CUDA Memory Types

Overview

1. Memory Access Efficiency
2. CUDA Memory Types
3. Reducing Global Memory Traffic
4. Example: Matrix-Matrix Multiplication Kernel
5. Parallelism Limited by Memory
6. Conclusions

Memory Access Efficiency

Memory Access Efficiency

So CUDA lets us schedule millions or more threads to be run in our kernel calls. While the programs we've written so far do run quite a bit faster, they don't run millions of times faster than the CPU version. Why is that?

Memory Access Efficiency

Part of this is the time it takes to copy the memory to and from the CUDA device.

Another part is that the CUDA device doesn't run all million threads simultaneously, but rather a subset of those threads.

Memory Access Efficiency

Even so, CUDA devices are capable of running thousands or more threads at the same time — yet we don't typically see a 1000x speedup. Why is that?

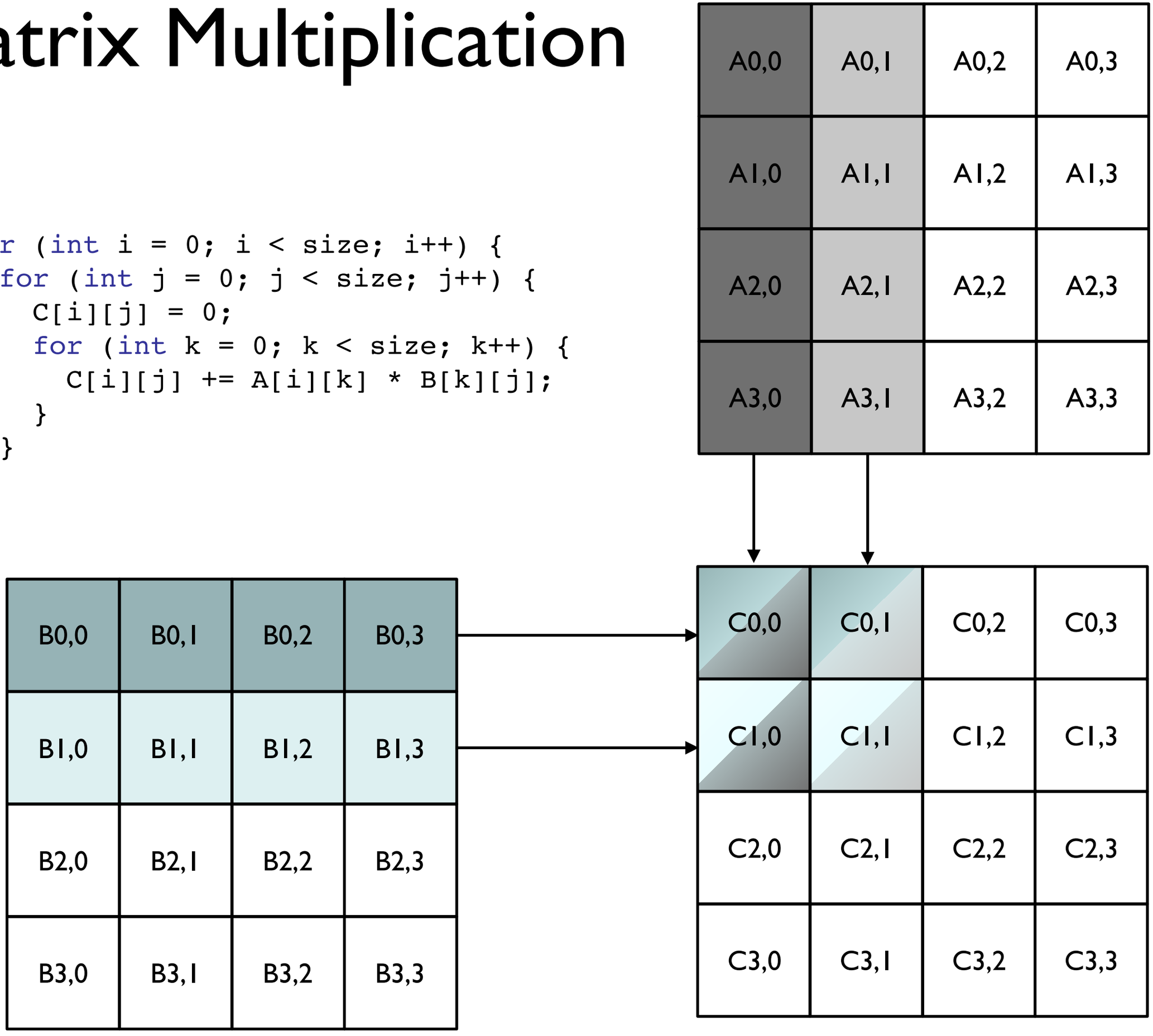
Memory Access Efficiency

The problem is in part due to the fact that the kernel threads are accessing global memory, which in most CUDA devices is implemented with DRAM (dynamic random access memory).

