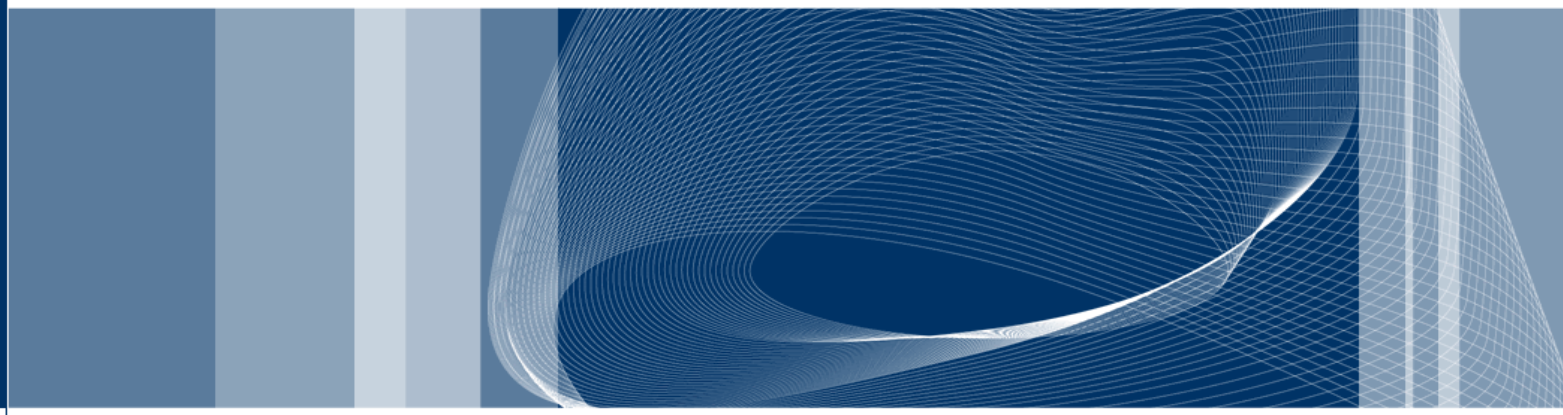
POLITECNICO DI MILANO



Performance optimization with CUDA

Algoritmi e Calcolo Parallelo

Daniele Loiacono

References

- This set of slides is mainly based on:
  - CUDA Technical Training, Dr. Antonino Tumeo, Pacific Northwest National Laboratory
  - Slide of *Applied Parallel Programming* (ECE498@UIUC) http://courses.engr.illinois.edu/ece498/al/
- Useful references
  - *Programming Massively Parallel Processors: A Hands-on Approach*, David B. Kirk and Wen-mei W. Hwu
  - http://www.gpgpu.it/ (CUDA Tutorial)
  - *CUDA Programming Guide* http://developer.nvidia.com/object/gpucomputing.html
  - *CUDA C Best Practices Guide* http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

Overview

Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host

Optimize Memory Access

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory
- Take advantages of shared memory
  - Hundreds of times faster than global memory
  - Threads can cooperate via shared memory
  - Use one / a few threads to load / compute data shared by all threads
  - Use it to avoid non-coalesced access: stage loads and stores in shared memory to re-order noncoalesceable addressing
- In shared memory, avoid high-degree bank conflicts

Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

Memory Optimization: Guidelines

Overview

- Optimizing host-device data transfers
- Coalescing global data accesses
- Using shared memory effectively

Host-Device Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 8GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX 280)
- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory
- Group transfers
  - One large transfer much better than many small ones

Page-Locked Data Transfers

- `cudaMallocHost()` allows allocation of page-locked ("pinned") host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2
- See the "bandwidthTest" CUDA SDK sample
- Use with caution!!
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits

Global and Shared Memory

- Global memory not cached on G8x GPUs
  - High latency, but launching more threads hides latency
  - Important to minimize accesses
  - Coalesce global memory accesses (more later)
- Shared memory is on-chip, very high bandwidth
  - Low latency
  - Like a user-managed per-multiprocessor cache
  - Try to minimize or avoid bank conflicts (more later)

