EECS 224 - HW3

Reduction

I average the bandwidth result of 5 trials for different kernels. The result is shown in the table:

Kernal	Naïve	Stride	Sequential	First_add	Unroll	Multiple
Bandwidth (GB/s)	9.99	14.93	17.59	34.96	54.58	83.43

- Naive

  > N: 8388608
  >
  > Timer: gettimeofday
  >
  > Timer resolution: ~ 1 us (?)
  >
  > Time to execute naive GPU reduction kernel: 0.003353 secs
  >
  > Effective bandwidth: 10.01 GB/s
  >
  > Time to execute naive CPU reduction: 0.147653 secs
  >
  > SUCCESS: GPU: 41.957069   CPU: 41.957066

  As the terminal output states, input array size is $8 * 1024 * 1024 = 8388608$, execution time is 0.003353 secs and effective bandwidth is 10.01 GB/s.

  The effective bandwidth is calculated as input vector size (in GB) divided by the execution time (in second).

- Stride

  About 50% percent faster than Naive kernel.

- Effective bandwidth of Sequential, First_add, Unroll and Multiple kernel is reported in the table.

Matrix Transpose

Because we are dealing with a 2D matrix, it is more natural to let threads and thread blocks structured in a 2D manner. Each thread blocks is responsible for transposing a small matrix patch of the big matrix. Given that the whole big matrix is a square matrix, the patch is also square.

Instead of using 1 thread to transpose 1 elements, I decide to let 1 thread deal with multiple elements. This approach is kind of similar to the cascading one in the reduction part. As tested, having each thread transposing 8 element is the most efficient configuration. Also take into account that it is efficient to have about 256 total # of threads in a thread block, I set the matrix patch size to 32 * 32. Accordingly, the block size is 32 * 4 * 1. To deal with the case when N is not devisable by 32, I $ceil$ the # of thread blocks needed.

I firstly wrote a intuitive approach. Thread blocks directly save an element in the input memory to its corresponding position of the output memory. Only one of the read/write operation toward the global memory could be coalesced.

With this intuitive approach, I achieved about 4.0 billion elements/sec bandwidth with $N = 2048$.

I then add the usage of shared memory to my code. Shared memory is fast and does not have the coalescing issue. It does not matter if we read the shared memory row by row or column by column. So I use an 2D scratch of size 32 * 32 in shared memory. I read the input matrix to shared memory row by row (coalesced) and also write it back to output global memory row by row (by reading the shared memory column by column). In this way, both read and write of the global memory is row by row (coalesced).

With this improved approach where both read and write to global memory are coalesced, I achieved about 7.2 billion elements/sec bandwidth with $N = 2048$.

Then I check Nvidia documentation for Compute Capability 2.0 (The Compute Capability of Tesla 2090 on gpu queue). It turns out that Tesla 2090 has 32 shared memory banks. If we use 32 * 32 scratch, all 32 threads within a warp attempt to read the same memory bank when reading the shared scratch column by column, leading to a 32-way bank conflict. I then adjust the scratch size to be 32 * 33. By wasting $\frac{1}{33}$ memory, the element in the same column is evenly distributed across the 32 memory banks of the shared memory. Thus, there is no bank conflicts when multiple threads access data at the same column simultaneously.

With this improved approach where both read and write to global memory are coalesced, I achieved about 13.5 billion elements/sec bandwidth with $N = 2048$.

Previous performance are tested on gpu queue. Finally, I test its performance on $gpu2$ queue with different matrix size N:

N	1000	2000	4000
Bandwidth (Billion elements / sec)	2.3	16.5	19.3

As is shown, as the matrix size grows, effective bandwidth also grows because the overhead (probably comes from thread synchronization) is amortized.