King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 14: CUDA Memories, Pt. 1

Markus Hadwiger, KAUST

Reading Assignment #7 (until Oct 20)

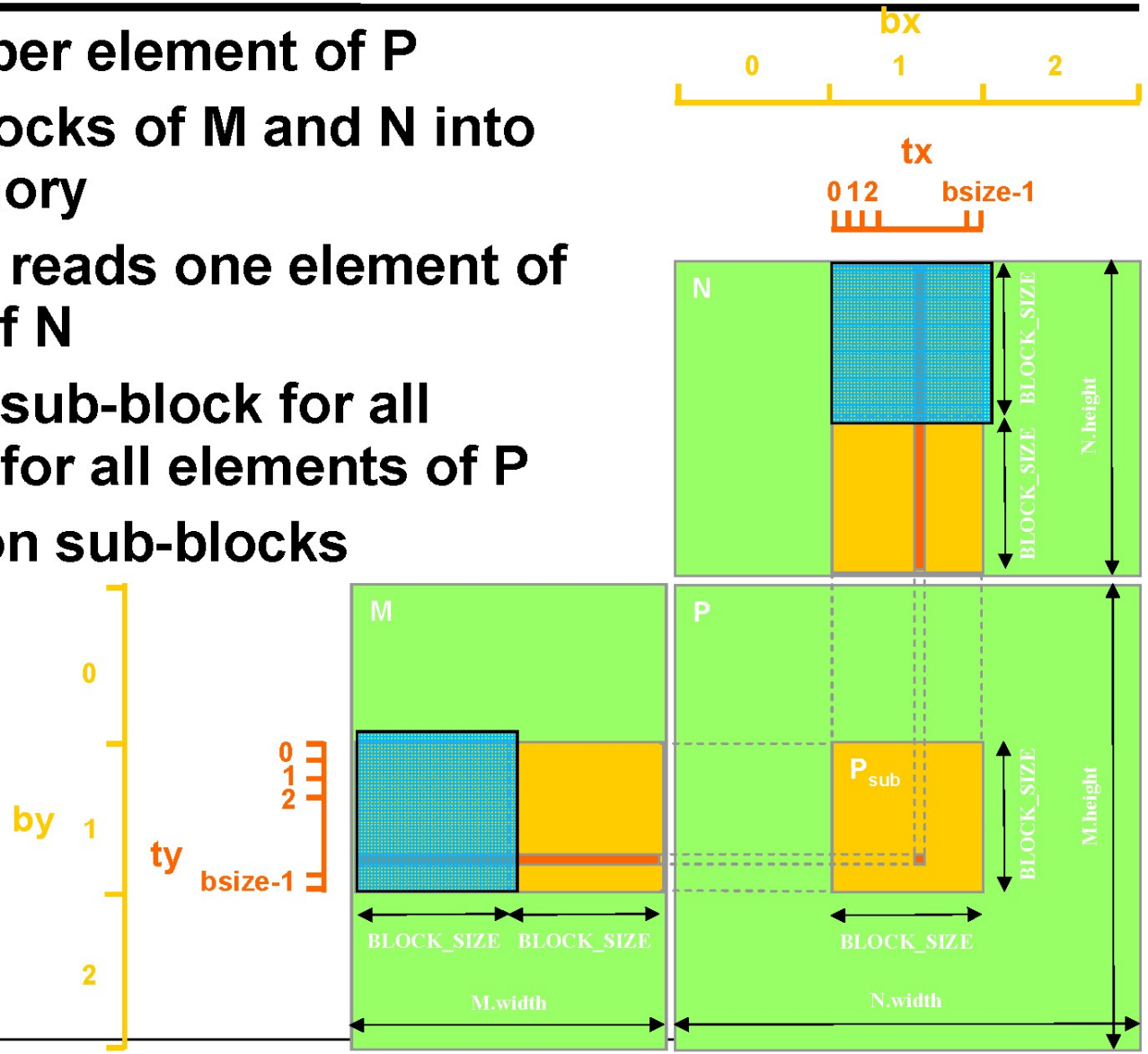
Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 5** (*Memory architecture and data locality*)

Code Example #2: Matrix Multiply

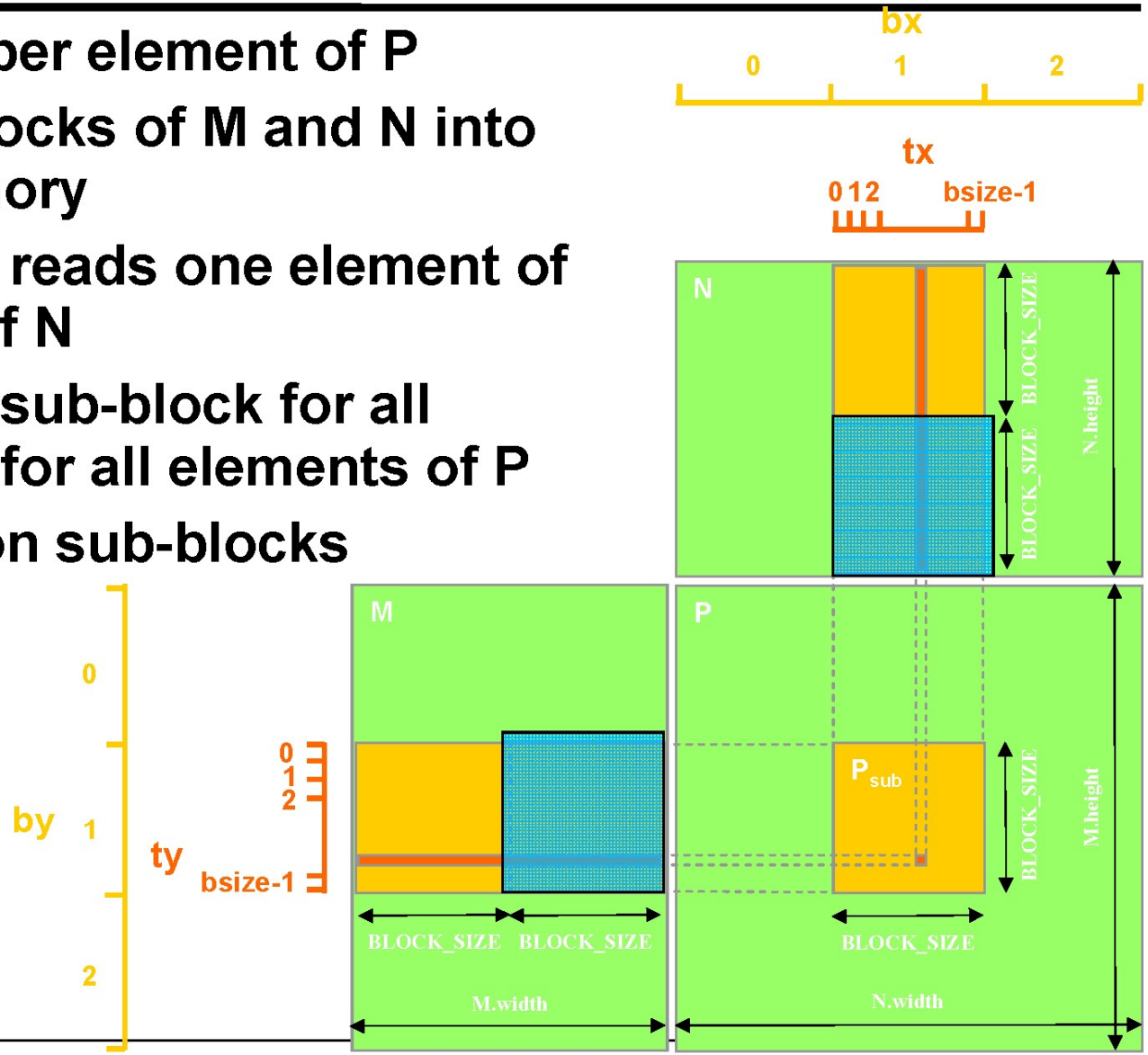
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks

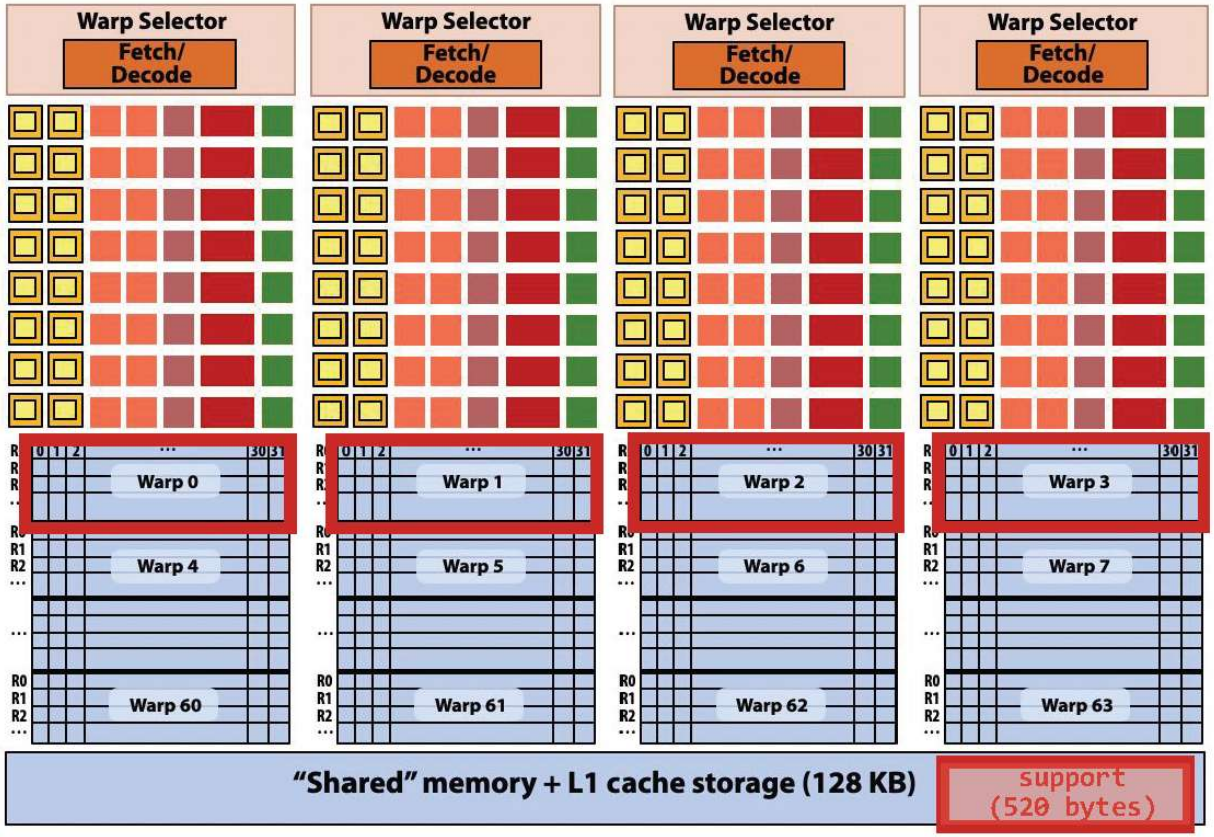


Example: Matrix Multiplication (4)

```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
      __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
      __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

      int bx = blockIdx.x; int tx = threadIdx.x;
      int by = blockIdx.y; int ty = threadIdx.y;

      int col = bx * BLOCK_SIZE + tx;
      int row = by * BLOCK_SIZE + ty;

      float out = 0.0f;
      for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {

          blockA[ ty ][ tx ] = matA[ row * w +   m * BLOCK_SIZE + tx       ];
          blockB[ ty ][ tx ] = matB[ col     + ( m * BLOCK_SIZE + ty ) * w ];
          __syncthreads();

          for ( int k = 0; k < BLOCK_SIZE; k++ ) {
              out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
          }
          __syncthreads();
      }

      matC[ row * w + col ] = out;
}
```

Caveat: for brevity, this code assumes matrix sizes are a multiple of the block size (either because they really are, or because padding is used; otherwise guard code would need to be added)

Running on a V100 (Volta) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

SM core operation each clock:

- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

Limits in CUDA Programming Guide

	Compute Capability													
Technical Specifications	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6	8.7	8.9	9.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	32		16	128	32	16	128	16	128					
Maximum dimensionality of grid of thread blocks	3													
Maximum x -dimension of a grid of thread blocks [thread blocks]	$2^{31}-1$													
Maximum y- or z-dimension of a grid of thread blocks	65535													
Maximum dimensionality of thread block	3													
Maximum x- or y-dimensionality of a block	1024													
Maximum z-dimension of a block	64													
Maximum number of threads per block	1024													
Warp size	32													
Maximum number of resident blocks per SM	32								16	32	16		24	32
Maximum number of resident warps per SM	64								32	64	48			64
Maximum number of resident threads per SM	2048								1024	2048	1536			2048

Limits in CUDA Programming Guide

Chapter 20.2
(CUDA 13,
Oct 2, 2025

	Compute Capability								
Technical Specifications	7.5	8.0	8.6	8.7	8.9	9.0	10.0	11.0	12.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	128								
Maximum dimensionality of grid of thread blocks	3								
Maximum x -dimension of a grid of thread blocks	2^{31}-1								
Maximum y- or z-dimension of a grid of thread blocks	65535								
Maximum dimensionality of thread block	3								
Maximum x- or y-dimensionality of a block	1024								
Maximum z-dimension of a block	64								
Maximum number of threads per block	1024								
Warp size	32								
Maximum number of resident blocks per SM	16	32	16		24	32		24	
Maximum number of resident warps per SM	32	64	48			64		48	
Maximum number of resident threads per SM	1024	2048	1536			2048		1536	
Number of 32-bit registers per SM	64 K								
Maximum number of 32-bit registers per thread block	64 K								
Maximum number of 32-bit registers per thread	255								
Maximum amount of shared memory per SM	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB			100 KB
Maximum amount of shared memory per thread block[27]	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB			99 KB
Number of shared memory banks	32								
Maximum amount of local memory per thread	512 KB								
Constant memory size	64 KB								

Limits in CUDA Programming Guide

Chapter 20.2
(CUDA 13,
Oct 2, 2025

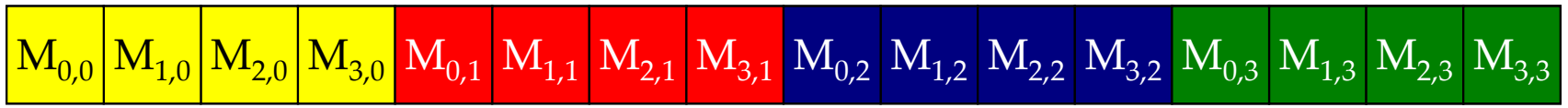
	Compute Capability								
Technical Specifications	7.5	8.0	8.6	8.7	8.9	9.0	10.0	11.0	12.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	128								
Maximum dimensionality of grid of thread blocks	3								
Maximum x -dimension of a grid of thread blocks	2^{31}-1								
Maximum y- or z-dimension of a grid of thread blocks	65535								
Maximum dimensionality of thread block	3								
Maximum x- or y-dimensionality of a block	1024								
Maximum z-dimension of a block	64								
Maximum number of threads per block	1024								
Warp size	32								
Maximum number of resident blocks per SM	16	32	16		24	32		24	
Maximum number of resident warps per SM	32	64	48			64		48	
Maximum number of resident threads per SM	1024	2048	1536			2048		1536	
Number of 32-bit registers per SM	64 K								
Maximum number of 32-bit registers per thread block	64 K								
Maximum number of 32-bit registers per thread	255								
Maximum amount of shared memory per SM	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB			100 KB
Maximum amount of shared memory per thread block[27]	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB			99 KB
Number of shared memory banks	32								
Maximum amount of local memory per thread	512 KB								
Constant memory size	64 KB								

What About Memory Performance? (more to come later…)

Memory Layout of a Matrix in C

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

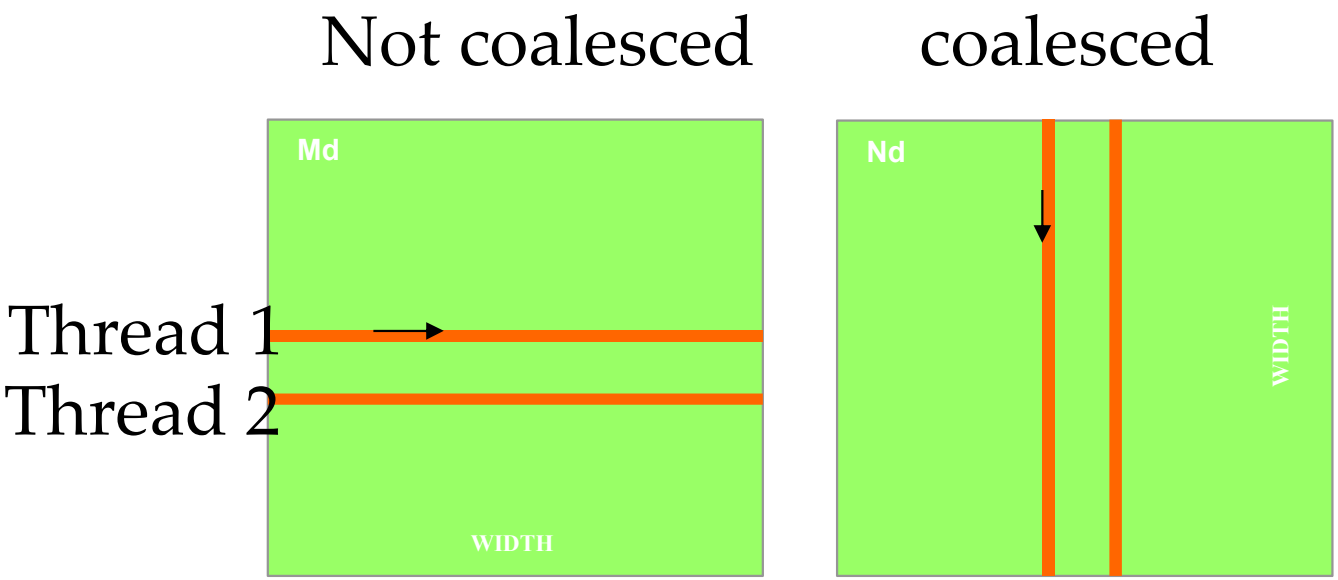
M



row-major order !