Memory Optimization: Coalescing

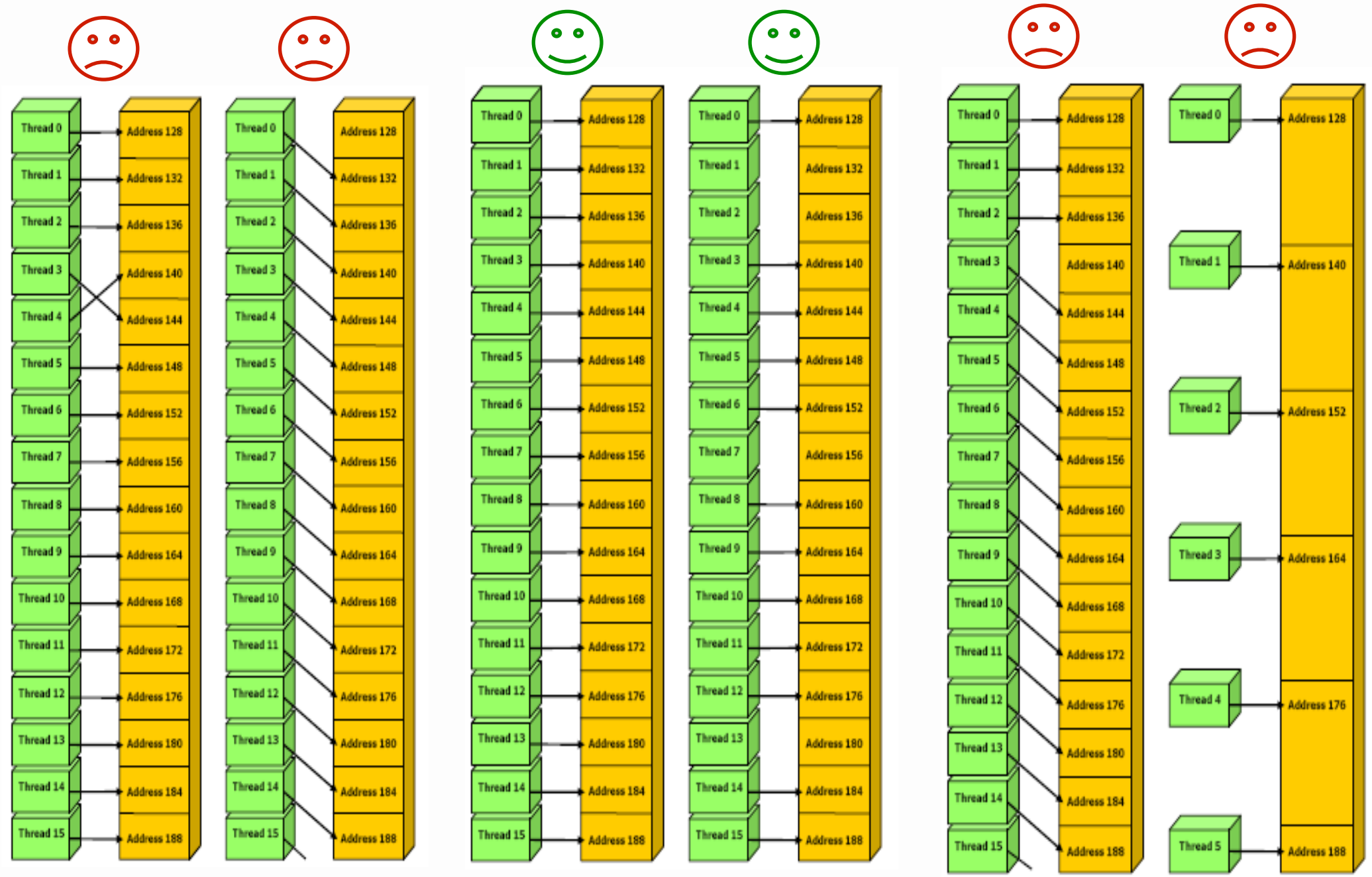
Global Memory Usage

- Why optimizing global memory usage?
  - Global memory is not always cached (e.g., on G8x/GT200)
  - Highest latency instructions: 400-600 clock cycles
  - Likely to be a performance bottleneck
  - Optimizations can greatly increase performance
- Optimizing the global memory usage means optimizing the access patterns of threads executed *at the same time* on GPU
- Which threads are executed *at the same time*?
  - Each block of thread is divided in 32-thread **warps**
  - Warps are groups of threads executed **physically in parallel** (SIMD)
  - The first or second half of warp are called **half-warp**
  - This is an implementation decision, not part of the CUDA programming model

Coalescing (compute capability 1.0 / 1.1)

- A coordinated read by a half-warp (16 threads)
- A contiguous region of global memory:
  - 64 bytes - each thread reads a word: int, float, …
  - 128 bytes - each thread reads a double-word: int2, float2, …
  - 256 bytes – each thread reads a quad-word: int4, float4, …
- Additional restrictions:
  - Starting address for a region must be a multiple of region size
  - The kth thread in a half-warp must access the kth element in a block being read
- Exception: not all threads must be participating
  - Predicated access, divergence within a halfwarp

