Advanced Topics in CUDA C

S. Sundar and M. Panchatcharam

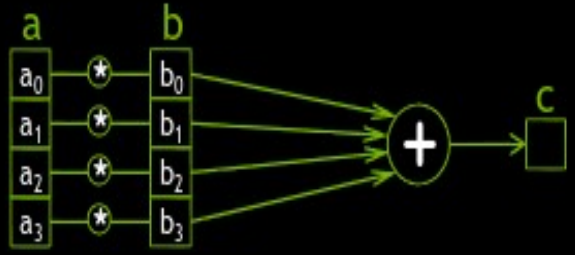
August 9, 2014

Outline

Dot Product

Dot Product

- Unlike vector addition, dot product is a *reduction* from vectors to a scalar



$$c = \vec{a} \cdot \vec{b}$$

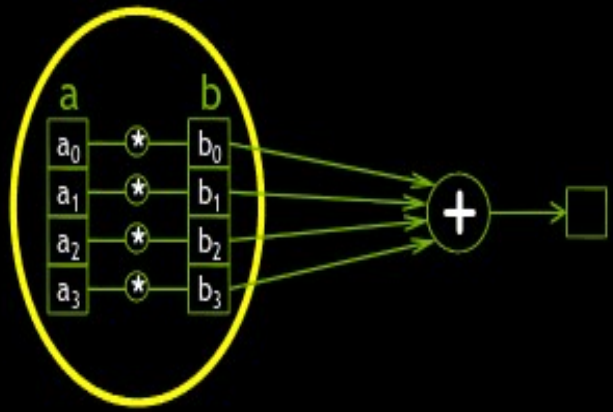
$$= (a_0, a_1, a_2, a_3) \cdot (b_0, b_1, b_2, b_3)$$

$$= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3$$

Dot Product

Dot Product

- Parallel threads have no problem computing the pairwise products:



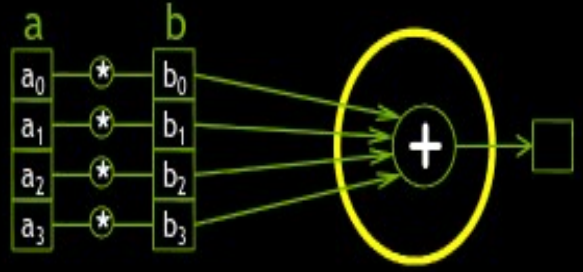
- So we can start a dot product CUDA kernel by doing just that:

```
__global__ void dot( int *a, int *b, int *c )    {
    // Each thread computes a pairwise product
    int temp = a[threadIdx.x] * b[threadIdx.x];
```

Dot Product

Dot Product

- But we need to share data between threads to compute the final sum:

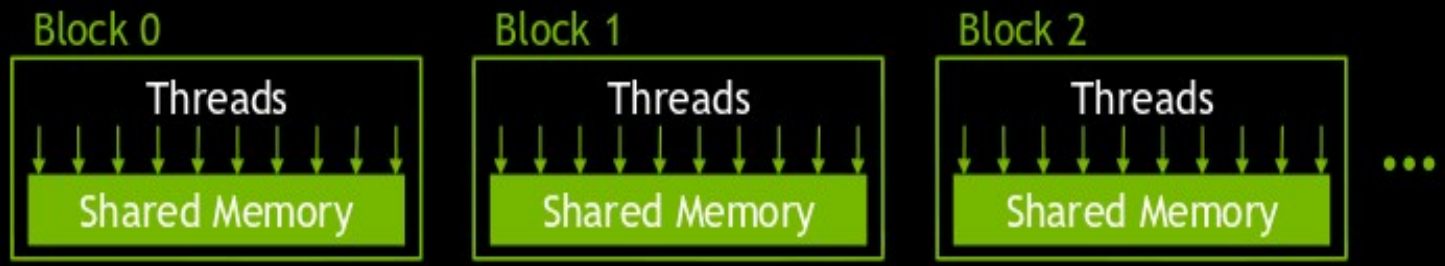



```
__global__ void dot( int *a, int *b, int *c )   {
    // Each thread computes a pairwise product
    int temp = a[threadIdx.x] * b[threadIdx.x];

    // Can't compute the final sum
    // Each thread's copy of 'temp' is private
}
```

Dot Product

Sharing Data Between Threads

- Terminology: A block of threads shares memory called...*shared memory*
- Extremely fast, on-chip memory (user-managed cache)
- Declared with the `__shared__` CUDA keyword
- Not visible to threads in other blocks running in parallel



Dot Product

Parallel Dot Product: dot()

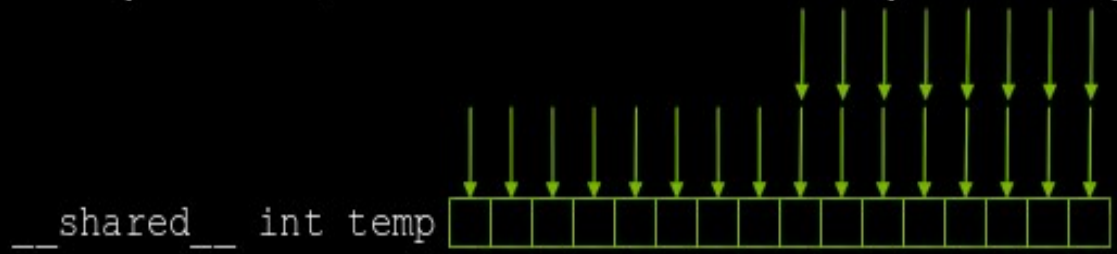
- We perform parallel multiplication, serial addition:

```
#define N  512
__global__ void dot( int *a, int *b, int *c ) {
        // Shared memory for results of multiplication
        __shared__ int temp[N];
        temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

        // Thread 0 sums the pairwise products
        if( 0 == threadIdx.x ) {
            int sum = 0;
            for( int i = 0; i < N; i++ )
                sum += temp[i];
            *c = sum;
        }
}
```

