

高性能计算的新发展

基于图形处理器的并行计算及CUDA编程

Ying Liu, Associate Prof., Ph.D

Graduate University of Chinese Academy of Sciences
Research Center on Fictitious Economy and Data Science

Performance Optimization

- Optimization strategies
- Memory coalescing
- Memory bank conflicts
- Prefetching
- Loop unrolling

CUDA Optimization Strategies

- Maximize the independence between threads
- Maximize the computational intensity (math/bandwidth)
- Minimize data transfer between GPU and CPU
 - 4GB/s peak (PCI-e x16) vs. 86.4 GB/s peak (G80)
 - Use shared memory wisely

CUDA Optimization Strategies

- In decreasing order of importance
 - 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
 - Have as few branching conditional loops as possible
 - Have small loops unrolled
 - Have no unnecessary `__syncthreads()` calls

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 `CC -maxrregcount = N`
 - $N = \text{desired maximum registers / kernel}$
 - At some point “spilling” into local memory may occur
 - Reduce performance – local memory is slow
 - Check .cubin file for local memory usage

Performance Optimization

- Optimization strategies
- **Memory coalescing**
- Memory bank conflicts
- Prefetching
- Loop unrolling

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

Coalescing

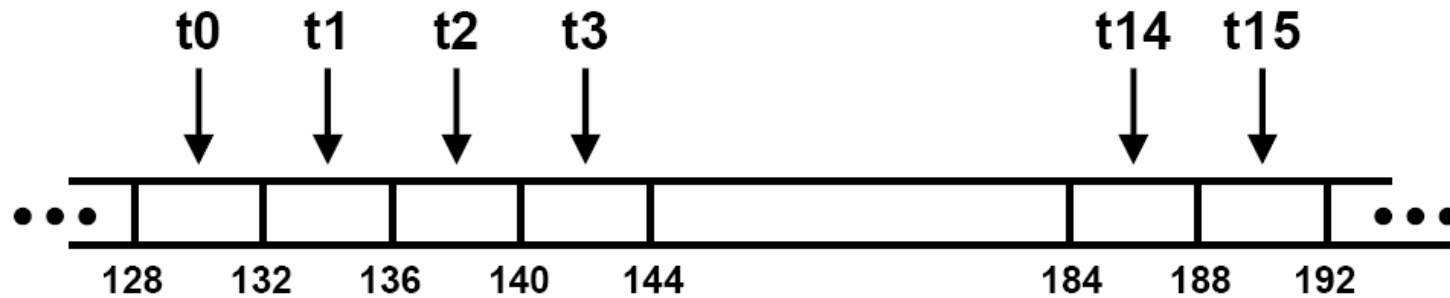
- Threads in a half-warp can be **coalesced** into a single memory transaction
- Size of a memory transaction
 - 64 bytes - each thread reads a word: int, float, ...
 - 128 bytes - each thread reads a double-word: int2, float2, ...

Coalescing

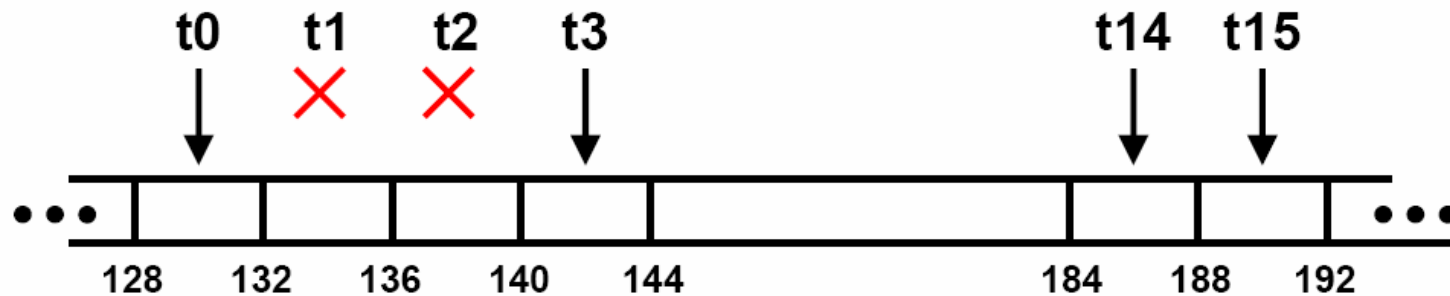
■ Requirements

- 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 of size equal to the memory
- 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)

Coalesced Access: Reading floats

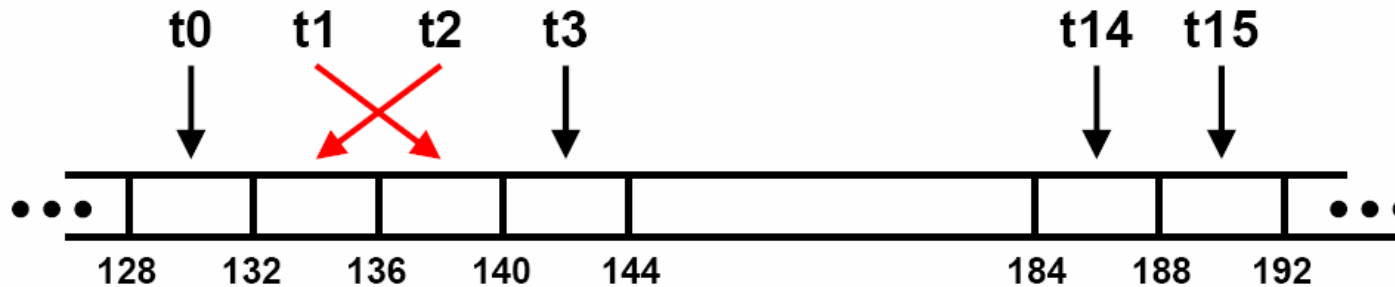


All threads participate

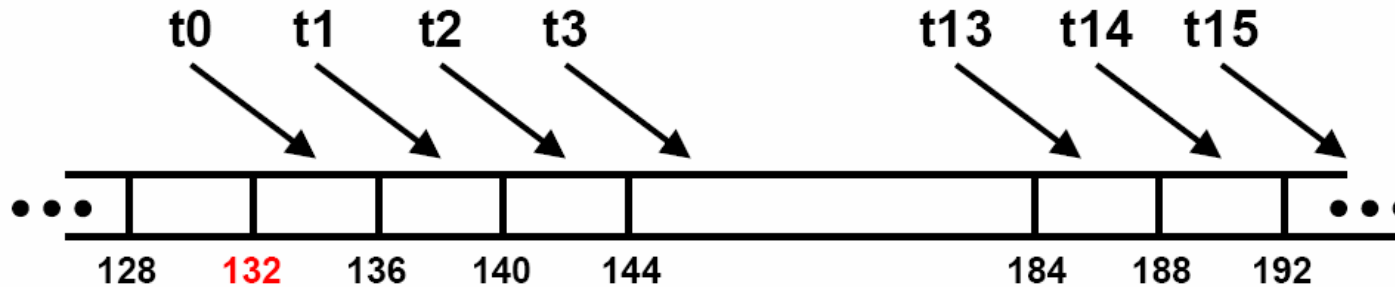


Some Threads Do Not Participate

Uncoalesced Access: Reading floats



Permuted Access by Threads



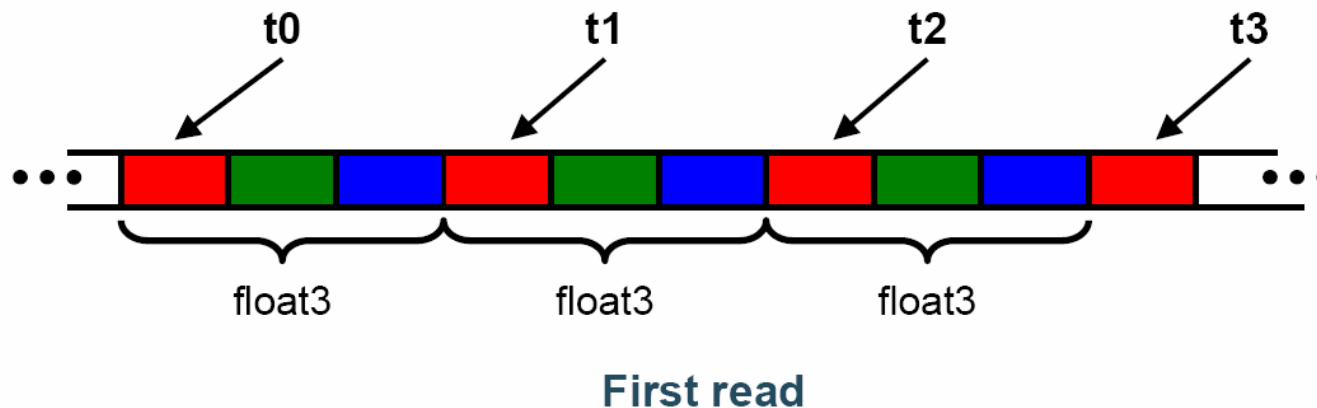
Misaligned Starting Address (not a multiple of 64)