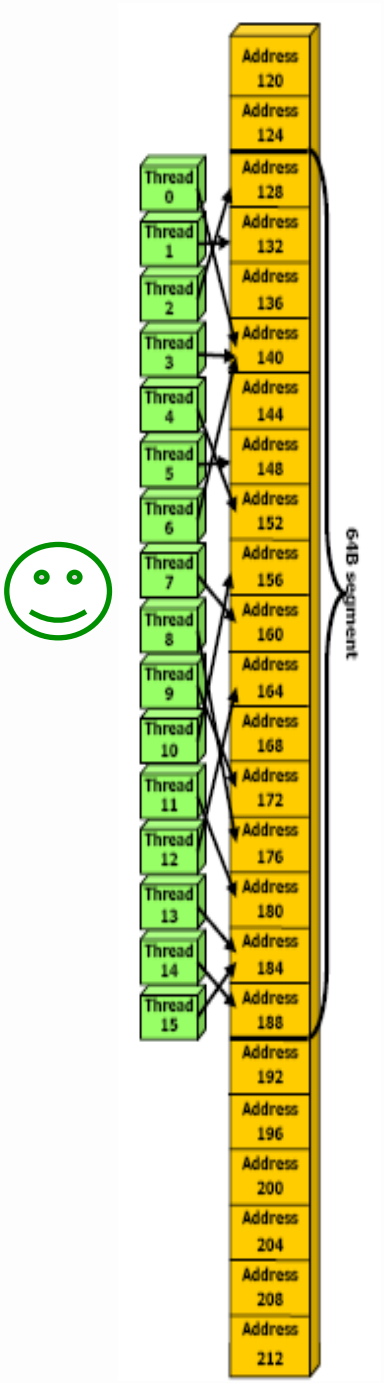
Coalescence (1.0/1.1): examples

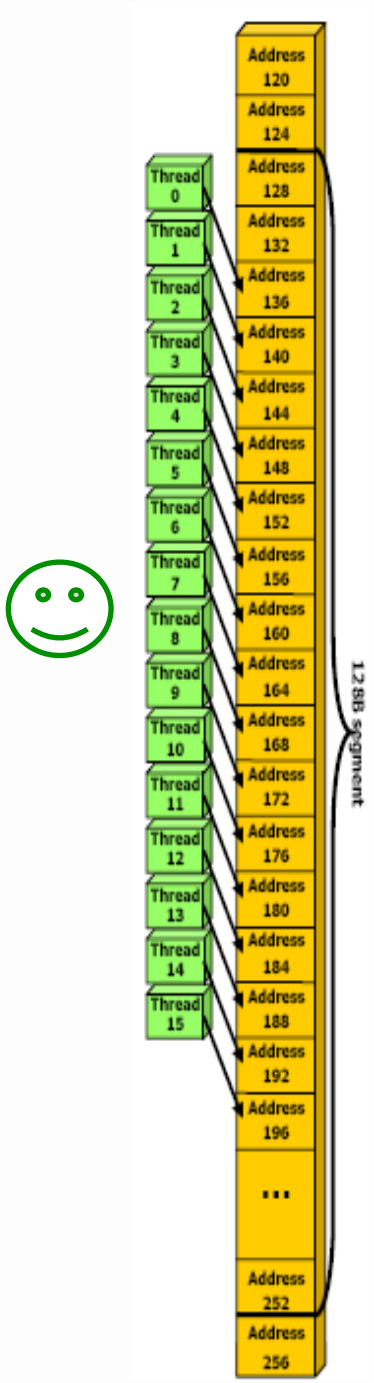


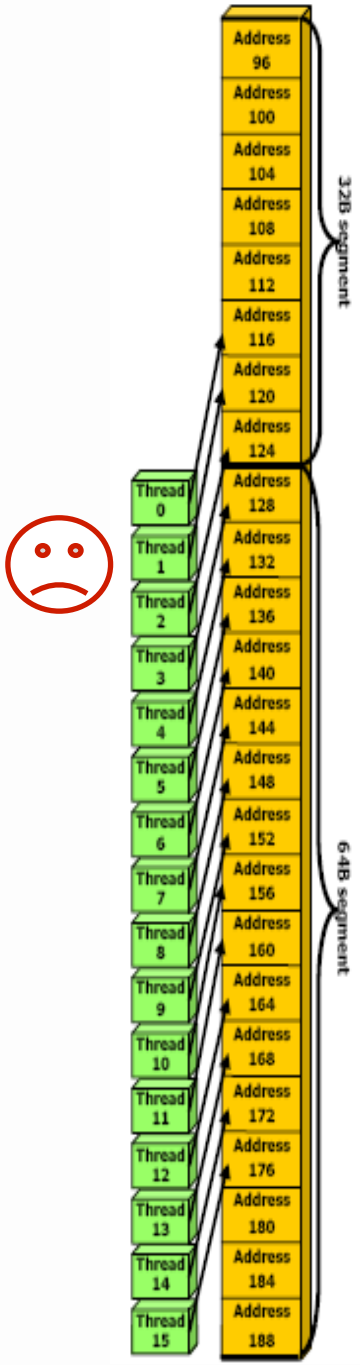
Coalescing (compute capability >= 1.2)

- A single memory transaction is issued for a half warp if words accessed by all threads lie in the same segment of size equal to:
  - 32 bytes if all threads access 8-bit words
  - 64 bytes if all threads access 16-bit words
  - 128 bytes if all threads access 32-bit or 64-bit words
- Achieved for any pattern of addresses requested by the half-warp
  - including patterns where multiple threads access the same address
- If a half-warp addresses words in n different segments, n memory transactions are issued (one for each segment)

