

基于图形处理器的并行计算及CUDA编程

Ying Liu, Associate Prof., Ph.D

School of Computer and Control, University of Chinese
Academy of Sciences

Key Lab of Big Data Mining & Knowledge Management,
Chinese Academy of Sciences

CUDA Performance Optimization

- Optimization strategies
- Maximize utilization
- Maximize memory throughput
- Maximize instruction throughput
- Others

Optimization Strategies

- Maximize utilization
- Maximize memory throughput
- Maximize instruction throughput
- Others

CUDA Performance Optimization

- Optimization strategies
- **Maximize utilization**
- Maximize memory throughput
- Maximize instruction throughput
- Others

Loop Parallelization

- We will look at some simple and commonly occurring code fragments and consider how to parallelize them
- Techniques involve to remove dependencies
 - Local variables to remove dependencies
 - Code transformations
 - Use of particular scheduling techniques

Local Variables

■ Code Fragment

```
float *a, *b, x;  
int n;
```

```
for (i=0 ;i<n; i++) {  
    x = a[i]*3;  
    b[i] = x*b[i];  
}
```

Local Variables

```
__global__ void parallel_func () {  
  
    __shared__ float a[ ], b[ ], x;  
    int tx;  
  
    tx = threadIdx.x;  
    x = a[tx]*3;  
    b[tx] = x*b[tx];  
}
```

wrong!

Local Variables

```
__global__ void parallel_func () {  
  
    __shared__ float a[], b[];  
    int tx;  
    float x;  
  
    tx = threadIdx.x;  
    x = a[tx]*3;  
    b[tx] = x*b[tx];  
}
```


Loop-Carried Variables

- Code fragment

```
Indx = 0;  
for (i=0; i<n; i++) {  
    indx += i;  
    a[i] = b[indx];  
}
```

Loop-Carried Variables

```
__global__ void parallel_func (float *a, float *b) {  
  
    int tx, indx;  
  
    tx = threadIdx.x;  
    indx = (tx*(tx+1))/2;  
    a[tx] = b[indx];  
}
```

Indirect Indexing

■ Code fragment

```
for (i=0;i<n;i++) {  
    ix = indexx[i];  
    iix = ixoffset[ix];  
    total[iix] += delta;  
}
```

Indirect Indexing

```
for (i=0;i<n;i++) {  
    ix = indexx[i];  
    iix[i] = ixoffset[ix];  
}  
  
for (i=0;i<n;i++) {  
    total[iix[i]] += delta;  
}
```

Loop Reordering

- Code fragment

```
for (k=0;k<n;k++) {  
    for (i=0;i<n;i++) {  
        for (j=0;j<n;j++) {  
            a[i][j]+=b[i][k]+c[k][j];  
        }  
    }  
}
```

Loop Reordering

```
__global__ void parallel_func (float *a, float*b, float
    *c, int n ) {
    int i;

    tx = threadIdx.x;
    ty = threadIdx.y;
    for (i=0; i<n; i++)
        a[ty][tx] += b[ty][i] + c[i][tx];
}
```

Grid/Block Size Heuristics

- # of blocks / # of stream multiprocessors (SM) > 1
 - Multiple blocks executed on a SM
 - Reduce the possibility that all the blocks in `_syncthreads()`
 - Share the shared memory and registers of SM
- # of blocks > 100
 - Scale to future devices
 - 1000 blocks per grid will scale across multiple generations

Thread Block

- # of threads in a block must be a multiple of warp size
 - Avoid under-populated warps
- A large number of threads
 - Hide the memory access latency
 - Less registers allocated for each thread
 - Not work due to lack of registers
- Heuristics
 - Minimum: 64 threads per block
 - Only if multiple concurrent blocks
 - 128 to 256 threads a better choice
 - Usually still enough registers to compile and invoke successfully
 - All depends on your application/computation!
 - Experiment!

Hiding Memory Access Latency

- Global memory access: 400-600 cycle latency
 - Global memory access instructions are stalled
- Remedies:
 - Increase # threads within a block
 - Improve computational intensity
 - Coalescing memory accesses to neighboring addresses
- Example
 - 4 global memory accesses cost $4 * 400 = 1,600$ cycles
 - 4 concurrent threads, 1 read per thread, minimum 400 cycles

Register

- Restrictions:
 - Number of registers per kernel
 - 8192 per SM, partitioned among concurrent threads
 - Amount of shared memory
 - 16KB per SM, partitioned among concurrent blocks
- Check .cubin file for # registers / kernel
 - `nvcc -keep <filename>.cu`
- Compile with `nvcc -maxrregcount = N`
 - N = desired maximum registers / kernel
 - At some point “spilling” into local memory may occur
 - Reduce performance – local memory is slow
 - Check .cubin file for local memory usage

Asynchronous Concurrent Execution

- Some runtime functions are asynchronous
 - Control is returned to the application before the device has completed the requested task

```
int main(int argc, char **argv) {  
    .....  
    operation0;  
    operation1;  
    operation2;  
    operation3;  
    .....  
}
```

If operation2 is asynchronous and all others are synchronous, then operation3 will be directly executed after operation2 is invoked, no matter whether operation2 has completed or not

Asynchronous Concurrent Execution

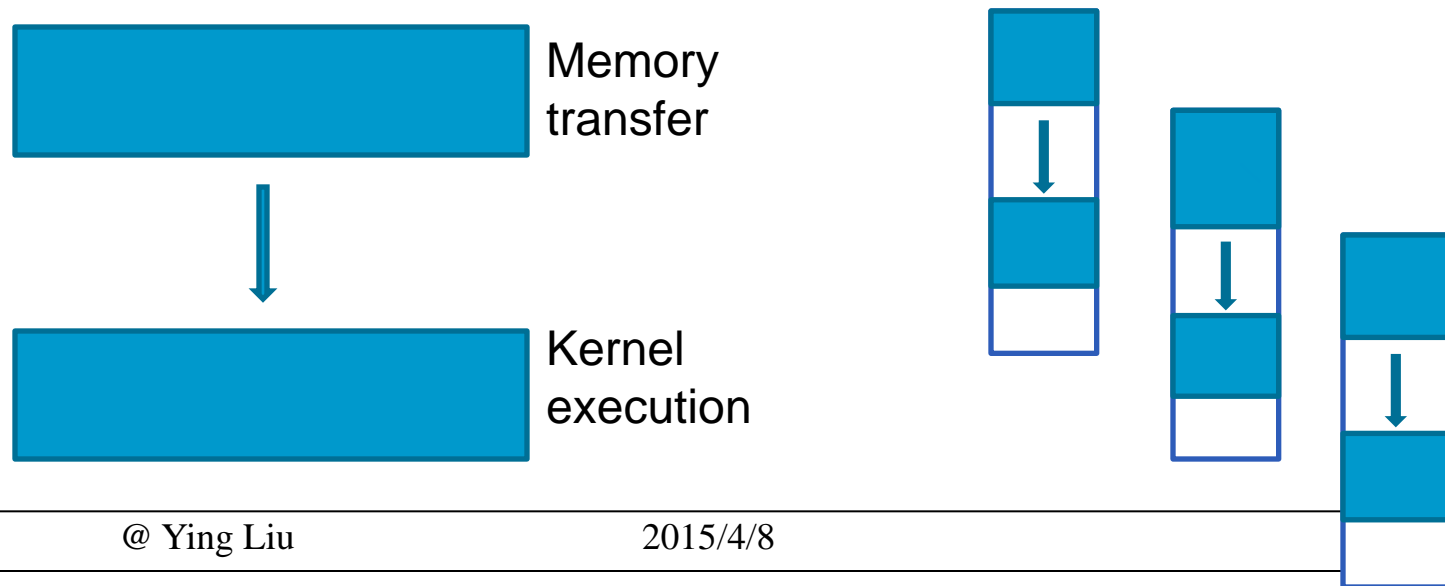
- Applications may query this capability by calling *cudaGetDeviceProperties()* and checking *deviceOverlap*

Device 0: "Tesla C1060"

Major revision number:	1
Minor revision number:	3
Total amount of global memory:	4294705152 bytes
Number of multiprocessors:	30
Number of cores:	240
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	16384
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Maximum memory pitch:	262144 bytes
Clock rate:	1.30 GHz
Concurrent copy and execution:	Yes

Asynchronous Concurrent Execution

- In order to maximize parallel execution between the multiprocessors, maximize the number of concurrent kernels by using streams
- Applications manage concurrency through *streams*. A *stream* is a sequence of operations that execute in order
- Different streams, may execute their operations out of order with respect to one another or concurrently



Asynchronous Concurrent Execution

- **CUDA Stream Management API:**
 - **cudaStreamCreate:** create an async stream
 - **cudaStreamQuery:** queries a stream for completion-status
 - **cudaStreamSynchronize:** waits for stream tasks to complete
 - **cudaStreamDestroy:** destroys and cleans-up a stream object

Asynchronous Concurrent Execution

```
// allocate and initialize an array of stream handles
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

// execution with streams
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size,
                    cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();

// release resources
for (int i = 0; i < nstreams; i++)
    cudaStreamDestroy(streams[i]);
```

CUDA Performance Optimization

- Optimization strategies
- Maximize utilization
- Maximize memory throughput
- Maximize instruction throughput
- Others

Data Transfer between Host and Device

- Minimize data transfer between CPU and GPU
 - 8 GB/sec
- Use page-locked host memory
- Use virtual memory

Device Memory Access

- Global memory 400-600 clock cycles latency
 - Performance bottleneck
- Coalesced vs. non-coalesced
 - Experiment
 - Kernel: read a float, increment, write back
 - 3M floats (12MB), times averaged over 10K runs
 - 12K blocks x 256 threads
 - 356 μ s – coalesced
 - 357 μ s – coalesced, some threads don't participate
 - 3,494 μ s – permuted/misaligned thread access
- Conclusion
 - Orders of magnitude improvement!
 - Critical to small or memory-bound kernels

Memory Bandwidth

- Device memory should be minimized
 - Device memory is much higher latency and lower bandwidth than on-chip memory
- Global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (16 threads) can be **coalesced** into a single memory transaction

Memory Bandwidth

- Device reads 32-bit, 64-bit or 128-bit words from global memory into registers in a single transaction
 - Data type must be `sizeof(type) = 4, 8, 16` bytes
 - Data type must be aligned
- Size of a memory transaction
 - 64 bytes - each thread reads a word: `int`, `float`, ...
 - 128 bytes - each thread reads a double-word: `int2`, `float2`, ...
 - 32 bytes (compute capability 1.2+) - each thread reads a short `int`, ...

Coalescing

■ Requirements

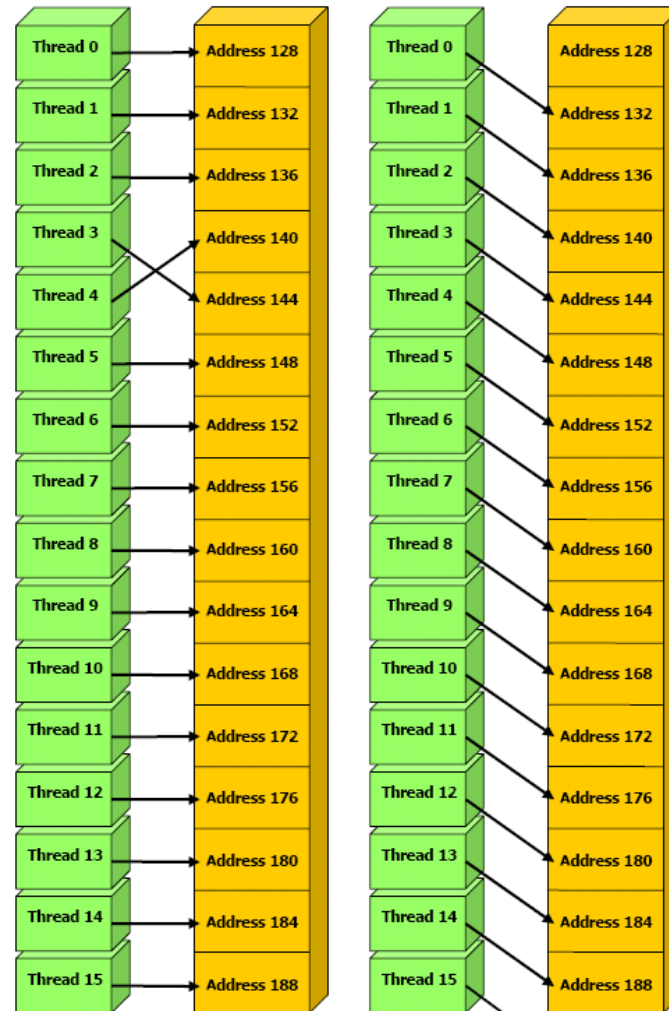
- The starting address of global memory is usually aligned to times of 64, 128 bytes
- `Sizeof(type)` must be 4, 8, or 16
- All 16 words must lie in the same segment
- All threads in a warp access the words in sequence
- Divergent warp is allowed
- Multiple threads access the same address (for devices with Compute Capability 1.2 and higher)

