CpSc 418 **Homework 5 Solution Set** 55 points

Version: April 7, 2018

Solution sample code can be found at

https://www.ugrad.cs.ubc.ca/~cs418/2017-2/hw/5/hw5-soln.zip.

The sample code does not include all of the tests that we ran against your code.

1. **Are We Having Another Moment?** (25 points). Implement the reduce pattern on CUDA to calculate the central moment approximations

$$E[X] \approx \frac{\sum_{i=0}^{n-1} x_i}{n} \qquad \text{and} \qquad \mu_k \approx \frac{\sum_{i=0}^{n-1} (x_i - E[X])^k}{n}, \tag{1}$$

for all moments $k = 2, 3, \ldots, k_{\max}$ where $k_{\max}$ is a (bounded) run-time parameter.

(a) (4 points) Why mixed precision?

Answer: Why double precision for calculations? Single precision will not be accurate enough. As discussed in K&H chapter 6, single precision floating point has a roundoff error of about 10^{-7}. Since our data set may have $n \approx 10^8$ elements, it is likely that $\mathcal{O}(n)$ calculations in single precision will yield few accurate digits. In contrast, double precision floating point has a roundoff error of about 10^{-15}, so we may get some accurate digits if we keep our calculations well-scaled.

Why single precision input data? Double precision takes up twice as much memory storage and bandwidth. Double precision requires 8 bytes while single precision only 4 bytes, so while we may be able to afford the extra memory (and corresponding global memory bandwidth demand) for small arrays (such as the output moments), we cannot for large arrays (such as the input data set). Mixed precision implementations are often needed in large reduction problems for these very reasons, although it is admittedly a headache to keep the precisions correct in the code.

Note that these explanations are much longer than what we required in your answers.

(b) (12 points) Correctness.

Answer: See the sample solution code in the `Moments-Soln/` subdirectory. In addition to the CUDA implementation, several features were also added to `moments-main.c` to help with debugging and validating the implementation:

- Another initialization routine `seq_float_vector()` which returns a vector containing an arithmetic sequence of values with specified start and step between consecutive values. The expectation and moments of such sequences are relatively easy to calculate by hand even for large vectors.
- CPU implementations of both the expectation and moment calculation. These are straightforward sequential loops, so it is easier to build confidence that they are correct by code inspection.
- A routine `check_double_vectors()` to compare the CPU and GPU results.
- A routine `total_fp_ops()` that returns the answer to part (c) to make it easier to report the FLOPS for a given run.

- Code to parse input arguments (to make it easier to try different parameters when optimizing for throughput in part (d)) and code to provide timing and throughput reports.

All of this scaffolding code is repetitive and annoying to write, but well worth the investment of an hour if it can save you several hours of debugging. Furthermore, I always find it best to code a CPU implementation before starting on the GPU implementation—not only does it ensure I understand the problem and its inputs and outputs, but sometimes the CPU version is fast enough and I can just quit.

(c) (2 points) Operations: Assume $n \gg 1$ and $k_{\max} \geq 2$, and consider the computational cost of (1) as a function of n and/or $k_{\max}$. How many additions are needed? How many multiplications are needed? Your answer must include any constants on the leading term(s) in n and/or $k_{\max}$, but you can use $\mathcal{O}()$ for the remaining term(s).

Answer: We report the computational cost of the serial algorithm. Although the parallel algorithm may perform additional operations (such as redundant computations to avoid communication) in order to improve parallelism, those operations are "computational overhead" and really should not be counted as improving the throughput (otherwise I could "improve throughput" by running the save-last recurrence from HW4 on a bunch of extra blocks to keep the SMs busy with irrelevant work whenever the reduction threads are stalled).