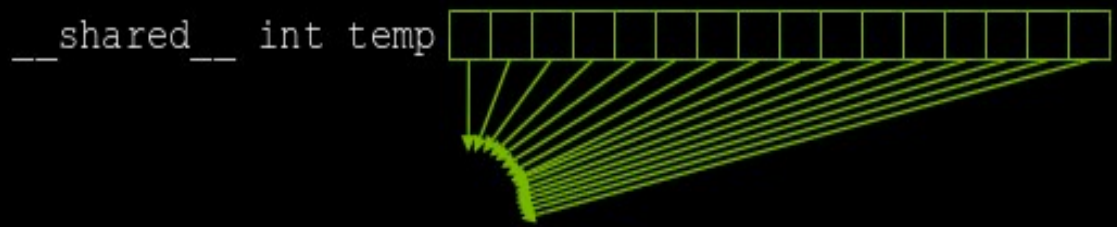
Dot Product

Faulty Dot Product Exposed!

- Step 1: In parallel, each thread writes a pairwise product

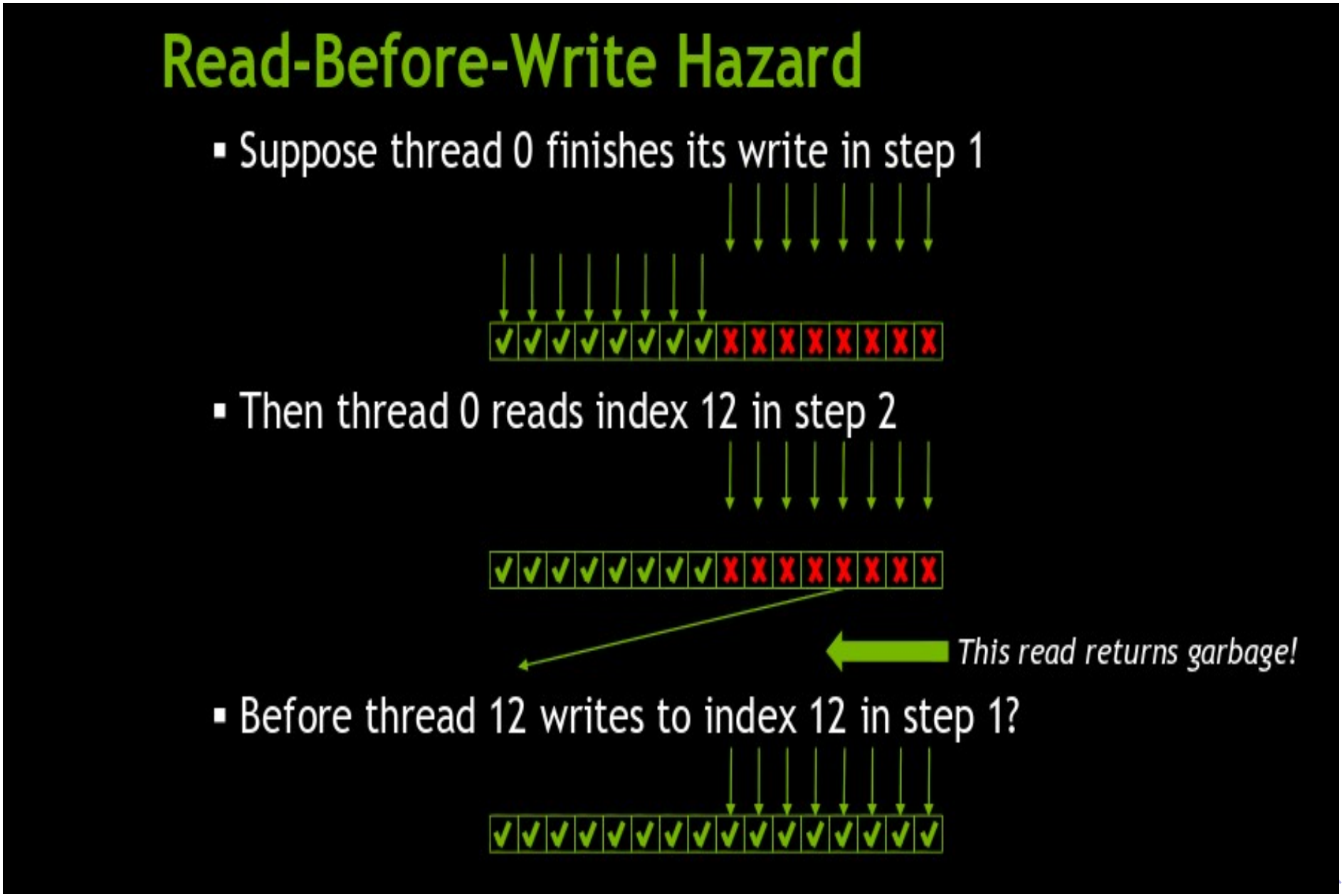


- Step 2: Thread 0 reads and sums the products



- But there's an assumption hidden in Step 1...