Global Memory



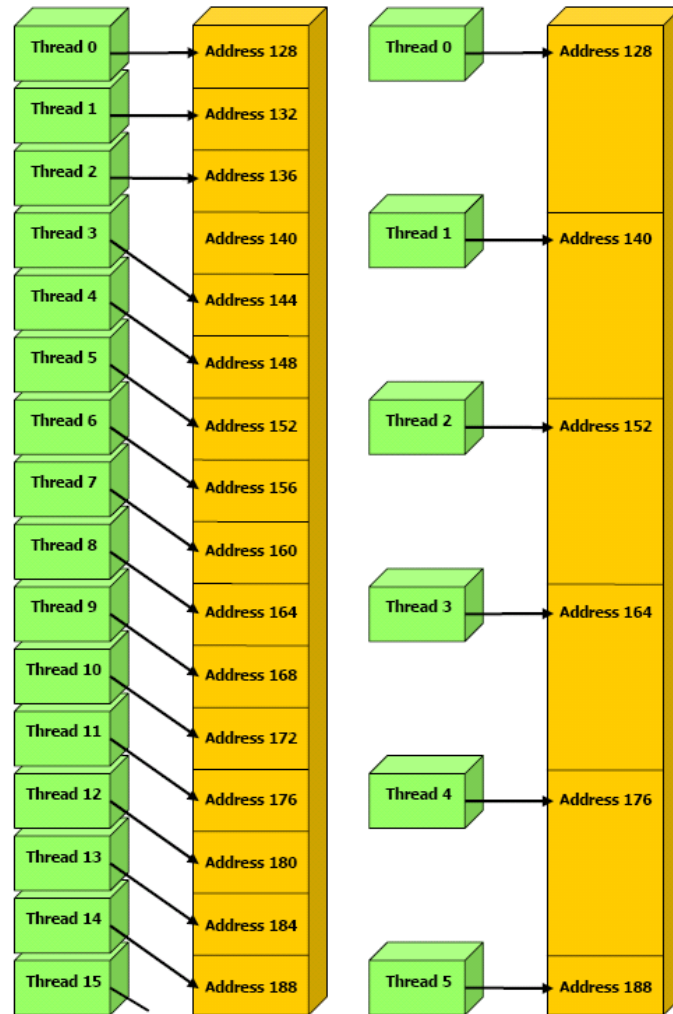
Coalesced Access

Global Memory



Non-coalesced Access

Global Memory



Non-coalesced Access

Global Memory

- The alignment specifiers `_align_(8)` or `_align_(16)`

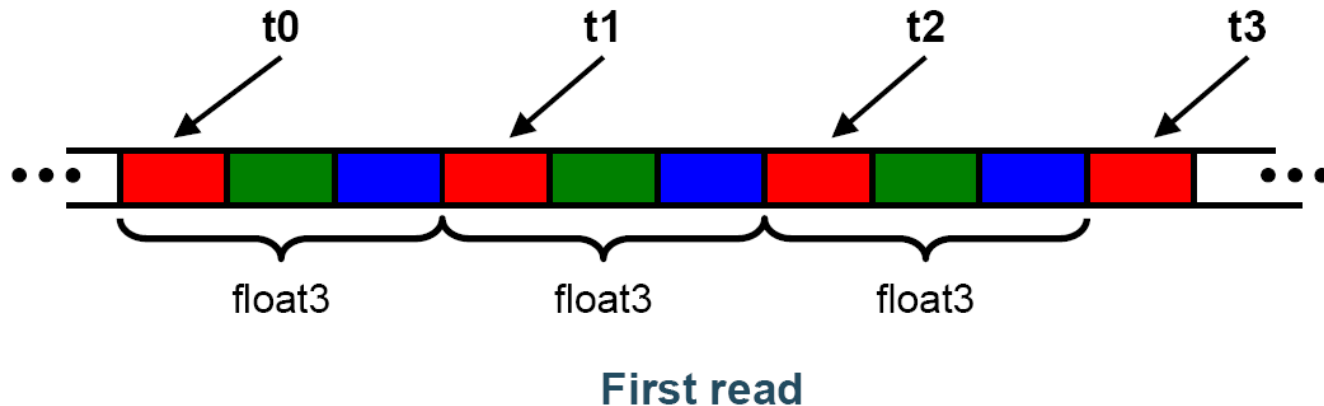
```
Struct _align_ (8) {  
    float a;  
    float b;  
}  
Struct _align_ (16) {  
    float a;  
    float b;  
    float c;  
}  
Struct _align_ (16) {  
    float a;  
    float b;  
    float c;  
    float d;  
    float e;  
}
```

Uncoalesced float3 Code

```
__global__ void accessFloat3(float3 *d_in,  
    float3* d_out)  
{  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    float3 a = d_in[index];  
    a.x += 2;  
    a.y += 2;  
    a.z += 2;  
    d_out[index] = a;  
}
```

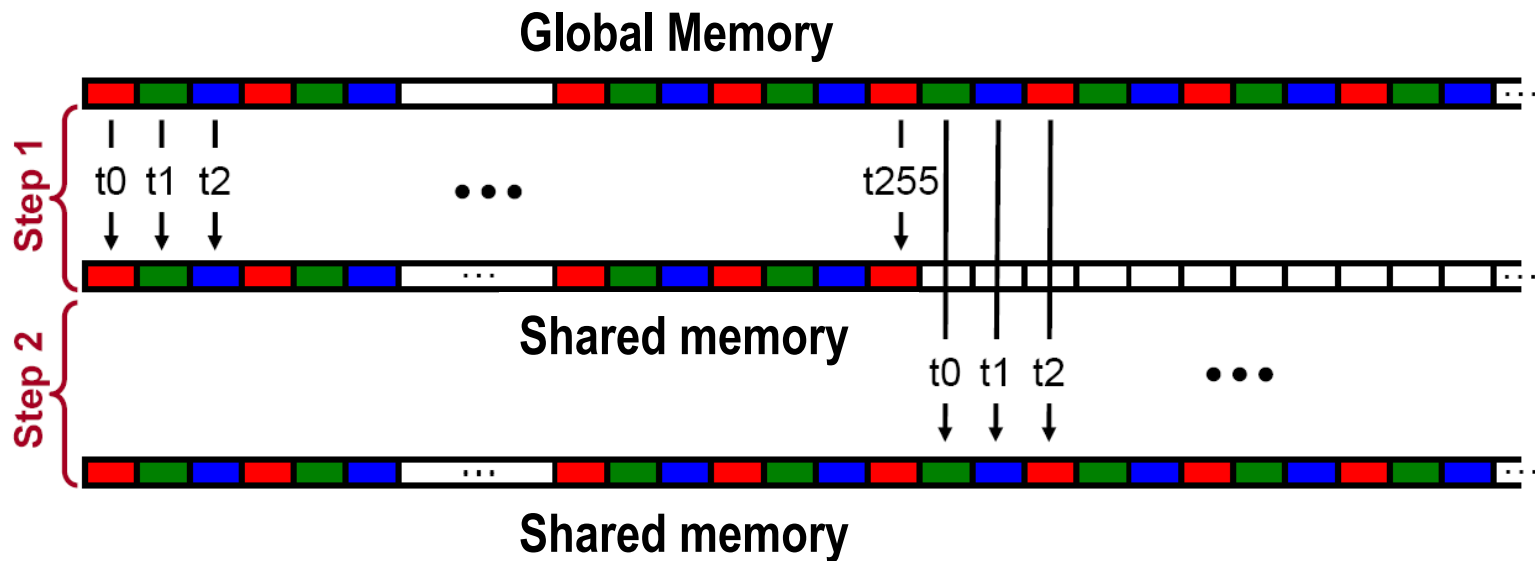
Uncoalesced float3 Code

- float3 (12 bytes): float3 f = d_in[threadIdx.x];
- Each thread ends up executing 3 reads
 - $\text{sizeof}(\text{float3}) \neq 4, 8, \text{ or } 16$
 - Half-warp reads three 64B **non-contiguous** regions



Coalescing float3 Access

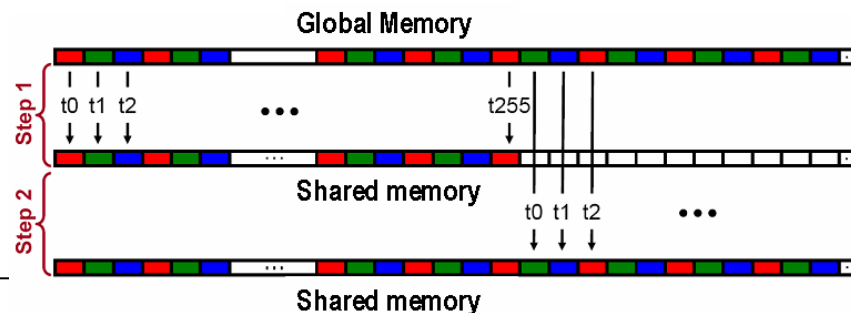
- A 3-step approach (256 threads/block)



Similarly, Step3 starting at offset 512

Coalescing float3 Access

- Use shared memory to allow coalescing
 - 256 threads per block
 - A thread block needs $\text{sizeof(float3)} \times 256$ bytes of SMEM
 - Each thread reads 3 scalar floats:
 - likely be processed by other threads, so sync
- Processing
 - Each thread retrieves its float3 from SMEM array
 - Cast the SMEM pointer to (float3*)
 - Use thread ID as index
 - Rest of the compute code does not change!



Coalescing float3 Access Code

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x] = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index] = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

