CpSc 418 **Homework 4 Solution Set** 33 points

Solution sample code can be found at

http://www.ugrad.cs.ubc.ca/~cs418/2017-2/hw/4/code.html.

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

1. **Cracking RSA Keys** (15 points). Coming soon!

2. **Getting Started in CUDA** (18 points). We used the recurrence map:

$$x_i^{(j)} = (x_{i-1}^{(j)} + x_{i-2}^{(j)}) \mod 2^k \tag{1}$$

where $j = 0, 1, \ldots n-1$ is the index of which sequence we are computing, $i = 0, 1, \ldots, m-1$ is the element within the sequence, and $k \in \{2, 3, \ldots, 32\}$ is small enough that we won't overflow an unsigned 4-byte integer. The parentheses are placed around the superscript to indicate that it is an index not an exponent. We considered three versions of computing these sequences:

- "save-last": Given $x_0^{(j)}$ and $x_1^{(j)}$ for all j, compute and return $x_{m-1}^{(j)}$ for all j.
- "save-all": Given $x_0^{(j)}$ and $x_1^{(j)}$ for all j, compute and return $x_i^{(j)}$ for all j and all i
- "cum-sum": Given $z_1^{(j)}$ and $y_i^{(j)}$ for all j and $i = 0, 1, \ldots, m-3$, compute and return $z_{m-1}^{(j)}$ for all j which solves the following recurrence

$$z_i^{(j)} = (z_{i-1}^{(j)} + y_{i-2}^{(j)}) \mod 2^k.$$

(a) (3 points) Compare and explain CPU run times. Here are the times for $n = 20000$ and $m = 10000$ using the posted code (median of 5 runs on `lin13` when it was otherwise unloaded):

Version	Run time (seconds)
CPU save-last	0.169
CPU save-all	0.646
CPU cum-sum	0.388

The three versions show similar relative run times for other values of m and n that are not too small. The relative run times are almost certainly due to the different memory access patterns of the three versions:

- Save-last requires only two memory reads at the beginning of the iteration and one memory write at the end; consequently, most iterations can be performed entirely in the registers without any memory access to even the caches.
- Save-all requires a memory write on every iteration. While the cost of these writes will be somewhat reduced by the caches, they nonetheless significantly impact performance.
- Cum-sum requires a memory read on every iteration, so performance will again be significantly impacted. The fact that the impact is not as large as for save-all is likely due to spatial locality improving the caches' average read access time compared to their write time.

(b) (9 points) Implement the three recurrences on the GPU. See the sample solution code. The implementation for "cum-sum" follows the common GPU implementation pattern:

- The host function `cum_sum_gpu()` allocates input and output arrays of appropriate size on the GPU using the `NewArrayGPU()` macro (see `cuda-helper.h` for the definition of this macro), copies the input data from the host array to the GPU array using `cudaMemcpy()` and launches a kernel with suitable grid and block sizes.

- The kernel function `cum_sum_kernel()` is almost identical to the CPU implementation `cum_sum_cpu()`, except that index `j` (the sequence index) is now tied to the thread index rather than being generated by a loop. We choose to assign one thread to each sequence because the sequences can be computed independently (a key property of data parallelism).
  In contrast, assigning one thread to compute each element in a sequence (index `i`) would be a bad idea because element $x_i^{(j)}$ depends on previous elements $x_{i-1}^{(j)}$ and $x_{i-2}^{(j)}$; consequently, if these three were computed by different threads then the thread computing element $x_i^{(j)}$ would need to both wait for and communicate with the threads computing elements $x_{i-1}^{(j)}$ and $x_{i-2}^{(j)}$. Such synchronization and communication between threads is definitely inefficient and sometimes impossible on the GPUs.
- Once the kernel completes, the host function `cum_sum_gpu()` copies the output data back into CPU memory using `cudaMemcpy()` and frees the device memory using `cudaFree()`.

The same pattern is followed for the solutions of the other two recurrences.

No attempt was made to optimize the performance of any of the recurrences on the GPU, since this question was intended only to get you to practice the basic steps of porting code from a CPU to a GPU implementation. You may want to think about how (or even whether) performance could be improved for any of these recurrences using the techniques we have discussed in subsequent weeks.

(c) (3 points) Compare and explain GPU run times. Here are the times for the sample solution and $n = 20000$ and $m = 10000$ (median of 5 runs on `lin13` when it was otherwise unloaded):

Version	Run time (seconds)
GPU save-last	0.00092
GPU save-all	0.293
GPU cum-sum	0.162

The three versions show similar relative run times for other values of m and n. The relative run times follow a pattern similar to that on the CPU, except that instead of being 2–4 times faster than the other two, save-last is 170–300 times faster. It is also worth noting that save-all and cum-sum are only about twice as fast on the GPU as the CPU, while save-last is about 180 times faster on the GPU than the CPU.

- Save-last requires only two memory reads at the beginning of the iteration and one memory write at the end; consequently, most iterations can be performed entirely in the registers without any memory access to even the caches. Without the need for memory accesses, the GPU can use its thousands of SPs (and correspondingly huge register banks) to run thousands of sequences in parallel and hence dramatically outperform both the CPU and the other two GPU versions.
- Save-all requires a memory write on every iteration. These writes must all go through to global memory, so the run time is constrained by the global memory bandwidth. The fact that the GPU version is twice as fast as the CPU version is likely due to the higher memory bandwidth on the GPU.
- Cum-sum requires a memory read on every iteration, so performance will again be significantly impacted. The fact that the impact is not as large as for save-all is likely due to spatial locality improving the caches' average read access time. The GPU version of cum-sum likely beats the CPU version due to the higher global memory bandwidth.

(d) (3 points) Replace (1) with the recurrence

$$\begin{aligned} x_i^{(j)} &= \left((x_{i-1}^{(j)})^2 + 3x_{i-2}^{(j)} + w_{i-1}^{(j)}\right) \mod 2^k \\ w_i^{(j)} &= (w_{i-1}^{(j)} + 5x_{i-1}^{(j)}) \mod 2^k \end{aligned} \tag{2}$$

where $w_0^{(j)}$ is provided and $k < 16$ so we avoid overflow.

First we observe that (1) requires one addition and one modulo operation per iteration, for a total of two operations per iteration. The new recurrence (2) requires one square of a previous iterate, two multiplications by a constant, three additions and two modulo operations per iteration. Depending on how many of the multiplications can be fused with additions, the new recurrence will require 5–8 operations per iteration.

With that in mind, here are some hypotheses on what would happen to the GPU run times for each of the three versions of the recurrence:

- Save-last: This version was compute bound previously, so increasing the number of operations per iteration will have a dramatic effect on run time. The run time will increase by a factor of at least 2–4, and it might be worse depending on how well the compiler can hide the latency of some operations. However, this version will still be far faster than the other two even if it is slowed by a factor of 2–4.
- Save-all: This version was memory bound before and the new recurrence still requires one write per iteration; consequently, a little extra computation per iteration is unlikely to have a huge effect. Any increase in run time is unlikely to be significant.
- Cum-sum: This version was memory bound before and the new recurrence still requires one read per iteration; consequently, a little extra computation per iteration is unlikely to have a huge effect. Any increase in run time is unlikely to be significant.