Coalescence (1.2): examples





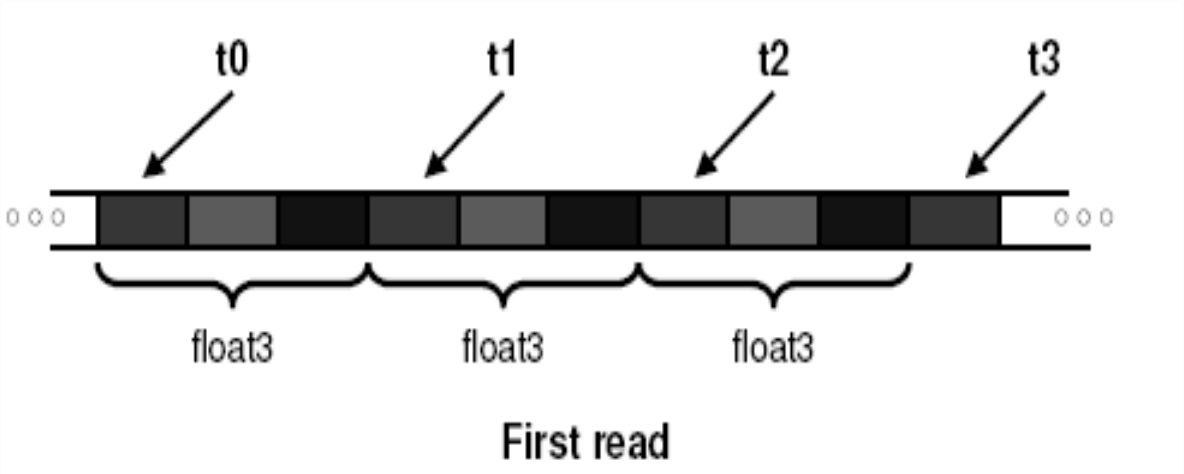


Coalescing: Timing Results

- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads:
  - 356µs – coalesced
  - 357µs – coalesced, some threads don’t participate
  - 3494µs – permuted/misaligned thread access

Uncoalesced Access: float3 Case

- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) != 4, 8, or 16
  - Half-warp reads three 64B non-contiguous regions



Uncoalesced float3 Code

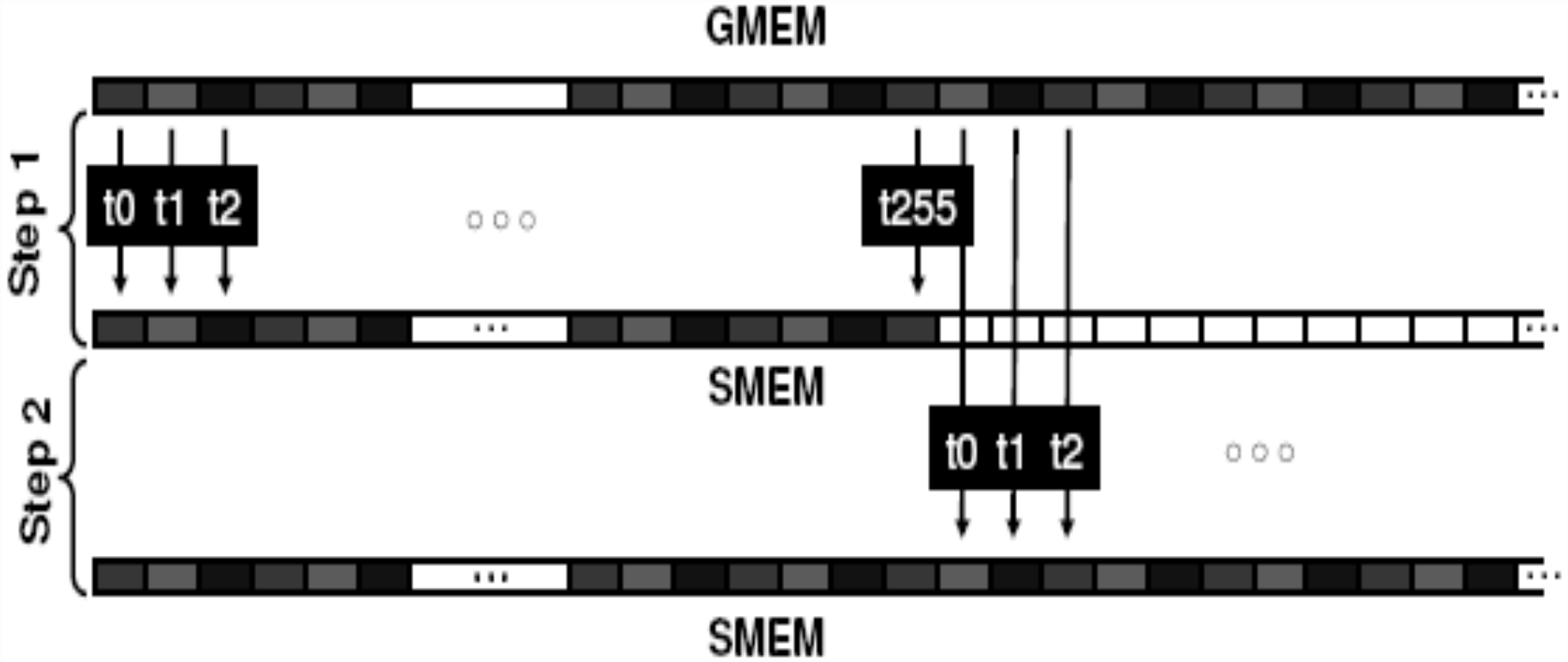
```
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   float3 a = d_in[index];
   a.x += 2;
   a.y += 3;
   a.z += 4;
   d_out[index] = a;
}
```

Shared Memory

Shared Memory

- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order noncoalesceable addressing

Coalescing float3 Access



Similarly, Step3 starting at offset 512

Coalesced Access: float3 Case

- Use shared memory to allow coalescing
  - Need `sizeof(float3)*(threads/block)` bytes of SMEM
  - Each thread reads 3 scalar floats:
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will 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!

