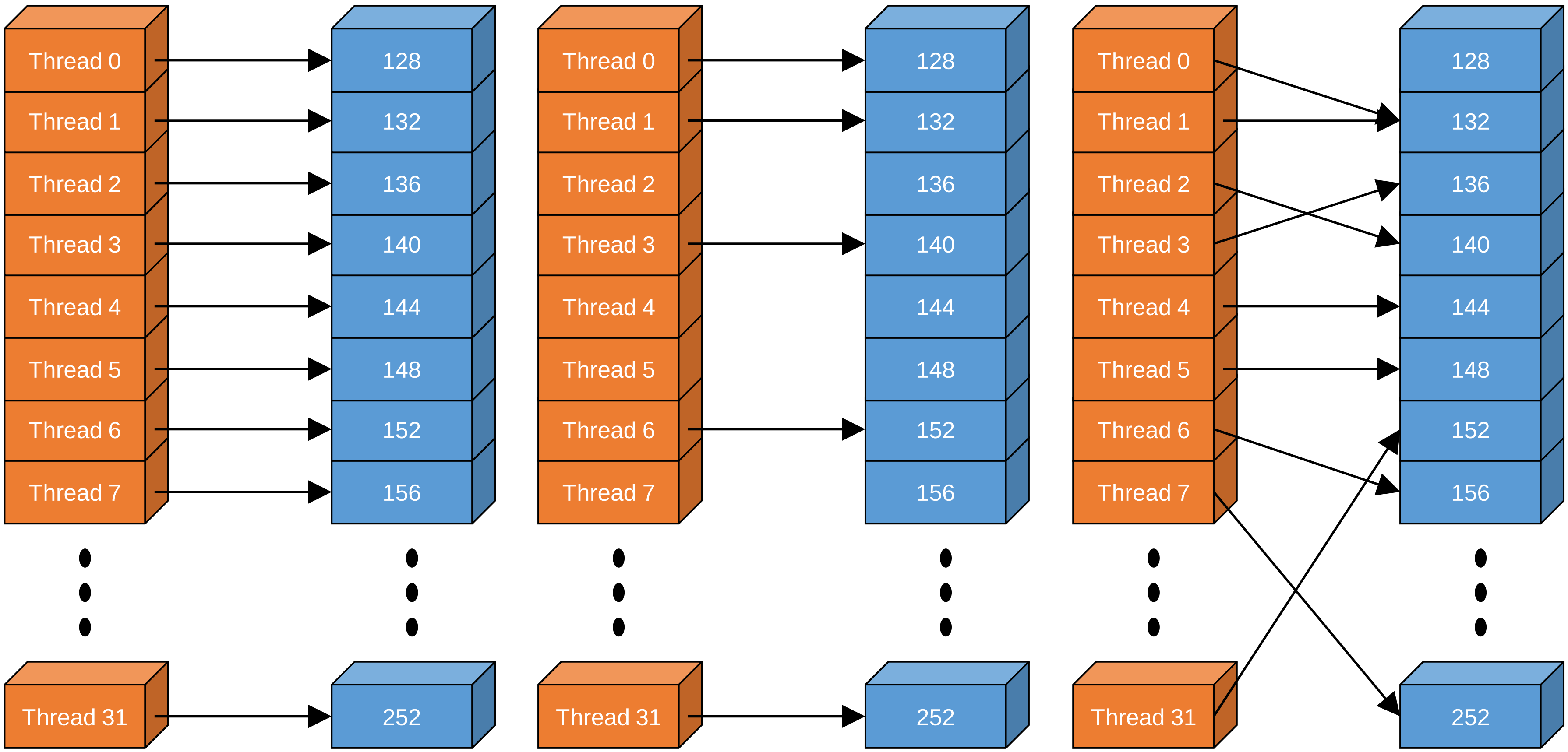
Coalesced Global Memory Access

- L1 cache has a line size of 128 bytes
  - ➔ 1 cache line per 128 byte segment
  - used per default for CC 2.x only

- L2 cache has a line size of 32 bytes
  - ➔ 4 cache lines per 128 byte segment
  - used per default for CC 3.x
  - used per default for CC 5.x, no L1 present for write access