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;  
}
```

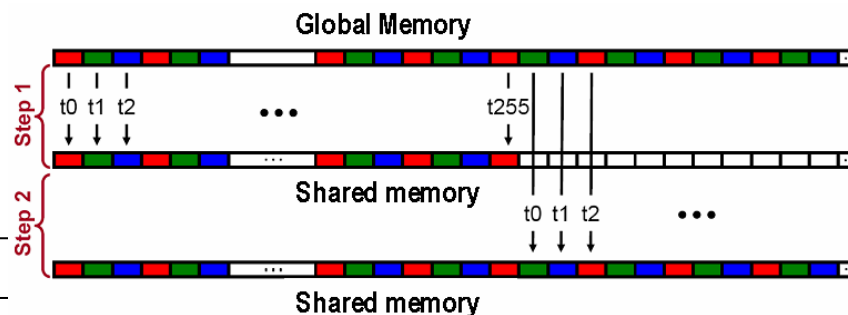
Uncoalesced float3 Code

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



Coalescing float3 Access

- Use shared memory to allow coalescing
 - 256 threads per block
 - A thread block needs $\text{sizeof}(\text{float3}) \times 256$ 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

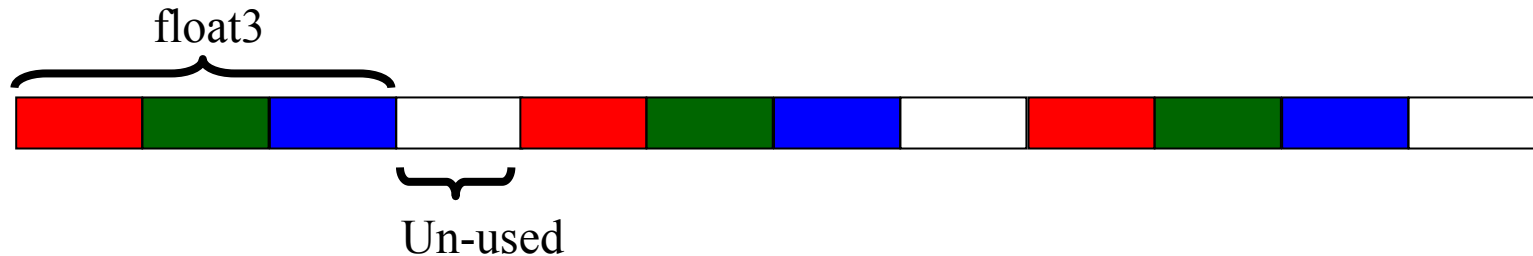
Write the result through SMEM

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

- Compiler enforces the alignment specifiers

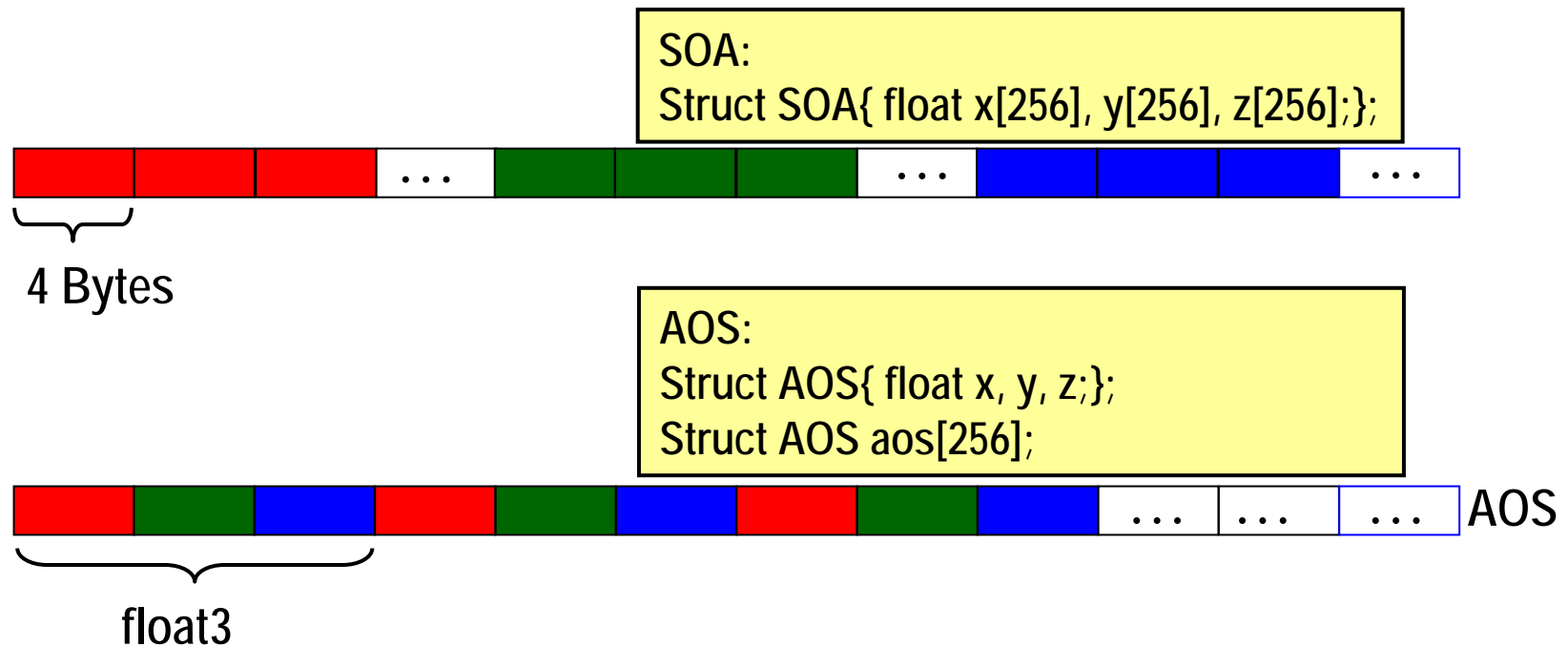
- `__align__(X)`, where $X = 4, 8,$ or 16