DRAM lookups have long latencies (100s of clock cycles; where an add operation usually takes 1 clock cycle), so in many cases warp scheduling (discussed in last lecture) is not good enough to over come this.

Matrix Multiplication

```
for (int i = 0; i < size; i++) {
  for (int j = 0; j < size; j++) {
    C[i][j] = 0;
    for (int k = 0; k < size; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```



A Simple Example

For example, lets look at a simple kernel for matrix multiplication:

```
//Computes matrix multiplication on a GPU.
__global__ void matrixMulKernel(float* A, float* B, float* C, int width) {
  //calculate the row and column for this element of the matrix
  int row = threadIdx.y + (blockDim.y * blockIdx.y);
  int col = threadIdx.x + (blockDim.x * blockIdx.x);

  if ((row < width) && (col < width)) {
    float result = 0;
    for (int k = 0; k < width; k++) {
      result += A[(row * width) + k] * B[(k * width) + col];
    }
    C[(row * width) + col] = result;
  }
}
```

A Simple Example

```
//Computes matrix multiplication on a GPU
__global__ void matrixMulKernel(float* A, float* B, float* C, int width) {
  //calculate the row and column for this element of the matrix
  int row = threadIdx.y + (blockDim.y * blockIdx.y);
  int col = threadIdx.x + (blockDim.x * blockIdx.x);

  if ((row < width) && (col < width)) {
    float result = 0;
    for (int k = 0; k < width; k++) {
      result += A[(row * width) + k] * B[(k * width) + col];
    }
    C[(row * width) + col] = result;
  }
}
```

The vast majority of work is done in this for loop. If we look at it, we can see that theres one add (result += A*B), one multiply (A[…] * B[…]), and two global memory lookups (getting the element of A and getting the element of B).

A Simple Example

```
//Computes matrix multiplication on a GPU
__global__ void matrixMulKernel(float* A, float* B, float* C, int width) {
  //calculate the row and column for this element of the matrix
  int row = threadIdx.y + (blockDim.y * blockIdx.y);
  int col = threadIdx.x + (blockDim.x * blockIdx.x);