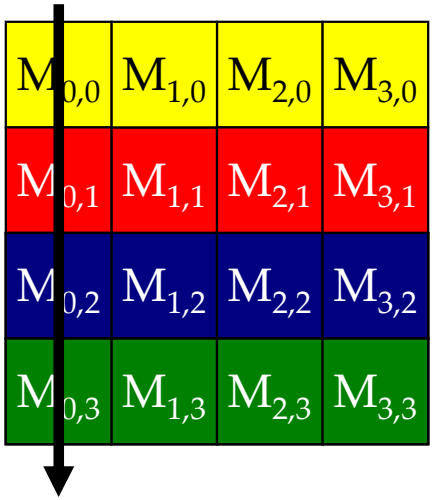
Memory Coalescing

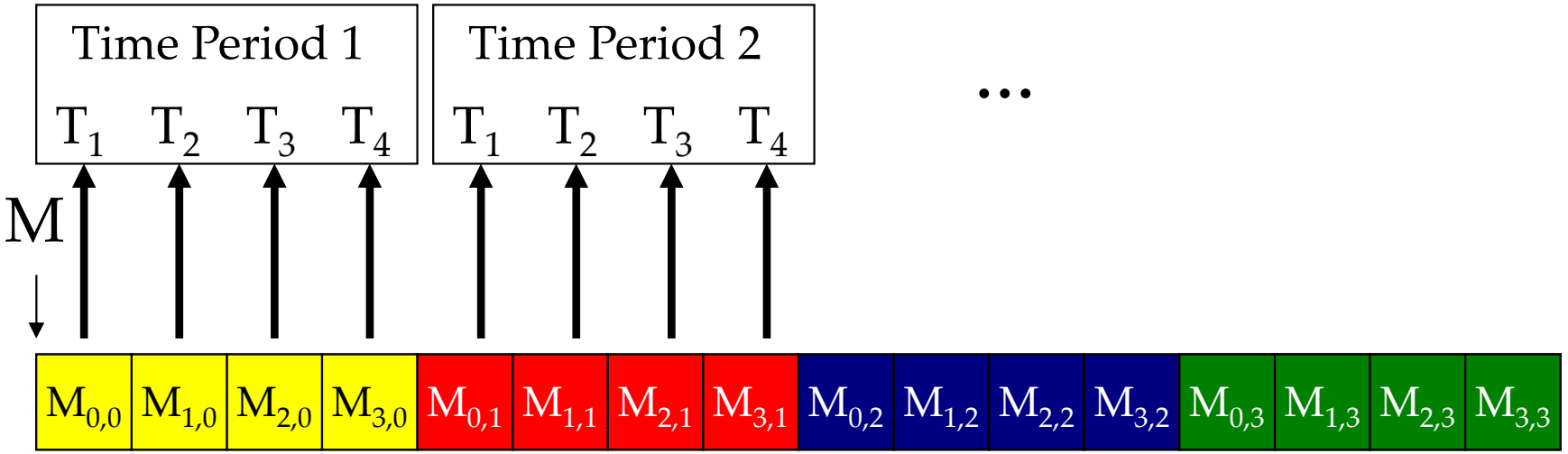
- When accessing global memory, peak performance utilization occurs when all threads in a half warp (full warp on Fermi+) access continuous memory locations.
- Requirements relaxed on >=1.2 devices; L1 cache on Fermi!



