|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | 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 |
|  | | | | Questions:   * Atomic Operations: imageWidth\*imageHeight + ceil(imageWidth\*imageHeight/BLOCK\_SIZE)\*HISTOGRAM\_LENGTH * Glob Mem W: ceil(imageWidth\*imageHeight/ BLOCK\_SIZE)\*HISTOGRAM\_LENGTH * Glob Mem R: imageWidth\*imageHeight + ceil( imageWidth\*imageHeight/BLOCK\_SIZE)\*HISTOGRAM\_SIZE * Block Partitioning: each thread handles a section * Thread Interleaving: threads jump b/w sections * L2 cache optimization is hardware based * int **atomicAdd**(int \* address, int val); * Problem with atomic operations: latency determines throughput |
| 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.** * **65536 blocks and 1024 threads per block and 32 threads per warp** * Excess register usage in CUDA kernel can reduce performance ---> local variables map to registers   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. | | | | |
|  | | http://webgpu.com/mp/24/imgs/figure.png   * JdsColStartIdx = matColStart | jdsColIdx = matCols | jdsData = matData | jdsRowNNZ = matRows | jdsRowPerm = matRowPerm | | |
| CUDA Streams:    **cudaStreamSynchronize**(stream\_id) -> only the stream provided  **cudaHostAlloc**(void(\*\*)&host, numByte, cudaHostAllocDefault); **cudaHostFree**(host); | |  | | |
| CNN:   * (TILE\_SIZE\*MASK\_WIDTH)2/( TILE\_SIZE+ MASK\_WIDTH -1)2 = bandwidth reduction for 2D internal elements using tiled shared mem * Ceil((INPUT\_X-MASK\_WIDTH + 1)/(TILE\_SIZE)) \* Ceil((INPUT\_Y-MASK\_WIDTH + 1)/(TILE\_SIZE)) = number of blocks * total number of input element after replication is H\_out\*W\_out\*K\*K times for each input feature map * total number of elements in each original input feature map is (H\_out+K-1) \* (W\*out+K-1) * replication factor: (INPUT\_MAPS\*H\_out\*W\_out\*K\*K)/(INPUT\_MAPS\*H\_IN\*W\_IN) * Example in img below, replication factor: (3\*2\*2\*2\*2)/(3\*3\*3) | p | | | |
| CNN Matrix Multiplication: Pass to Matrix Multiplication Kernel After Unrolling    Basic Kernel Code (Not to use with Matrix Multiplication: |  | | | |