Compute code is not changed

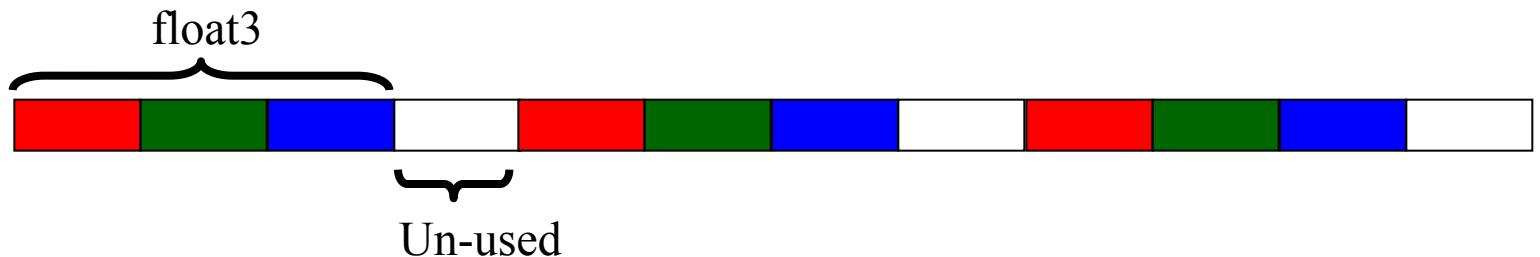
Write the result through SMEM

Coalescing: Structures of Size $\neq 4, 8, \text{ or } 16\text{B}$

■ Compiler enforces the alignment specifiers

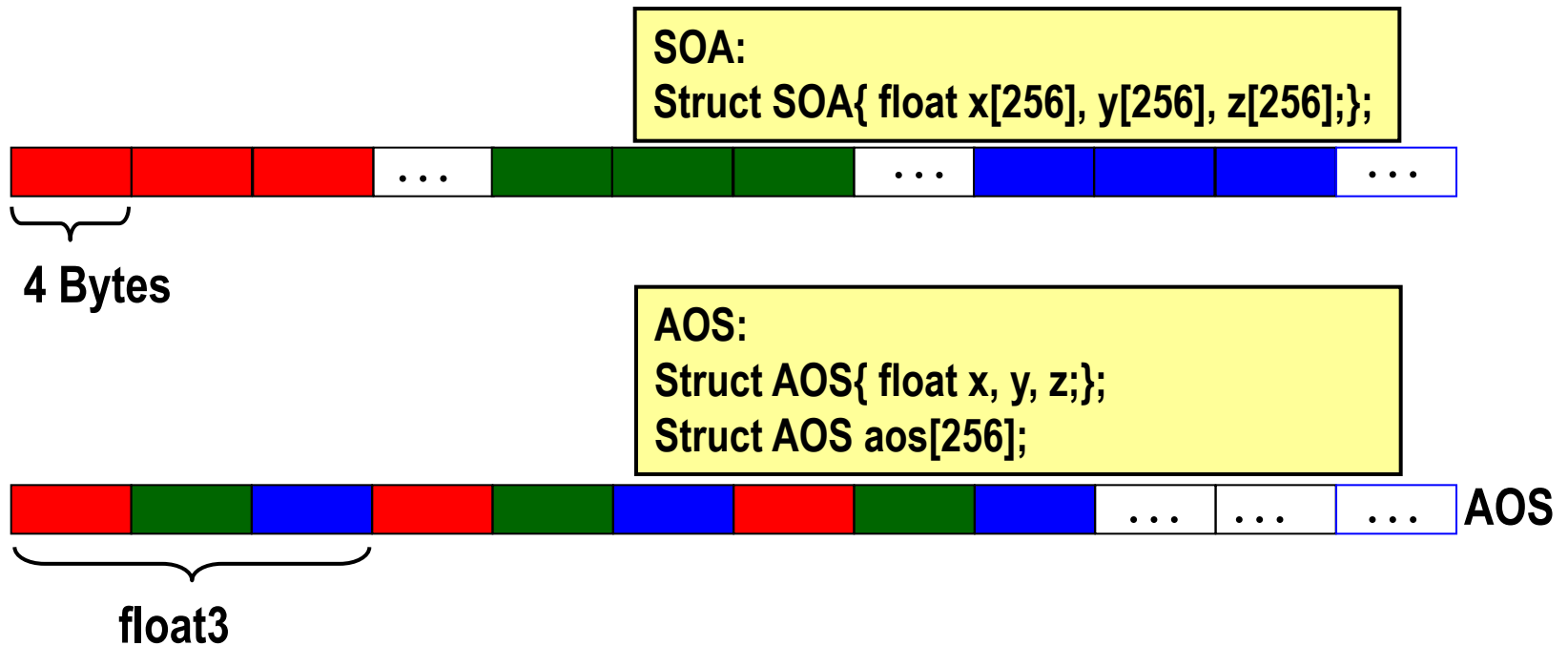
■ `__align__(X)`, where $X = 4, 8, \text{ or } 16$

- `struct __align__(16) {
 float x;
 float y;
 float z;
};`
- Waste some memory space



Coalescing: Structures of Size $\neq 4, 8, \text{ or } 16\text{B}$

- Use a structure of arrays instead of AoS

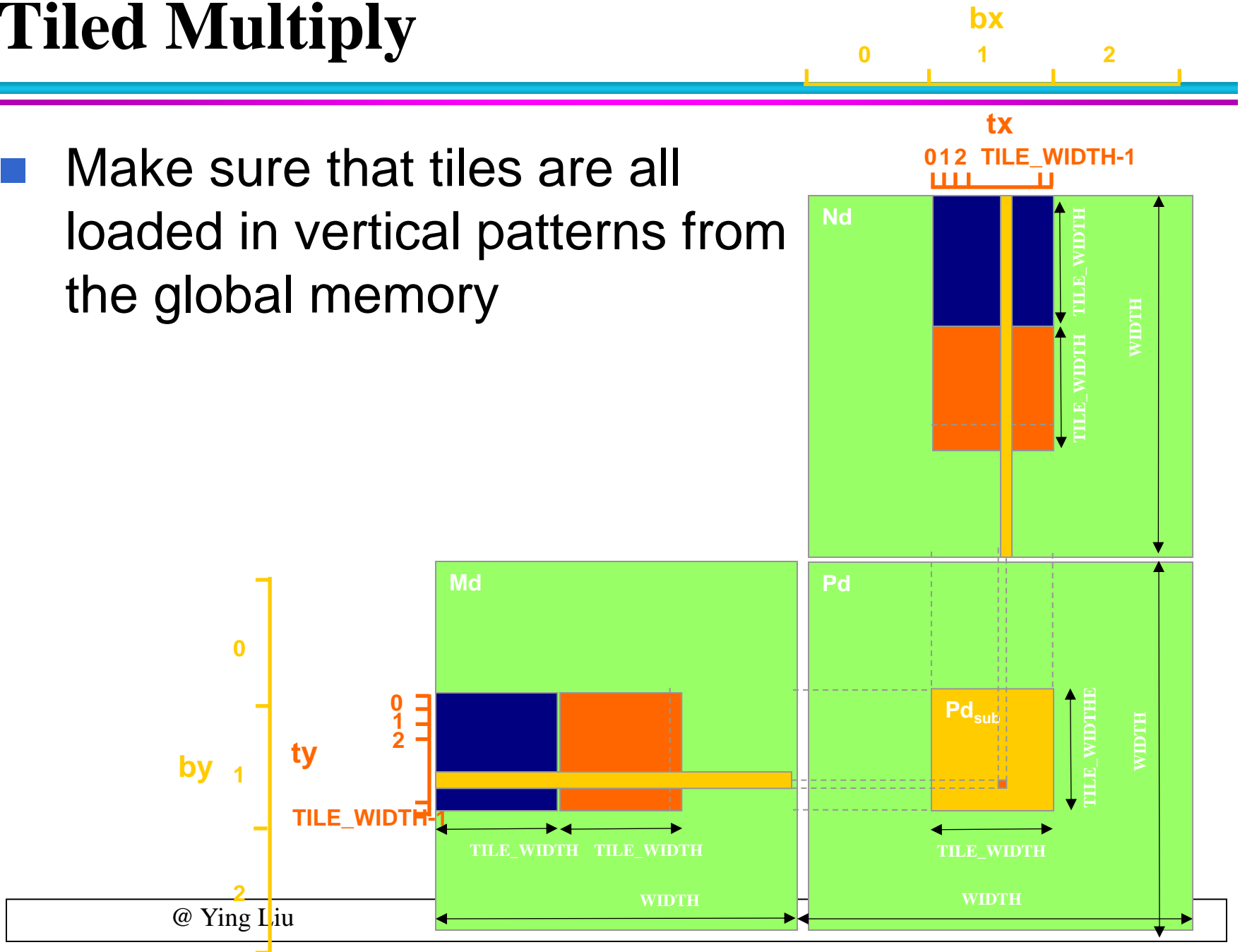


Solve Non-Coalesced Memory Access

- If all threads in a half-warp read the identical address, use constant memory
- For sequential access patterns, but `sizeof(struct) ≠ 4, 8, or 16` bytes:
 - Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)
 - Or force structure alignment by compiler
 - Using `__align(X)`, where $X = 4, 8, \text{ or } 16$
 - Or use shared memory to achieve coalescing

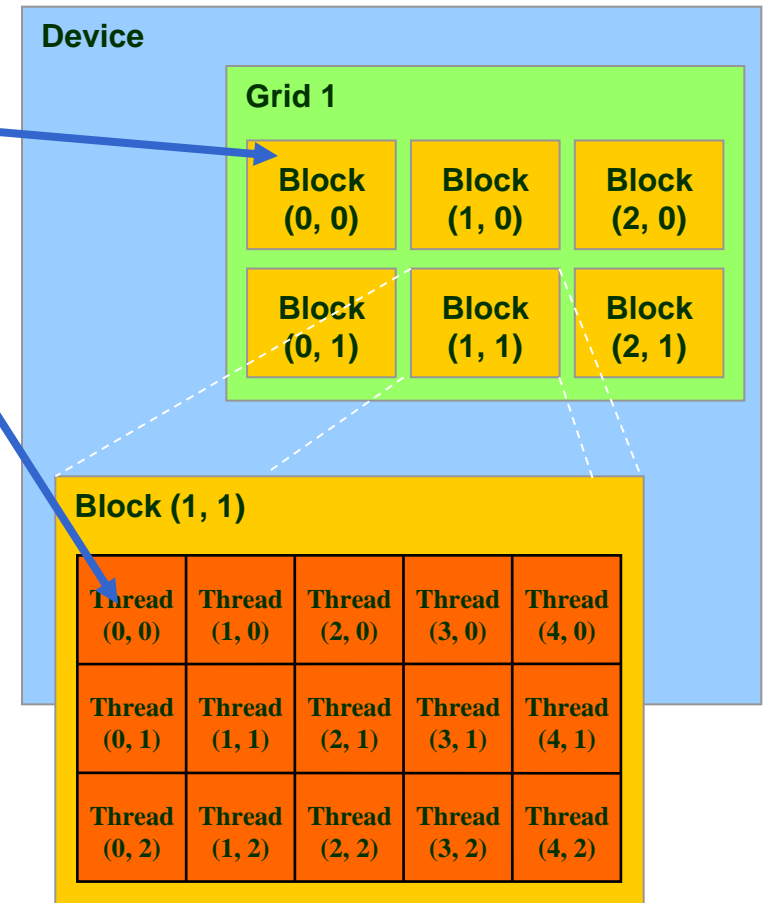
Tiled Multiply

- Make sure that tiles are all loaded in vertical patterns from the global memory



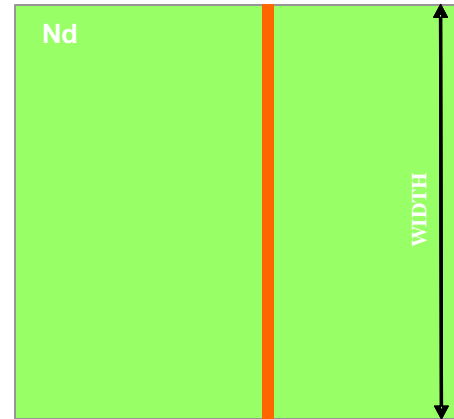
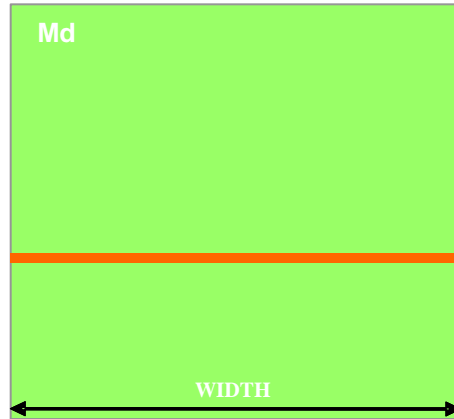
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
 - Block ID: 1D or 2D
 - Thread ID: 1D, 2D, or 3D

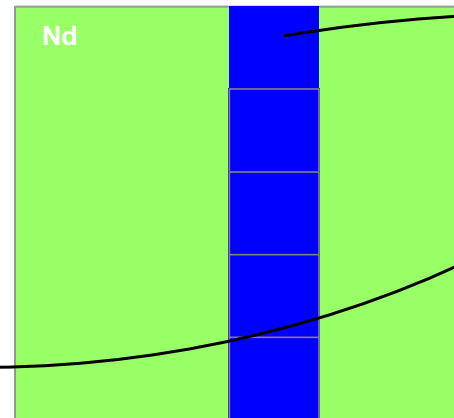
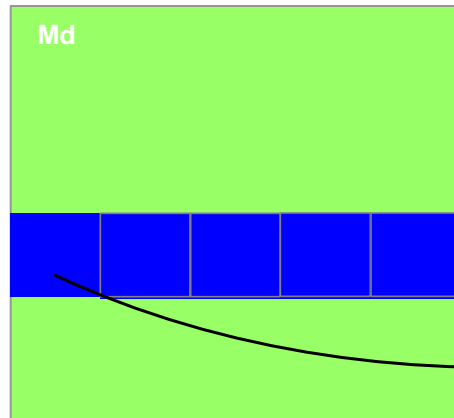