Dot Product


Read-Before-Write Hazard
Suppose thread 0 finishes its write in step 1
Then thread 0 reads index 12 in step 2
This read returns garbage!
Before thread 12 writes to index 12 in step 1?

Dot Product

Synchronization

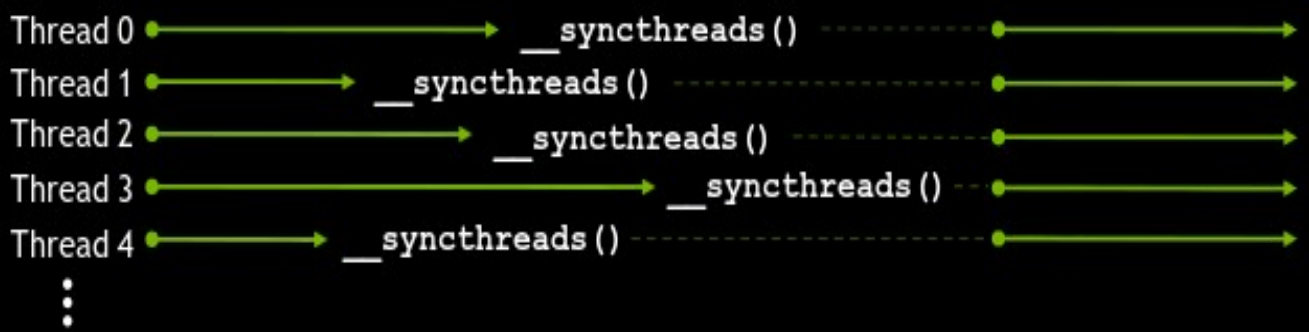
- We need threads to wait between the sections of `dot()`:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // * NEED THREADS TO SYNCHRONIZE HERE *
    // No thread can advance until all threads
    // have reached this point in the code

    // Thread 0 sums the pairwise products
    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

Dot Product

__syncthreads()

- We can synchronize threads with the function `__syncthreads()`
- Threads in the block wait until *all* threads have hit the `__syncthreads()`



- Threads are *only* synchronized within a block

Dot Product

Parallel Dot Product: dot()

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

- With a properly synchronized `dot()` routine, let's look at `main()`

Dot Product

Parallel Dot Product: main()

```
#define N  512
int main( void ) {
    int *a, *b, *c;                 // copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = N * sizeof( int );   // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Dot Product

Parallel Dot Product: main()

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch dot() kernel with 1 block and N threads
    dot<<< 1, N >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Dot Product

Multiblock Dot Product

- Recall our dot product launch:

```
// launch dot() kernel with 1 block and N threads
dot<<< 1, N >>>( dev_a, dev_b, dev_c );
```

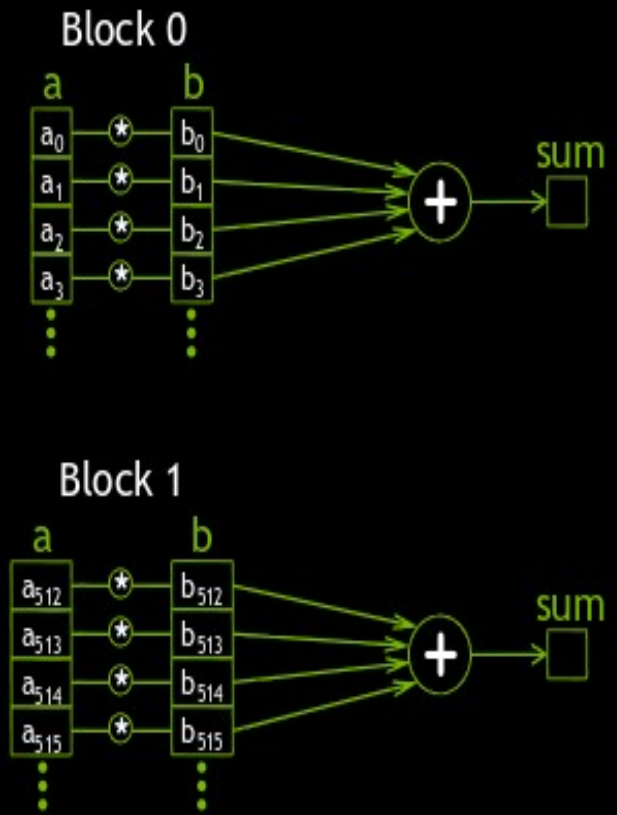
- Launching with one block will not utilize much of the GPU

- Let's write a multiblock version of dot product

Dot Product

Multiblock Dot Product: Algorithm

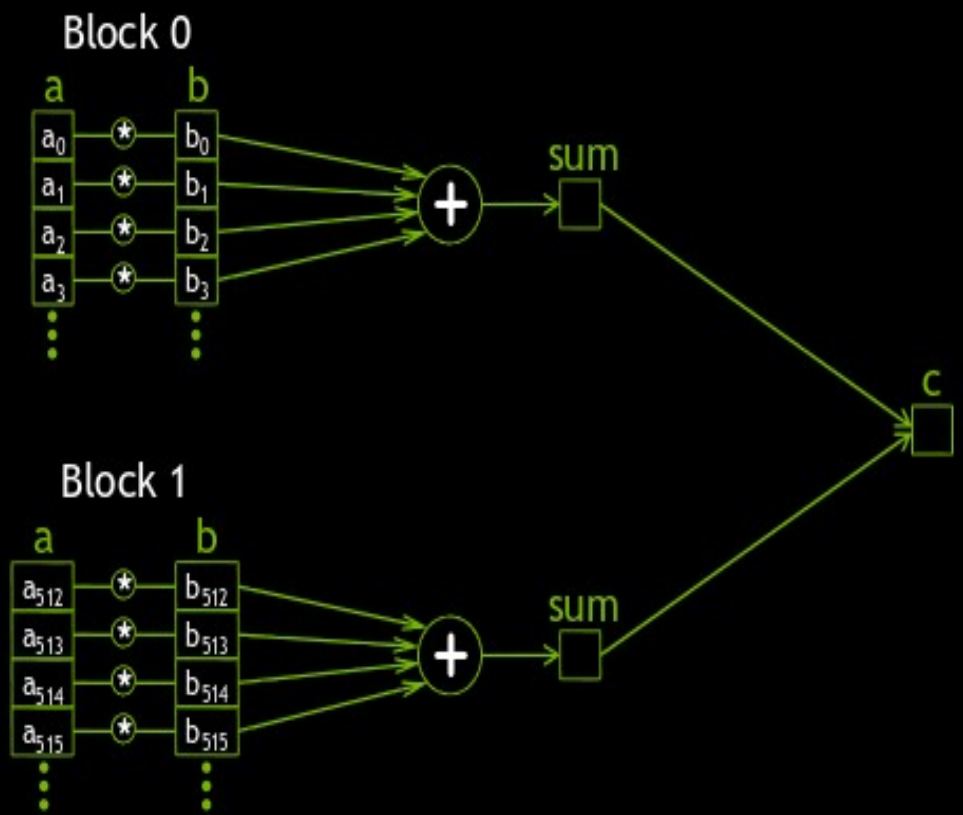
- Each block computes a sum of its pairwise products like before:



Dot Product

Multiblock Dot Product: Algorithm

- And then contributes its sum to the final result:



Dot Product

Multiblock Dot Product: dot()

