Daniel Garon
Dr Sanjay Rajopadhye
CS 475
2023-04-07

# PA4: CUDA First Steps

## Vector MAX: Coalescing Memory Accesses

The provided vecMax.cu program performs a partial max reduction on a 1D array. It invokes the vecMaxKernel00 in which partial reductions are computed and stored in an array. Equal size contiguous segments of the input array are divided among a specified number of blocks. Within each block, equal size contiguous segments are divided among a specified number of threads. The threads then perform a max reduction on their segment, storing their respective answers in the *reductions* array. The main program then computes the max of the partial answers.

The following image illustrates how the reads, and resulting max reduction, are divided in vecMaxKernel00 using an example input array of size 16, and a call to the kernel which specifies two blocks, and two threads per block:

| blockIDx.x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| threadIDx.x | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Element | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

*Figure 1: non-coalesced memory access pattern*

In vecMaxKernel01, I modified the program so that each thread within each block will read adjacent array elements within a segment of the array equal to the size of the number of threads, i.e., *blockDim.x*. After each iteration, the beginning of the segment is incremented by *blockDim.x* and the process repeats. As before, each thread stores their respective answer to the max reduction in the output *reductions* array.

The following image illustrates how the computation is divided among the threads in vecMaxKernel01.cu using an example input array of size 16, and a call to the kernel which specifies two blocks, and two threads per block:

| blockIDx.x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| threadIDx.x | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Element | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

*Figure 2: coalesced memory access pattern*

| | Time (s) |
|---|---|
| vecMaxKernel00 | 0.364 |
| vecMaxKernel01 | 0.041 |

*Table 1: vecMax execution times*

The alignment of the memory accesses allows the hardware to perform multiple reads in a single transaction. In the actual program, 128 threads read 128 adjacent elements. I believe this allows the reads to be organized such that 32 four-byte floats are read per transaction. This allows for all 32 threads in a warp to read an element with no bank conflicts.

With 80 blocks at 128 threads-per-block and an input size of 1,280,000,000, the result of the coalesced memory accesses is a speedup of 8.8.

**Shared CUDA Matrix Multiply**

The matmult program performs a matrix multiplication on two input matrices *A, B* and stores the result in a third matrix *C*. The computation is divided such that each block is responsible for a square portion of matrix *C*.

The kernel is invoked with a 2D grid of blocks, size 100 x 100. Within each block, a 2D team of threads is allocated, size 16 x 16. Each block iterates over a *block sized* segment of rows of *A* and columns of *B*, performing a dot product. Each of the 256 threads is responsible for one of the 256 results in its respective block of *C*.

The modified version of the program distributes the computations such that rather than a one-to-one correspondence between block dimensions and the number of results to be computed, a square footprint, twice the width / height of a block is allocated to each block. Each block, with the same 256 threads is responsible for a patch four times as large. Thus, each block is computing 1,024 results, four per thread.

The execution time improves by a constant factor of approximately 2.5 among the tested input sizes. The semantics of the computation is unchanged, and the algorithm retains a cubic time complexity.

| Matrix Size | 1600 x 1600 | 3200 x 3200 | 6400 x 6400 |
|---|---|---|---|
| matmultKernel00 | 0.017 | 0.118 | 0.968 |
| matmultKernel01 | 0.006 | 0.048 | 0.376 |

*Table 2: matmult execution times*

The average GFLOPS for the initial version of the program is approximately 500. The larger footprint increases the GFLOPS to approximately 1,400. The performance of both versions of the program was fairly consistent for the matrix sizes tested.

Some of the changes I experimented with to observe the effect on performance:

- Increasing the block size to 32 x 32 and the footprint size to 64 x 64 resulted in a performance decrease with GFLOPS dropping back down to approximately 1,000.
- Increasing the footprint size to 48 x 48 with a block size of 16 x 16 resulted in a small performance increase over matmult01. In this scheme, each thread is responsible for nine values in the results matrix.
- Increasing the block size to 32 x 32 for matmult00. This resulted in a small performance increase to approximately 650 GFLOPS.
- Aligning the threads during the copy phase so that 32 contiguous elements are copied over, with the shared matrices filled in row-major order. In matmultKernel01, the footprint-sized shared matrices are populated in block-sized segments, 16 wide. This resulted in a small improvement averaging approximately 1,480 GFLOPS.
- Increasing the width of shared_A and the height of shared_B so that we require fewer synchronization steps and maximize the use of the shared memory. This resulted in no improvement over matmult01.

Based on the performance results for the different versions of these two programs, it seems that we want to structure our code so that we (read: the compiler) make the best use of the hardware. One of the most important things we can optimize are the memory accesses. If we use a memory access pattern that allows for coalescing, we significantly reduce the number of memory transactions. We can also balance the number of arithmetic operations so that global memory latency can be hidden by the thread scheduler. I believe this to be the primary reason for improvement in matmult01.