Memory Access Pattern

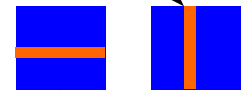
Original
Access
Pattern



Tiled
Access
Pattern



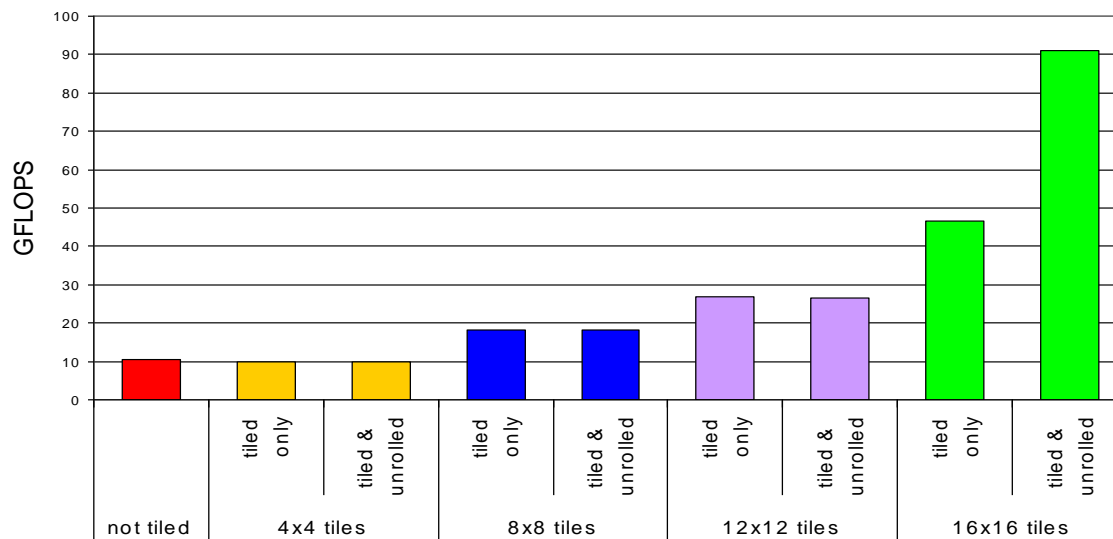
Copy into
scratchpad
memory



Perform
multiplication
with scratchpad
values

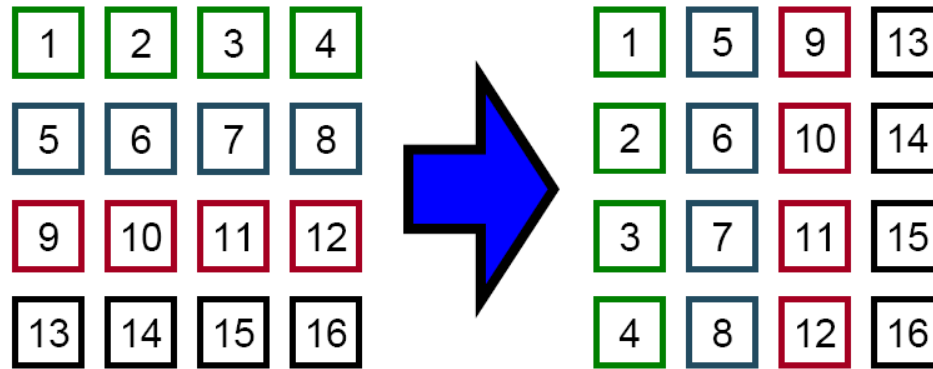
Tiling Size Effects

- For good bandwidth utilization, accesses should be aligned and consist of 16 contiguous words
- Tile size 16X16 minimal required to achieve full coalescing
 - Both reduction of global memory accesses and more efficient execution of the accesses