Coalesced float3 Code

Read the input through SMEM

Compute code Is not changed

Write the result through SMEM

```
__global__ void accessInt3Shared(float *g_in,
    float *g_out)
{
  int dim = blockDim.x;
  int index = 3 * blockIdx.x * dim +
   threadIdx.x;
  __shared__ float s_data[dim*3];
  s_data[threadIdx.x] = g_in[index];
  s_data[threadIdx.x+dim] = g_in[index+dim];
  s_data[threadIdx.x+2*dim]= g_in[index+dim*2];
  __syncthreads();
  float3 a = ((float3*)s_data)[threadIdx.x];

  a.x += 2;
  a.y += 3;
  a.z += 4;

  ((float3*)s_data)[threadIdx.x] = a;
  __syncthreads();
  g_out[index] = s_data[threadIdx.x];
  g_out[index+dim] = s_data[threadIdx.x+dim];
  g_out[index+dim*2] = s_data[threadIdx.x+dim*2];
}
```

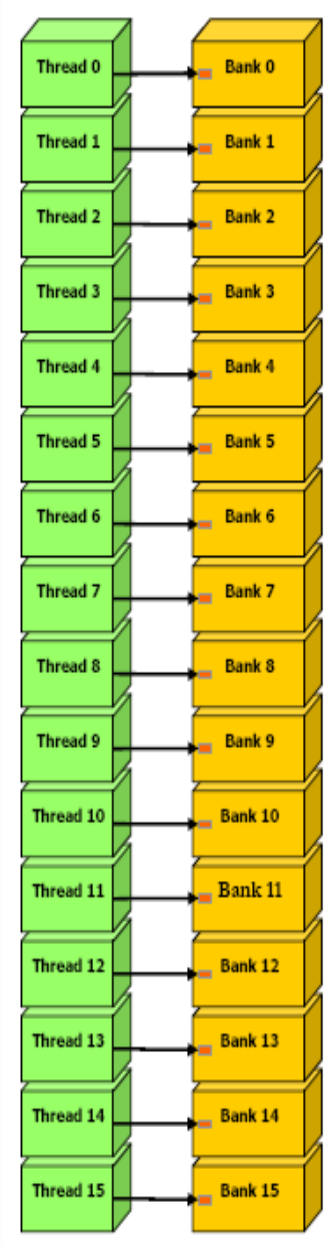
Coalescing: Timing Results

- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads reading floats:
  - 356µs – coalesced
  - 357µs – coalesced, some threads don’t participate
  - 3494µs – permuted/misaligned thread access
- 4K blocks x 256 threads reading float3s:
  - 3302µs – float3 uncoalesced
  - 359µs – float3 coalesced through shared memory