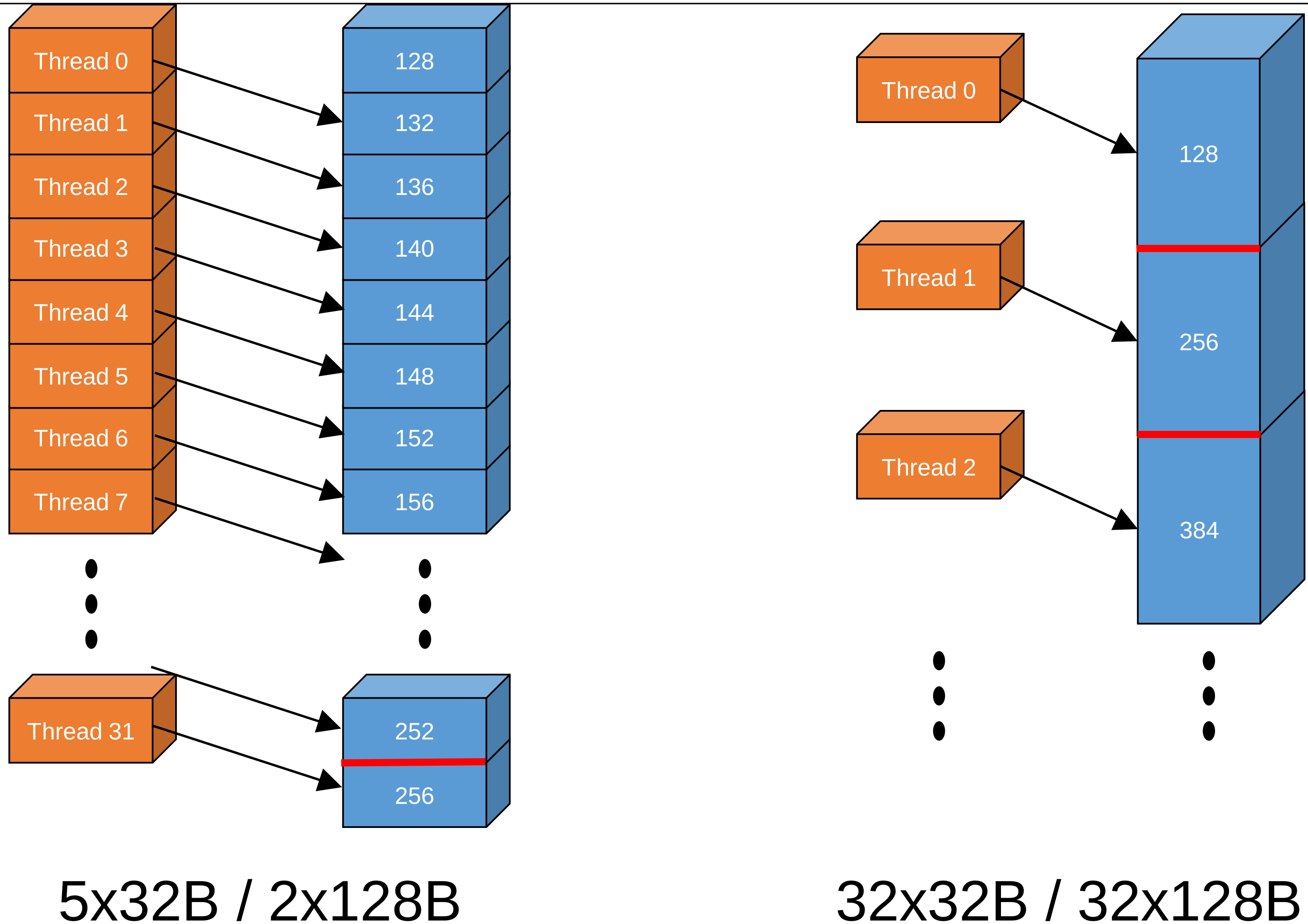
- # transactions = # different cache lines accessed by a warp in a single load instruction

Coalesced Global Memory Access



4x32B / 1x128B   <=4x32B / 1x128B   <=4x32B / 1x128B

Non-Coalesced Global Memory Access



Enabling/Disabling caches

- L1 caching for global memory can be disabled at compile time for CC 2.x
- L1 caching for global memory can be enabled at compile time for some CC 3.x GPUs
- inline assembly allows to mix global memory accesses which are cached in L1 cache or only L2 cache in the same kernel if and only if caching is enabled
- in general it is not recommended to change the default behavior