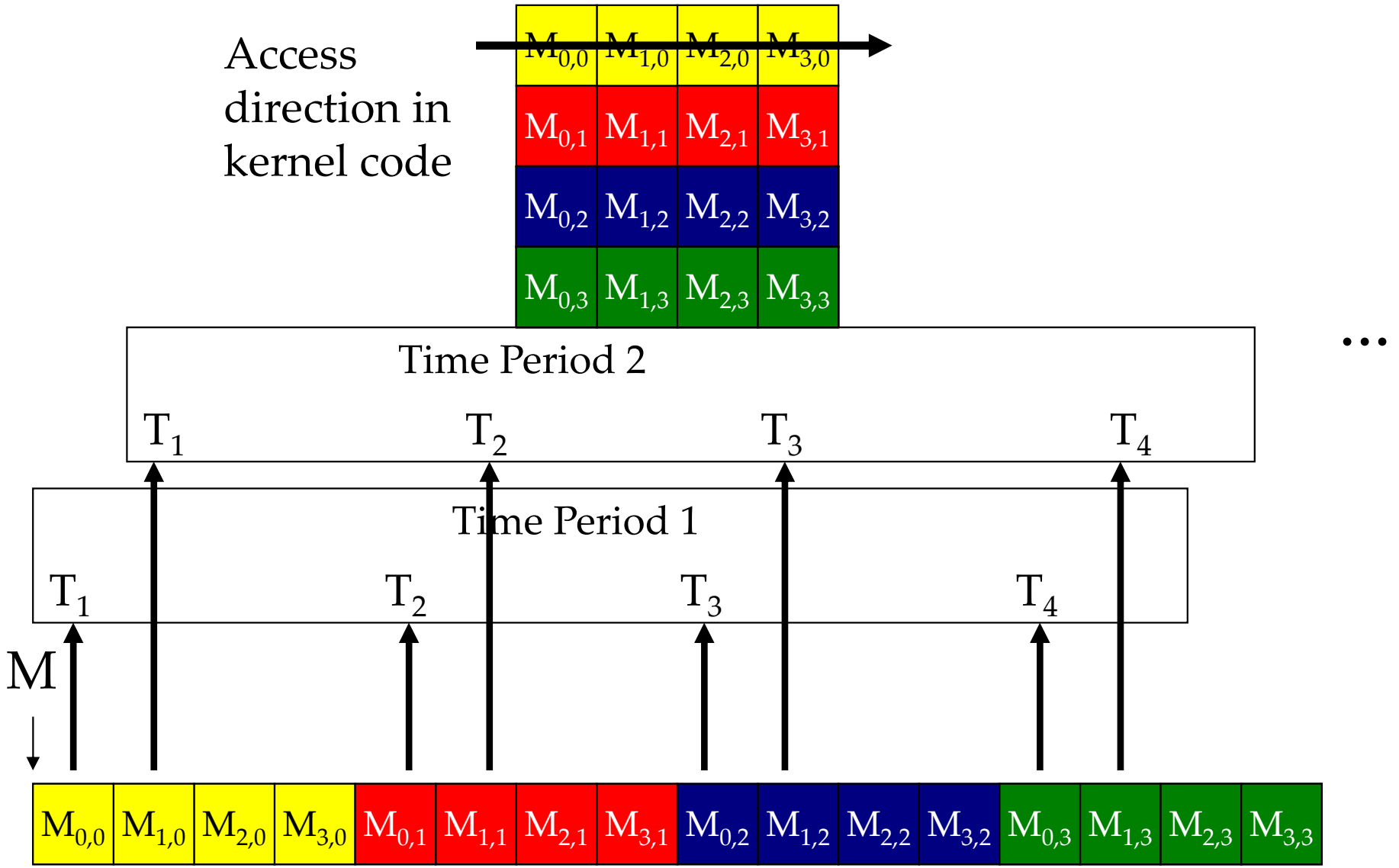
Memory Layout of a Matrix in C

Access direction in kernel code





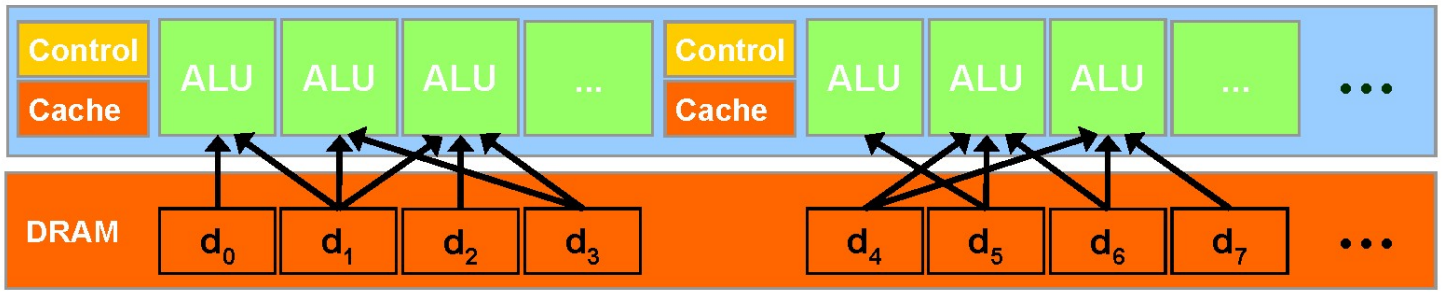
Memory Layout of a Matrix in C



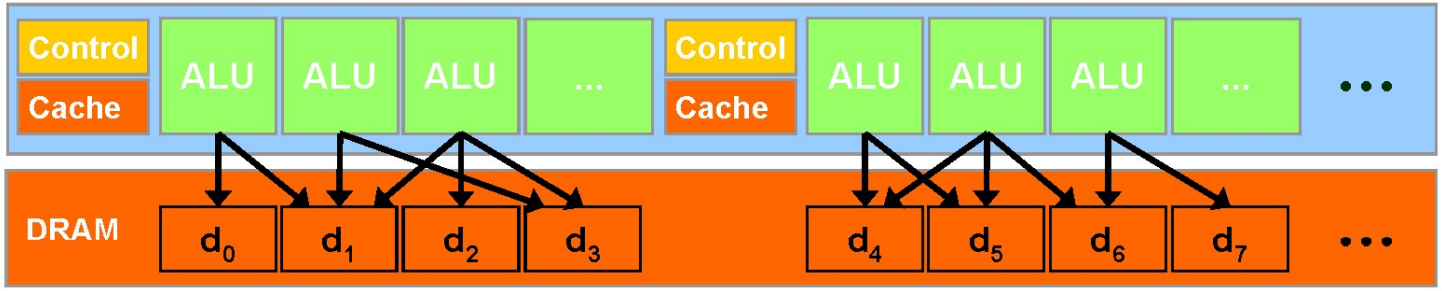
CUDA Memory

CUDA Highlights: Scatter

- **CUDA provides generic DRAM memory addressing**
  - Gather:



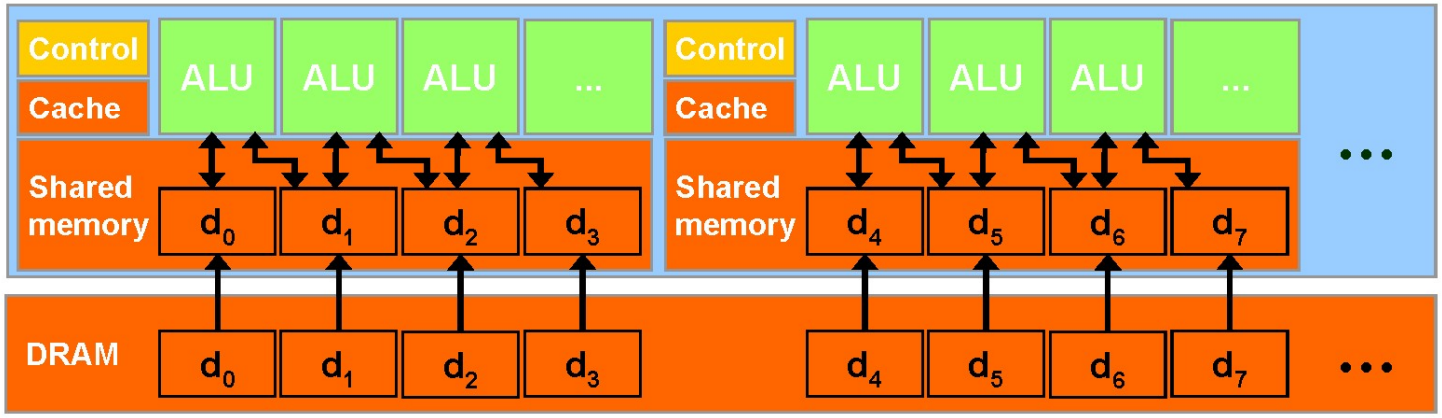
  - And scatter: no longer limited to write one pixel



→ **More programming flexibility**

CUDA Highlights: On-Chip Shared Memory

- **CUDA enables access to a parallel on-chip shared memory for efficient inter-thread data sharing**

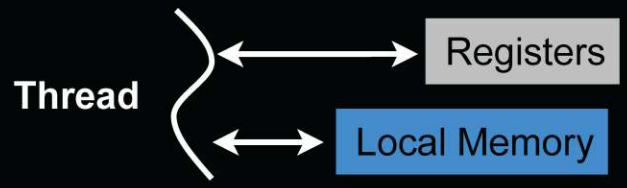


→ **Big memory bandwidth savings**

CUDA Memory: Overview

Kernel Memory Access

- **Per-thread**

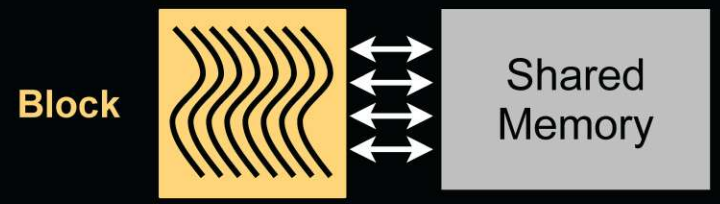


On-chip

Off-chip, ~~uncached~~

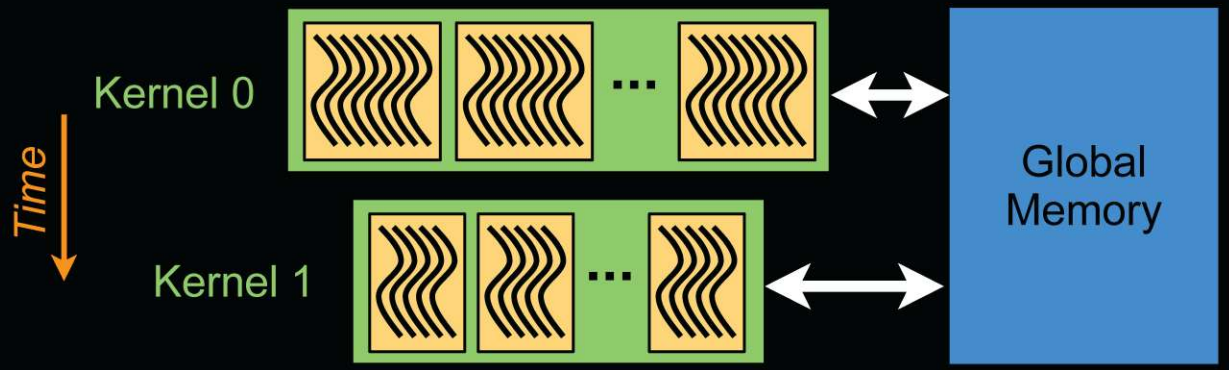
now cached in L1!
(Fermi or newer)

- **Per-block**



- On-chip, small
- Fast

- **Per-device**



- Off-chip, large
- ~~Uncached~~
- Persistent across kernel launches
- Kernel I/O

now cached in L2!
(Fermi or newer)

Memory and Cache Types

Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

CUDA Memory: Shared Memory

Memory and Cache Types

Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

Shared Memory Allocation

- **2 modes**
- **Static size within kernel**

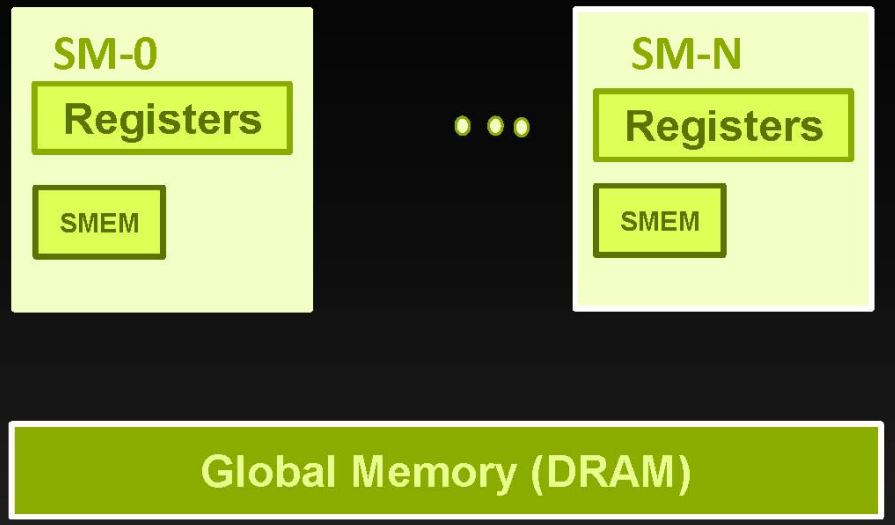
```
__shared__ float vec[256];
```

- **Dynamic size when calling the kernel**

```
// in main
int VecSize = MAX_THREADS * sizeof(float4);
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, …);

// declare as extern within kernel
extern __shared__ float vec[];
```

Shared Memory

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion



Shared Memory/L1 Sizing

- Shared memory and L1 use the same 64KB
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: ~~cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()~~

later: use *carveout*

use cudaFuncSetAttribute()

- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)
- Large SMEM can improve performance when:
  - Occupancy is limited by SMEM

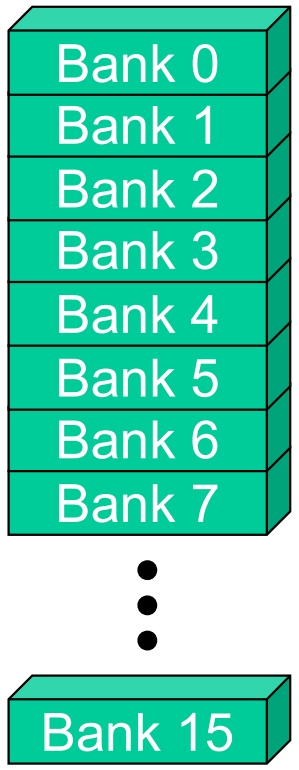
Shared Memory

- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Organization:
  - 32 banks, 4-byte (or 8-byte) banks
  - Successive words accessed through different banks

8-byte bank size is/was only available on Kepler !
(using the now deprecated cudaSharedMemConfig()
all other architectures so far use a fixed bank size of 4 bytes!