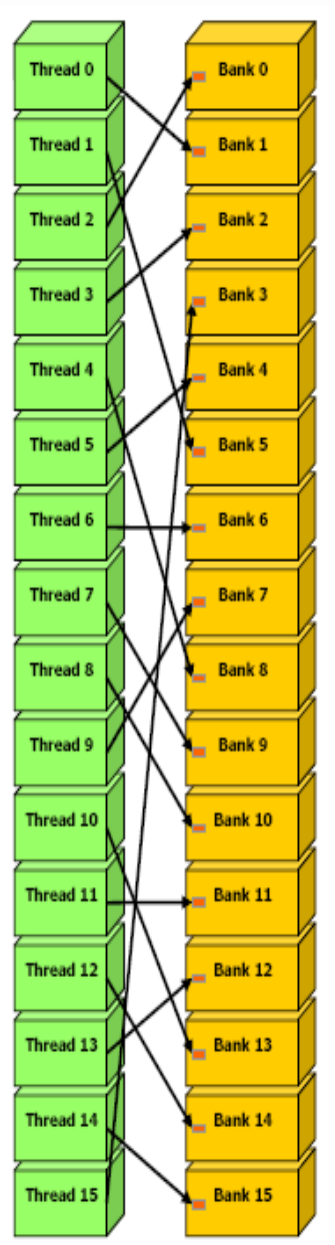
Parallel Memory Architecture

- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

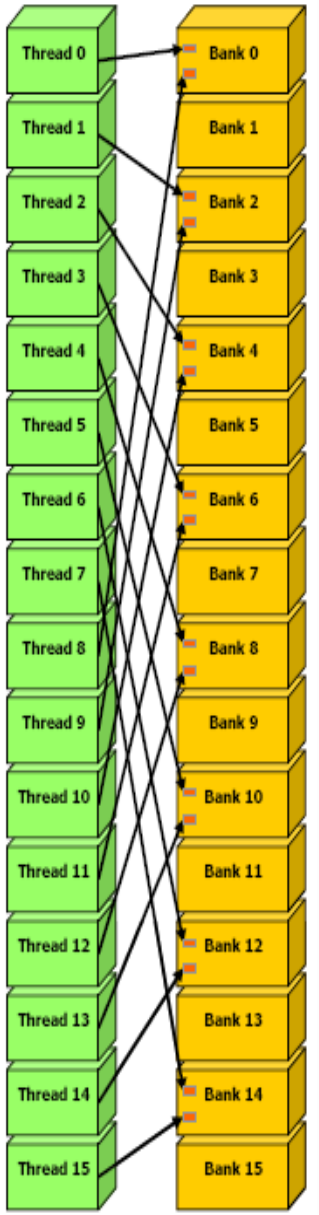
Bank Addressing Examples



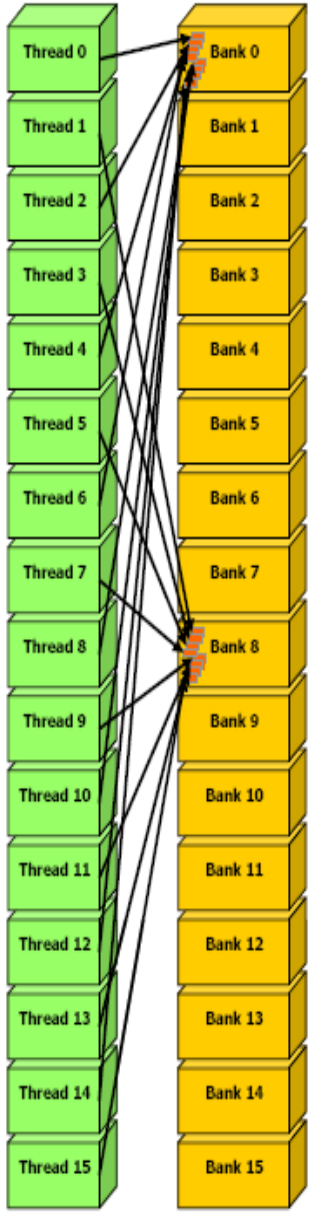
- No bank conflicts
  - Left: linear addressing stride == 1
  - Right: random 1:1 permutation



Bank Addressing Examples



- Left: 2-way Bank Conflicts
  - Linear addressing stride == 2
- Right: 8-way Bank Conflicts
  - Linear addressing stride == 8



Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- The fast cases:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow cases:
  - 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

How addresses map to banks on G80/ GT200

- Bandwidth of each bank is 32 bit per 2 clock cycles
- Successive 32-bit words are assigned to successive banks
- G80/GT200 have 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
  - No bank conflicts between different half-warps, only within a single half-warp

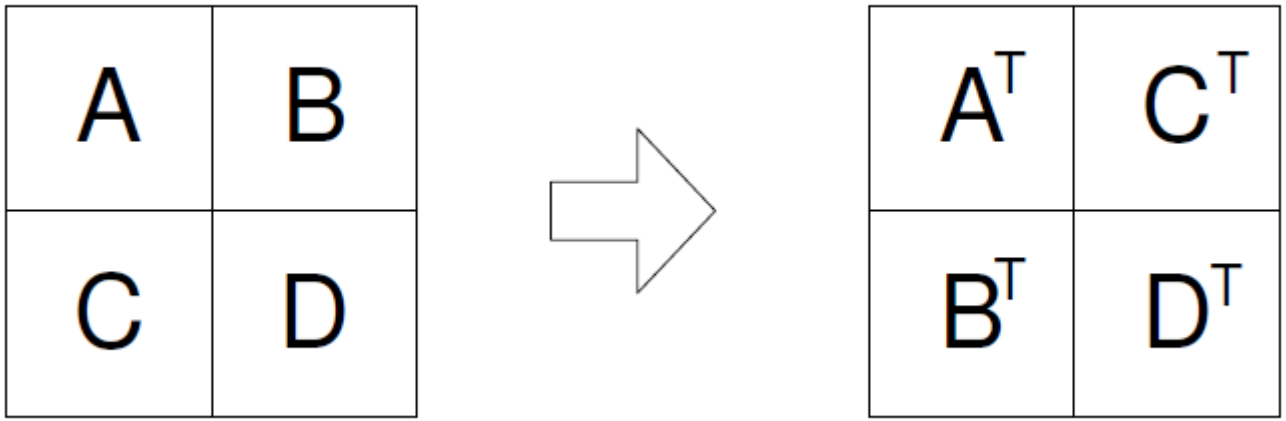
A common case

```
__shared__ float shared[32];
float data = shared[BaseIndex + s * tid];
```

- s is the stride
- Threads tid and tid+n access the same banks if:
  - s*n is a multiple of the number of banks m (m=16)
  - n is a multiple of m/d, where d is the greatest common divisor of m and s
- No bank conflicts if:
  - size(half_warp) <= m/d = 16 / d
  - m/d = 16 (d = 1) -> s must be odd!

Matrix Transpose Example

- Each thread block transposes an equal sized block of matrix M
- Assume M is square (n x n)
- What is a good blocksize?
- CUDA places limitations on number of threads per block
  - 512 threads per block is the maximum allowed by CUDA