- Expectation requires $n-1$ additions and 1 division. Although the division is much more expensive than a single addition, for $n \gg 1$ it will be negligible. We ignore the divisions in the moment calculations for the same reason.
- Moment 2 requires n subtractions (one for each input element to compute the difference from the mean; subtractions count the same as additions), n multiplications (to compute the square) and $n-1$ additions (to compute the sum.
- After computing $(x_i - E[x])^2$ for element i, we can compute $(x_i - E[x])^3$ for the same element with just one more multiplication. Therefore, by reusing the partial products from moment 2, moment 3 requires n further multiplications and $n-1$ further additions.
- The same holds true for higher moments as well, so each moment beyond 2 requires n multiplications and $n-1$ additions.
- Total for μ_k for $k = 2, 3, \ldots, k_{\max}$:

$$\begin{aligned} \text{Additions: } & n - 1 + n + (k_{\max} - 1)(n-1) = nk_{\max} + \mathcal{O}(n + k_{\max}) \\ \text{Multiplications: } & (k_{\max} - 1)n = nk_{\max} + \mathcal{O}(n) \\ \text{Total: } & 2nk_{\max} + \mathcal{O}(n + k_{\max}) \end{aligned}$$

As a side observation: It does not appear that any of the calculations in the GPU kernel can be performed as fused multiply-adds (FMAs). Although at each iteration of the `k` loop we are performing a multiply (to increase the power) and then an add (into the partial sum for the current moment), we need to separately keep track of the result of the multiply in order to use it as the partial product for the next higher moment, so we will need a separate multiplication. Without FMAs, we should expect our theoretical peak throughput to drop by a factor of two immediately.

(d) (6 points) Throughput: **Answer:** See the sample solution code in the `Moments-Soln/` subdirectory. Key design features:

- I chose to use just a single block to perform all of the work so that I didn't need to worry about synchronization or sharing partial sums between blocks. Each thread in the block therefore had to handle `n / BLOCK_SIZE` input elements. The threads did the reduction on their assigned elements sequentially. The partial sum(s) for each thread were kept in shared memory to make the final reduction easier.
- I chose to use one kernel to compute the mean and one to compute all of the rest of the moments. By computing all of the moments together, each input data element only needed to be loaded from global memory once (plus once more for the mean). Because I am using only a single block, these two kernels could have been done in a single kernel by computing the mean, then a `__syncthreads()`, then computing the moments; however, I chose to break them up for clarity.
- After all threads had completed the sequential reduction on their portion of the input elements, the final result was computed using a reduction tree that is a slight modification of K&H figure 5.15.
- Threads were assigned non-contiguous input elements: A given thread's input elements were separated by `BLOCK_SIZE`. That arrangement meant that consecutively indexed threads (in particular, threads sharing a warp) would access consecutively indexed memory locations every time they read from the input array, and consequently, the global memory accesses could be coalesced.
- In the kernel which computes the moments, each thread is assigned `max_moments-1` shared memory locations in which to store the partial sums for moments 2, 3, ..., `max_moments`. For a given thread, these shared memory locations are also assigned non-continguously: the partial sums for all threads for moment 2 are stored consecutively, followed by the partial sums for all threads for moment 3, and so on. This choice ensures that consecutively indexed threads (in particular, threads sharing a warp) will access consecutively indexed shared memory locations and hence will **not** have shared memory bank conflicts (there is no coalescing of access to shared memory). However, this choice does slightly complicate the indexing process both for the reduction tree and for extracting the final sums for each moment.

Throughput was measured using the flop count described above and the median of five (or more) runs. Full points for achieving 4 GFLOPS, with a reduction of 1 point for each power of two reduction in speed; for example, throughputs of 1–2 GFLOPS will receive 4 points. Bonus points will be determined once we see what peak throughputs were achieved.

Note that we may measure a different throughput than you. That may be because you used a different operation count or it may be because your timing routines were placed in a different location. In particular, if you did not include `compute_all_gpu()` **and** either `after_run_gpu()` or `data_from_gpu()` inside your timing loop, you may have only measured the time to launch the kernel, not the time to complete it.