Parallel Memory Architecture

- 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



now: 32 banks!

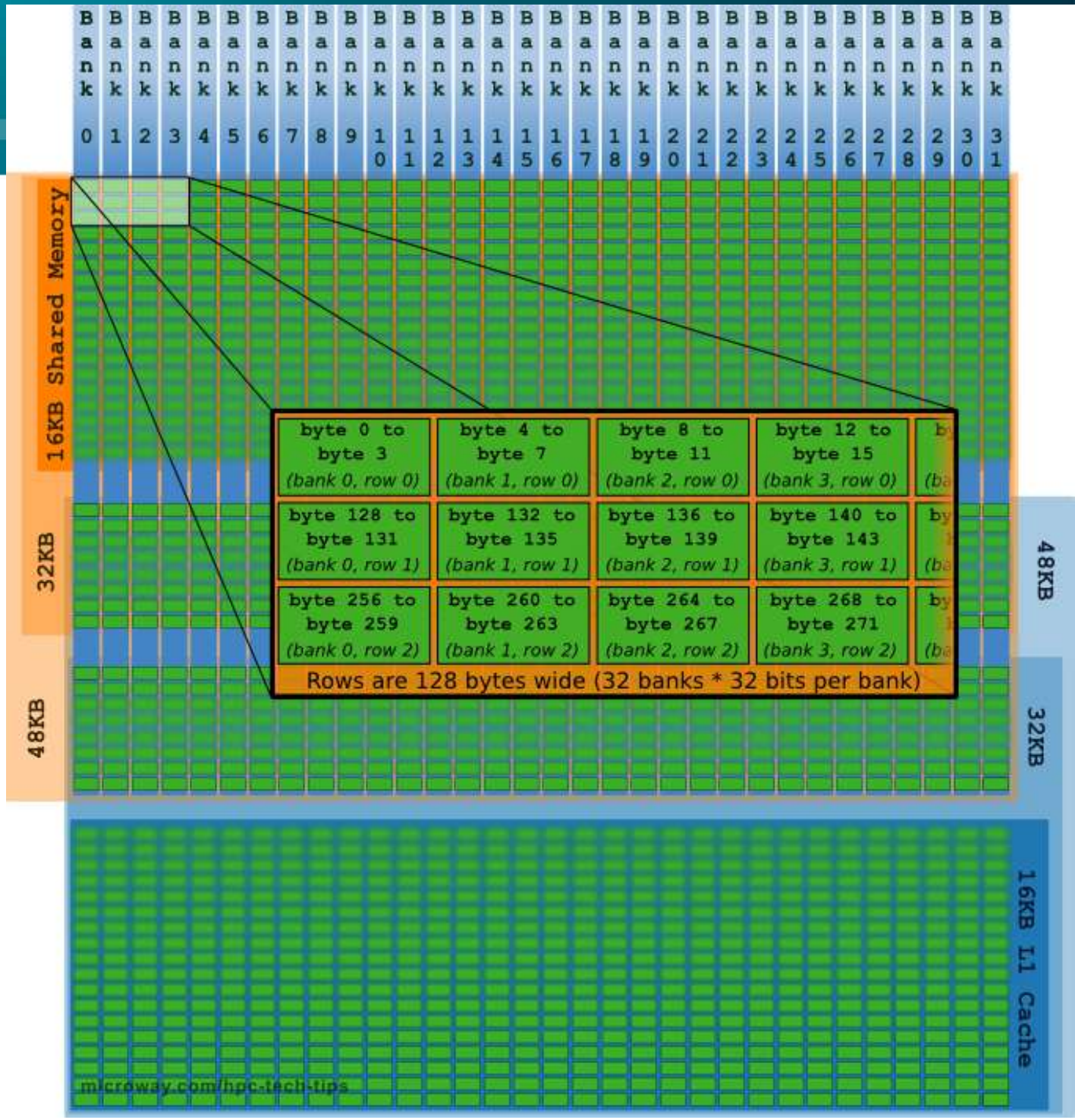
Memory Banks

all architectures
since Fermi:

32 banks

bank size:
4 bytes / bank

(only on Kepler:
optionally
configurable to
8 bytes / bank)

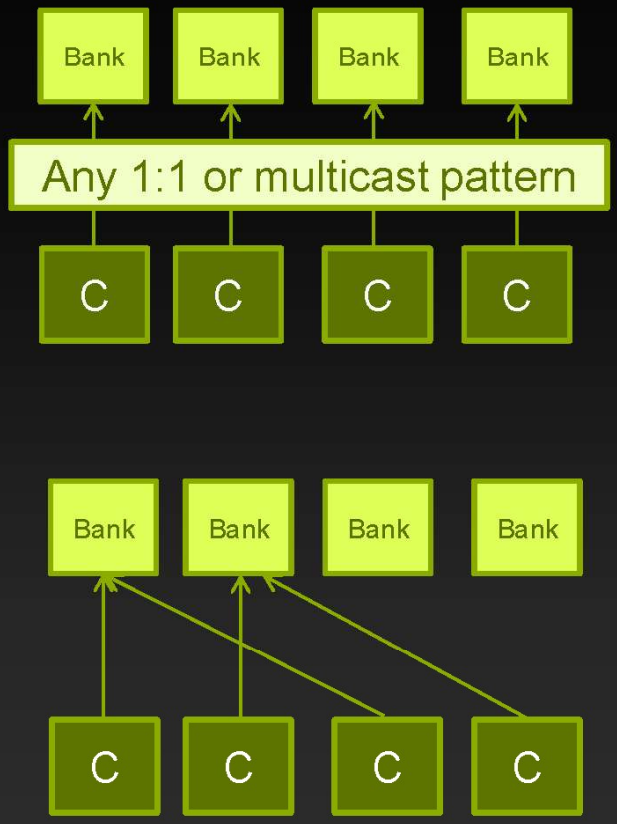


Shared Memory

- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Performance:
  - smem accesses are issued per warp
  - Throughput is 4 (or 8) bytes per bank per clock per multiprocessor
  - serialization: if *N* threads of 32 access different words in the same bank, *N* accesses are executed serially
  - multicast: *N* threads access the same word in one fetch
    - Could be different bytes within the same word

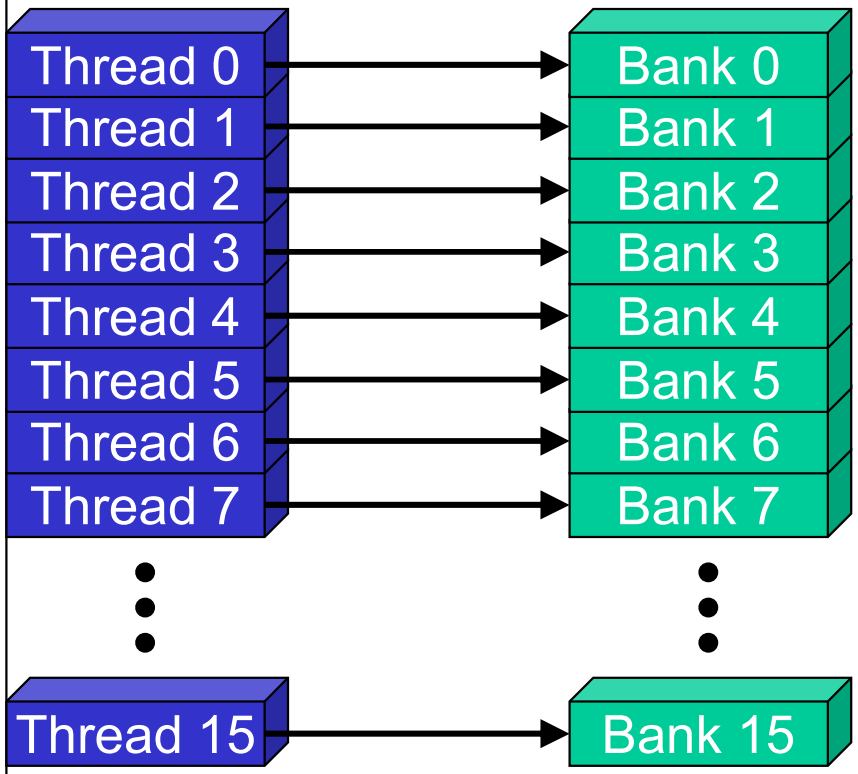
Shared Memory Organization

- Organized in 32 independent banks
- Optimal access: no two words from same bank
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize

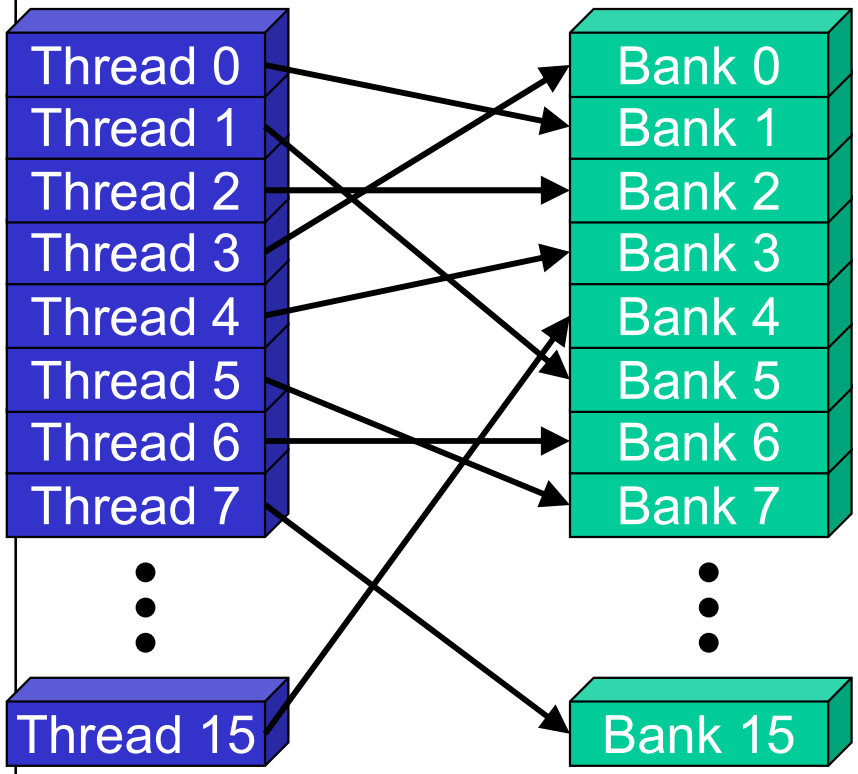


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1



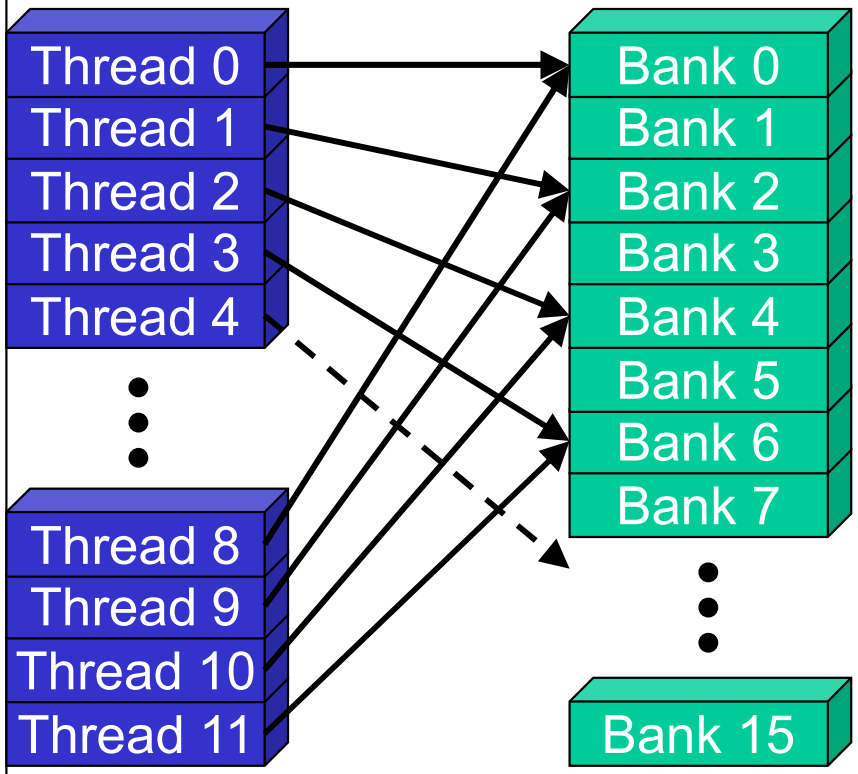
- No Bank Conflicts
  - Random 1:1 Permutation



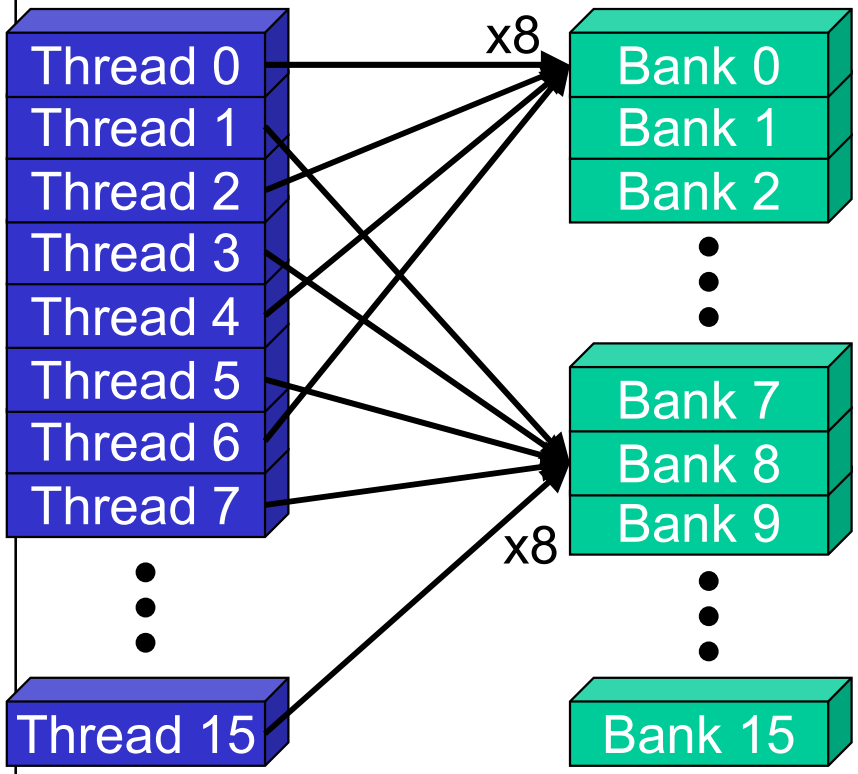
now: 32 banks!

Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



now: 32 banks!

How addresses map to banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks now: 32 banks!
  - So bank = address % 16 now: % 32
  - Same as the size of a half-warp now: full warps
    - No bank conflicts between different half-warps, only within a single half-warp

Fermi and newer have 32 banks,
considers full warps instead of half warps!

Shared Memory Bank Conflicts

- **Shared memory is as fast as registers if there are no bank conflicts**

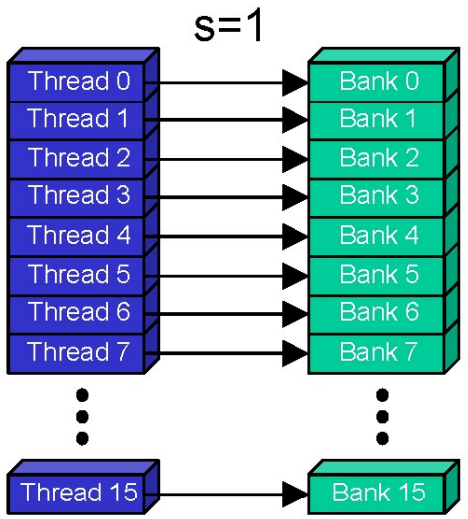
- **The fast case:**
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- **The slow case:**
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

full warps instead of half warps on Fermi and newer!