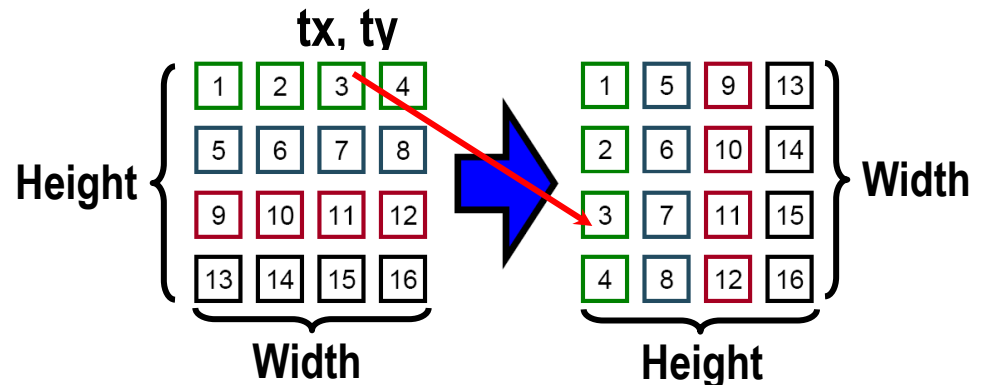
Matrix Transpose

- SDK Sample (“transpose”) illustrates memory coalescing

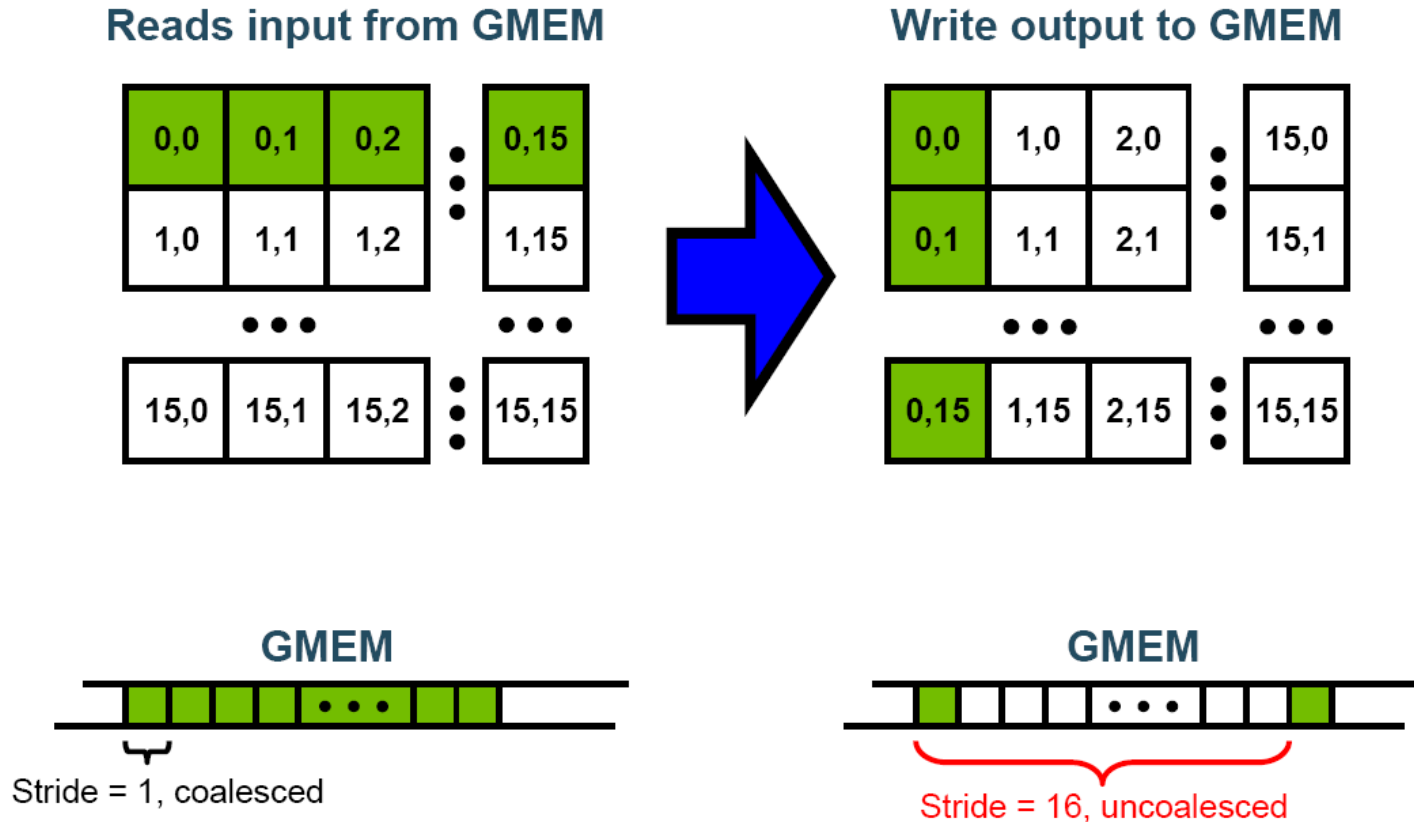


Uncoalesced Transpose

```
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

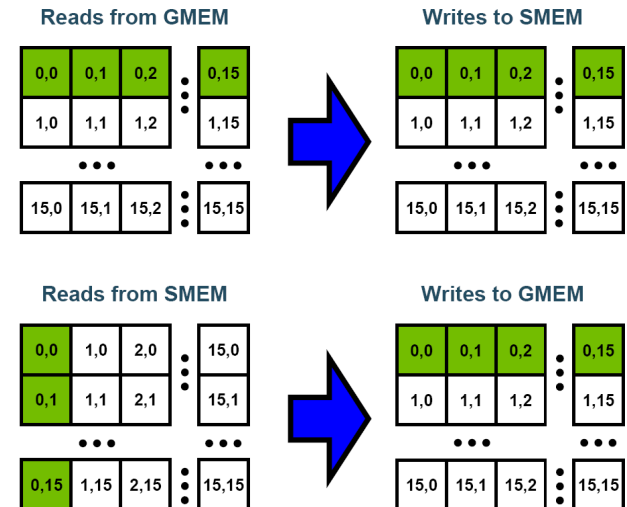


Uncoalesced Transpose

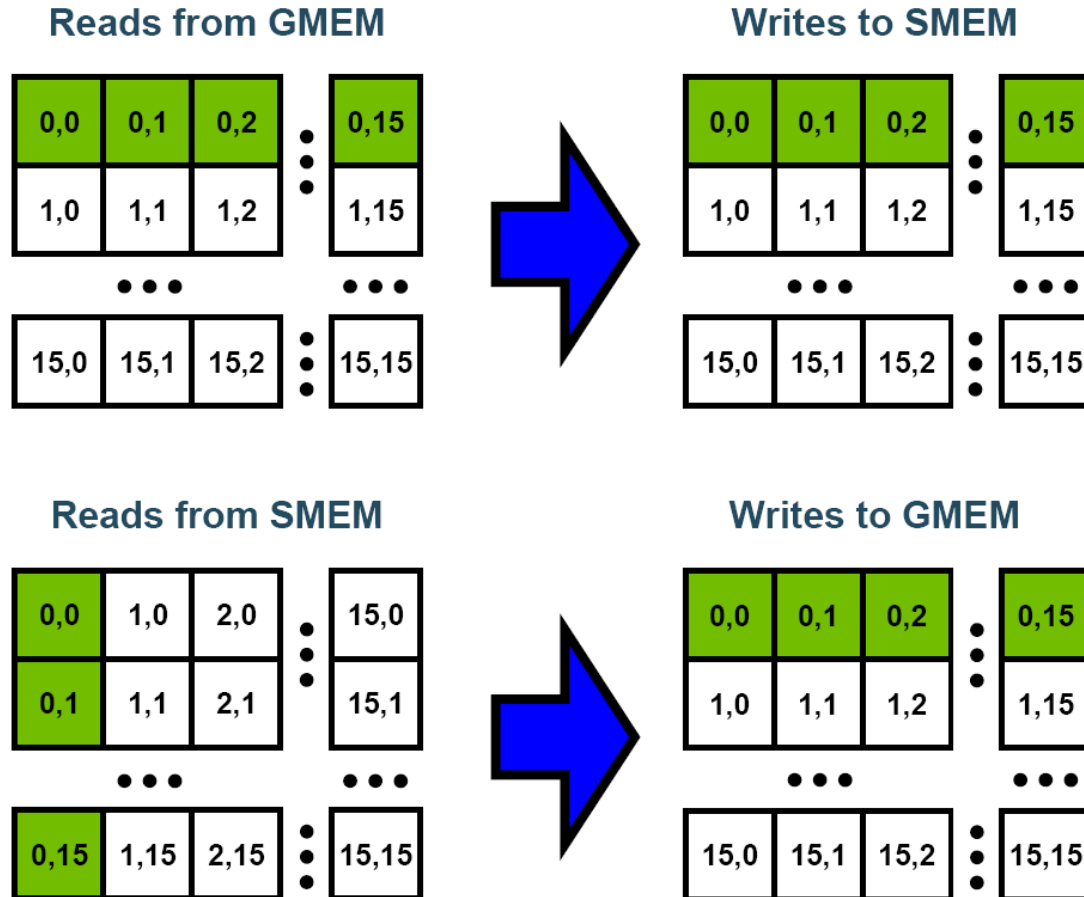


Coalesced Transpose

- Assume matrix is decomposed into square tile
- Thread block (bx, by):
 - Read the (bx, by) input tile, store into SMEM
 - Write the SMEM data to (by, bx) output tile
- Thread (tx, ty):
 - Read element (tx, ty) from input tile
 - Write element (tx, ty) into output tile
- Coalescing is achieved if:
 - Block/tile dimensions are multiples of 16



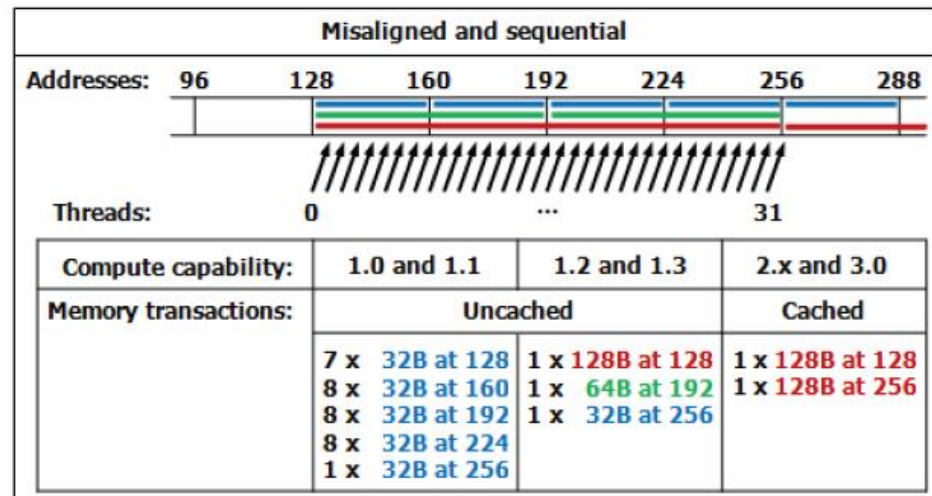
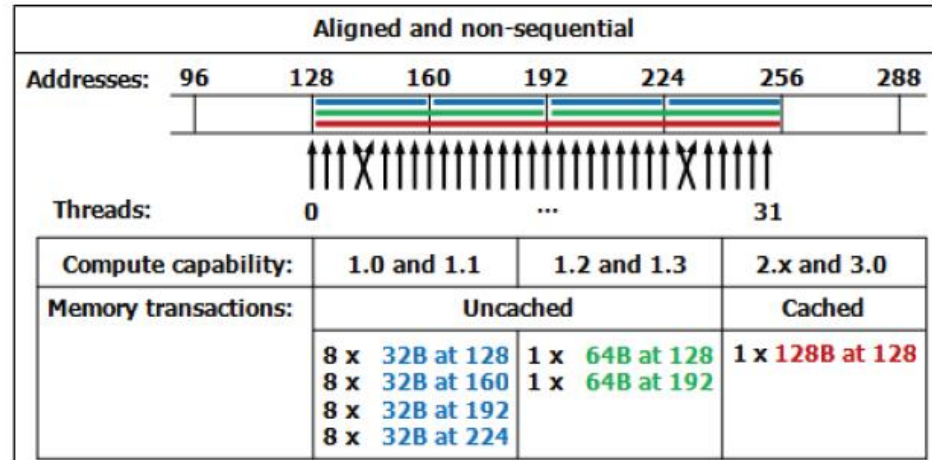
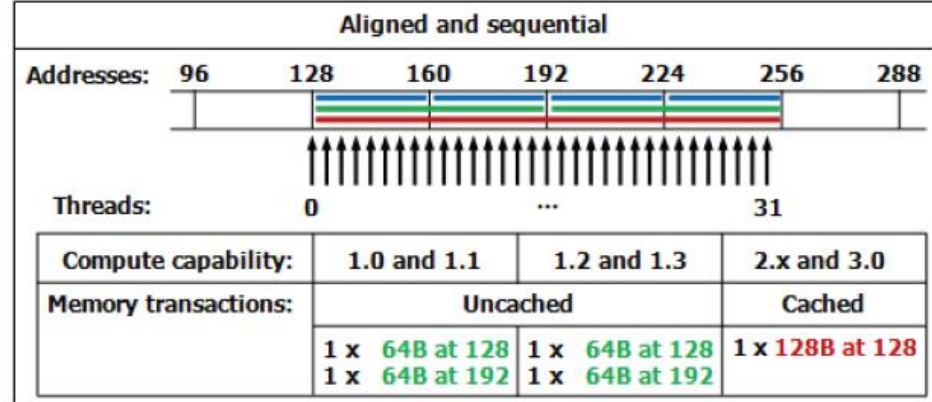
Coalesced Transpose



Coalesced Transpose

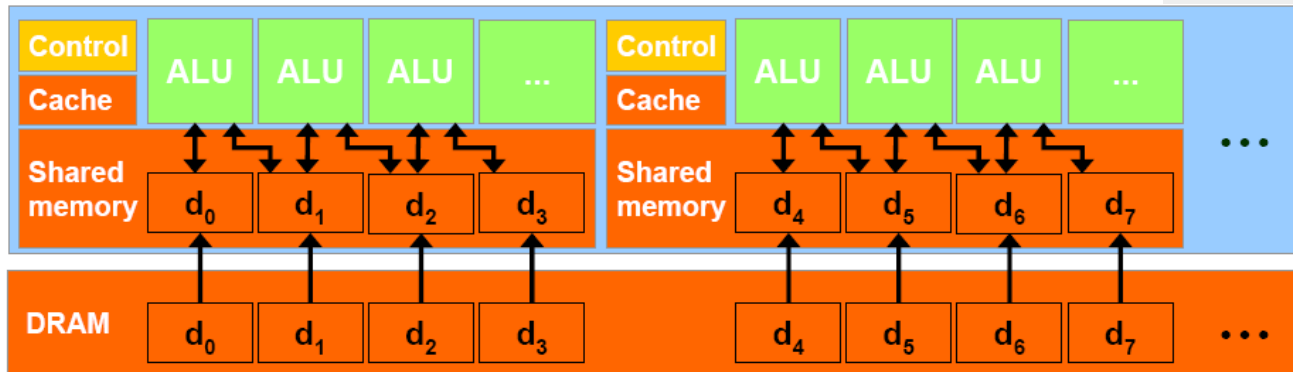
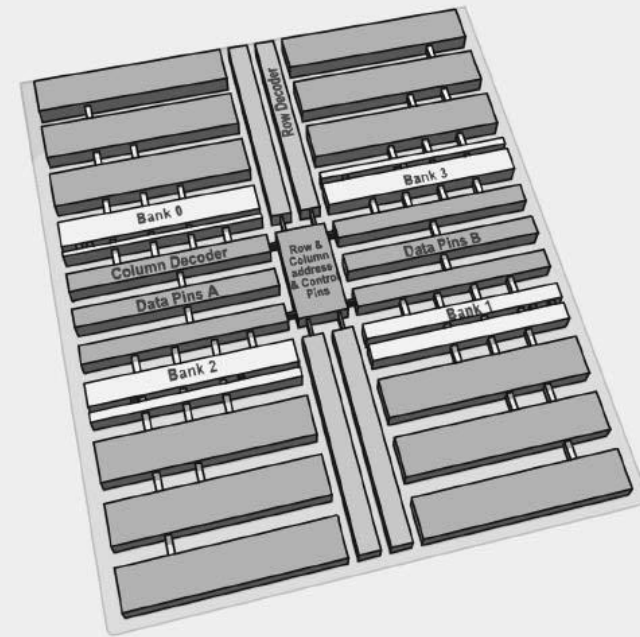
```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];
    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;
    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
    unsigned int index_out, index_transpose;
    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
        block[index_block] = idata[index_in];
        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();
    if (xIndex < width && yIndex < height)
        odata[index_out] = block[index_transpose];
}
```

Global Memory Accesses in Different Compute Capability



Shared Memory

- Shared memory is divided into equally-sized memory modules (banks)
- 16 banks on devices (capability 1.x)
- A memory can service as many simultaneous accesses as it has banks when no bank conflict
- One read/write per two cycles per bank



➔ **Big memory bandwidth savings**

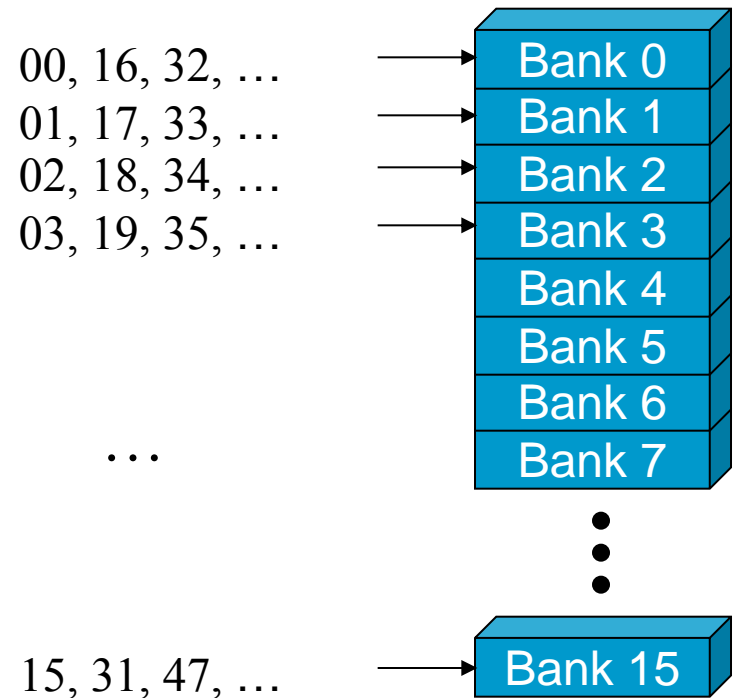
How Addresses Map to Banks on G80

- The bandwidth of each bank is 32 bits (4Bytes) per two clock cycles
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks

- So bank = 4-byte address % 16
- Same as the size of a half-warp
 - A shared memory request for a warp is split into two requests
 - No bank conflicts between different half-warps

- If two requests fall in the same bank, **bank conflict!!!**

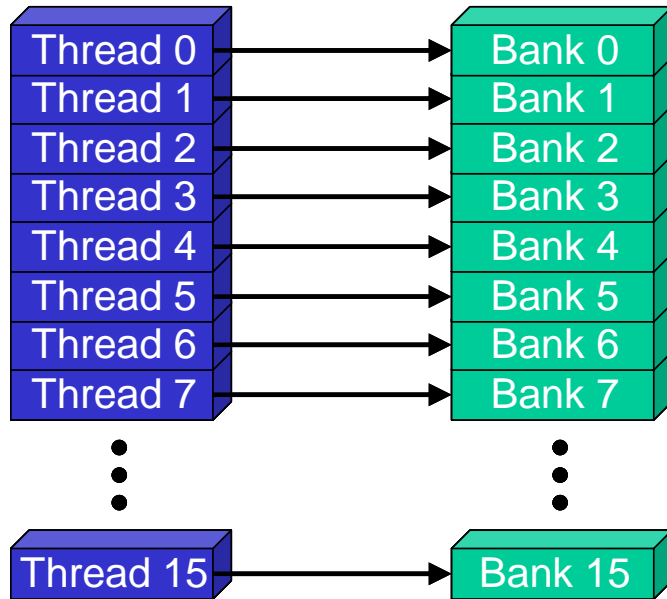
- Must be serialized



Bank Addressing Example

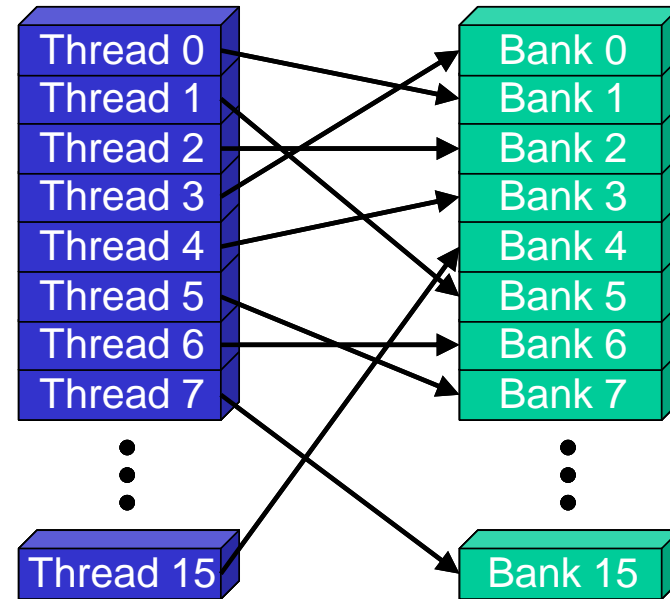
■ No Bank Conflicts

- Linear addressing stride == 1 ($s=1$)



■ No Bank Conflicts

- Random 1:1 Permutation

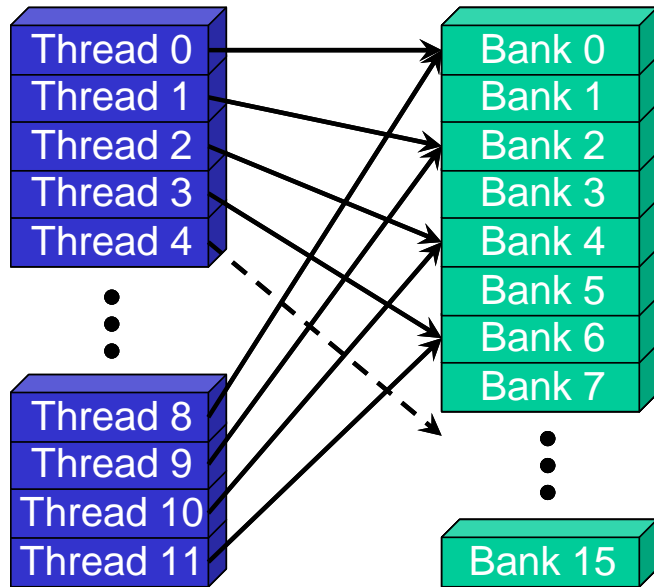