(e) (1 point) Efficiency: What fraction of peak throughput did your solution achieve on the GTX 1060 3GB GPUs in the lab? If you did not get a FLOPS count for your own code, you can report the fraction of peak throughput achieved by Ian's $5.5(10^9)$ FLOPS solution.

Answer: The peak throughput of the GTX 1060 3GB GPUs is reported in a number of places; for example, the GeForce 10 Series Wikipedia page. For double precision it is 108

GFLOPS (when not in boost mode). Therefore my code achieved

$$\frac{5.5(10^9)}{108(10^9)} \approx 5.1\% \text{ peak throughput.}$$

As promised, the reduce operation is not something that achieves high efficiency on GPUs.

2. **Convolution** (30 points). Implementing and exploring 2D convolution.

   (a) (8 points): Basic kernel. **Answer:** See the sample solution code in the `Conv-Soln/` subdirectory. Design features:

   - As recommended in K&H chapter 7, `set_up_mask()` places the mask data into constant memory on the GPU. The mask width parameters p and q (and their half widths) are also stored into GPU constant memory to avoid having to pass them as arguments to the convolution kernel; however, it would also be reasonable to store them into static global variables on the CPU and then pass them as arguments to the GPU at kernel launch.
   - Because the mask data is stored in the statically allocated constant memory, nothing needs to be done in `clean_up_mask()`.
   - We assign one thread to each output element. The code in the basic kernel `conv_basic_kernel()` is copied directly from the CPU implementation `conv_cpu()`, except that the outer two loops (in `i` and `j`) are replaced by the thread indexes and there is a test to ensure that we do not execute the convolution for threads whose output elements are outside the output array bounds.

   (b) (12 points): Tiled kernel. **Answer:** See the sample solution code in the `Conv-Soln/` subdirectory. Design features:

   - Each thread in the block is assigned one output element; consequently, once the halo elements are included the tile will contain more elements than there are threads in the block. This choice was made to increase the ratio of output elements to halo elements, but an alternative is to match the tile size to the block size and then only use some threads for output elements.
   - The tiled kernel works in three steps:
     i. Load input elements into the tile (including halo elements).
     ii. Compute the convolution for each output element.
     iii. Write the convolution result to the output array.

     Note that threads may be assigned to work with different elements in each of these steps.
   - Variables `i` and `j` maintain global thread indexes, which are useful for working with the input and output arrays. Variables `it` and `jt` maintain block thread indexes, which are useful for working with the tile in shared memory.
   - Because there are more input elements in the tile than threads, each thread may need to load multiple elements. The indexing gets messy. Threads with consecutive indexes are assigned input elements with consecutive addresses in order to ensure coalescing of global memory accesses. We achieve this arrangement by using variables related to `it` (and hence to `threadIdx.x`) as the row index into our column-major arrays.

- For halo elements which fall outside the input array, we set them to 0.0 during the tile loading stage. It is necessary to have code somewhere which identifies portions of the convolution which fall outside the input array. By handling this condition during the tile loading step, only one thread has to check the condition for any given tile element. If the condition were checked during the convolution, every thread which touches that element would have to check the condition. Note that in this solution every thread which touches that element does have to perform a multiplication by 0.0 and then an add to 0.0 during the convolution, but multiplies and adds (and especially fused multiply-adds) are faster than branching.
- The convolution section is almost identical to the basic kernel except that the indexing into the tile is based on thread block indexes rather than thread global indexes.

(c) (10 points): Explore the effect of one or two of the following parameters on GPU throughput by holding the other parameters constant and varying your chosen parameter(s):

- (Up to 2 points) Input data size $m = n$.
- (Up to 3 points) Mask size $p = q$.
- (Up to 4 points) Block size.
- (Up to 5 points) Tile size.
- (Up to 6 points) Number of output elements assigned to each thread.

Answer: There are many possible combinations of answers depending on which parameters you chose to explore and whether you used a basic or tiled version of the kernel. The answers below are representative but not complete.