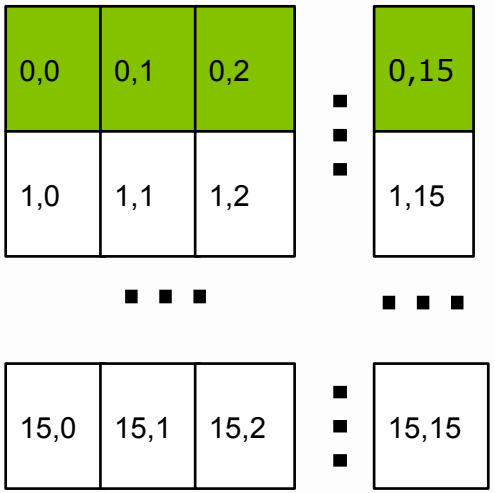


Matrix transpose example

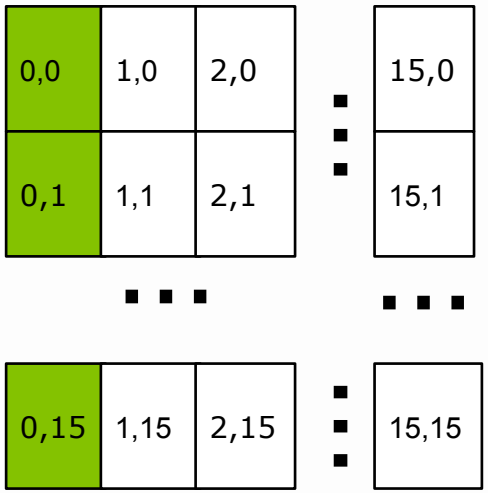
```
__global__ void transpose_naive(float *odata, float* idata, int
   width, int height, int pitch_in, int pitch_out)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   if (xIndex < width && yIndex < height)
   {
       unsigned int index_in  = xIndex + pitch_in * yIndex;
       unsigned int index_out = yIndex + pitch_out * xIndex;
       odata[index_out] = idata[index_in];
   }
}
```

Uncoalesced transpose

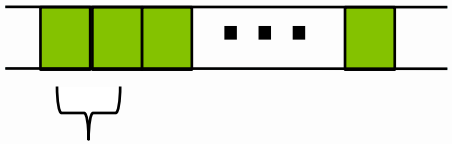
Reads inputs from GMEM



Writes outputs to GMEM

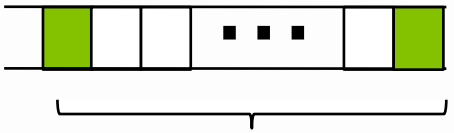


GMEM



Stride = 1, coalesced

GMEM



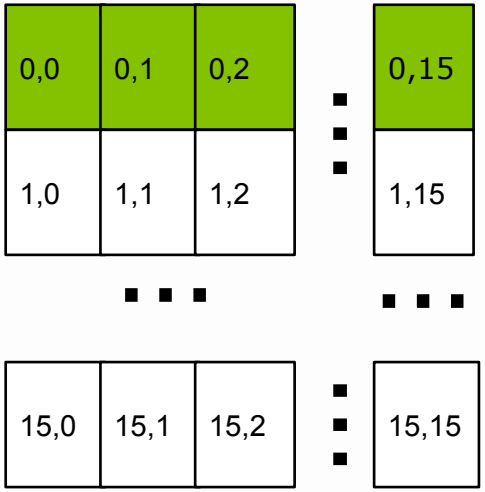
Stride = M, uncoalesced

Coalesced Transpose

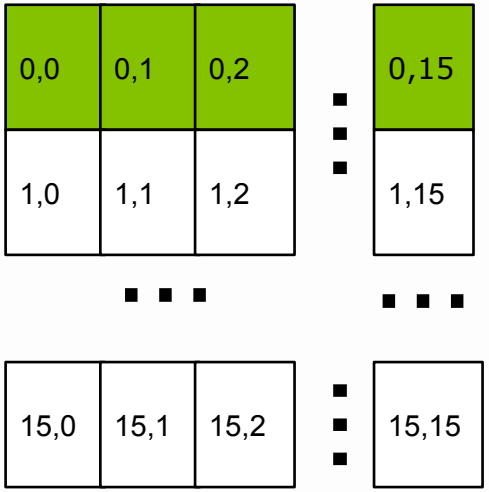
- Matrix is partitioned into square tiles
- Threadblock (bx,by):
  - Read the (bx,by) input tile, store into SMEM
  - Write the SMEM data to (by,bx) output tile
    - Transpose the indexing into SMEM
- Thread (tx,ty):
  - Reads element (tx,ty) from input tile
  - Writes element (tx,ty) into output tile
- Coalescing is achieved if:
  - Block/tile dimensions are multiples of 16