```
__shared__ float shared[256];  
float foo = shared[base + s * threadIdx.x];
```

Bank Addressing Example

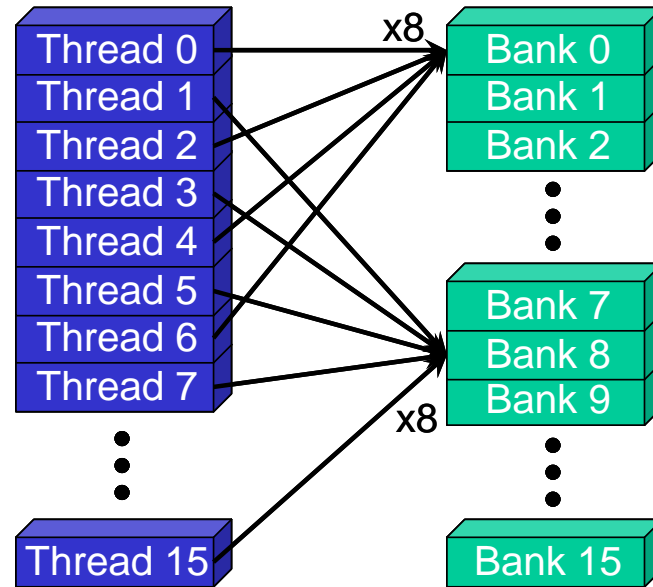
■ 2-way bank conflicts

- Linear addressing
stride == 2 ($s=2$)



■ 8-way bank conflicts

- Linear addressing
stride == 8 ($s=8$)



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


Memory Bank Conflicts

- If no bank conflicts, shared memory is fast
- 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

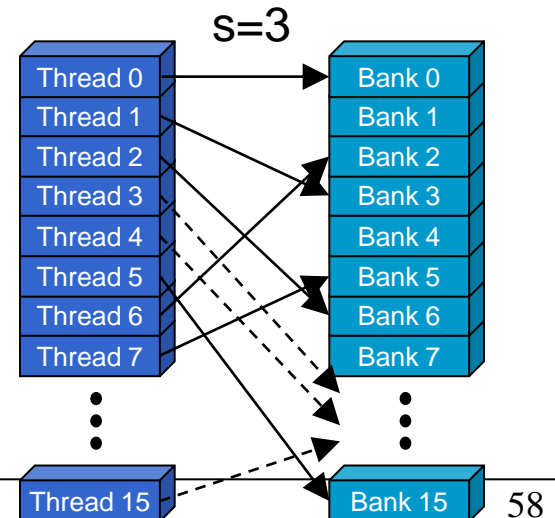
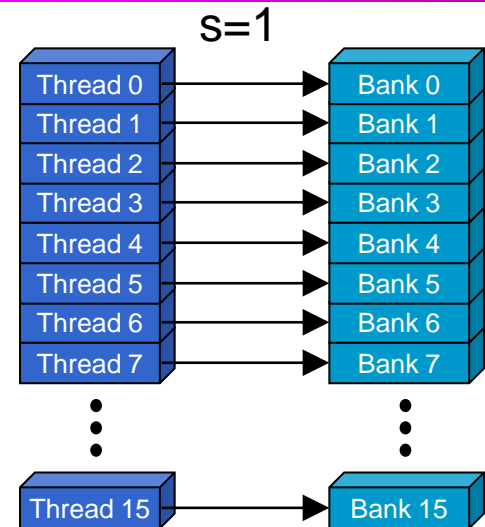
Linear Addressing

- 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**



Data Types and Bank Conflicts

■ Linear addressing

■ Element smaller than 32-bits

- 4-way bank conflicts

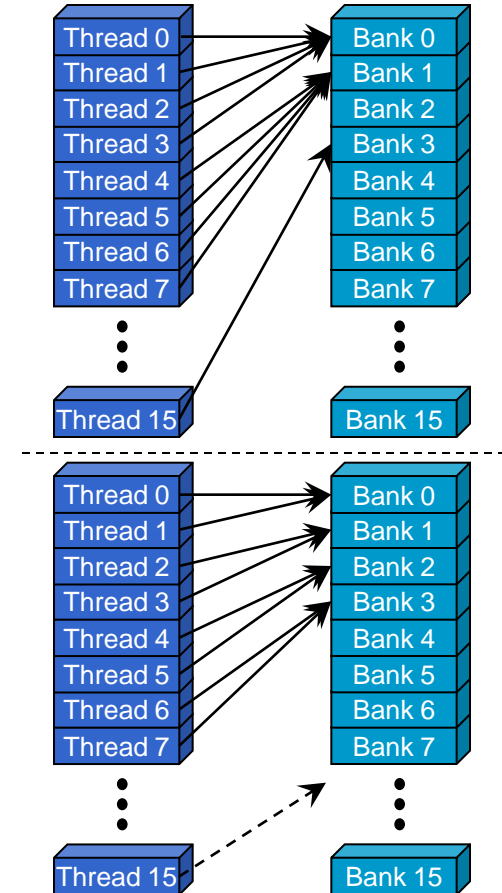
```
__shared__ char shared[];
```

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

- 2-way bank conflicts

```
__shared__ short shared[];
```

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

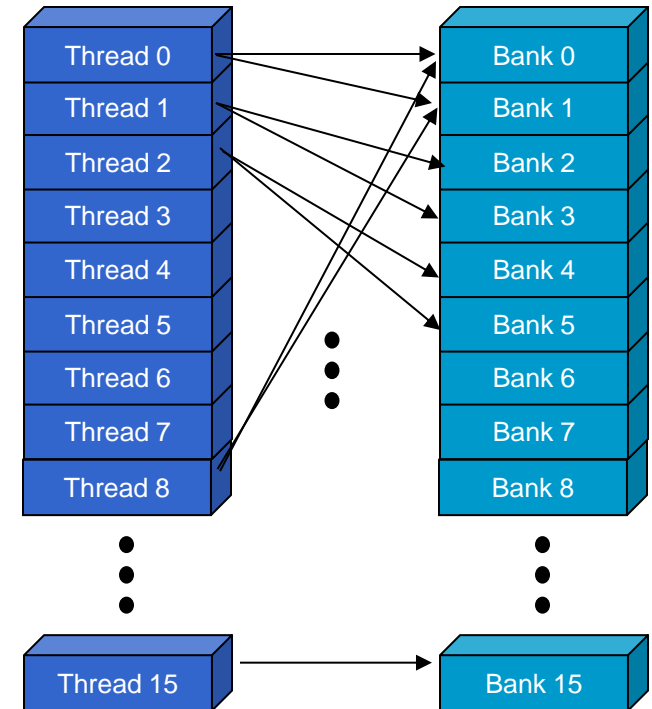


Structs and Bank Conflicts

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

```
struct myType {  
    float f;  
    int c;  
};  
__shared__ struct myType myTypes[32];  
struct myType m = myTypes[baseIndex  
    + threadIdx.x];
```

- This has 2-way bank conflicts for myType
 - 2 accesses per thread

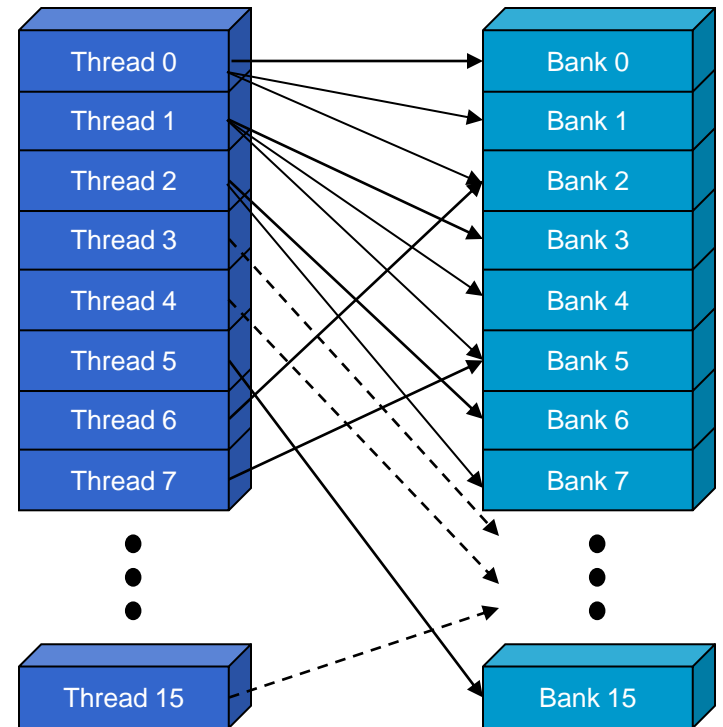


Structs and Bank Conflicts

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

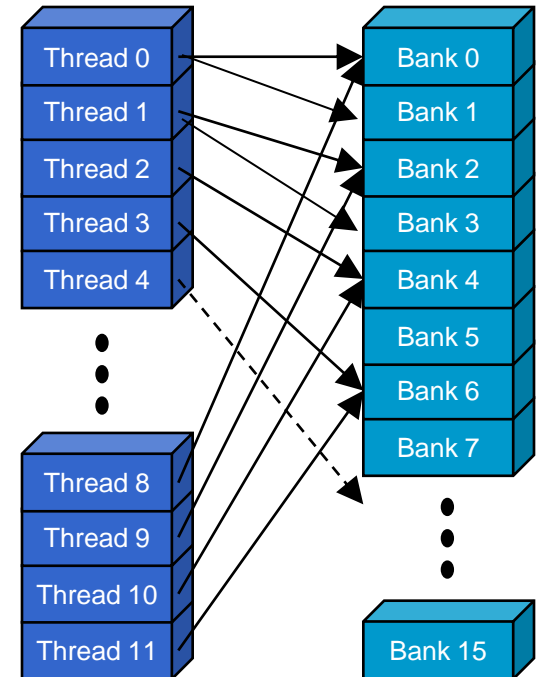
```
struct vector { float x, y, z; };  
__shared__ struct vector vectors[32];  
struct vector v = vectors[baseIndex +  
threadIdx.x];
```

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



Bank Conflict - 1D

- Each thread loads 2 words to shared memory:
 - 2-way-interleaved loads result in 2-way bank conflicts

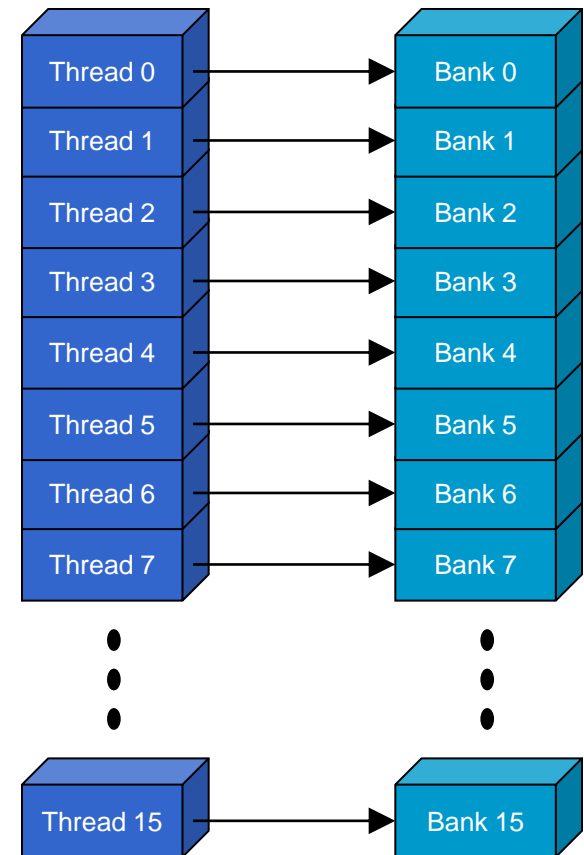


A Better Access Pattern

- Each thread loads one element in every consecutive group of `blockDim` elements

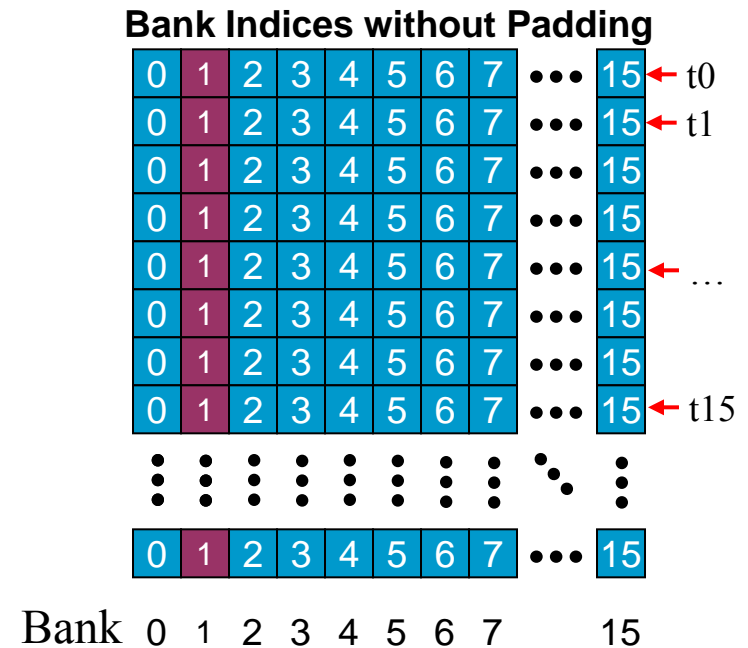
```
shared[tid] = global[tid];
```

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



Bank Conflict - 2D

- 2D floating array
 - image processing, e.g.
- 16x16 block
 - Half-warp threads access one column (example: column 1 in purple)
 - 16-way bank conflicts



Bank Conflict - 2D

- Solution 1: **pad the rows**
 - Pad an element at the end of each row
- Solution 2: transpose the matrix **before processing**
 - Suffer bank conflicts during transpose
 - But possibly save them later

Bank Indices with Padding

0	1	2	3	4	5	6	7	...	15	0
1	2	3	4	5	6	7	8	...	0	1
2	3	4	5	6	7	8	9	...	1	2
3	4	5	6	7	8	9	10	...	2	3
4	5	6	7	8	9	10	11	...	3	4
5	6	7	8	9	10	11	12	...	4	5
6	7	8	9	10	11	12	13	...	5	6
7	8	9	10	11	12	13	14	...	6	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	0	1	2	3	4	5	6	...	14	15

Shared Memory Optimization

- Threads read SMEM with stride = 16
 - Bank conflicts

Reads from SMEM



- Solution
 - Allocate an “extra” column
 - Read stride = 17
 - Threads read from consecutive banks



Coalesced Transpose with Shared Memory Optimization

