1. By using matrix tiling of 32x32, it will reduce memory bandwidth usage by 32 times aka the tile size for input matrices M and N since each row and each column can be reused up to 32 times to compute values without the need to re-fetch the same data from the global memory.
2. For case A where peak FLOPS = 200 GFLOPS and memory bandwidth = 100GB/s

AI = 36 FLOPS / (7\*4 bytes) = 1.29 FLOPS / byte

Achievable FLOPS = 100 GB/s \* 1.29 = 129 GFLOPS

Since 129 GFLOPS is less than the peak computing capacity of 200 GFLOPS, this kernel is memory-bound

For case B where peak FLOPS = 300 GFLOPS and memory bandwidth = 250GB/s

Achievable FLOPS = 250 GB/s \* 1.29 = 322.5 GFLOPS

Since the achievable FLOPS of 322.5 is higher than the peak computing FLOPS of 300 GFLOPS, this kernel is compute-bound.

1. Each block has 128 threads and the grid dimension is (N = 1024 + 128 – 1) / 128 = 9 blocks. In total, there would be 9 \* 128 = 1152 threads and there is one version of var i per thread so there will be 1152 versions of i

Each thread will have its own separate instance of x[] and since there’re 1152 threads, there will be 1152 versions of x[]

Y\_s is in shared memory and accessible by all threads within the same block and there are 9 blocks so there will be 9 version of the variable y\_s

B\_s[] is in shared memory and accessible by all threads within the same block and there are 9 blocks so there will be 9 version of the variable y\_s

Y\_s uses 4 bytes since its type is float which uses 4 bytes. B\_s[] array of 128 float elements will use 4 \* 128 = 512 bytes. In total, the shared memory usage per block is 512 + 4 = 516 bytes.

From the code, the computation steps consist of 4 multiplication and 5 additions of floats for a total of 9 FLOPS for each loop and there are 4 loops which means that there are 36 FLOPS per thread. The kernel access global memory 4 times to read from a and 1 time to write to b per loop iteration so total bytes transfer per thread is 5 operations \* 4 bytes = 20 bytes. Finally, we’re able to calculate the AI = 36 / 20 = 1.8 FLOPS / bytes to get the compute-to-global memory access ratio.

1. Case A

Number of Block = 2048 / 64 = 32 blocks required which meets 32 blocks allowed

Number of Register = 27 \* 64 \* 32 = 55, 296 registers required which meets the 65,535 registers allowed

Shared memory capacity = 4KB \* 32 = 128KB required which does not meet the 96KB limit for shared memory capacity

The limiting factor is the shared memory and this configuration can’t achieve full occupancy

Case B

Number of Block = 2048 / 256 = 8 blocks required which meets the 32 blocks allowed

Number of Registers = 31 \* 256 \* 8 = 63,488 registers required which meets the 65,535 registers allowed

Shared memory capacity = 8KB \* 8 = 64 KB required which meets the 96KB shared memory

There is no limiting factor and this configuration can achieve full occupancy.

1. The access to array a in line 5 is **coalesced** because each thread in a warp accesses consecutive memory addresses in a where I increments by 1 for each threadIdx.x in the block

**Coalescing not applicable** to shared memory in a\_s in line 5

The access to array b in line 7 is **coalesced** because each thread in a warp accesses consecutive memory addresses in b where I increments by 1 for each threadIdx.x in the block

The access to array c in line 7 is **uncoalesced** because the threads are accessing non-consecutive memory locations with a stride of 4

**Coalescing not applicable** to shared memory in bc\_s in line 7

**Coalescing not applicable** to shared memory in a\_s in line 10

The access to array d in line 10 is **coalesced** because each thread in a warp accesses consecutive elements of d with respect to the offset of 8

**Coalescing not applicable** to shared memory in bc\_s in line 11

The access to array e in line 11 is **uncoalesced** because the threads are accessing non-consecutive memory locations with a stride of 8