  if ((row < width) && (col < width)) {
    float result = 0;
    for (int k = 0; k < width; k++) {
      result += A[(row * width) + k] * B[(k * width) + col];
    }
    C[(row * width) + col] = result;
  }
}
```

So we have a *compute to global memory access (CGMA) ratio* of 1:1 (2 ops to 2 accesses). This ratio is quite poor, and means the code will be limited by global memory access.

A Simple Example

Further, suppose the bandwidth to global memory is 200GB/s. With a CGMA of 1:1, the kernel will execute no more than 50 giga floating point operations per second (gigaFLOPS).

This is because each float takes 4 bytes of memory, so at the very best it will be possible to load (200GB/S / 4 bytes) = 50 gigaFLOPS.

While this quite a bit of gigaFLOPS, high end GPUs can perform 3 teraFLOPS (3,000 gigaFLOPS); so in this case they would be severely underutilized. To get 3 teraFLOPS, we'd need a CGMA of 60:1.

CUDA Memory Types

Memory Types

We can overcome some of these performance degradations by exploiting the different memory types that CUDA devices provide.

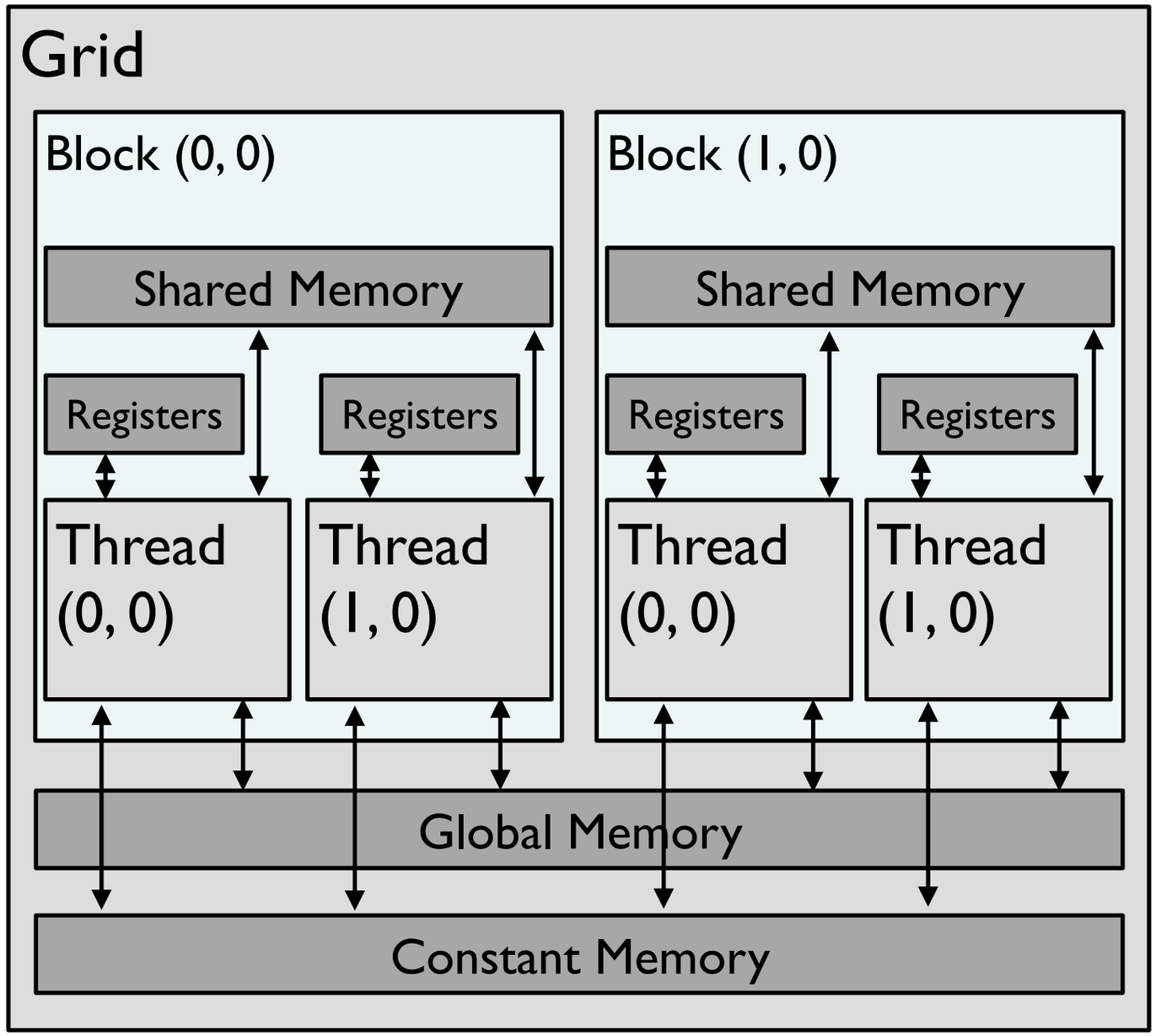
In general, the different memory types are:

- registers
- shared memory
- global memory
- constant memory

CUDA Memories

Device code can:
- read/write per-thread registers
- read/write per-thread local memory
- read/write per-block shared memory
- read/write per-grid global memory
- read only per-grid constant memory

Host code can:
- transfer data to and from global and constant memory



Note that you can get the size of shared, global and constant memory using the CUDA device property function.

General Considerations

The general take away from this is that there are different levels of memory, memory that can only be accessed by a thread (registers), memory that can only be accessed by a threads within a block (shared memory) and memory that can be accessed by all threads within a grid (constant and global memory).