Linear Addressing

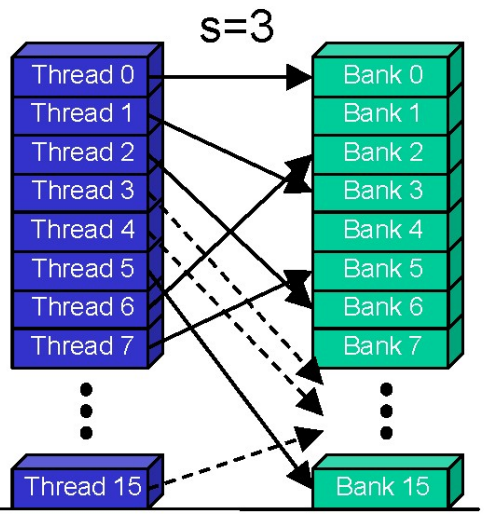
- **Given:**

```
__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**
  - 16 on G80, so `s` must be odd

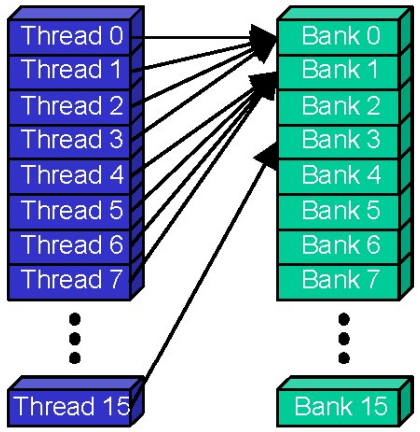
now: 32 but same rule: s must be odd!



Data Types and Bank Conflicts

- **This has no conflicts if type of `shared` is 32-bits:**

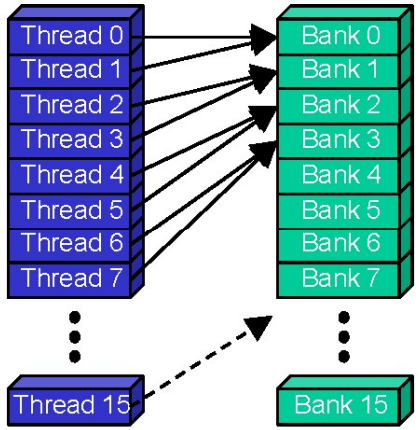
```
foo = shared[baseIndex + threadIdx.x]
```



- **But not if the data type is smaller**
  - 4-way bank conflicts:

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

not true on Fermi+, because of multi-cast!



  - 2-way bank conflicts:

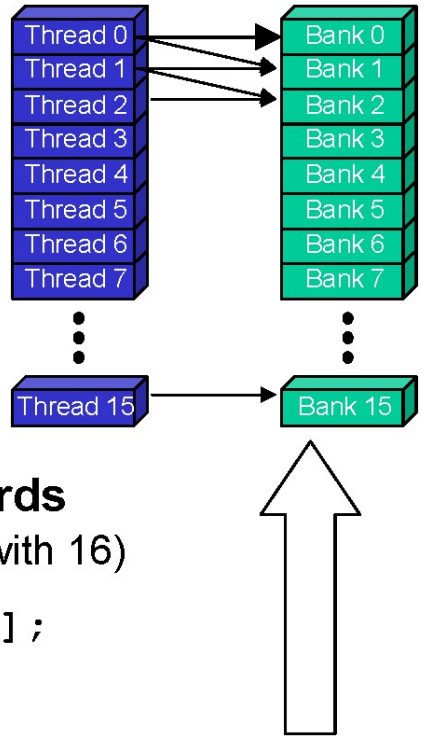
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```

not true on Fermi+, because of multi-cast!

Structs and Bank Conflicts

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

```
struct vector { float x, y, z; };
struct myType {
    float f;
    int c;
};
__shared__ struct vector vectors[64];
__shared__ struct myType myTypes[64];
```



- **This has no bank conflicts for vector; struct size is 3 words**
  - 3 accesses per thread, contiguous banks (no common factor with 16)

```
struct vector v = vectors[baseIndex + threadIdx.x];
```

- **This has 2-way bank conflicts for myType;**
  (each bank will be accessed by 2 threads simultaneously)

```
struct myType m = myTypes[baseIndex + threadIdx.x];
```

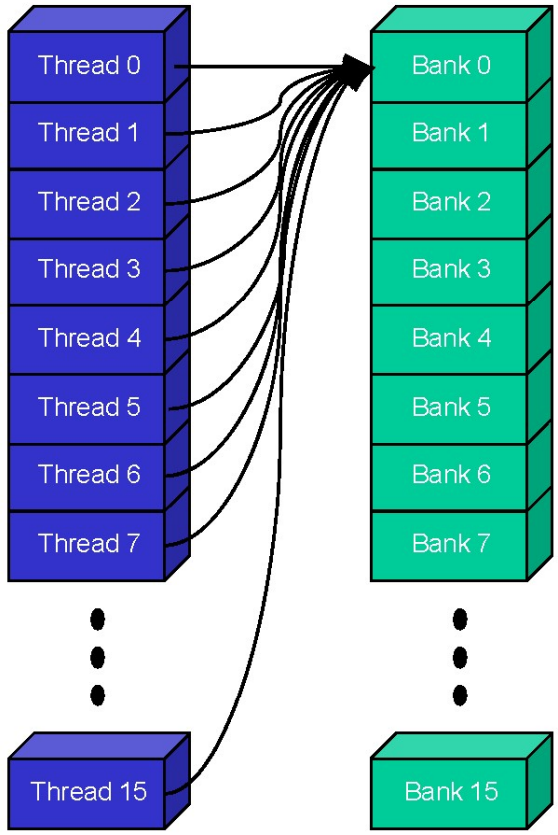
Broadcast on Shared Memory

- **Each thread loads the same element – no bank conlict**

```
x = shared[0];
```

- **Will be resolved implicitly**

multi-cast on Fermi and newer!

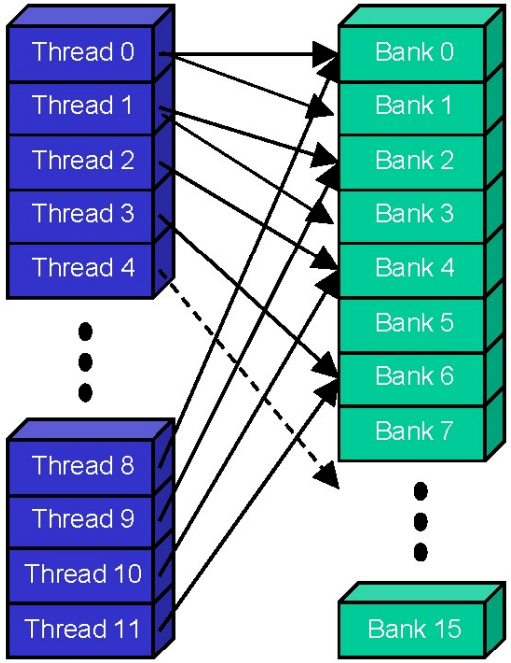


Common Array Bank Conflict Patterns 1D

- **Each thread loads 2 elements into shared mem:**
  - 2-way-interleaved loads result in 2-way bank conflicts:

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

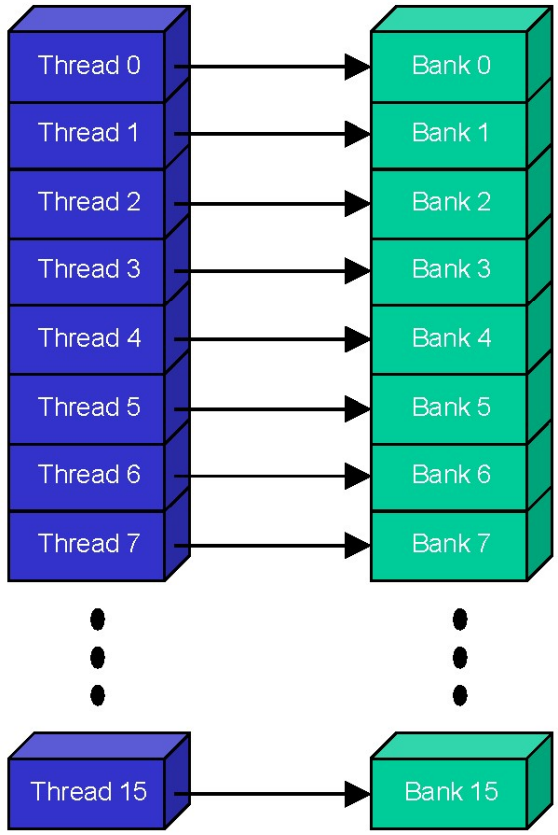
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects



A 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];
```



NVIDIA Architectures (since first CUDA GPU)

Tesla [CC 1.x]: 2007-2009

- G80, G9x: 2007 (Geforce 8800, ...)
  GT200: 2008/2009 (GTX 280, ...)

Fermi [CC 2.x]: 2010 (2011, 2012, 2013, …)

- GF100, ... (GTX 480, ...)
  GF104, ... (GTX 460, ...)
  GF110, ... (GTX 580, ...)

Kepler [CC 3.x]: 2012 (2013, 2014, 2016, …)

- GK104, ... (GTX 680, ...)
  GK110, ... (GTX 780, GTX Titan, ...)

Maxwell [CC 5.x]: 2015

- GM107, ... (GTX 750Ti, ...); [Nintendo Switch]
  GM204, ... (GTX 980, Titan X, ...)

Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, …)

- GP100 (Tesla P100, ...)
- GP10x: x=2,4,6,7,8, ...
  (GTX 1060, 1070, 1080, Titan X *Pascal*, Titan Xp, ...)

Volta [CC 7.0, 7.2]: 2017/2018

- GV100, ...
  (Tesla V100, Titan V, Quadro GV100, ...)

Turing [CC 7.5]: 2018/2019

- TU102, TU104, TU106, TU116, TU117, ...
  (Titan RTX, RTX 2070, 2080 (Ti), GTX 1650, 1660, ...)

Ampere [CC 8.0, 8.6, 8.7, 8.8]: 2020

- GA100, GA102, GA104, GA106, ...; [Nintendo Switch 2]
  (A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

Hopper [CC 9.0], **Ada Lovelace** [CC 8.9]: 2022/23

- GH100, AD102/103/104/106/107, ...
  (H100, H200, GH200, L20, L40, L40S, L2, L4,
  RTX 4080 (12/16 GB), RTX 4090, RTX 6000 (Ada), ...)

Blackwell [CC 10.0, 10.1(→11.0), 10.3, 12.0, 12.1]: 2024/2025

- GB100, GB200, GB202/203/205/206/207, G10, ...
  (RTX 5080/5090, HGX B200/B300, GB200/GB300 NVL72,
  RTX 4000/5000/6000 PRO Blackwell, B40, ...)

see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

Compute Capab. 5.x (Maxwell)

K.4.3. Shared Memory

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle.

A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank). In that case, for read accesses, the word is broadcast to the requesting threads and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).

Figure 22 shows some examples of strided access.

Figure 23 shows some examples of memory read accesses that involve the broadcast mechanism.

Compute Capab. 6.x (Pascal)

K.5.3. Shared Memory

Shared memory behaves the same way as in devices of compute capability 5.x (See Shared Memory).

Compute Capab. 7.x (Volta/Turing) [1]

K.6.4. Shared Memory

Similar to the Kepler architecture, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *Volta* architecture (compute capability 7.0), the unified data cache has a size of 128 KB, and the shared memory capacity can be set to 0, 8, 16, 32, 64 or 96 KB. For the *Turing* architecture (compute capability 7.5), the unified data cache has a size of 96 KB, and the shared memory capacity can be set to either 32 KB or 64 KB. Unlike *Kepler*, the driver automatically configures the shared memory capacity for each kernel to avoid shared memory occupancy bottlenecks while also allowing concurrent execution with already launched kernels where possible. In most cases, the driver's default behavior should provide optimal performance.

Compute Capab. 7.x (Volta/Turing) [2]

20.6.4. Shared Memory

The amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *Volta* architecture (compute capability 7.0), the unified data cache has a size of 128 KB, and the shared memory capacity can be set to 0, 8, 16, 32, 64 or 96 KB. For the *Turing* architecture (compute capability 7.5), the unified data cache has a size of 96 KB, and the shared memory capacity can be set to either 32 KB or 64 KB. Unlike Kepler, the driver automatically configures the shared memory capacity for each kernel to avoid shared memory occupancy bottlenecks while also allowing concurrent execution with already launched kernels where possible. In most cases, the driver's default behavior should provide optimal performance.

Because the driver is not always aware of the full workload, it is sometimes useful for applications to provide additional hints regarding the desired shared memory configuration. For example, a kernel with little or no shared memory use may request a larger carveout in order to encourage concurrent execution with later kernels that require more shared memory. The new `cudaFuncSetAttribute()` API allows applications to set a preferred shared memory capacity, or `carveout`, as a percentage of the maximum supported shared memory capacity (96 KB for *Volta*, and 64 KB for *Turing*).

`cudaFuncSetAttribute()` relaxes enforcement of the preferred shared capacity compared to the legacy `cudaFuncSetCacheConfig()` API introduced with Kepler. The legacy API treated shared memory capacities as hard requirements for kernel launch. As a result, interleaving kernels with different shared memory configurations would needlessly serialize launches behind shared memory reconfigurations. With the new API, the carveout is treated as a hint. The driver may choose a different configuration if required to execute the function or to avoid thrashing.

Compute Capab. 7.x (Volta/Turing) [3]

```
// Device code
__global__ void MyKernel(...)
{
    __shared__ float buffer[BLOCK_DIM];
    ...
}

// Host code
int carveout = 50; // prefer shared memory capacity 50% of maximum
// Named Carveout Values:
// carveout = cudaSharedmemCarveoutDefault;   //  (-1)
// carveout = cudaSharedmemCarveoutMaxL1;     //   (0)
// carveout = cudaSharedmemCarveoutMaxShared; // (100)
cudaFuncSetAttribute(MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout,
↪carveout);
MyKernel <<<gridDim, BLOCK_DIM>>>(...);
```

In addition to an integer percentage, several convenience enums are provided as listed in the code comments above. Where a chosen integer percentage does not map exactly to a supported capacity (SM 7.0 devices support shared capacities of 0, 8, 16, 32, 64, or 96 KB), the next larger capacity is used. For instance, in the example above, 50% of the 96 KB maximum is 48 KB, which is not a supported shared memory capacity. Thus, the preference is rounded up to 64 KB.

Compute Capab. 7.x (Volta/Turing) [4]

Compute capability 7.x devices allow a single thread block to address the full capacity of shared memory: 96 KB on *Volta*, 64 KB on *Turing*. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, as such they must use dynamic shared memory (rather than statically sized arrays) and require an explicit opt-in using `cudaFuncSetAttribute()` as follows.

```
// Device code
__global__ void MyKernel(...)
{
    extern __shared__ float buffer[];
    ...
}

// Host code
int maxbytes = 98304; // 96 KB
cudaFuncSetAttribute(MyKernel, cudaFuncAttributeMaxDynamicSharedMemorySize, maxbytes);
MyKernel <<<gridDim, blockDim, maxbytes>>>(...);
```

Otherwise, shared memory behaves the same way as for devices of compute capability 5.x (See *Shared Memory*).

Compute Capab. 8.x (Ampere/Ada) [1]

20.7.3. Shared Memory

Similar to the *Volta architecture*, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the **NVIDIA Ampere GPU Architecture**, the unified data cache has a size of 192 KB for devices of compute capability 8.0 and 8.7 and 128 KB for devices of compute capabilities 8.6 and 8.9. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132 or 164 KB for devices of compute capability 8.0 and 8.7, and to 0, 8, 16, 32, 64 or 100 KB for devices of compute capabilities 8.6 and 8.9.

An application can set the `carveout`, i.e., the preferred shared memory capacity, with the `cudaFuncSetAttribute()`.

```
cudaFuncSetAttribute(kernel_name, cudaFuncAttributePreferredSharedMemoryCarveout,
↪carveout);
```

Compute Capab. 8.x (Ampere/Ada) [2]

The API can specify the carveout either as an integer percentage of the maximum supported shared memory capacity of 164 KB for devices of compute capability 8.0 and 8.7 and 100 KB for devices of compute capabilities 8.6 and 8.9 respectively, or as one of the following values: `{cudaSharedmemCarveoutDefault`, `cudaSharedmemCarveoutMaxL1`, or `cudaSharedmemCarveoutMaxShared`. When using a percentage, the carveout is rounded up to the nearest supported shared memory capacity. For example, for devices of compute capability 8.0, 50% will map to a 100 KB carveout instead of an 82 KB one. Setting the `cudaFuncAttributePreferredSharedMemoryCarveout` is considered a hint by the driver; the driver may choose a different configuration, if needed.

Devices of compute capability 8.0 and 8.7 allow a single thread block to address up to 163 KB of shared memory, while devices of compute capabilities 8.6 and 8.9 allow up to 99 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see *Shared Memory* for the **NVIDIA Volta GPU Architecture**.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

Compute Capab. 9.x (Hopper)

20.8.3. Shared Memory

Similar to the *NVIDIA Ampere GPU architecture*, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *NVIDIA H100 Tensor Core GPU architecture*, the unified data cache has a size of 256 KB for devices of compute capability 9.0. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132, 164, 196 or 228 KB.

As with the *NVIDIA Ampere GPU architecture*, an application can configure its preferred shared memory capacity, i.e., the `carveout`. Devices of compute capability 9.0 allow a single thread block to address up to 227 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see *Shared Memory* for the **NVIDIA Volta GPU Architecture**.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

Compute Capab. 10.x / 12.x (Blackwell)

20.9.3. Shared Memory

The amount of the unified data cache reserved for shared memory is configurable on a per kernel basis and is identical to *compute capability 9.0*. The unified data cache has a size of 256 KB for devices of compute capability 10.0. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132, 164, 196 or 228 KB.

As with the *NVIDIA Ampere GPU architecture*, an application can configure its preferred shared memory capacity, i.e., the `carveout`. Devices of compute capability 10.0 allow a single thread block to address up to 227 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see *Shared Memory* for the Volta architecture.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

Thank you.