- ```
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

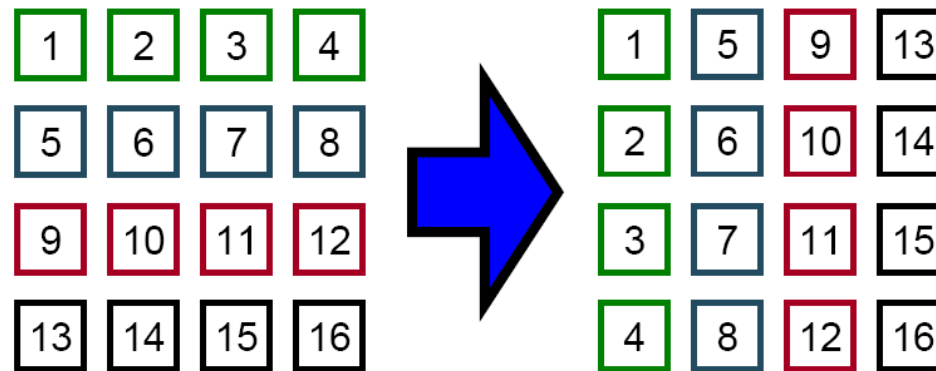
---

- 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

# Matrix Transpose

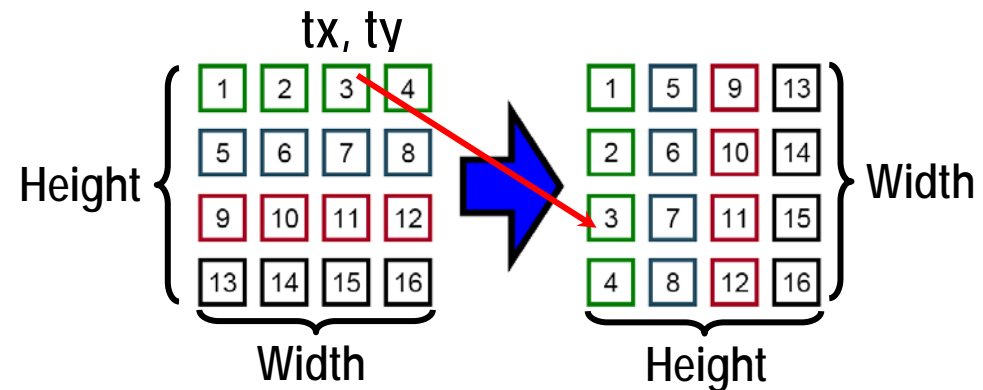
---

- SDK Sample (“transpose”) illustrates memory coalescing

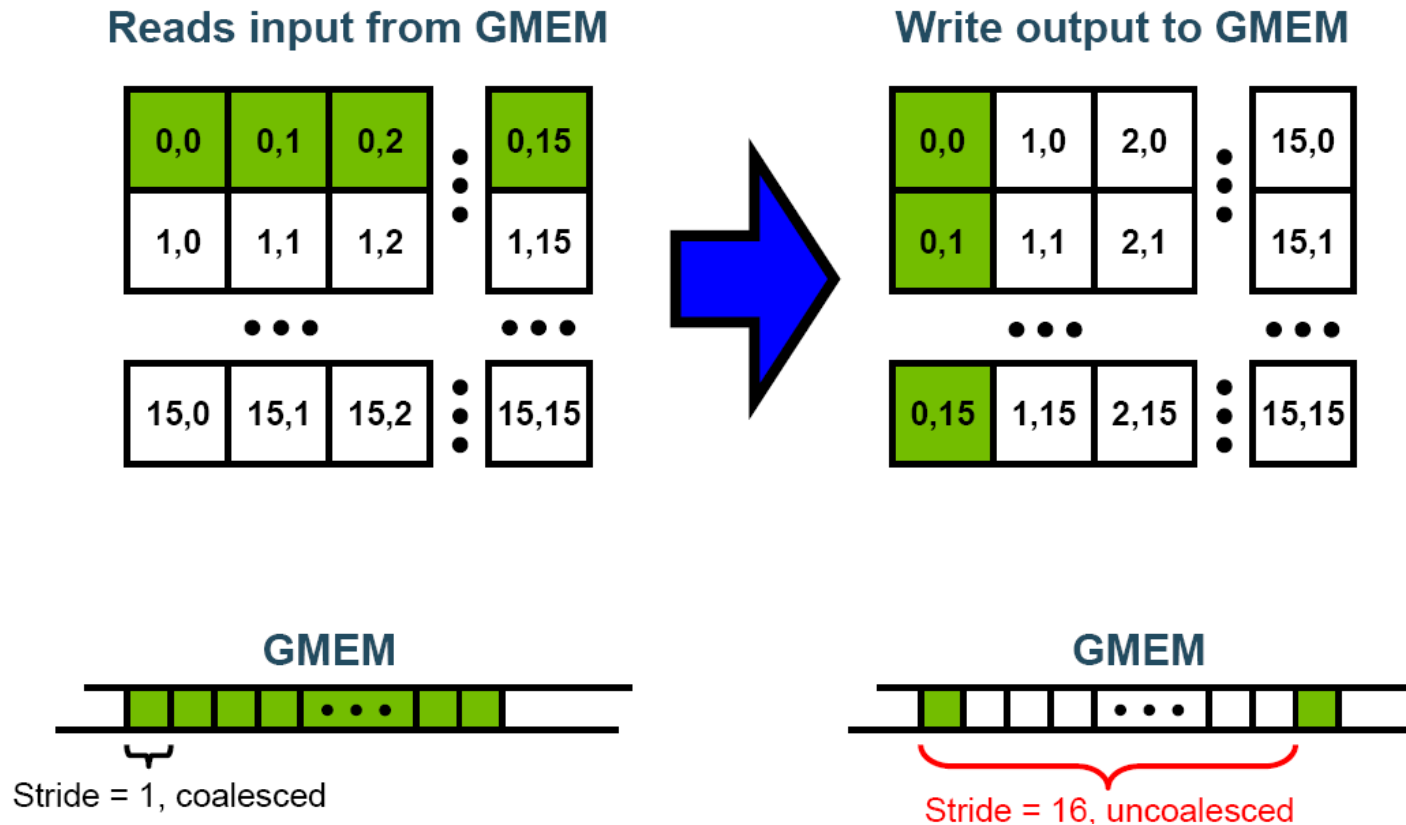


# Uncoalesced Transpose

```
__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];
 }
}
```

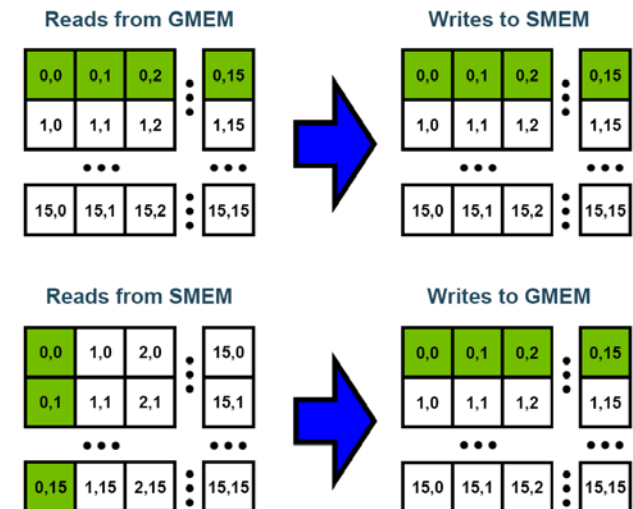


# Uncoalesced Transpose

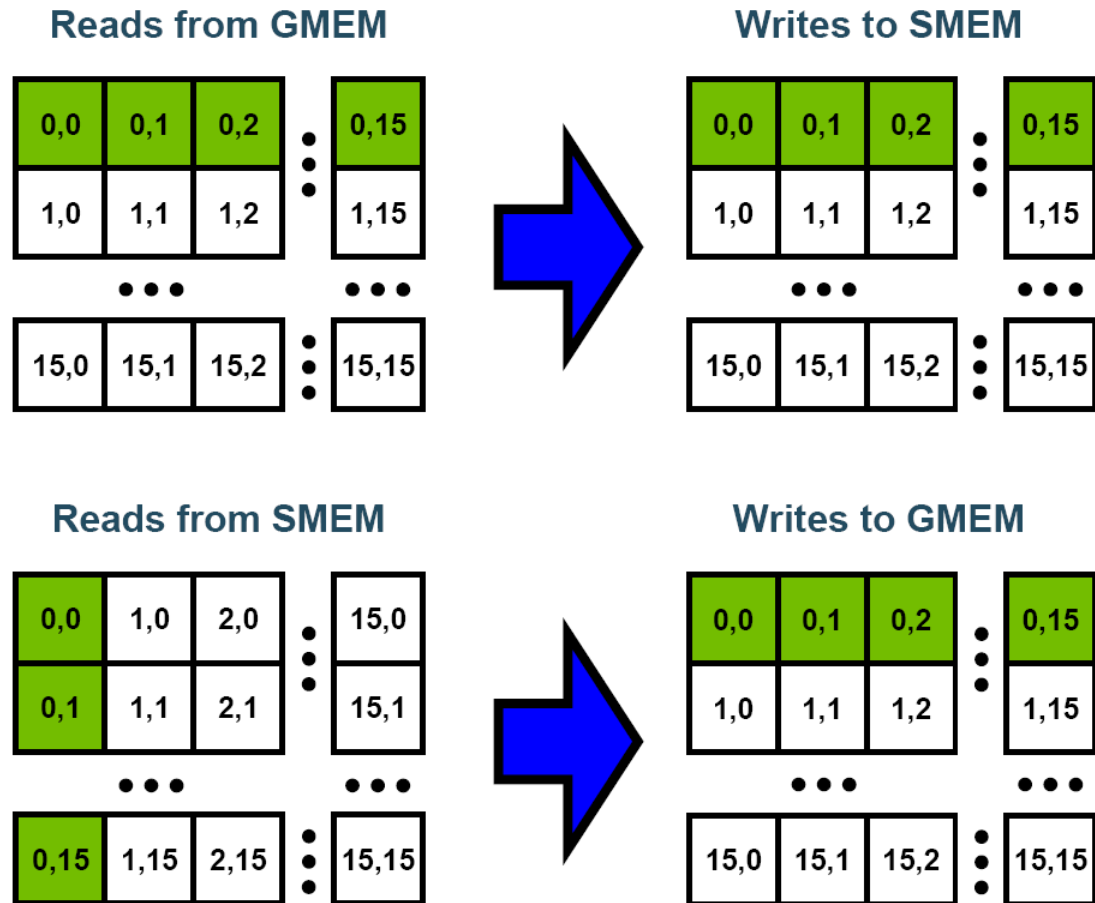


# Coalesced Transpose

- 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):
  - Reads element (tx, ty) from input tile
  - Writes 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];
}
```

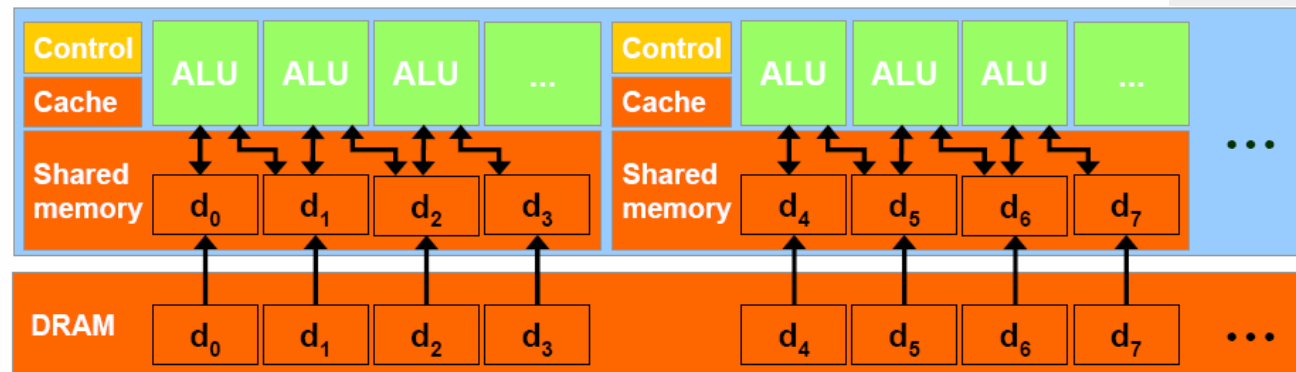
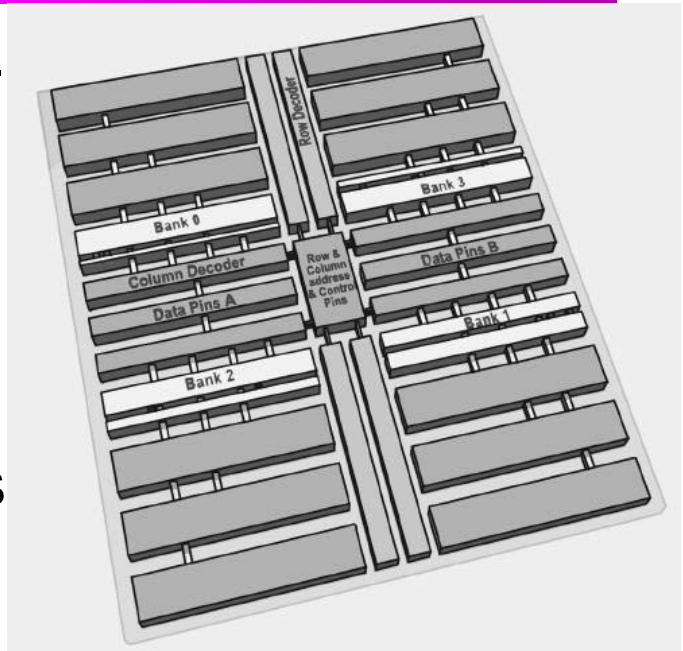
# Performance Optimization

---

- Optimization strategies