```
__global__ void transpose_exp(float *odata, float *idata, int width, int height){
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if( (xIndex < width)&&(yIndex < height) ){
        unsigned int index_in = xIndex + yIndex * width;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }
    __syncthreads();
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if( (xIndex < height)&&(yIndex < width) ){
        unsigned int index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Transpose Timings

- Speedups with coalescing
 - 128x128: 0.011ms vs. 0.022ms (2.0X speedup)
 - 512x512: 0.07ms vs. 0.33ms (4.5X speedup)
 - 1024x1024: 0.30ms vs. 1.92ms (6.4X speedup)
 - 1024x2048: 0.79ms vs. 6.6ms (8.4X speedup)
- (Note: 10% speedup is achieved by eliminating shared memory bank conflicts)

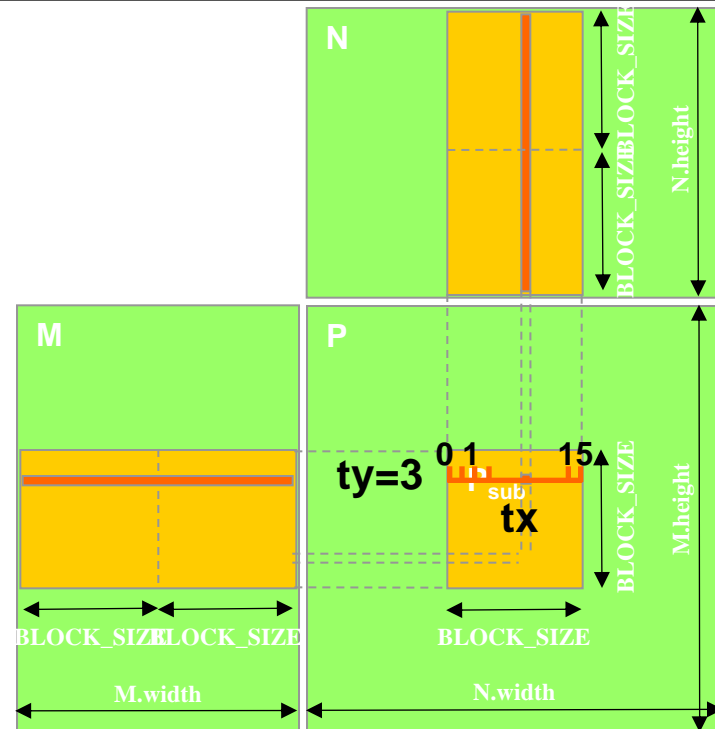
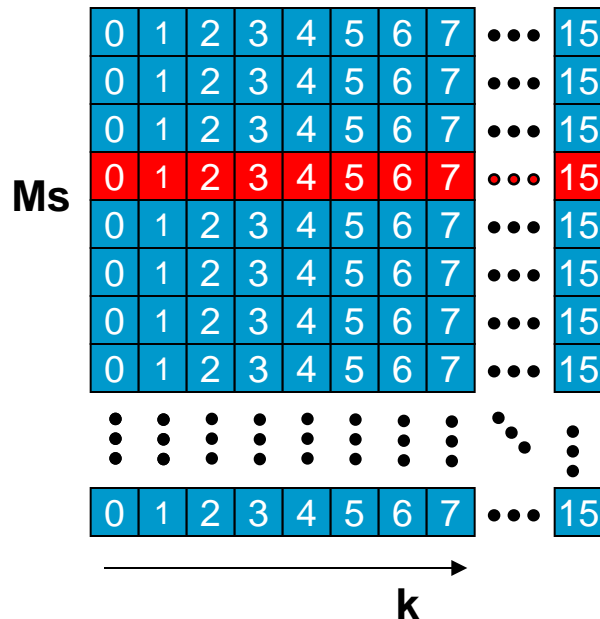
How about Matrix Multiplication?

- Matrix $P = M \times N$, (16x16)

When reading M s, half-warps access the same bank, **broadcast!**

```
for (int k = 0; k < 16; ++k)    //ty is constant over a half-warp
```

```
    Csub += Ms[ty][k] * Ns[k][tx]; //here k is fixed over a half-warp
```



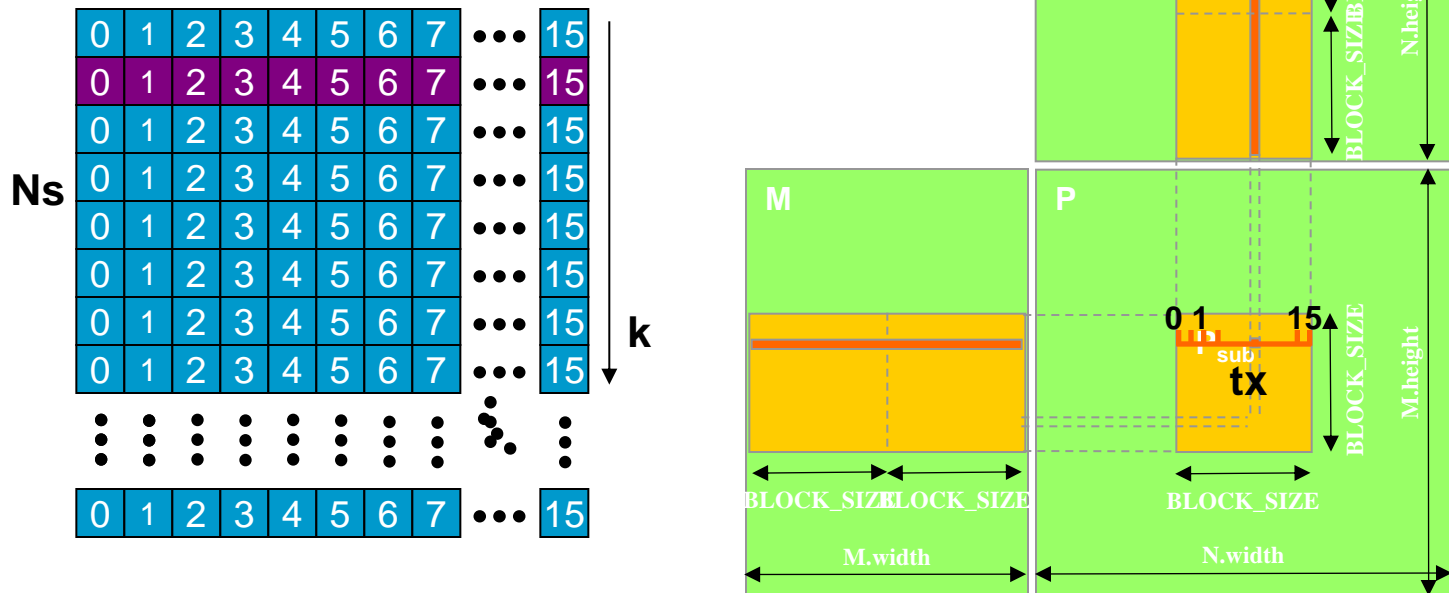
How about Matrix Multiplication?

- Matrix $P = M \times N$, (16x16 block)

When reading N s, half-warps access different banks, **no conflicts!**

```
for (int k = 0; k < 16; ++k)    //tx is constant over a half-warp
```

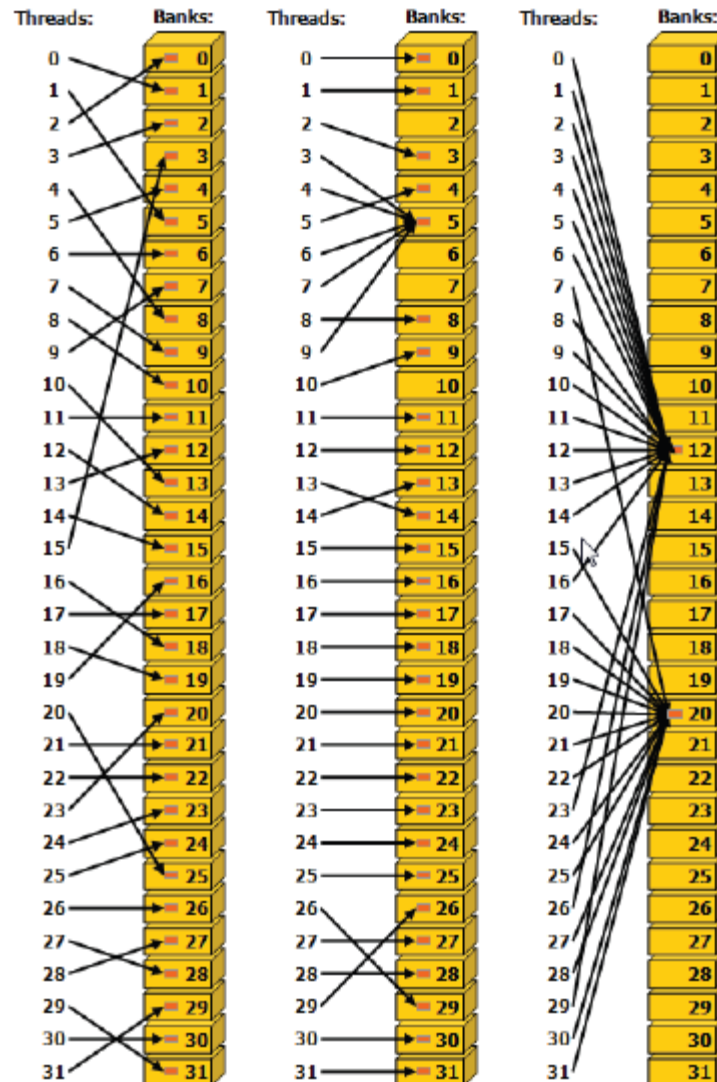
```
    Csub += Ms[ty][k] * Ns[k][tx];
```