- Input data size $m = n$. See figure 1. The hypothesis is that the larger the data set, the more threads which can run in parallel and hence the higher throughput. That hypothesis appears to hold true for input sizes from 64 to 2048, but the throughput drops abruptly for input sizes above that. It is not immediately clear what has happened, although it is possible that caching plays a role in reduced performance for larger input sets. The difference between basic and tiled implementations holds steady at a factor of two, which is rather disappointing considering that our quick analysis in class indicated that the tiling should increase CGMA by a factor near to pq, which in this case should be several hundred.
- Mask size $p = q$. See figure 2. The hypothesis is that the larger the mask size, the more computation to be done. Furthermore, the larger the mask the greater the CGMA in the tiled versions, because each output value depends on more input values and hence there is more reuse of the input values loaded into the tile. The figure shows a slight dependence on mask size for small masks, but throughput is almost flat for mask sizes larger that 21. More disappointingly, there does not appear to be any benefit from larger masks for the tiled version—for all but the smallest mask the tiled version always has roughly twice the throughput. Somewhat surprisingly, the tiled version does not suffer even as the mask becomes very big, despite the fact that for mask size 41, over 80% of the tile is halo elements.
- Block size. See figure 3. The hypothesis is that larger block sizes may be harder to schedule onto SMs (thus degrading performance) but will make better use of shared memory to increase CGMA. Based on the experiments, it seems the latter slightly outweighs the former. A block size of 4 does very poorly, since a 4×4 block has only 16 threads and hence fails to have even one complete warp. After that the power of 2 block sizes (8, 16, 32) gradually increase in throughput, while the two samples

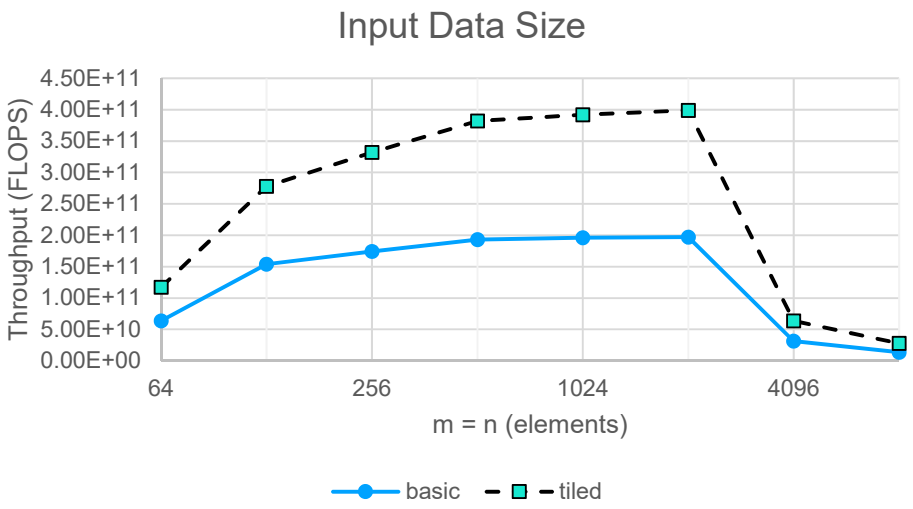

Figure 1: Throughput for basic and tiled convolution kernels for varying input data sizes with fixed mask size $p = q = 21$ and block size 32×32.

between these block sizes suffer slightly lower throughput. Potential causes for the latter effect are memory accesses poorly aligned with cache lines, or possibly for the block size of 12 a half-full warp.

- Tile size. In Ian's implementation, the tile size is determined by the block size and maximum mask size. For the runs described in the previous part, the maximum mask size was set to 41, so for block size $b \times b$, the tile size would have been $(b+40) \times (b+40)$. Therefore, see the "tiled" data in figure 3 for an exploration of tile size on throughput.
- Number of output elements assigned to each thread. Exploring this parameter would require a rewrite of the convolution code to allow each thread to handle more than a single output element. The hypothesis is that perhaps by assigning more work to each thread, it will be possible to expose more independent operations per thread and thereby get better (or the same level of) latency hiding while reducing the number of blocks and perhaps getting better use of the SMs (and in particular their caches). Whether that hypothesis is true will have to await somebody with the energy and time to recode their implementation.