- Memory coalescing
- **Memory bank conflicts**
- Prefetching
- Loop unrolling

# Shared Memory

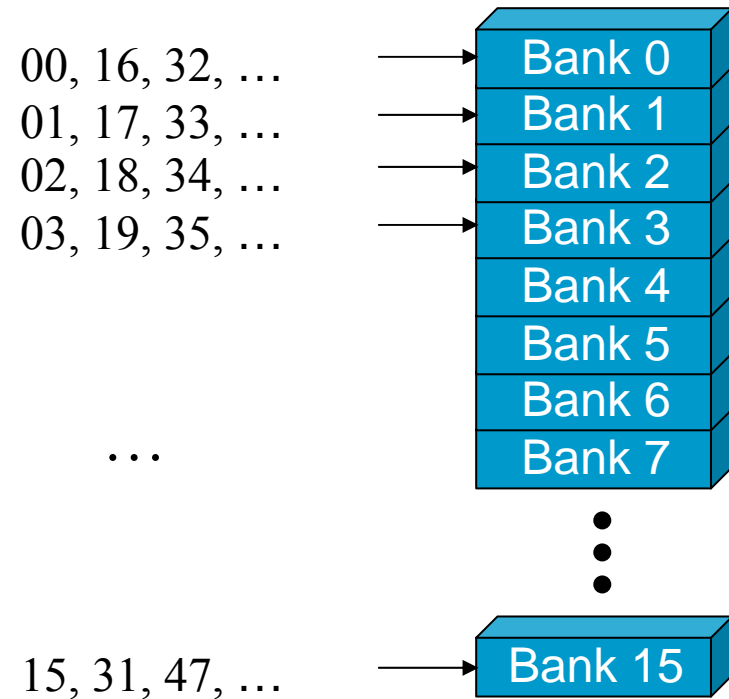
- Shared memory is divided into equally-sized memory modules (banks)
- 16 banks on devices of compute capability 1.x
- A memory can service as many simultaneous accesses as it has banks
- One read/write per cycle per bank



➔ Big memory bandwidth savings

# How Addresses Map to Banks on G80

- The bandwidth of each bank is 32 bits (4Bytes) per clock cycle
- 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!!!**

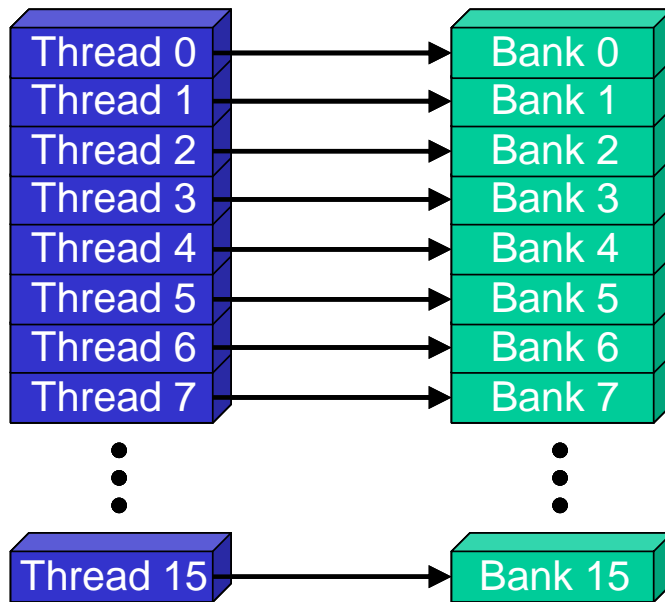


- Must be serialized

# Bank Addressing Example

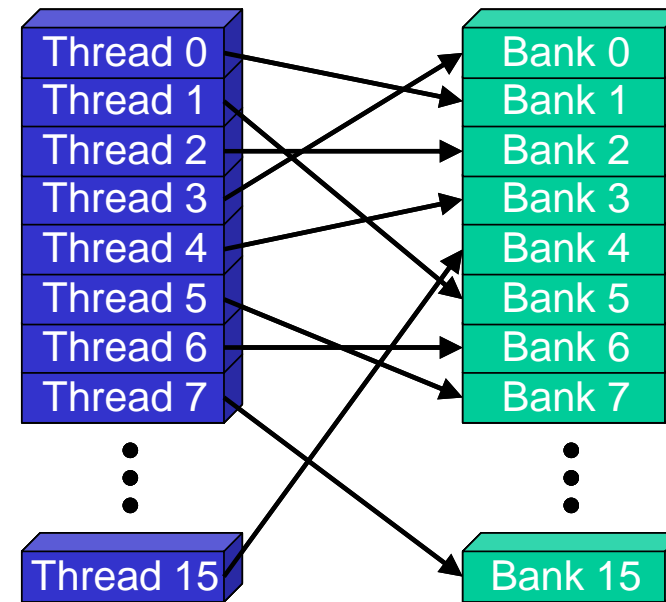
## ■ No Bank Conflicts

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



## ■ No Bank Conflicts

- Random 1:1 Permutation

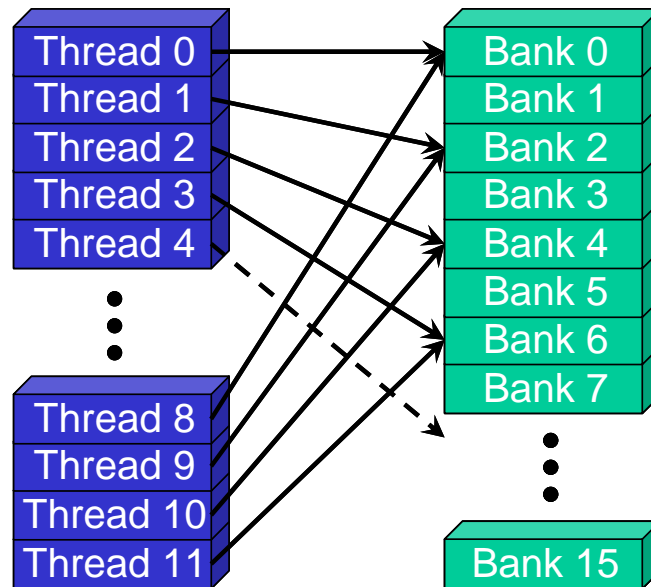