Constant memory is read only, so it can be accessed faster than global memory (as the device doesn't have to worry about problematic writes).

In general, registers are faster than shared memory, which is faster than constant memory, which is faster than global memory. A decent rule of thumb is that each level of memory is an order of magnitude slower than the next.

General Considerations

Some other takeaways:

The aggregated bandwidth of registers is ~2 orders of magnitude (100) times more than global memory, and doesn't require off-chip global memory access. This can really improve the CGMA ratio.

Using registers involves fewer instructions than global memory. To use global memory first it needs to fetched into the instruction register (IR) and the number of operations that can be fetched and executed each clock cycle is limited.

Using registers also takes less energy, which is becoming increasingly important for programs (especially for mobile devices).

Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	thread	Kernel
Automatic array variables	Local	thread	Kernel
__device__ __shared__ int shared_var;	Shared	block	Kernel
__device__ int global_var;	Global	grid	Application
__device__ __constant__ int constant_var;	Constant	grid	Application

Scope describes the threads within the kernel that can access the variable. Variables with a thread scope can only be accessed by that particular thread. Variables with a block scope can only be accessed by threads within the same block. Variables with a grid scope can be accessed by any thread within the kernel.

Lifetime describes how long the value for the variable stays alive. Kernel lifetime means that after the kernel execution the variable as gone. Application lifetime means the variable can be reused over multiple kernels.

Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	thread	Kernel
Automatic array variables	Local	thread	Kernel
__device__ __shared__ int shared_var;	Shared	block	Kernel
__device__ int global_var;	Global	grid	Application
__device__ __constant__ int constant_var;	Constant	grid	Application

Note that arrays created within threads (automatic array variables) are actually stored in global memory, however other threads cannot access them. This means that there is a heavy performance penalty for using them without any benefit. However, they rarely need to be used.

Reducing Global Memory Traffic: Tiling

Tiling

The term tiling comes from the analogy of a large wall or floor (global memory) can be covered by tiles (subsets that each fit into shared memory).

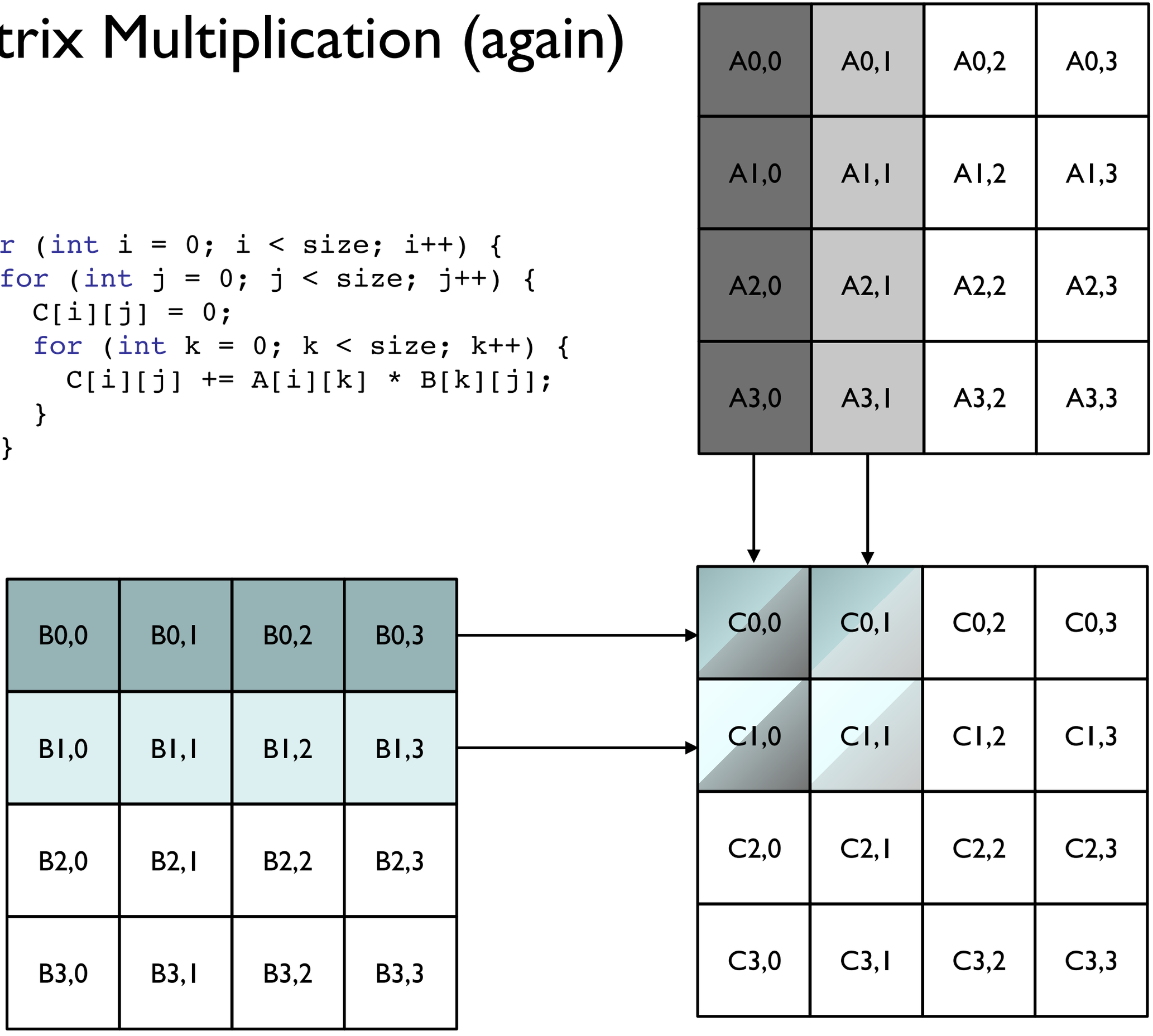
Tiling

Global memory is large but slow, while shared memory is small but fast. Further, global memory can be accessed by all threads within a grid, while shared memory can only be accessed by all threads within a block.

Many GPU kernels can be sped up using *tiling*. Tiling is done by partitioning the data to be worked on into subsets which can fit into shared memory. The major benefit of this is the global data can be loaded into shared memory in parallel by all the threads in the block (as opposed to sequentially by each thread as in our previous matrix multiplication example) and then accessed very quickly.

Matrix Multiplication (again)

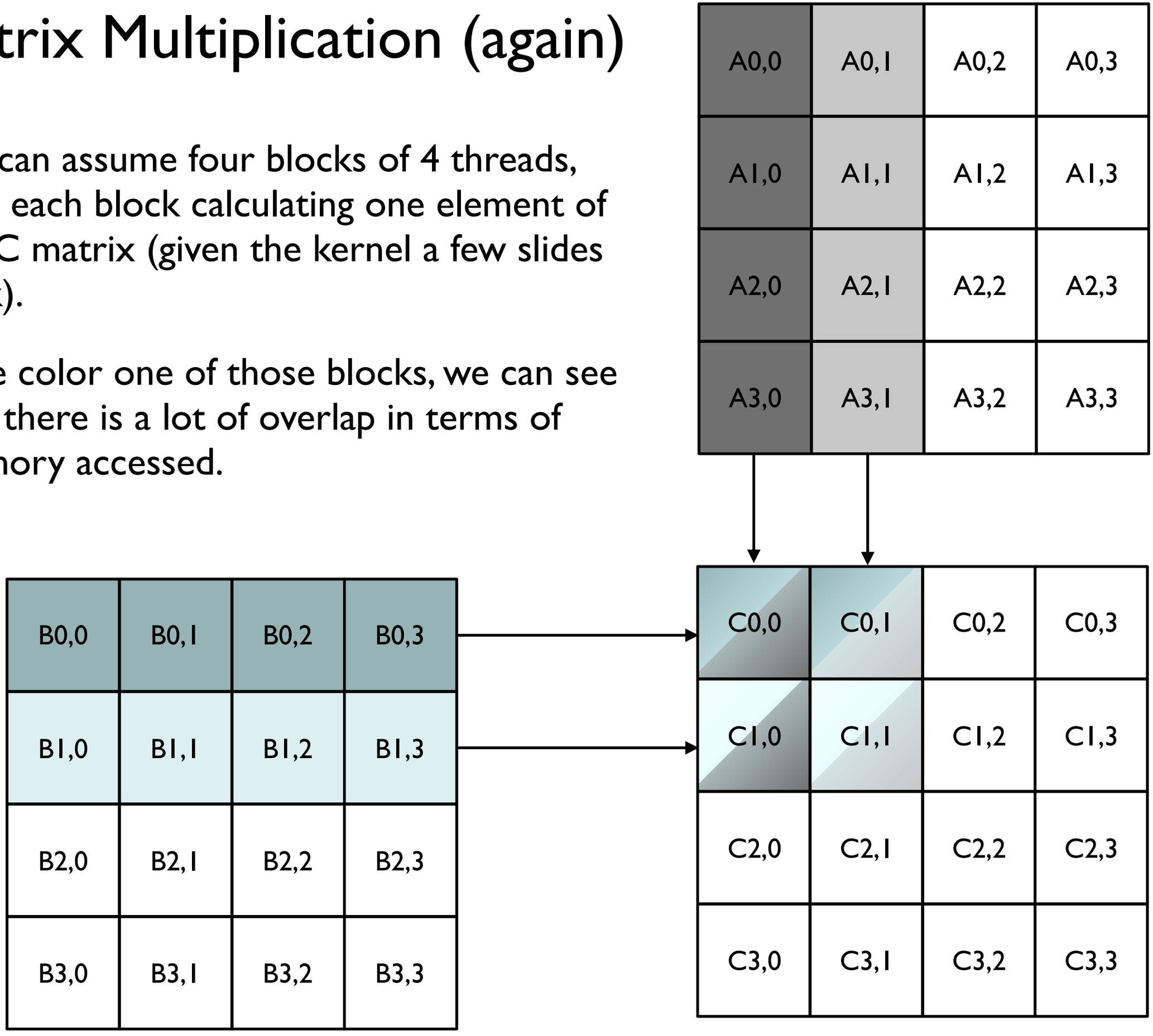
```
for (int i = 0; i < size; i++) {
  for (int j = 0; j < size; j++) {
    C[i][j] = 0;
    for (int k = 0; k < size; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```



