REPORT – CSC 512 - PROJECT 5

Glen Ashwin Menezes
Unity Id: gmeneze

1) Performance Issues:

The original code had several performance issues :-

1. Each thread in a block computes the sum for all elements, this results in each thread in the block doing the same work of computing sum. This could be done in a more efficient way using a shared array in CUDA and reduction techniques like Butterfly or Sequential addressing reduction.
2. The elements of the in[] array are not accessed in an optimal manner. The block of elements are accessed column by column (y-coordinate * width + x-coordinate) instead of row by row. This results in poor utilization of cache. Thus, memory optimization is needed.
3. The presence of if-else conditions near the end of the CUDA routine results in thread divergence. The control flow path taken by different threads is different.
4. The variable width passed to the CUDA function is a constant. It is the same for all warps. It would be a good idea to define it as a device constant and it should help improve performance.

2) Optimizations:

The following optimizations were performed :-

1. Loop unrolling - opt1.cu
   - The nested for loop is used by each thread to compute the sum of all elements in the block. This is not needed to be done by each thread.
   - I have used a __shared__ array which is shared among all the threads in the block. Each thread updates one index (corresponding to it's threadId) in this array. This way, each thread is responsible for only one element (obtained by multiplying one element of in[] and mul[] arrays).
   - A barrier (syncthreads) is added after this point to ensure that further processing is done after each thread is done computing value explained above.
   - After this, sequential addressing reduction technique is applied to reduce (sum) the array parallel-y. This evenly distributes computation among all threads in an efficient way.
     Reference :- NVidia – Optimizing parallel reduction in CUDA
   - This optimization falls under 3 (Increase the amount of parallelism) and 4 (Minimize the number of instructions)

2. Memory Optimization - opt2.cu

- The elements of the in[] array are not accessed in an optimal manner as explained in point 2 in Performance Issues. Cache performance can be improved by changing the way we are accessing elements.
- If we think of the data layout, threadIdx.x represents the row number and threadIdx.y represents the column number. The original code accesses the data column by column. This results in poor cache performance, as when there is a cache hit several consecutive elements in the array are cached. By accessing column by column we don't advantage of this and the cache is refreshed repeatedly.
- The array index computations were changed to achieve better cache performance.
- This optimization falls under 1 (Memory optimizations).

3. Thread Divergence minimization - opt3.cu
   - The presence of if-else conditions near the end of the CUDA program causes thread divergence.
   - I have eliminated this by reducing all conditions to a single expression.
   - The expression satisfies all the conditions enforced by if-else in the original code.
   - Thus, we avoid splitting up the control flow for threads based on their thread-Ids.
   - This optimization falls under 2 (Minimize thread divergences).

4. Device constant - opt4.cu
   - The variable width passed to the CUDA routine is a device constant. That is, it is a constant across all warps.
   - Thus it makes sense to define it as a device constant (__constant__) instead of passing it as a parameter.
   - The constant memory space resides in device memory and is cached in the constant cache. For all threads of a half warp, reading from the constant cache is as fast as reading from a register.
   - This optimization falls under category 1 (Memory optimizations).

3) Performance:

Version	GTX 480 Time	GTX 780 Time
norm.cu	0.016258s	0.016253s
opt1.cu	0.004046s	0.003249s
opt2.cu	0.015097s	0.014763s
opt3.cu	0.015438s	0.015489s
opt4.cu	0.016074s	0.016021s
optAll.cu	0.002473s	0.001119s

As we can see, there is a huge improvement in performance.

4) Experience on Tool:

I had some trouble using nvvp tool as it was extremely slow and cumbersome. I had to resort to using nvprof which was extremely useful as it helped me identify performance issues in the program. I used its outputs to query the advising tool. The suggestions from the tool were very useful and helped me identify and fix the issues.