```
__shared__ float shared[256];
float foo = shared[base + s * threadIdx.x];
```

# Bank Addressing Example

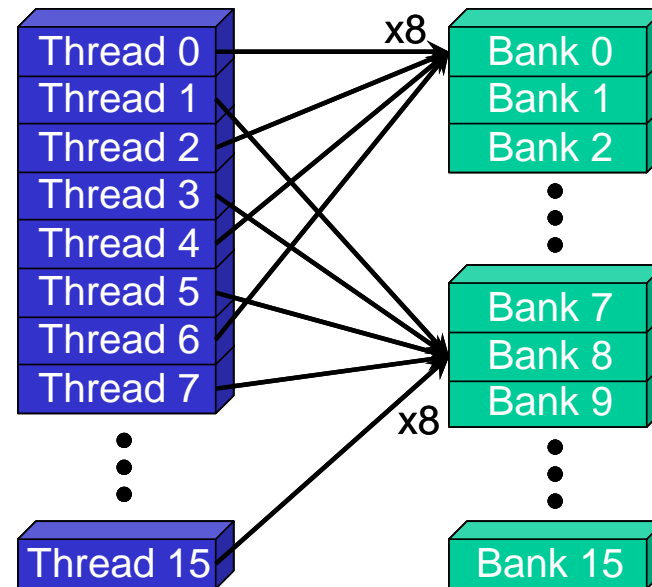
## ■ 2-way bank conflicts

- 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 the same fast as register
- 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

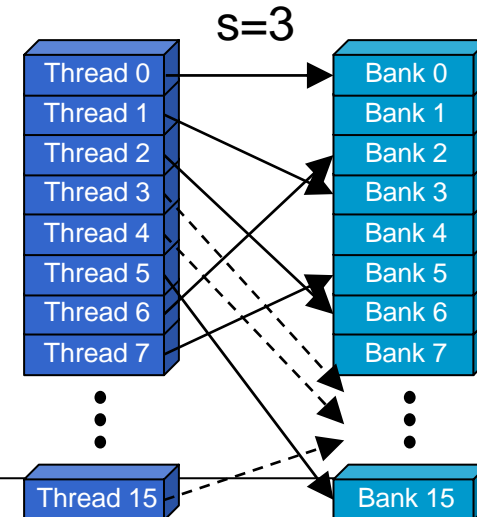
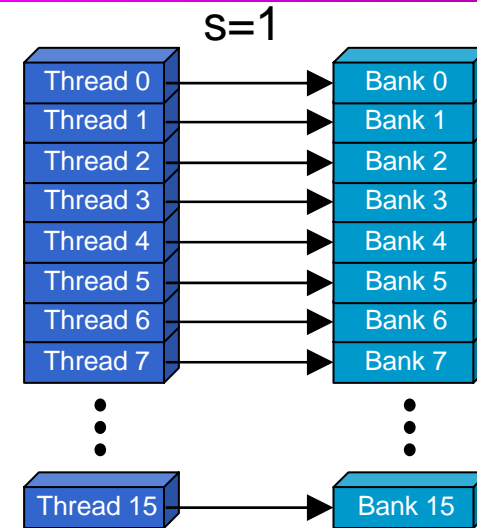
- Given:

```
__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

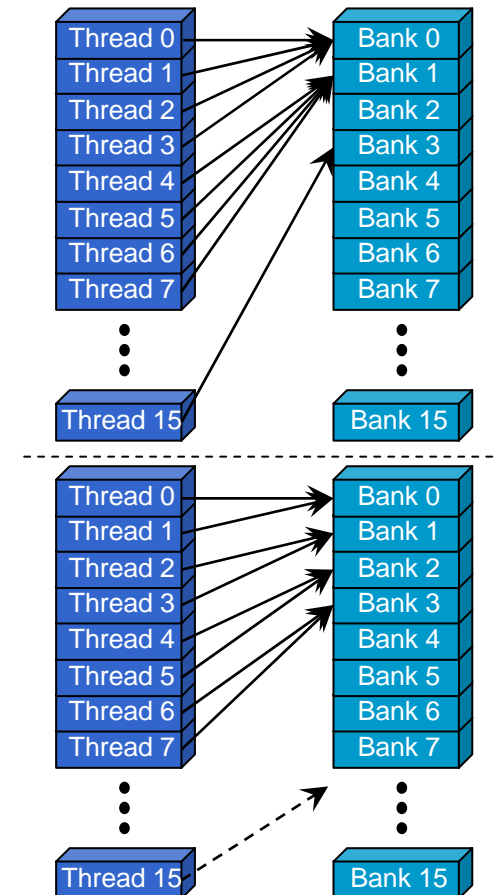
```
__shared__ char shared[];
```

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

- 2-way bank conflicts

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

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



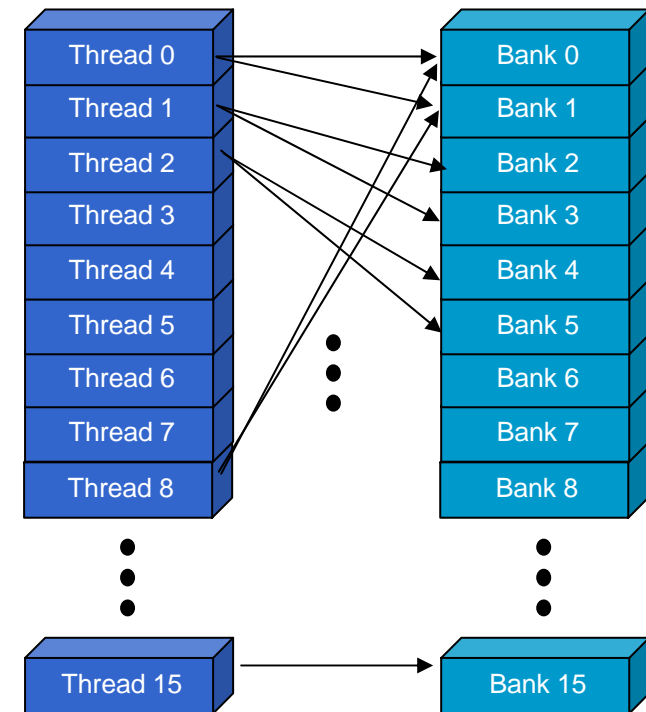
# Structs and Bank Conflicts

- 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

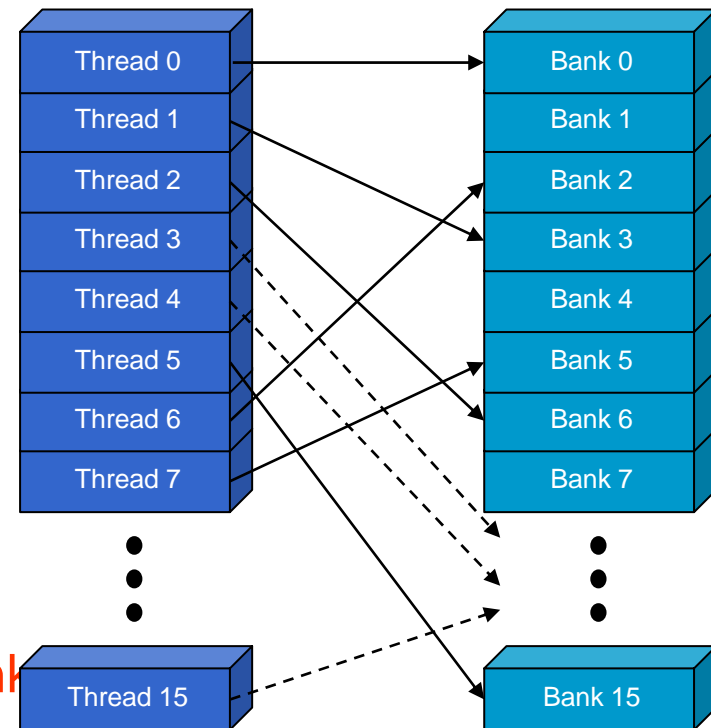


# Structs and Bank Conflicts

- 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];
```

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



# Bank Conflict - 1D

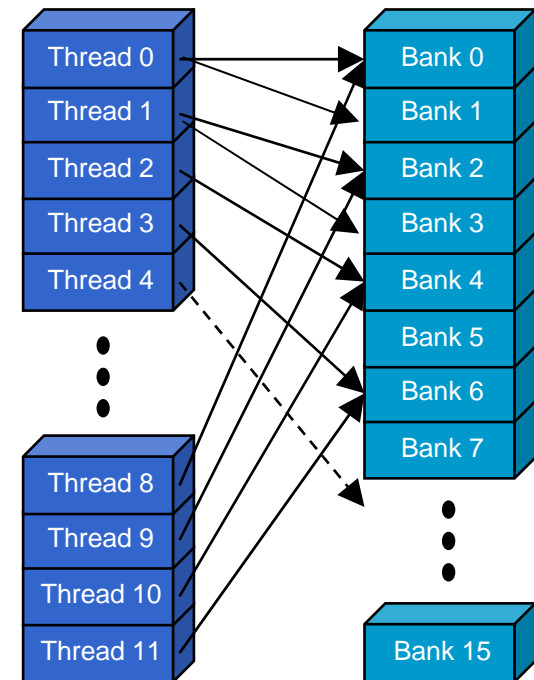
- Each thread loads 2 words to shared memory:
  - 2-way-interleaved loads result in 2-way bank conflicts:

```
int tid = threadIdx.x;
```

```
shared[2*tid] = global[2*tid];
```

```
shared[2*tid+1] = global[2*tid+1];
```

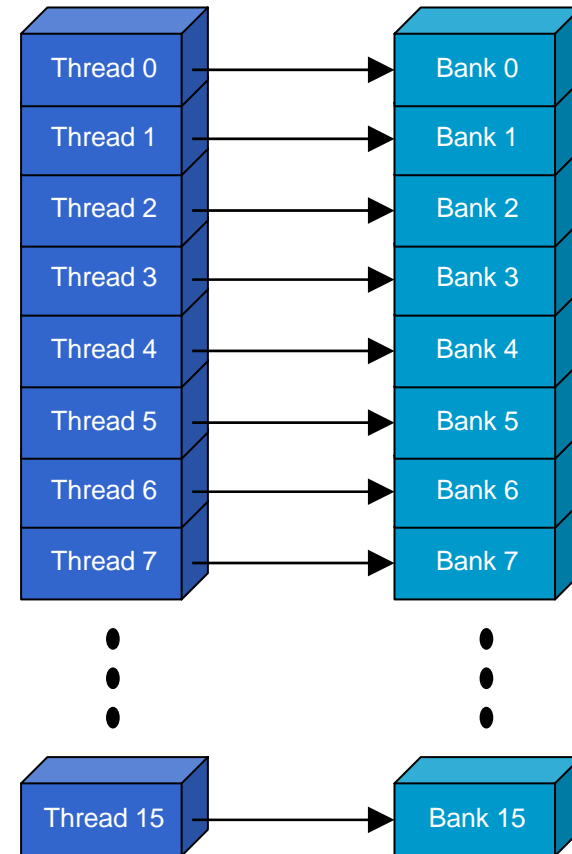
- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic
  - Not the case in shared memory where there is no cache line effects but banking effects



# A Better Access Pattern

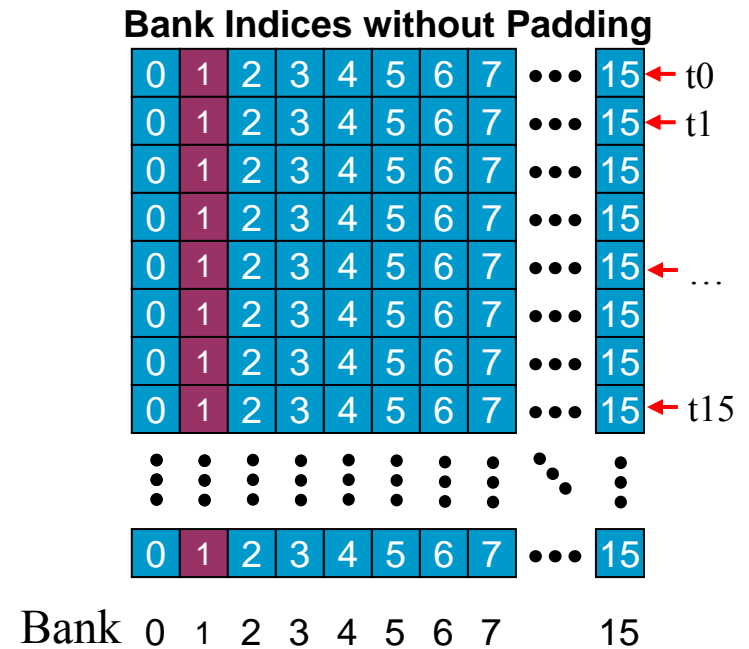
- Each thread loads one element in every consecutive group of `blockDim` elements

```
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 | ... | 7  | 8  |
| ⋮  | ⋮ | ⋮ | ⋮  | ⋮  | ⋮  | ⋮  | ⋮  | ⋮   | ⋮  | ⋮  |
| 15 | 0 | 1 | 2  | 3  | 4  | 5  | 6  | ... | 14 | 15 |

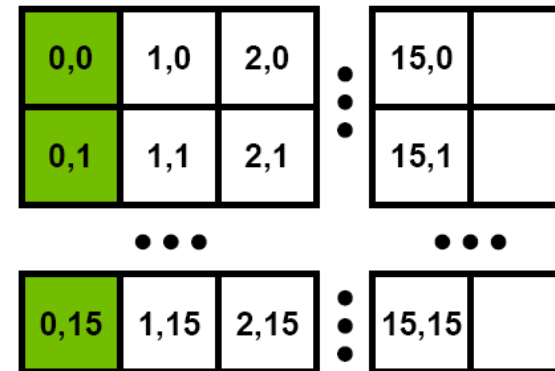
# Shared Memory Optimization

- 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)