Load/Store Clustering/Batching

- use load to hide load latency
  - (non-dependent load ops only)
  - use same thread to help hide own latency (ILP)

instead of	use
LD 0 (long latency)	LD 0 (long latency)
Dependent MATH 0	LD 1 (long latency - hidden)
LD 1 (long latency)	MATH 0
Dependent MATH 1	MATH 1

- compiler handles this if enough non-dependent loads, math and registers are available
  - that's why compiling a kernel might take a while

GCC Graphics, Capture and Massively Parallel Computing

ILP vs. TLP Example

- assume that a kernel has
  - 256-thread blocks
  - 4 independent instructions for each global memory load in the thread program
  - each thread uses 10 registers
  - global loads take 200 cycles
  - ➔12 Blocks can run on each SMX 3.x

ILP has become a major factor in improving performance with 3.x and 5.x

- if a compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load
  - Only 11 can run on each SMP
  - 200/(8*4) = 7 warps needed to tolerate the memory latency
  - 11 Blocks have 88 warps, performance can be actually higher!

Control Flow

- objectives
- to understand the implications of control flow on
  - branch divergence overhead
  - SMP execution resource utilization
- to learn better ways to write code with control flow
- to understand compiler/HW predication designed to reduce the impact of control flow

How Thread Blocks are Partitioned

- thread blocks are partitioned into warps
  - thread IDs within a warp are consecutive and increasing
  - warp 0 starts with thread ID 0
- partitioning is always the same
  - thus you can use this knowledge in control flow
  - the exact size of warps may change between hardware generations (but extremely unlikely!)
- DO NOT rely on any ordering between warps
  - if there are any dependencies between threads in different warps in the same block, you must use `__syncthreads()` to get correct results

Control Flow Instructions

- main performance concern with branching is divergence
  - threads within a single warp take different paths
  - different execution paths are serialized on all GPUs
  - control paths taken by the threads in a warp are traversed one at a time until there is no more
  - no penalty for divergence between warps, as long as all threads within a warp follow the same path

Control Flow Instructions

- common case: divergence when branch condition is a function of thread ID
- example with divergence:
  - `if (threadIdx.x > 2) { … }`
  - creates two different control paths for threads in a block
  - branch granularity < warp size
  - threads 0 and 1 follow different path than the rest of the threads in the first warp
- example without divergence:
  - `if (threadIdx.x / WARP_SIZE > 2) { … }`
  - also creates two different control paths for threads in a block
  - branch granularity is a whole multiple of warp size
  - all threads in any given warp follow the same path

Parallel Reduction

- given an array of values, “reduce” them to a single value in parallel

- examples
  - sum reduction: sum of all values in the array
  - max reduction: maximum of all values in the array

- typical parallel implementation:
  - recursively halve the number of threads
  - add two values per thread