Matrix Multiplication (again)

We can assume four blocks of 4 threads, with each block calculating one element of the C matrix (given the kernel a few slides back).

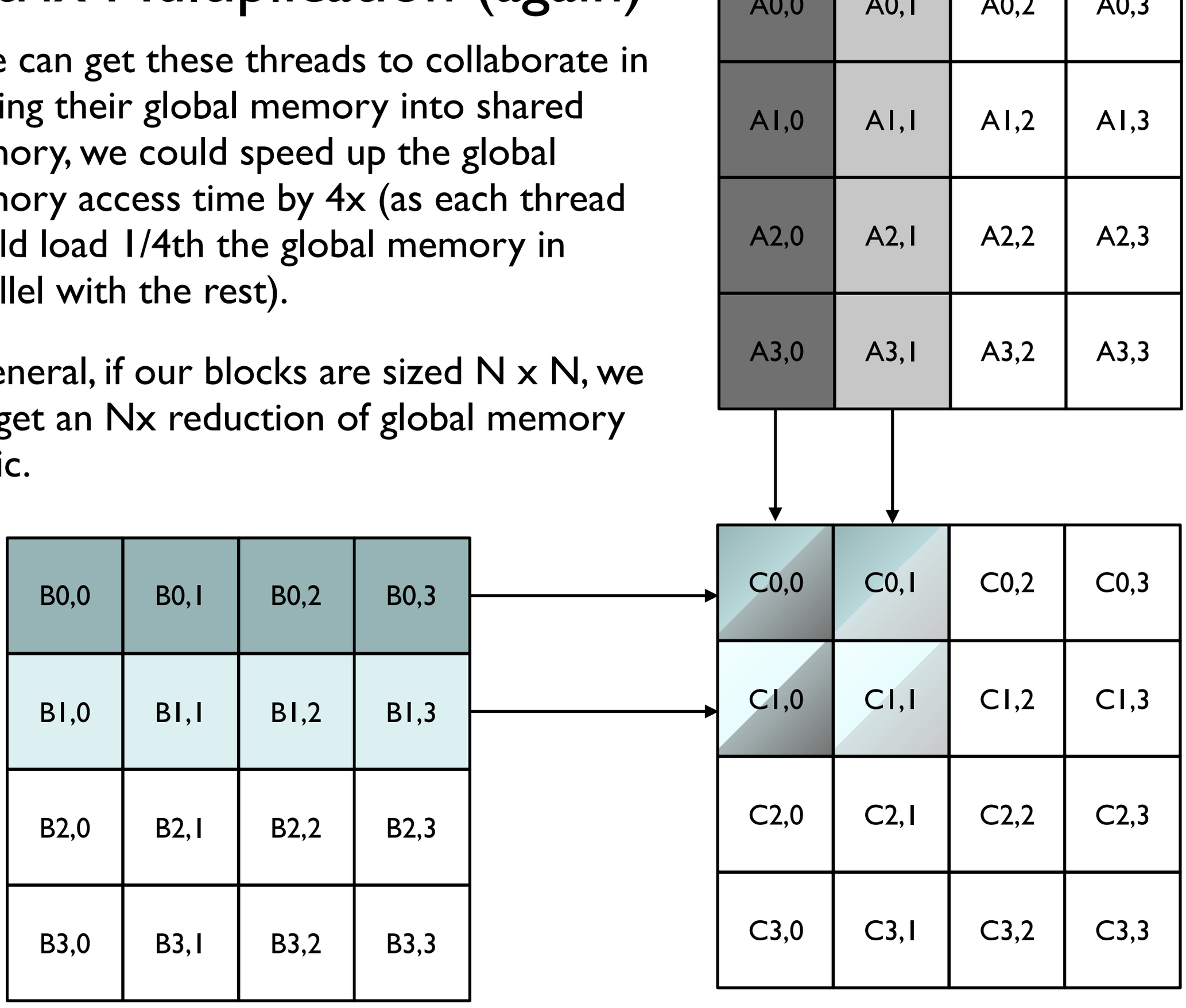
If we color one of those blocks, we can see that there is a lot of overlap in terms of memory accessed.



Matrix Multiplication (again)

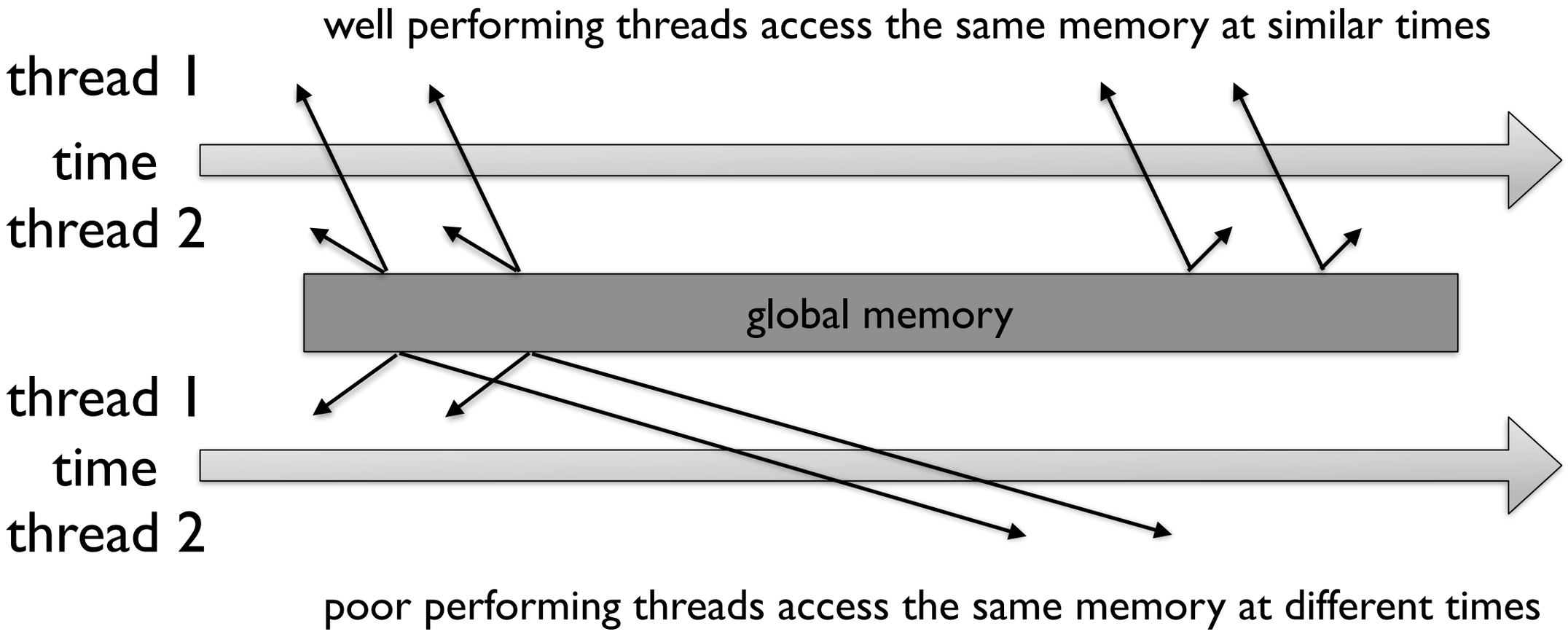
If we can get these threads to collaborate in loading their global memory into shared memory, we could speed up the global memory access time by 4x (as each thread would load 1/4th the global memory in parallel with the rest).