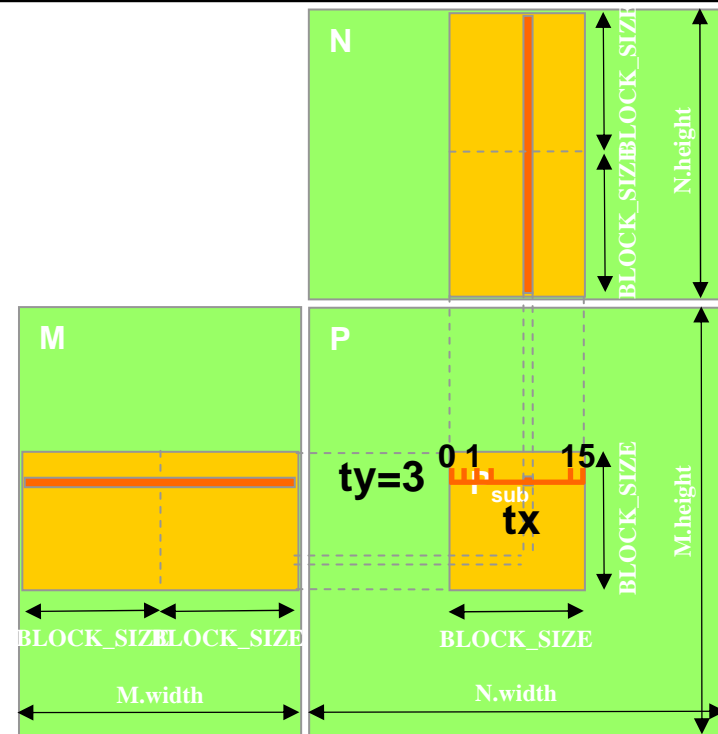
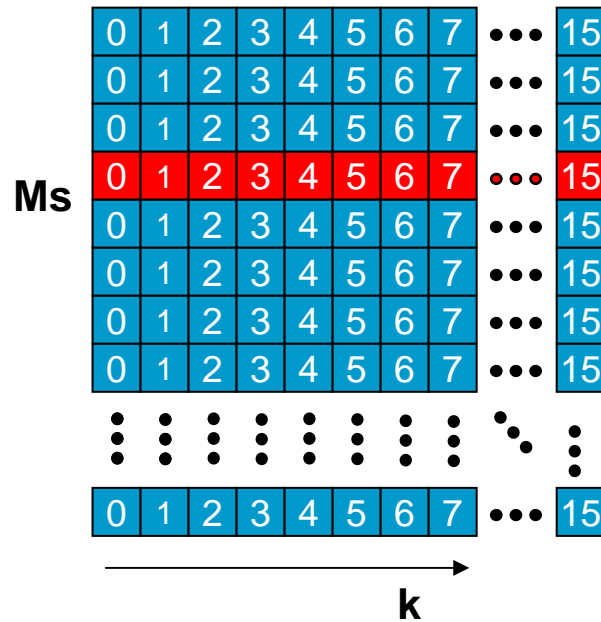
# Then How about Matrix Multiplication?

- 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
```



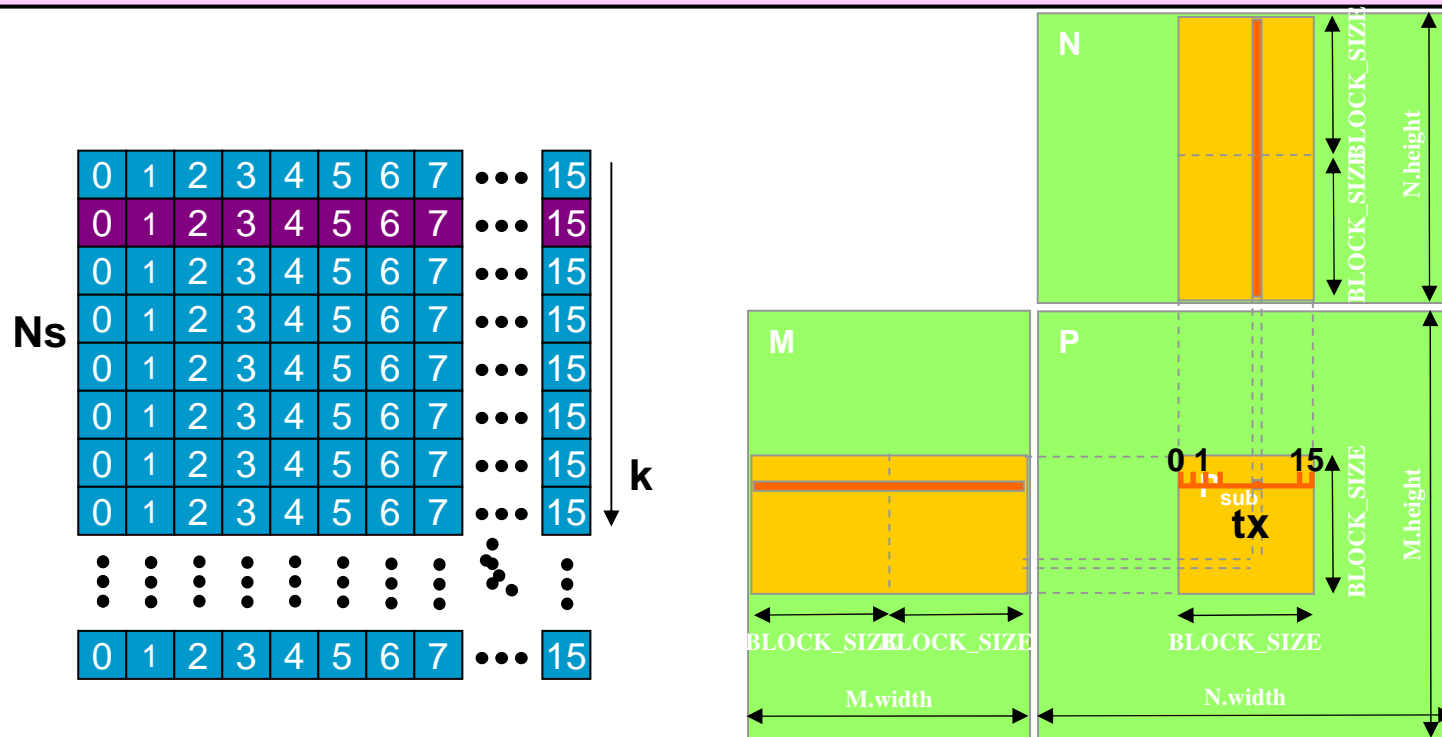
# How about Matrix Multiplication?

- 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];
```



# Performance Optimization

---

- Optimization strategies
- Memory coalescing
- Memory bank conflicts
- Loop unrolling
- Prefetching