  ➔takes log(n) steps for n elements

  ➔requires n/2 threads

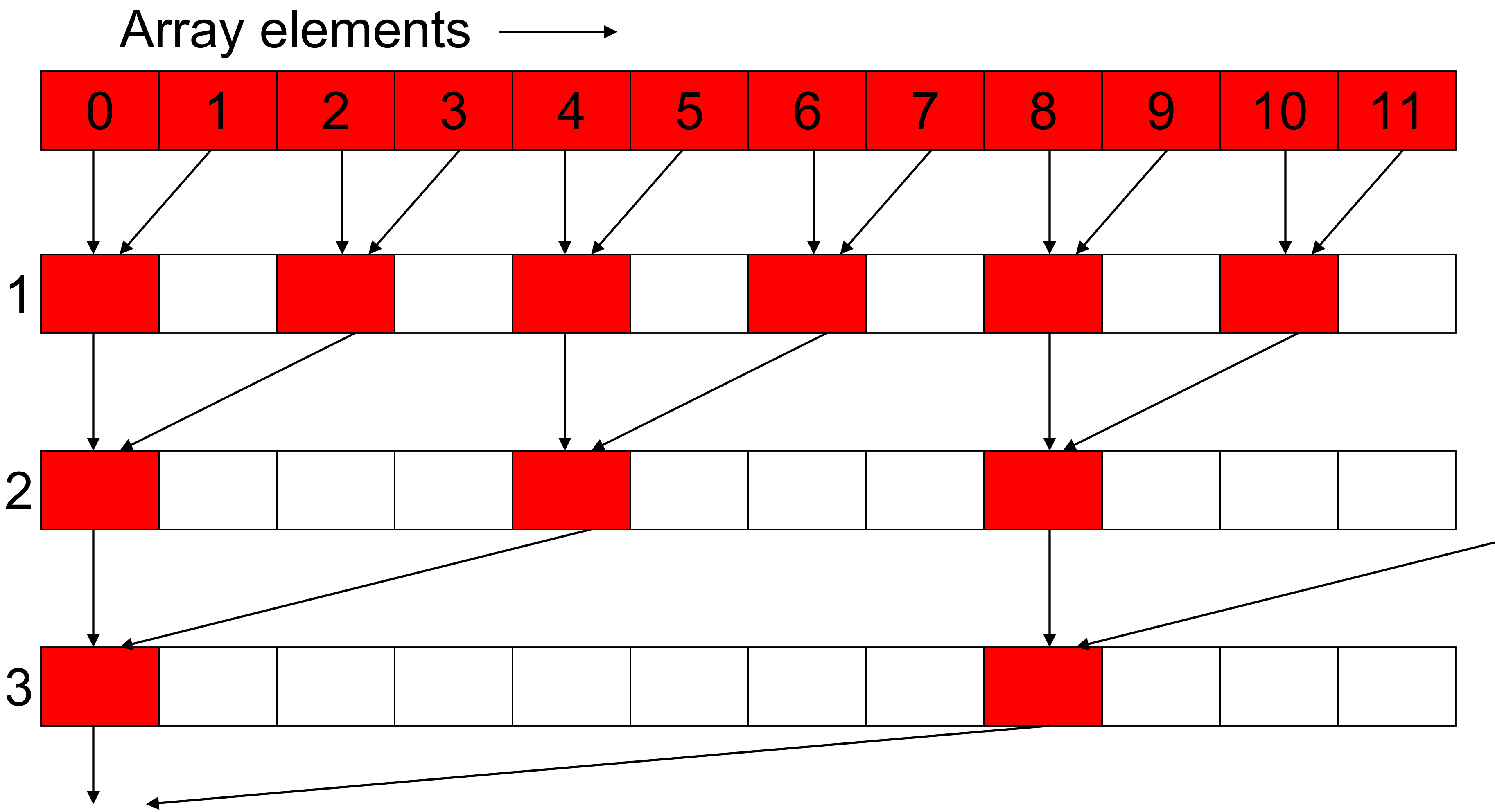
Vector Reduction Example

- assume an in-place reduction using shared memory
  - the original vector is in device global memory
  - the shared memory used to hold a partial sum vector
  - each iteration brings the partial sum vector closer to the final sum
  - the final solution will be in element 0