Coalesced Transpose

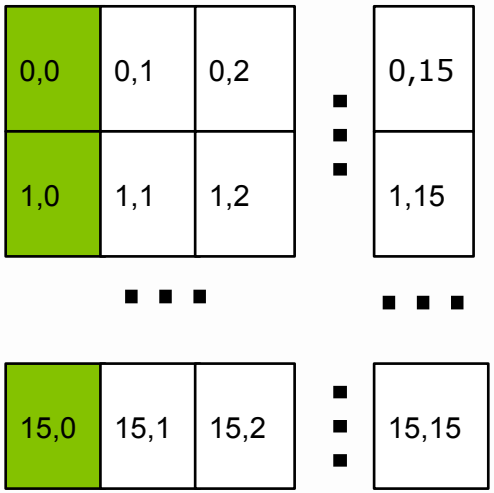
Reads from GMEM



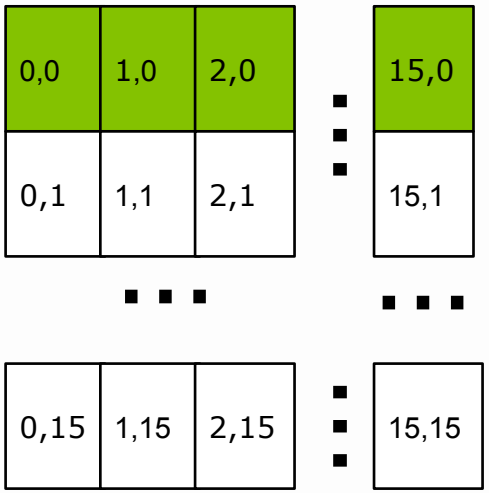
Writes to SMEM



Reads from SMEM

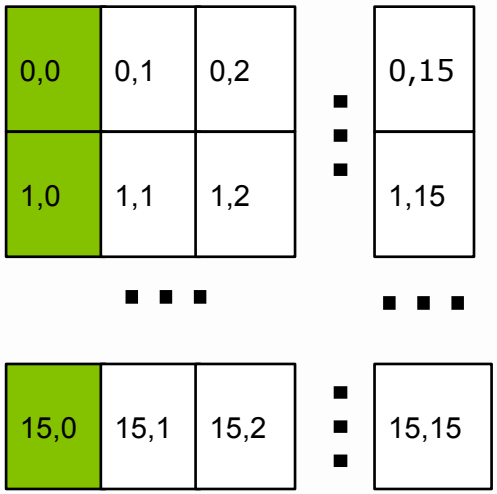


Writes to GMEM



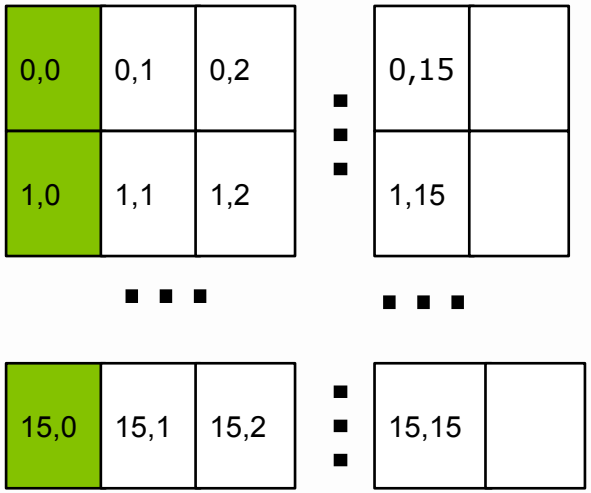
SMEM Optimization

Reads from SMEM



- Threads read SMEM with stride = 16
  - Bank conflicts

Reads from SMEM



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

Optimized transpose

```
#define BLOCK_DIM 16
__global__ void transpose(float *odata, float *idata, int width, int height,
    int pitch_in, int pitch_out){
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
    // read the matrix tile into shared memory
    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 = yIndex * pitch_in + xIndex;
        block[threadIdx.x][threadIdx.y] = idata[index_in];
    }
    __syncthreads();
    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width)){
        unsigned int index_out = yIndex * pitch_out + xIndex;
        odata[index_out] = block[threadIdx.y][threadIdx.x];
    }
}
```

A Common Programming Strategy

- Global memory resides in device memory (DRAM) - much slower access than shared memory
- So, a profitable way of performing computation on the device is to **tile data** to take advantage of fast shared memory:
  - **Partition** data into **subsets** that fit into shared memory
  - Handle **each data subset with one thread block** by:
    - Loading the subset from global memory to shared memory, **using multiple threads to exploit memory-level parallelism**
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

A Common Programming Strategy (Cont.)

- Carefully divide data according to access patterns
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

Execution Configuration

Occupancy Optimization

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy

- **Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently

- Limited by resource usage:
  - Registers
  - Shared memory

Grid/Block Size Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute
- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren't waiting at a `__syncthreads()` keep the hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations

Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
- More threads per block == better memory latency hiding
- But, more threads per block == fewer registers per thread
  - Kernel invocations can fail if too many registers are used
- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation, so experiment!

Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

BUT…

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - (It all comes down to arithmetic intensity and available parallelism)

Parameterize Your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of multiprocessors
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
- You can even make apps self-tuning
  - "Experiment" mode discovers and saves optimal configuration

Instructions and Flow Control

Runtime Math Library

- There are two types of runtime math operations
  - `__func()`: direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - `func()` : compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x,y)

- The `-use_fast_math` compiler option forces every `func()` to compile to `__func()`

How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0
- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- However, DO NOT rely on any ordering between warps
  - If there are any dependencies between threads, you must `__syncthreads()` to get correct results

Control Flow Instructions

- Main performance concern with branching is **divergence**
  - Threads within a single warp take different paths
  - Different execution paths must be serialized

- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `if (threadIdx.x > 2) { }`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) { }`
    - Branch granularity is a whole multiple of warp size

GPU results may not match CPU

- Many variables: hardware, compiler, optimization settings
- Floating-point arithmetic is not associative!
  - In symbolic math, (x+y)+z == x+(y+z) but this is not necessarily true for floating-point addition
    - e.g., try with *x = 10^30, y = -10^30 and z = 1*
  - When you parallelize computations, you potentially change the order of operations
  - Parallel results may not exactly match sequential results (this problem is not specific to GPU or CUDA)