```
#define N   (2048*2048)
#define THREADS_PER_BLOCK  512
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c , sum );
    }
}
```

- But we have a race condition...
- We can fix it with one of CUDA's atomic operations

Dot Product

Race Conditions

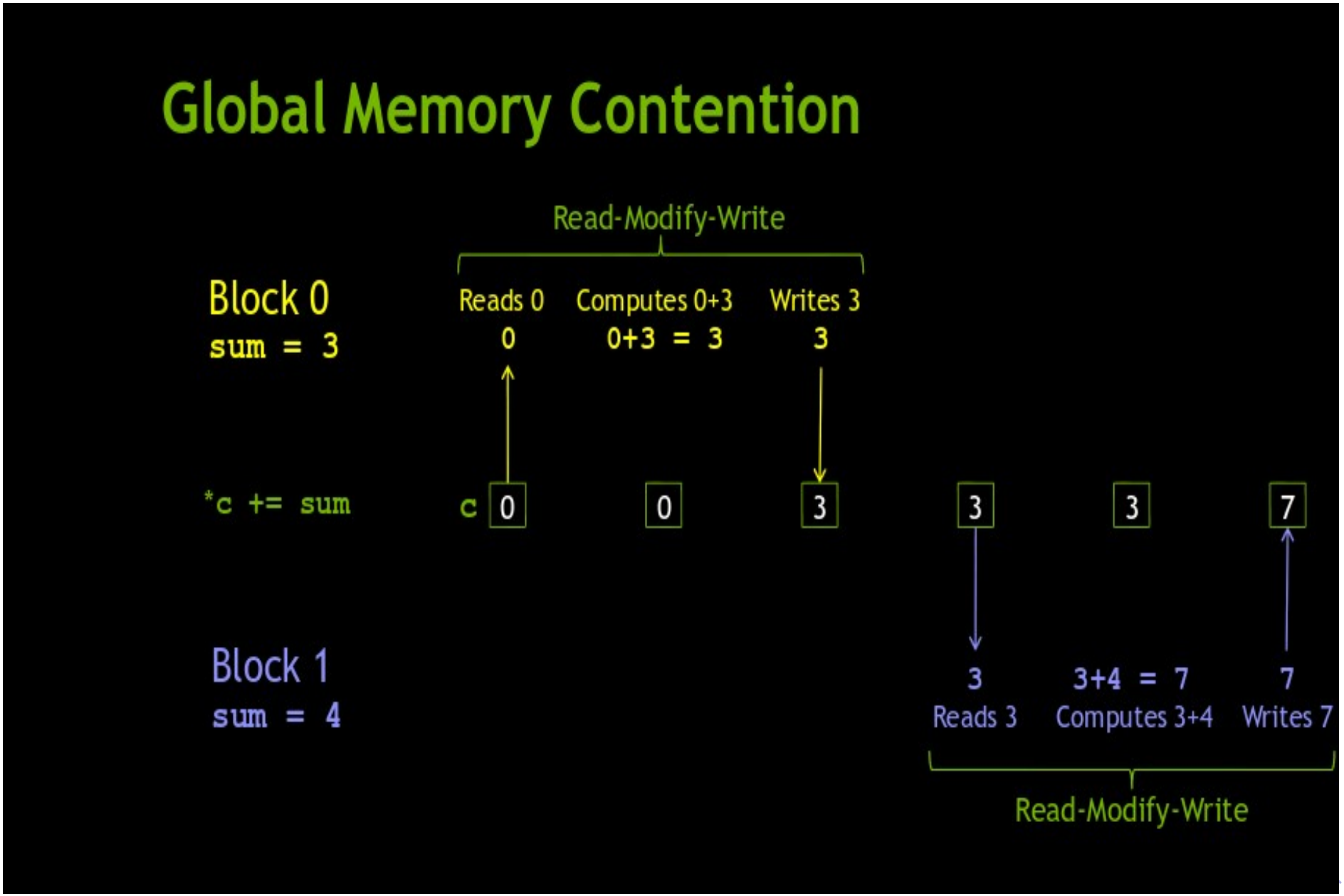
- Terminology: A *race condition* occurs when program behavior depends upon relative timing of two (or more) event sequences

- What actually takes place to execute the line in question: `*c += sum;`
  - Read value at address `c`
  - Add `sum` to value
  - Write result to address `c`

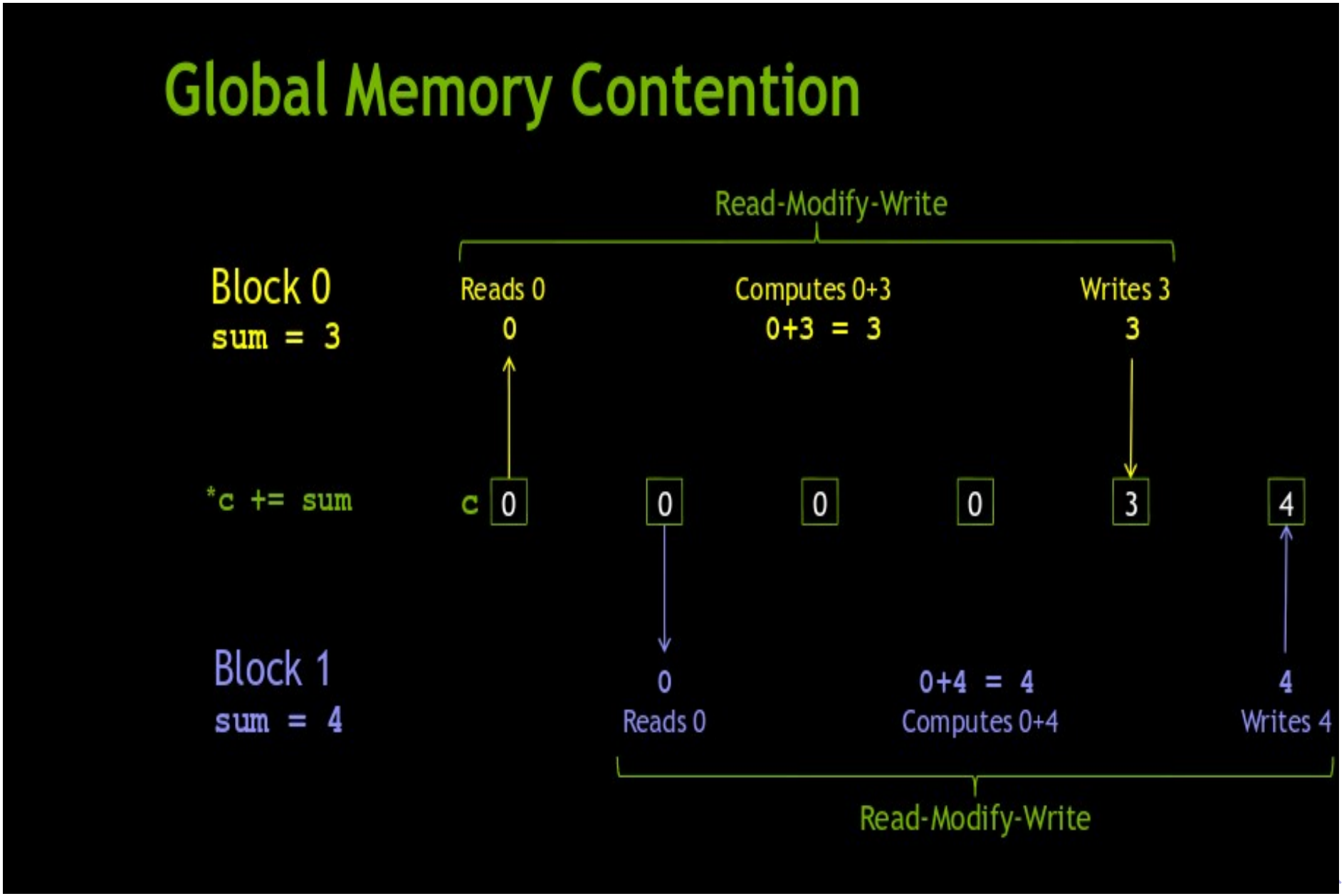
  } Terminology: *Read-Modify-Write*

- What if two threads are trying to do this at the same time?
  - Thread 0, Block 0
    - Read value at address `c`
    - Add `sum` to value
    - Write result to address `c`
  - Thread 0, Block 1
    - Read value at address `c`
    - Add `sum` to value
    - Write result to address `c`

Dot Product


Global Memory Contention
Read-Modify-Write
Block 0
sum = 3
Reads 0
0
Computes 0+3
0+3 = 3
Writes 3
3
*c += sum
c
0
0
3
3
3
7
Block 1
sum = 4
3
Reads 3
3+4 = 7
Computes 3+4
7
Writes 7
Read-Modify-Write

Dot Product


Global Memory Contention
Read-Modify-Write
Block 0
sum = 3
Reads 0
0
Computes 0+3
0+3 = 3
Writes 3
3
*c += sum
c
0
0
0
0
3
4
Block 1
sum = 4
0
Reads 0
0+4 = 4
Computes 0+4
4
Writes 4
Read-Modify-Write

Atomic Operations

Atomic Operations

Atomic Operations

- Terminology: Read-modify-write uninterruptible when *atomic*
- Many *atomic operations* on memory available with CUDA C
  - `atomicAdd()`
  - `atomicSub()`
  - `atomicMin()`
  - `atomicMax()`
  - `atomicInc()`
  - `atomicDec()`
  - `atomicExch()`
  - `atomicCAS()`
- Predictable result when simultaneous access to memory required
- We need to atomically add `sum` to `c` in our multiblock dot product

Dot Product

Multiblock Dot Product: dot()

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c , sum );
    }
}
```