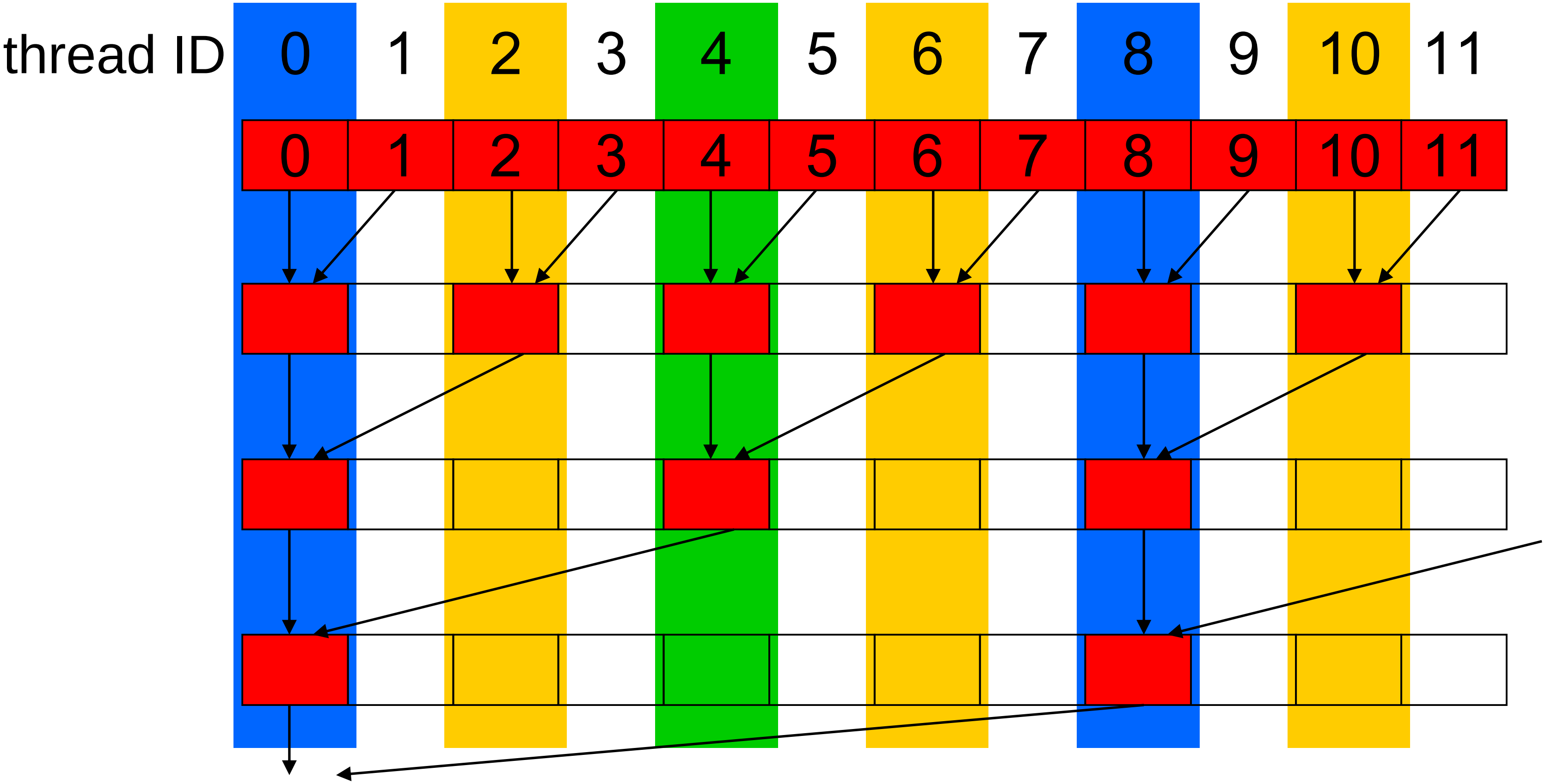
A Simple Implementation

```
// create array and load data into shared memory
__shared__ float partialSum[];
// actual load omitted

// [...]

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

Vector Reduction with Bank Conflicts



Vector Reduction w/ Branch Divergence



Some Observations

TECHNISCHE UNIVERSITÄT DARMSTADT

- in each iteration, two control flow paths will be sequentially traversed for each warp
  - threads that perform addition and threads that do not
  - threads that do not perform addition may cost extra cycles depending on the implementation of divergence

- no more than half of threads will be executing at any time
  - all odd index threads are disabled right from the beginning!
  - on average, less than ¼ of the threads will be activated for all warps over time
  - after the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence
  - this can go on for up to 4 more iterations ($512/32=16= 2^4$), where each iteration only has one thread activated until all warps retire

GCC Graphics, Capture and Massively Parallel Computing

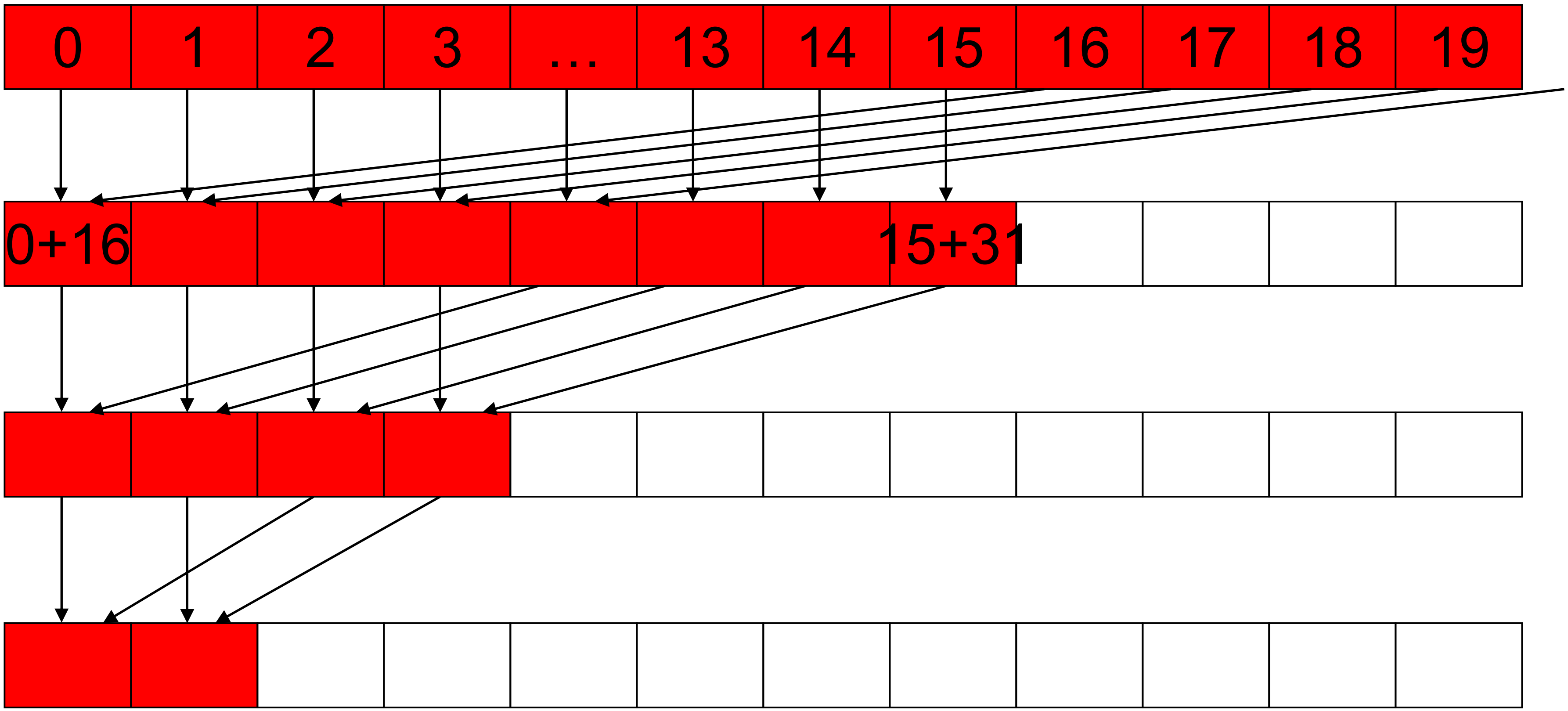
Shortcomings of the Implementation

