|  |  |  |  |
| --- | --- | --- | --- |
|  | | 3 Questions:   * 2\*numACols\*numCRows\*numCCols FLOPS * numCRows\*numCColumns global memory writes * numARows\*numACols\*ceil(numCCol/TILE\_WIDTH) + numBRows\*numBCols\*ceil(numCRows/TILE\_WIDTH) global memory reads | |
|  | | | 4 Questions:   * z\_size\*y\_size\*x\_size global memory writes * Some threads in the block were only used to load halo elements. * While others added in actual elements and then performed the convolution on the output tile elements. * Each thread calculates only one output. * If the mask is a cube with 5 elements on each side what is the trend of the average number of times each input element will be accessed from the shared memory during the calculation of an output tile as a function of the input tile width? * (C) Increases with the width of the tile size with a limit of 125 |
|  | | | 5.1 Questions:   * N global memory reads * ceil(N/(2\*BLOCK\_SIZE)) global memory writes * Min 0 or 1 FLOPS * Max log2(SECTION\_SIZE) |
|  | 5.2 Questions:   * Reduction step: (2\*BLOCK\_SIZE – 1)\*ceil(N/(2\*BLOCK\_SIZE)) FLOPS * Post-reduction step: (2\*BLOCK\_SIZE – 1 – log2(BLOCK\_SIZE))\*ceil(N/(2\*BLOCK\_SIZE)) FLOPS * N global memory reads * N + ceil(N/(2\*BLOCK\_SIZE)) global memory writes * Min 0 or 1 FLOPS * Max 2\*log2(SECTION\_SIZE) - 1 FLOPS * 2\*(1+log2(BLOCK\_SIZE)) number of synchronizations | | |
| Synchronization:   * In CUDA, a \_\_syncthreads() statement, if present, must be executed by all threads in a block. When a \_\_syncthread() statement is placed **in an if-statement, either all threads in a block execute the path that includes the \_\_syncthreads() or none of them does.** For an if-the-else statement, if each path has a \_\_syncthreads() statement, either all threads in a block execute the then-path or all of them execute the else-path. The two \_\_syncthreads() are different barrier synchronization points. If a thread in a block executes the then-path and another executes the else-path, they would be waiting at different barrier synchronization points. They would end up waiting for each other forever. It is the responsibility of the programmers to write their code so that these requirements are satisfied.   Streaming Multiprocessors:   * Once a kernel is launched, the CUDA run-time system generates the corresponding grid of threads. As we discussed in the previous section, these threads are assigned to execution resources on a block-by-block basis. In the current generation of hardware, the execution resources are organized into Streaming Multiprocessors (SMs). * **Combination of 8 BLOCKS and 1536 THREADS can be assigned to each SM. There is a finite number of registers per SM as well.** * Excess register usage in CUDA kernel can reduce performance ---> local variables map to registers   DRAM Burst/Memory Coalescing:   * Recognizing the burst organization of modern DRAMs, current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads into favorable patterns. This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. **That is, the most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations. In this case, the hardware combines, or coalesces, all these accesses into a consolidated access to consecutive DRAM locations.**   Warps, Control Divergence, and SIMD Model:   * **32 threads in a warp** ---> try to use multiples of 32 when creating block size ---> if you don’t, then the block will be padded with extra threads that do nothing * Warps are placed in row major order within a block * For an if-else construct, the execution works well when either all threads execute the if part or all execute the else part. **When threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these divergent paths.**   Thread Granularity:   * **Computation / Communication Ratio:** In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. Periods of computation are typically separated from periods of communication by synchronization events. * **Fine-grain Parallelism:** Relatively small amounts of computational work are done between communication events. Low computation to communication ratio. Facilitates load balancing. Implies high communication overhead and less opportunity for performance enhancement. If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. * **Coarse-grain Parallelism:** Relatively large amounts of computational work are done between communication/synchronization events. High computation to communication ratio. Implies more opportunity for performance increase. Harder to load balance efficiently. | | | |