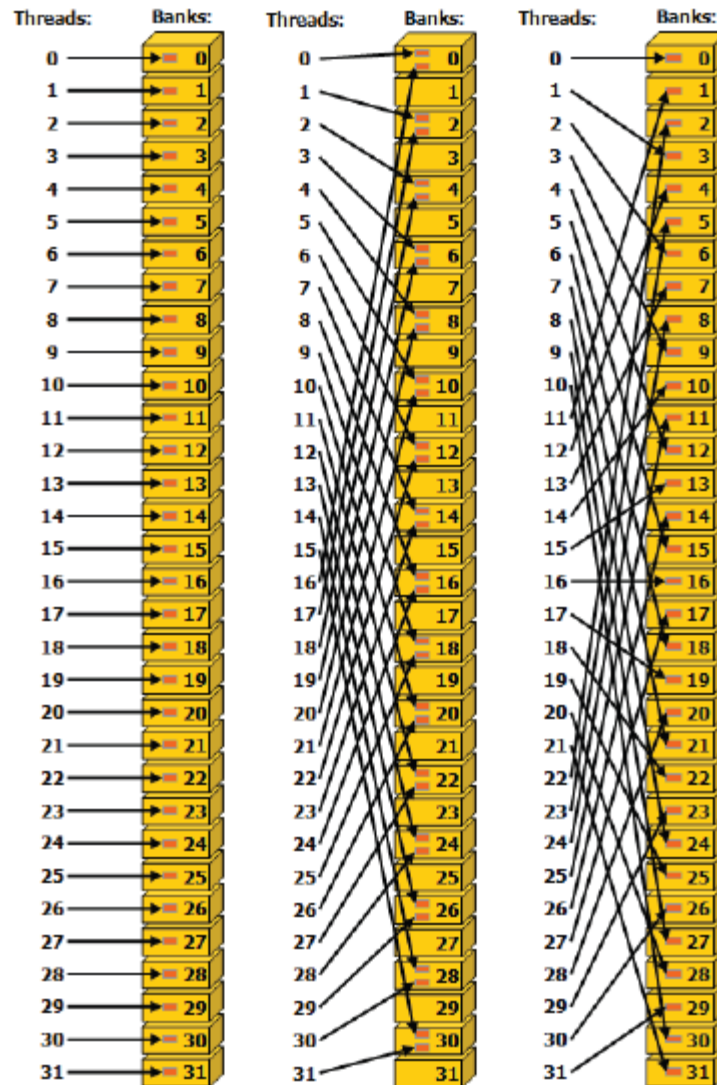
Shared Memory Access (Capability 2.x)

- Shared memory has 32 banks
- Successive 32-bit words map to successive banks
- Each bank has a bandwidth of 32 bits per two clock cycles
- 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

Shared Memory Access (Capability 2.x)



Shared Memory Access (Capability 2.x)



Shared Memory Access (Capability 3.x)

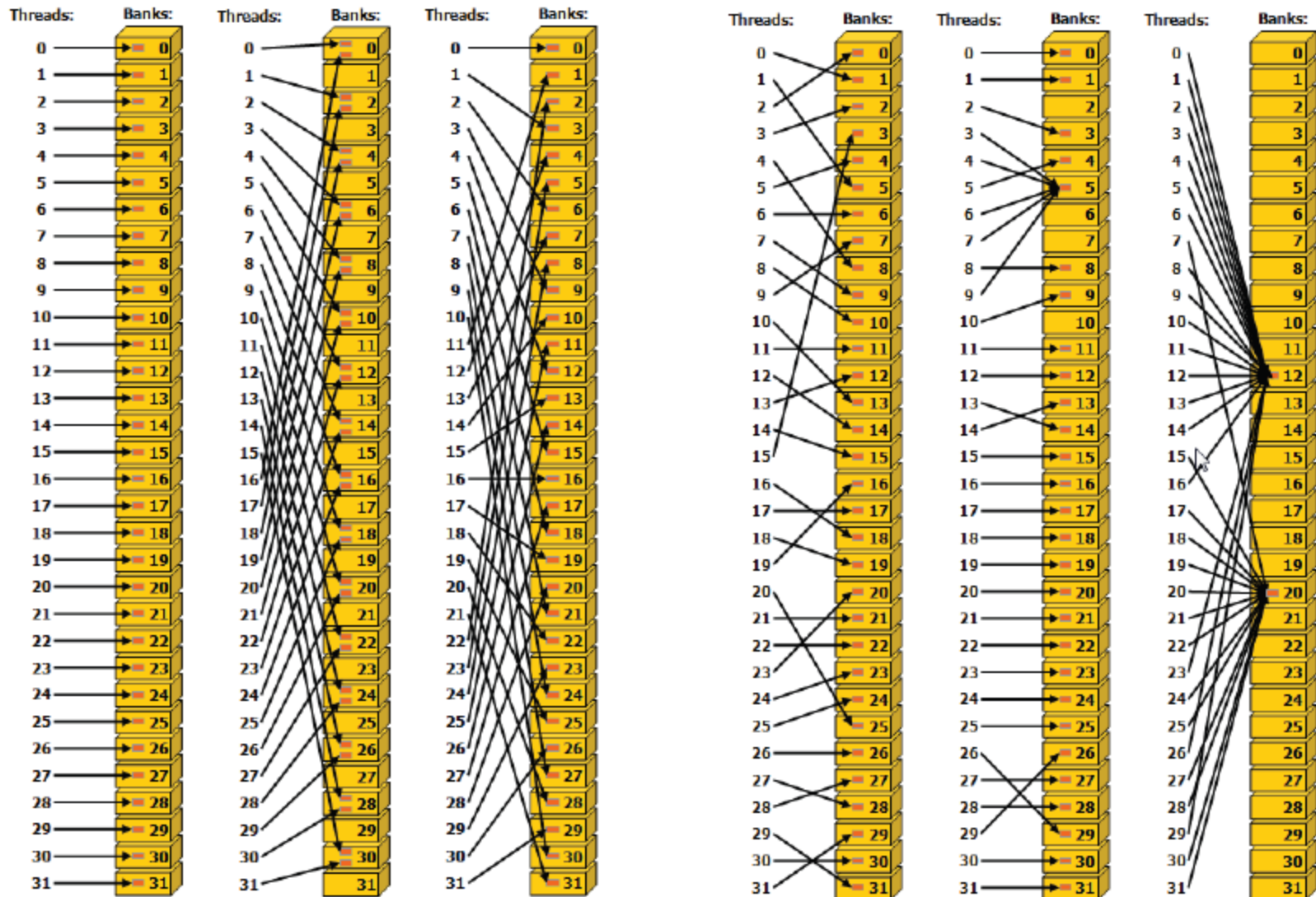
- Shared memory has 32 banks with two addressing modes
- 64-bit mode: successive 64-bit words map to successive banks
- 32-bit mode: successive 32-bit words map to successive banks
- Each bank has a bandwidth of 64 bits per two clock cycles

Shared Memory Access (Capability 3.x)

■ 64-bit mode

- A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 64-bit word (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 64-bit word is broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined).

Shared Memory Access (Capability 3.x)



CUDA Performance Optimization

- Optimization strategies
- Maximize utilization
- Maximize memory throughput
- Maximize instruction throughput
- Others

Instruction Throughput

- Use intrinsic functions, less accurate, but faster device-only version
 - `__pow`, `__log`, `__log2`, `__log10`, `__exp`,
`__sin`, `__cos`, `__tan`, etc.
- Minimize divergent warps
- Reduce the number of synchronizations

CUDA Performance Optimization

- Optimization strategies
- Maximize utilization
- Maximize memory throughput
- Maximize instruction throughput
- Others

Unrolling

```
Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads();

    // compute results for tile
    for (i = 0; i < 16; i++)
    {
        Ctemp += As[ty][i]
                 * Bs[i][tx];
    }

    __syncthreads();
}
C[indexC] = Ctemp;
```

(b) Tiled Version

```
Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads();

    // compute results for tile
    Ctemp +=
        As[ty][0] * Bs[0][tx];
    ...
    Ctemp +=
        As[ty][15] * Bs[15][tx];

    __syncthreads();
}
C[indexC] = Ctemp;
```

(c) Unrolled Version

Instruction Mix Considerations

- `for (int k = 0; k < BLOCK_SIZE; ++k)`
 `Pvalue += Ms[ty][k] * Ns[k][tx];`
- There are very few mul/add between branches and address calculation
- Loop unrolling can help

`Pvalue += Ms[ty][0] * Ns[0][tx] + ... + Ms[ty][15] *
Ns[15][tx];`

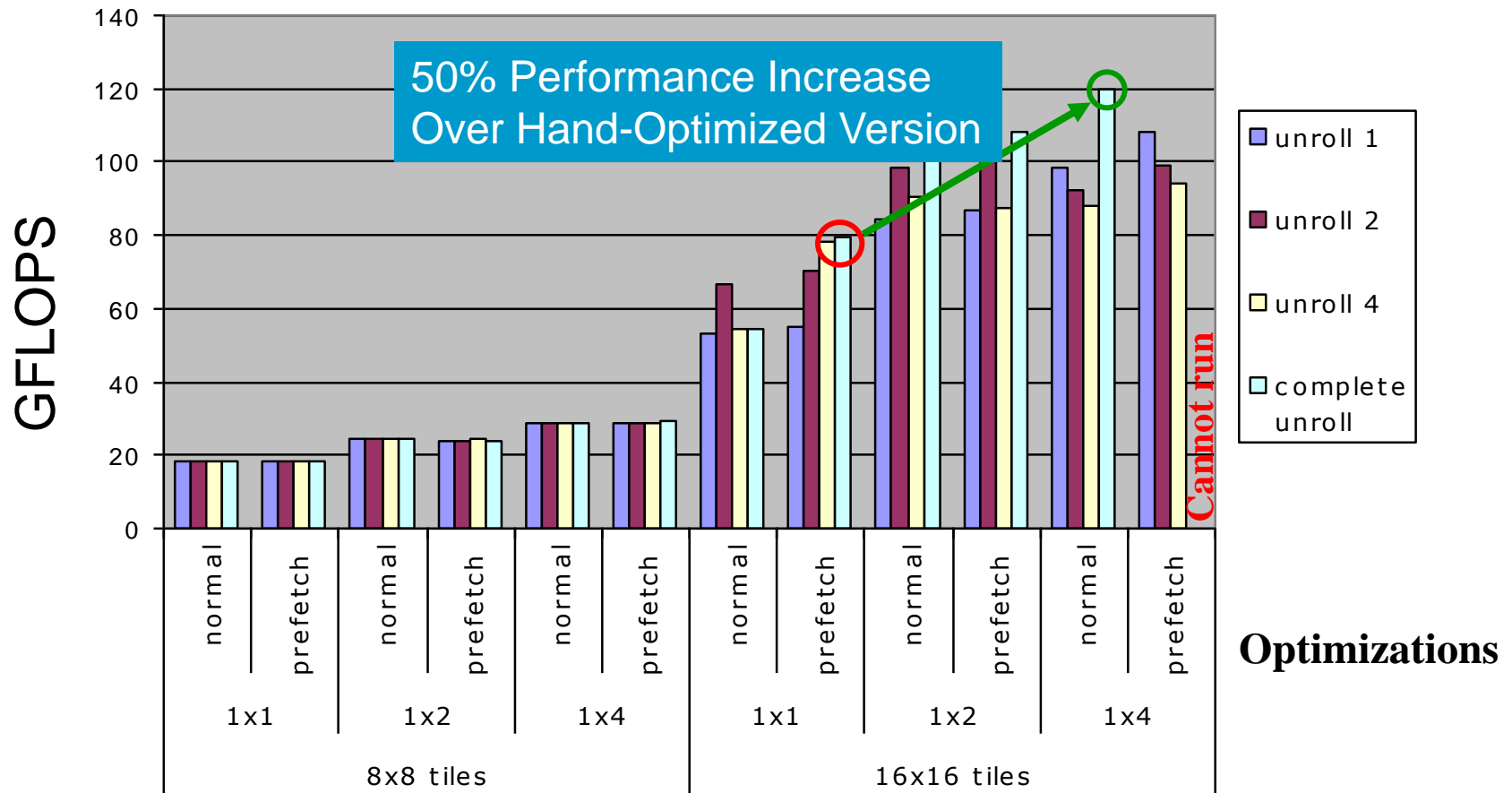
Prefetching

- One could double buffer the computation, getting better instruction mix within each thread
 - This is classic software pipelining in ILP compilers

```
Loop {  
    Load current tile to shared  
    memory  
  
    syncthreads()  
  
    Compute current tile  
  
    syncthreads()  
}
```

```
Load next tile from global  
memory  
  
Loop {  
    Deposit current tile to shared  
    memory  
    syncthreads()  
  
    Load next tile from global  
    memory  
  
    Compute current tile  
  
    syncthreads()  
}
```

Matrix Multiplication Space



Summary

- In decreasing order of importance
 - Algorithm parallelization
 - Asynchronous concurrent execution
 - Use a multiple of 32 threads per block and at least as many blocks as multiprocessors
 - Access global memory properly
 - Avoid shared memory bank conflicts
 - Use "-use-fast-math" option if you have operations like sine and cosine on the device
 - Have as few branching conditional loops as possible
 - Have small loops unrolled
 - Have no unnecessary `__syncthreads()` calls
- The defaults in the compiler have basically always been the best