# 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
```

# 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][k] \* Ns[k][tx] + ... + Ms[ty][k+15] \*  
Ns[k+15][tx];

# Performance Optimization

---

- Optimization strategies
- Memory coalescing
- Memory bank conflicts
- Loop unrolling
- Prefetching

# 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

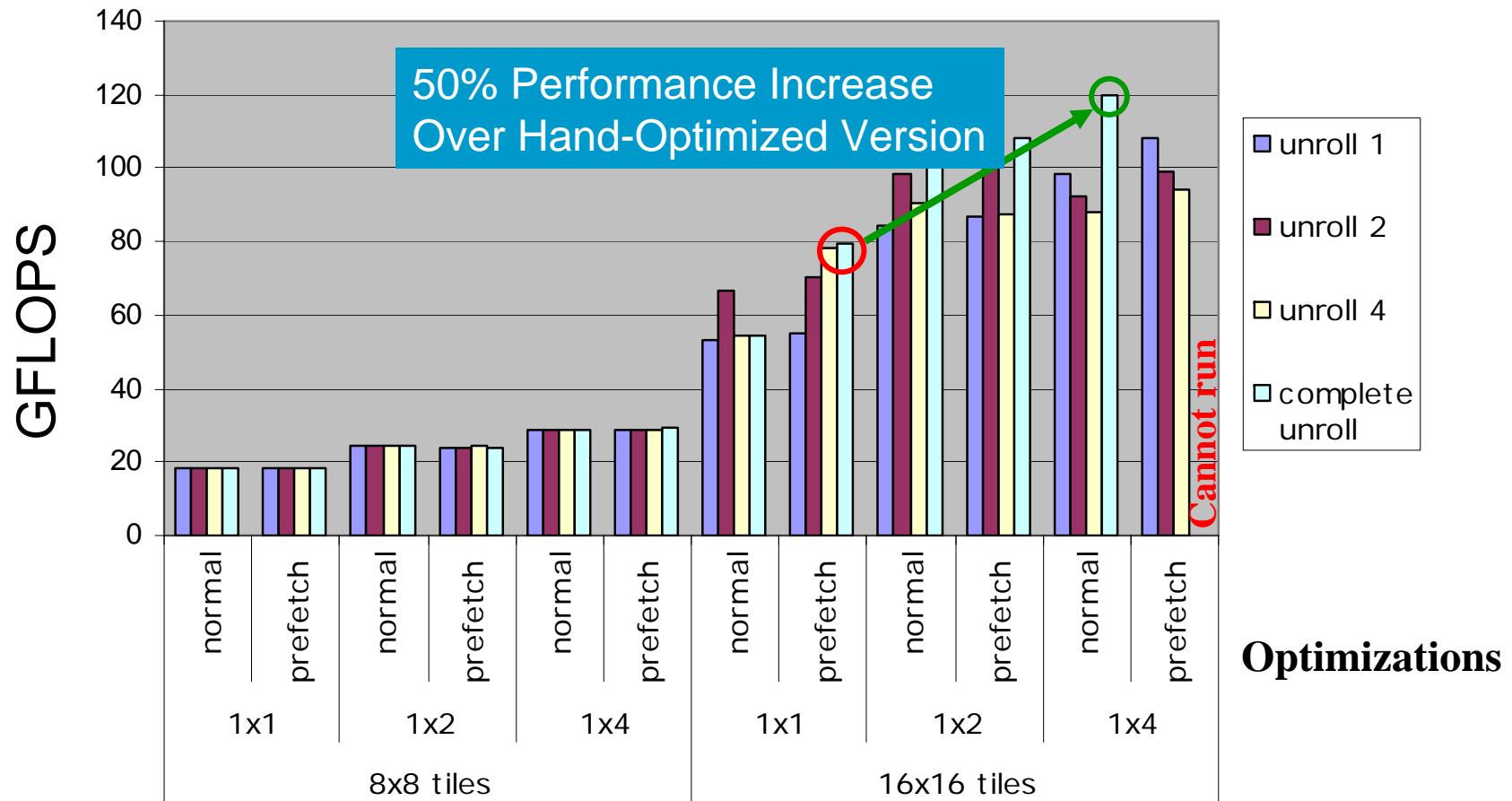
Loop {
Deposit current tile to shared
memory
syncthreads()

Load next tile from global
memory

Compute current subblock

syncthreads()
}
```

# Matrix Multiplication Space



# Asynchronous Concurrent Execution

---

- In order to facilitate concurrent execution between host and device, some runtime functions are asynchronous
- Control is returned to the application before the device has completed the requested task

For example:

```
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.

# Asynchronous Concurrent Execution

---

- Kernel launches through `__global__` functions or `cuLaunchGrid()` and `cuLaunchGridAsync()`;
- CUDA Memory Management API:
  - The functions that perform memory copies suffixed with **Async**;
  - The functions that perform device ↔ device memory copies;
  - The functions that set memory;
  - Some functions that perform page-locked host ↔ device memory copies. This capability is currently supported only for memory copies that do not involve CUDA arrays (used for texture fetching) or 2D arrays allocated through `cudaMallocPitch()`.

# Asynchronous Concurrent Execution

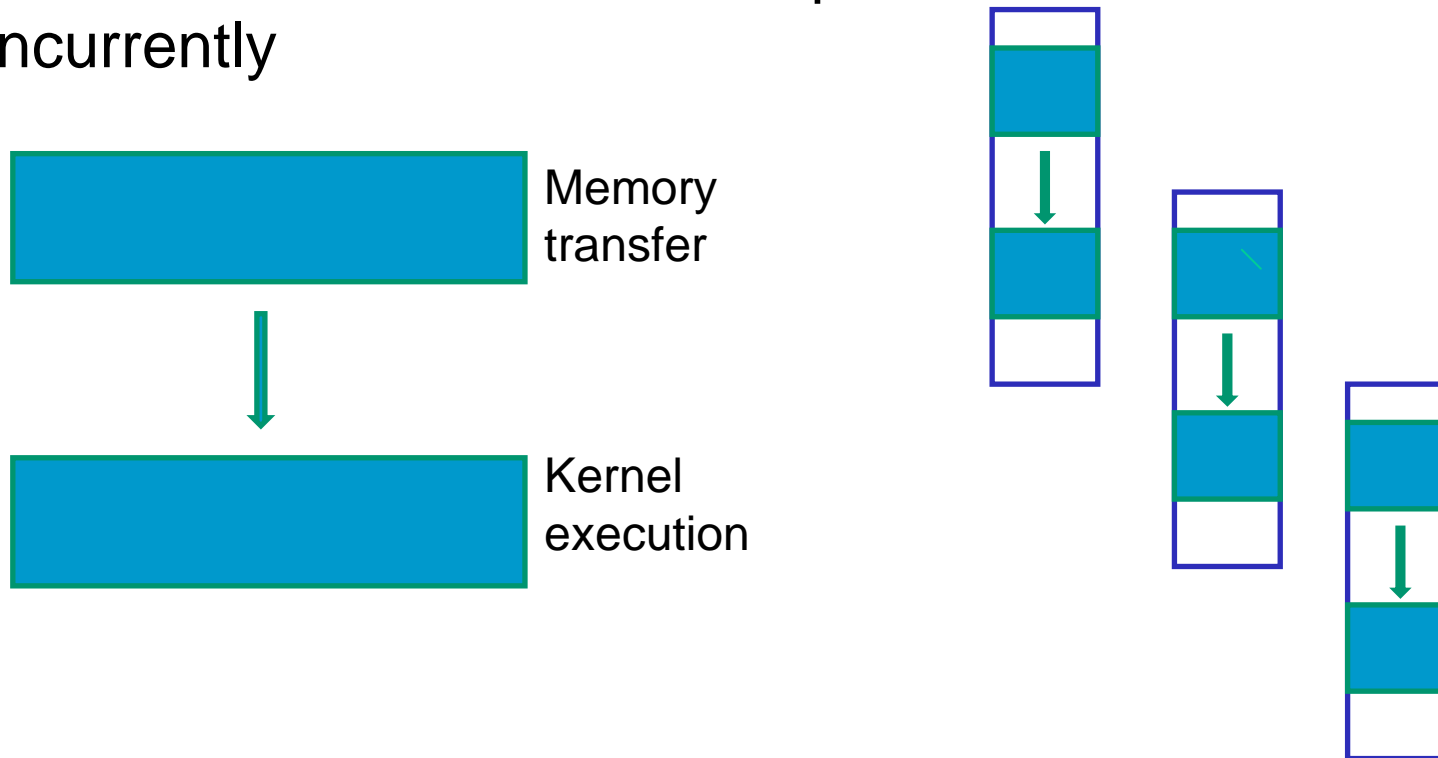
- Applications may query this capability by calling `cudaGetDeviceProperties()` and checking `deviceOverlap`

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      |
| Texture alignment:                             | 256 bytes         |
| Clock rate:                                    | 1.30 GHz          |
| <b>Concurrent copy and execution:</b>          | <b>Yes</b>        |

# Asynchronous Concurrent Execution

- Applications manage concurrency through *streams*. A *stream* is a sequence of operations that execute in order
- Different streams, on the other hand, 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]);
```

# Summary

---

- Focus on the "big" parts of the problem only and not to bother much about how slowly the "little" parts run
- Keep things as simple, clean and straightforward as possible
- Obey Nvidia's advice on memory access and block/grid configuration, e.g., and let the compiler and hardware have free reign over the code generation and execution
- Algorithm optimization/effective problem parallelization is probably far more profitable (with the exception of loop unrolling that has significantly helped in some cases)
- The defaults in the compiler have basically always been the best
- Use "-usefastmath" option if you have operations like sine and cosine on the device