- Now let's fix up `main()` to handle a multiblock dot product

Dot Product

Parallel Dot Product: main()

```
#define N  (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = N * sizeof( int );   // we need space for N ints

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Dot Product

Parallel Dot Product: main()

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch dot() kernel
    dot<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Memory Coalescing

- Whenever possible:
  - Read/Write global memory
  - Only once
  - Without stride or offset
- Use shared memory (150x faster than global)
- To coalesce global memory loads/stores
- To arrange data in useful patterns
- To avoid multiple global memory accesses

Thread Diverging

- Entire warp executes every branch path taken by any thread in serial
- Threads can diverge at any...
- `if`,`switch`,`do`,`for`,`while`
- Gotcha: Diverging threads within a warp are slow

Thread Diverging

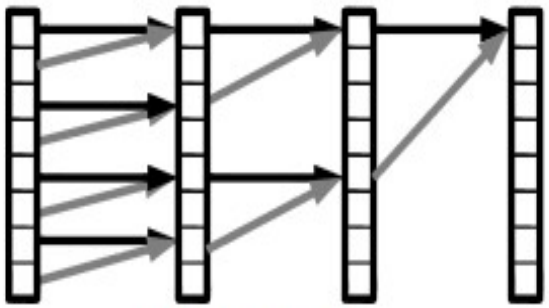
- Create a simple kernel where threads within a warp diverge between two or more paths
- Each path should do some work
- Compare to same amount of work done without thread divergence
- Use diverge.cu as a template

Shared Memory Banks

- Shared memory is stored in banks
- Successive 32-bit or 64-bit words are stored in successive banks
- `cudaDeviceGetSharedMemConfig()`
- `cudaDeviceSetSharedMemConfig()`
- If two threads from a warp access different addresses within a bank, the requests are serialized
- shared memory bank conflict
- There are 32 banks, data is distributed cyclicly

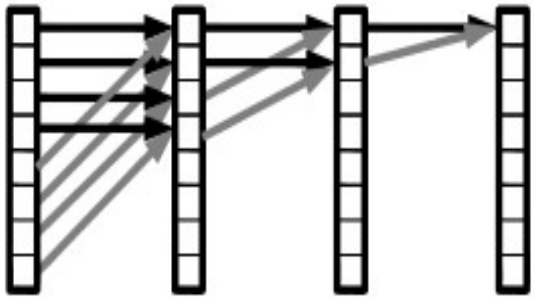
Which access pattern minimizes shared memory bank conflicts?

A)



Interleaved

B)



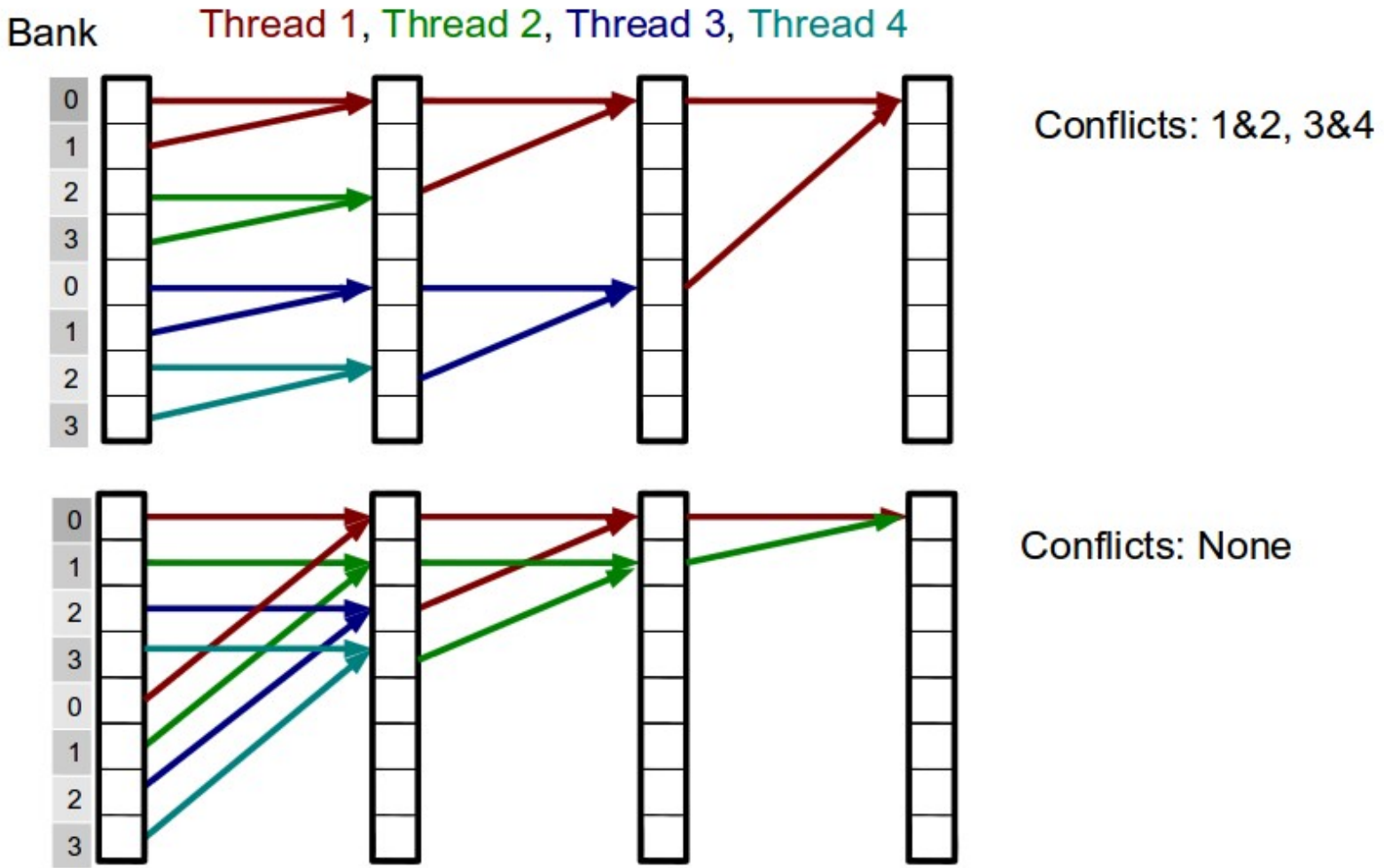
Sequential

C) Neither produces bank conflicts

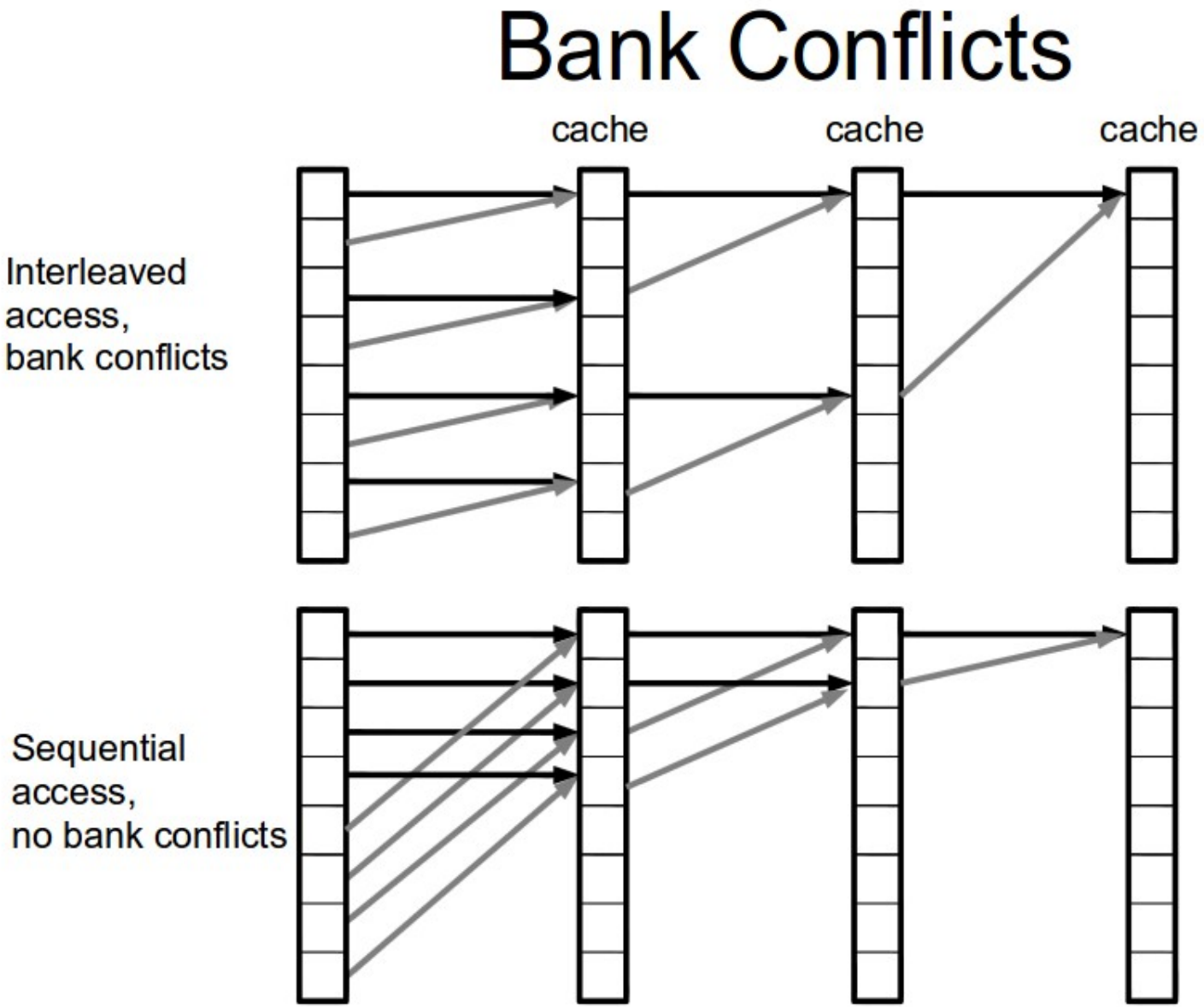
D) Both produce equal numbers of bank conflicts

Memory

To simplify diagrams: assume 4 banks, 4 threads per warp
Normally: 32 banks, 32 threads per warp



Bank conflicts



Why multi-GPU Programming

- Many systems contain multiple GPUs:
  - Servers (Tesla/Quadro servers and desksides)
  - Desktops (2- and 3- way SLI desktops, GX2 boards)
  - Laptops (hybrid SLI)
- Additional processing power
  - Increasing processing throughput
- Additional memory
  - Some problems do not fit within a single GPU memoryl

Multi-GPU Memory

- GPUs do not share global memory
  - One GPU cannot access another GPUs memory directly
- Inter-GPU communication
  - Application code is responsible for moving data between GPUs
  - Data travels across the PCIe bus
    - Even when GPUs are connected to the same PCIe switch

Run-Time API Device Management:

- A host thread can maintain one context at a time
  - GPU is part of the context and cannot be changed once a context is established
  - Need as many host threads as GPUs
  - Note that multiple host threads can establish contexts with the same GPU
    - Driver handles time-sharing and resource partitioning