```
// create array and load data into shared memory
__shared__ float partialSum[];
// actual load omitted

//[...]

unsigned int t = threadIdx.x;
unsigned int stride;
for (stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    // BAD interleaved decisions --> divergence
    if (t % (2 * stride) == 0)
        // BAD bank conflicts in memory access
        partialSum[t] += partialSum[t + stride];
}
```

Better Implementation

```
// create array and load data into shared memory
__shared__ float partialSum[]
// actual load omitted

// [...]

unsigned int t = threadIdx.x;
unsigned int stride;
for (stride = blockDim.x; stride > 1;) {
    __syncthreads();
    stride >> 1;
    if (t < stride)
        partialSum[t] += partialSum[t + stride];
}
```

No Divergence until ≤ 16 Sub-Sums



Some Observations

- now only the last 5 iterations will have divergence
- entire warps will be shut down as iterations progress
  - for a 512-thread block, 4 iterations to shut down all but one warp in each block
  - better resource utilization, will likely retire warps and thus blocks faster
- recall, no bank conflicts either

(Dangerous?) Further Refinement

- for last 6 loops only one warp active (i.e., tid's 0..31)
  - shared reads & writes SIMD synchronous within a warp
  - skip `__syncthreads()` and unroll last iterations

```
// make sure that warp size is still 32
assert(WARP_SIZE == 32);

unsigned int tid = threadIdx.x;
for (unsigned int d = n >> 1; d > 32; d >>= 1) {
    __syncthreads();
    if (tid < d)
        shared[tid] += shared[tid + d];
}
// [...]
```

Unsafe Further Refinement

- for last 6 loops only one warp active (i.e., tid’s 0..31)
  - shared reads & writes SIMD synchronous within a warp
  - skip `__syncthreads()` and unroll last iterations

```
// [...]
__syncthreads();
if (tid <= 32) {  // unroll last 6 steps
    shared[tid] += shared[tid + 32];
    shared[tid] += shared[tid + 16];
    shared[tid] += shared[tid + 8];
    shared[tid] += shared[tid + 4];
    shared[tid] += shared[tid + 2];
    shared[tid] += shared[tid + 1];
}
```

Safe Further Refinement

- for last 6 loops only one warp active (thread IDs 0..31)
  - threads in a single warp can communicate without shared memory using warp shuffling