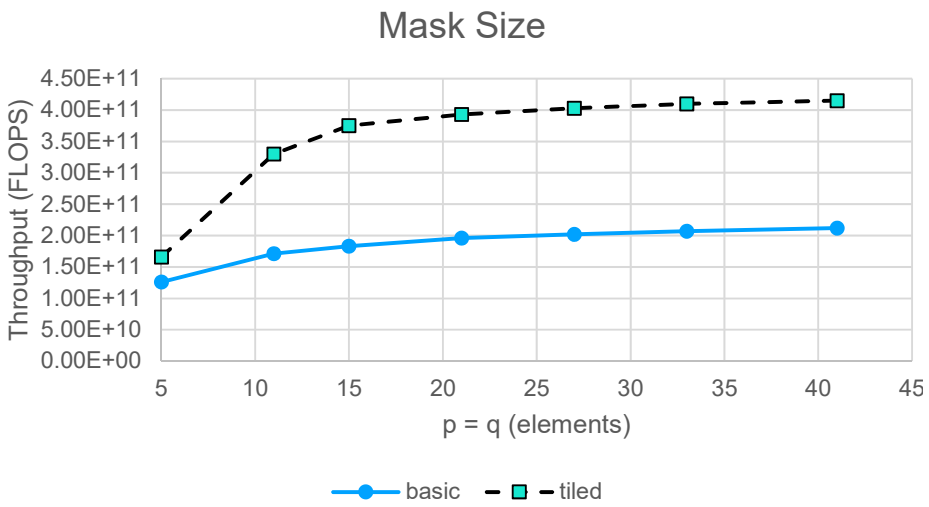

Figure 2: Throughput for basic and tiled convolution kernels for varying mask sizes with fixed input size $m = n = 1024$ and block size 32×32.

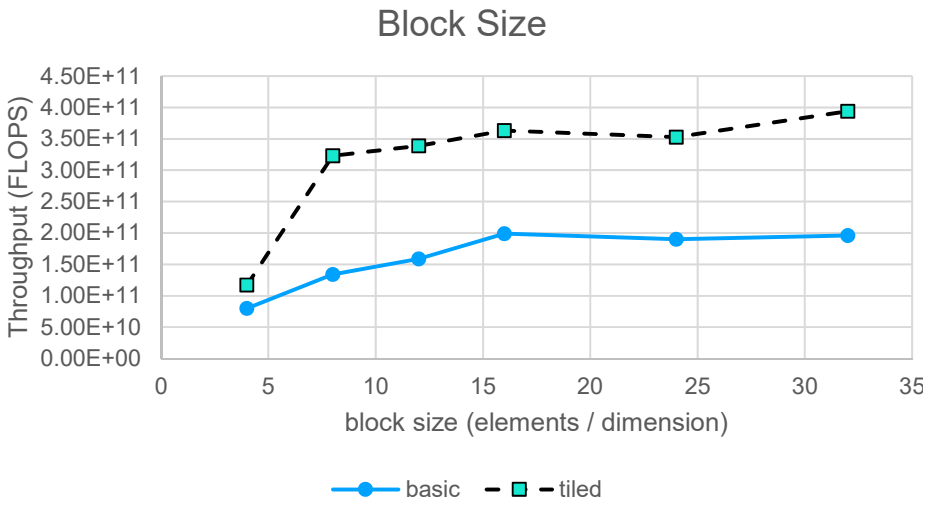

Figure 3: Throughput for basic and tiled convolution kernels for varying block sizes (or equivalently, tile sizes) with fixed input size $m = n = 1024$ and mask size $p = q = 21$.