- GPUs have consecutive integer IDs, starting with 0
- Device management calls:
  - cudaGetDeviceCount(int *num_devices)
  - cudaSetDevice(int devic_id)
  - cudaGetDevice(int *current_device_id)
  - cudaThreadExit( )

Choosing a Device

- Properties for a given device can be queried
  - No context is necessary or is created
  - cudaGetDeviceProperties(cudaDeviceProp *properties, int device_id)
  - This is useful when a system contains different GPUs
- Explicit device set:
  - Select the device for the context by calling cudaSetDevice () with chosen device ID
    - Must be called prior to context creation
    - Fails if a context has already been established
    - One can force context creation with cudaFree(0)
- Default behavior:
  - Device 0 is chosen when no explicit cudaSetDevice is called
    - Note this will cause multiple contexts with the same GPU
    - Except when driver is in the exclusive mode

Ensuring One Context Per GPU

- Two ways to achieve:
  - Application-control
  - Driver-control
- Application-control:
  - Host threads negotiate which GPUs to use
    - For example,OpenMP threads set device based on OpenMP thread ID
    - Pitfall: different applications are not aware of each other's GPU usage
  - Call cudaSetDevice() with the chosen device ID

Driver-Control

- To use exclusive mode:
  - Administrator sets the GPU to exclusive mode using SMI
    - SMI(System Management Tool) is provided with Linux drivers
  - Application: do not explicitly set the GPU
- Behaviour:
  - Driver will implicitly set a GPU with no contexts
  - Implicit context creation will fail if all GPUs have contexts
    - The first state-changing CUDA call will fail and return an error
- Device mode can be checked by querying its properties

Inter-GPU Communication

- Application is responsible for moving data between GPUs:
  - Copy data from GPU to host thread A
  - Copy data from host thread A to host thread B
    - Use any CPU library (MPI, ...)
  - Copy data from host thread B to its GPU
- Use asynchronous memcopies to overlap kernel execution with data copies
- Lightweight host threads (OpenMP,pthreads) can reduce host-side copies by sharing pinned memory
  - Allocate with cudaHostAlloc(...)

Texture Memory

Texture Memory Usage and optimization

- Legacy from graphics
- Optimized for 2D locality
- Performs better than GMEM for random accesses

Texture Description

- Has a dimensionality
- Has elements, called texel/s, of a particular type
  - Support native types and 2 or 4 component vectors like int4
- Has a read mode for indices normalization
  - cudaReadModeElementType
  - cudaReadModeNormalizedFloat
- Has an addressing mode to deal with out of range (OOR) accesses
  - cudaAddressModeWrap
  - cudaAddressModeClamp
  - cudaAddressModeMirror
  - cudaAddressModeBorder

Texture Description

- Filtering Mode
  - cudaFilterModePoint
  - cudaFilterLinear

Usage

- Bind the texture to a linear array
  - Bind the texture to CUDA array
  - Opaque container: can only access its content through texture

Texture