In general, if our blocks are sized N x N, we can get an Nx reduction of global memory traffic.



Tiling and Caching

If we can get these threads to collaborate in data loading, we also benefit from how DRAM works. If the threads are grabbing data from a similar region in global memory, when cache lines are loaded from DRAM they will most likely contain most of the requested data from all the threads, even further reducing the overhead of memory access.

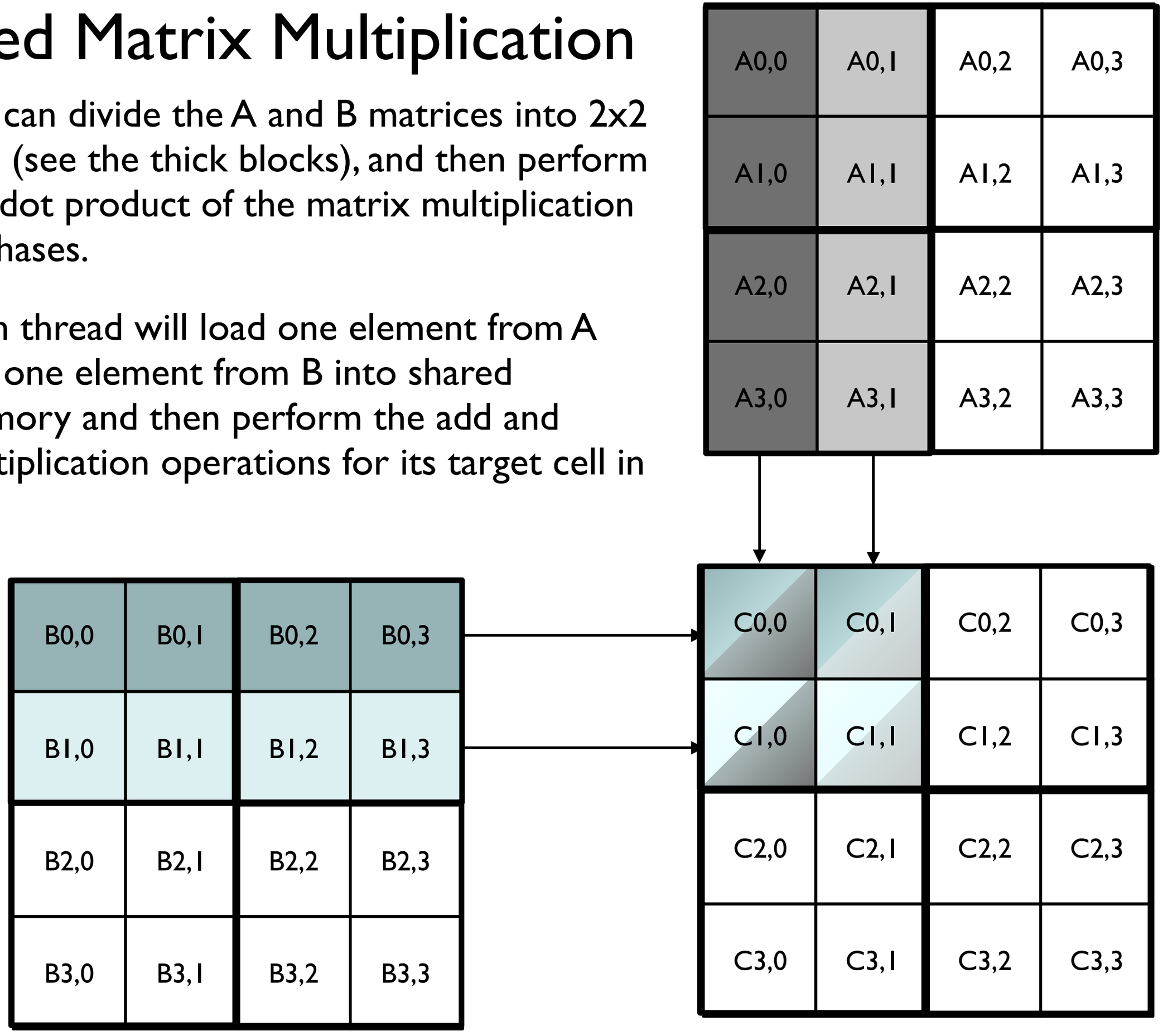


A Tiled Matrix Multiplication Kernel

Tiled Matrix Multiplication

We can divide the A and B matrices into 2x2 tiles (see the thick blocks), and then perform the dot product of the matrix multiplication in phases.