```
// [...]
__syncthreads();
if (tid <= 32) {  // unroll last 6 steps
    float tmp = shared[tid] + shared[tid + 32];
    tmp += __shfl_xor(tmp, 16);
    tmp += __shfl_xor(tmp, 8);
    tmp += __shfl_xor(tmp, 4);
    tmp += __shfl_xor(tmp, 2);
    tmp += __shfl_xor(tmp, 1);
    shared[tid] = tmp;
}
```

Advanced: Predicated Execution

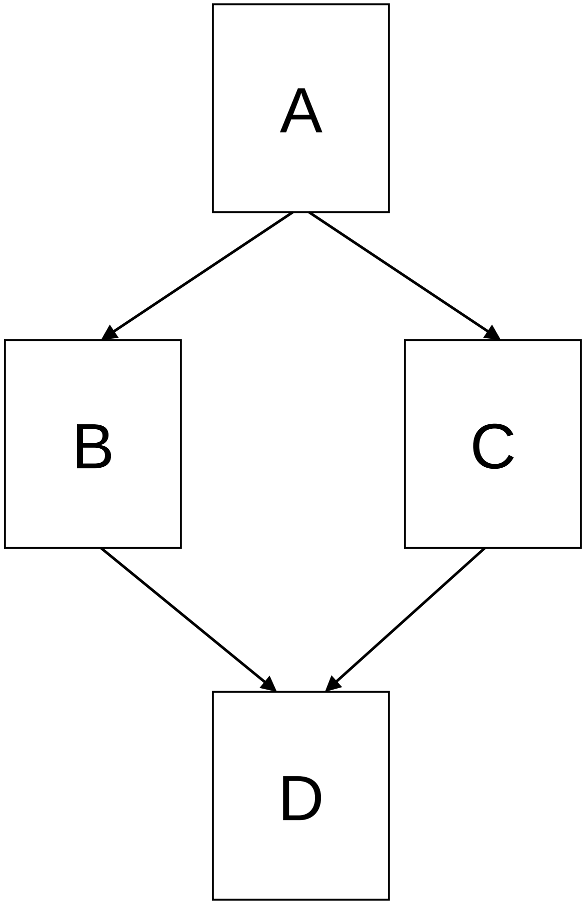
- predicated statement
  `<p1> LDR r1,r2,0`
  - if p1 is TRUE, instruction executes normally
  - if p1 is FALSE, instruction treated as NOP

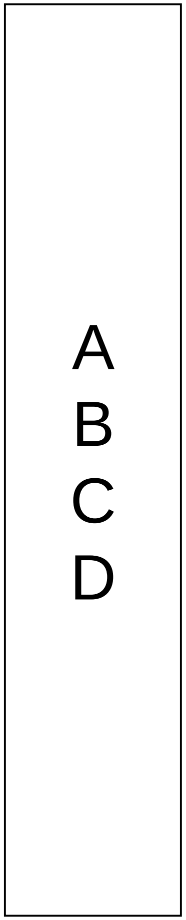
- example

```
:
if (x == 10)
c = c + 1;
:
```

```
      :
      LDR r5, x
      p1 <- r5 eq 10
<p1>  LDR  r1 <- C
<p1>  ADD  r1, r1, 1
<p1>  STR  r1 -> C
      :
```

Predication Very Helpful for if-else





Predication Very Helpful for if-else

- extra instructions will be issued at code execution
- there is, however, no branch divergence
- scheduler can optimize execution order

```
     :
     p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p1> inst 2 from B
<p1> :
     :
<p2> inst 1 from C
<p2> inst 2 from C
     :
```

```
     :
     p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p2> inst 1 from C
<p1> inst 2 from B
<p2> inst 2 from C
<p1> :
     :
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Instruction Predication

- comparison instructions set condition codes (CC)
- instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)

- compiler tries to predict if a branch condition is likely to produce many divergent warps
  - if guaranteed not to diverge: only predicates if < 4 instructions
  - if not guaranteed: only predicates if < 7 instructions
  - may replace branches with instruction predication

- all predicated instructions take execution cycles
  - those with false conditions don’t write their output or invoke memory loads and stores
  - saves branch instructions, so can be cheaper than serializing divergent paths