```
#define N 10000
texture <int,1 , cudaReadModeElementType> myTexture;
__global__ void kernel ( . . . ) {
//Fetch the right index
int a = tex1Dfetch( myTexture , idx );
. . .
}
int main () {
. . .
int *d_V = NULL;
cudaMalloc((void**)&d_V,N*sizeof(int));
//Bind the texture to some linear array
cudaBindTexture(0 , myTexture ,d_V,N*sizeof(int));
kernel <<< blocks , threads >>> ( . . . );
//Unbind once done
cudaUnbindTexture(myTexture);
. . .
}
```

Summary

- Avoid doing lots of transfer from CPU to GPU
- Use shared memory:
  - for storage of reusable data
  - for data shared by all the threads in a block
  - for buffering global memory loads and writes without any bank conflicts
- Use registers for reusable data local to a thread
- Avoid local memory by limiting the number of registers being used by a kernel
- Maximize occupancy to keep the GPU busy
- Strive for coalesced accesses to the global memory

- Use constant memory for read–only reusable data
- Make use of alternatives such as LDU or read–only caching when running out of constant memory, or when you want to use both CMEM and alternatives at the same time
- Use texture memory for random accesses in a kernel without disabling the L1 cache
- Use texture memory for data sets with 2D locality

THANK YOU