Each thread will load one element from A and one element from B into shared memory and then perform the add and multiplication operations for its target cell in C.



Tiled Matrix Multiplication

	Phase 1			Phase 2		
thread 0,0	A[0,0] -> Ashare[0,0]	B[0,0] -> Bshare[0,0]	C[0,0] += Ashare[0,0] * Bshare[0,0] + Ashare[0,1] * Bshare[1,0]	A[0,2] -> Ashare[0,0]	B[2,0] -> Bshare[0,0]	C[0,0] += Ashare[0,0] * Bshare[0,0] + Ashare[0,1] * Bshare[1,0]
thread 0,1	A[0,1] -> Ashare[0,1]	B[0,1] -> Bshare[0,1]	C[0,1] += Ashare[0,0] * Bshare[0,1] + Ashare[0,1] * Bshare[1,1]	A[0,3] -> Ashare[0,1]	B[2,1] -> Bshare[0,1]	C[0,1] += Ashare[0,0] * Bshare[0,1] + Ashare[0,1] * Bshare[1,1]
thread 1,0	A[1,0] -> Ashare[1,0]	B[1,0] -> Ashare[1,0]	C[1,0] += Ashare[1,0] * Bshare[0,0] + Ashare[1,1] * Bshare[1,0]	A[1,2] -> Ashare[1,0]	B[3,0] -> Ashare[1,0]	C[1,0] += Ashare[1,0] * Bshare[0,0] + Ashare[1,1] * Bshare[1,0]
thread 1,1	A[1,1] -> Ashare[1,1]	B[1,1] -> Ashare[1,1]	C[1,1] += Ashare[1,0] * Bshare[0,1] + Ashare[1,1] * Bshare[1,1]	A[1,3] -> Ashare[1,1]	B[3,1] -> Ashare[1,1]	C[1,1] += Ashare[1,0] * Bshare[0,1] + Ashare[1,1] * Bshare[1,1]

time →

Note that the different 2x2 blocks of memory are loaded in parallel in the first part of each phase, then the same add/multiple from the same parts of shared memory (but loaded from different global memory) are done in the second part of each phase. Note that each value loaded into shared memory is used twice.

Locality

This strategy allows us to exploit locality, we can copy data into shared memory once, and then reuse it multiple times. As accessing shared memory is much quicker this gives us a lot of speed up (and if our global memory reads are aligned properly then we can also benefit from reading global memory caching).

Tiled Matrix Multiplication

```
#define TILE_WIDTH 16
__global__ void matrixMulKernel(float* A, float* B, float* C, int width) {
  __shared__ float Ashare[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Bshare[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;

  //calculate the row and column for this element of the matrix
  int row = by * TILE_WIDTH + ty;
  int col = bx * TILE_WIDTH + tx;

  float result = 0;
  //loop over the A and B tiles required to compute the C element
  for (int m = 0; m < width / TILE_WIDTH; m++) {
    //collectively load the A and B tiles into shared memory
    Ashare[ty][tx] = A[(row * width) + (m * TILE_WIDTH) + tx];
    Bshare[ty][tx] = B[(((m * TILE_WIDTH) + ty) * width) + col];
    __syncthreads(); //wait for all the shared memory to be loaded

    for (int k = 0; k < TILE_WIDTH; k++) {
      result += A[ty][k] * B[k][tx];
    }
    __syncthreads(); //make sure all threads have done their calculation
                     //before modifying the shared memory.
  }
  C[(row * width) + col] = result;
}
```

Performance Improvement

With a TILE_WIDTH of 16, this improves the CGMA ratio from 1:1 to 16:1, or a ~16x speedup over the non-tiled version.

Parallelism Limited by Memory

Memory Limits to Parallelism

There are limits to the number of registers and shared memory any CUDA device can utilize, and this will limit the amount of parallelism of your kernels.

Suppose on some GPU, a SM can run up to 1,536 threads and has 16,384 registers. While that is a large number of registers, 16,384/1536 = 10.6666…

That means each thread can only use 10 registers. If the threads require more than 10 registers than the number of threads the SM can run concurrently will drop, based on the block granularity; so if each block had 512 threads, than threads using11 registers would be able to run 1024 threads instead of 1536. If threads required 17 registers, then only 512 threads could be run in parallel per SM.

Memory Limits to Parallelism

Shared memory is also limited in a similar fashion. Supposed a device has 16K of shared memory. If each SM can accommodate 8 blocks, then each block cannot use more than 2K memory.

In the matrix multiplication example, if we used 16 x 16 tiles, then each block needs (16 x 16 x 4(bytes per float)) = 1K storage for A, and another 1K for B. This means we could only use 8 blocks given 16k shared memory.

Conclusions