High Performance Computing
Homework 4: due 4/4/22
Andrew Lipnick

Question 2

I improved the

`sin4_intrin`

AVX code by adding the terms up through x^{11} (see code).

Extra Credit

Using the fact that $e^{i(\theta+\frac{\pi}{2})} = ie^{i\theta}$ we get that $e^{i(\theta+n\frac{\pi}{2})} = i^n e^{i\theta}$. Combining with the fact that $e^{i\theta} = \cos\theta + i\sin\theta$ we get that for any θ we can find n such that $\theta' = \theta + n\frac{\pi}{2} \in [-\frac{\pi}{4}, \frac{\pi}{4}]$. Then we can evaluate either sin or cos for θ'. By only evaluating angles in $[-\frac{\pi}{4}, \frac{\pi}{4}]$ we can get a high level of accuracy for any angle without needed to add more terms. To decide whether to use sin or cos (and if we need to multiply by -1, we look at we then look r, the remainder of dividing n by 4 (because $i^4 = 1$) and use the following results:

$$\begin{aligned} e^{i\theta} &= \cos\theta + i\sin\theta \\ ie^{i\theta} &= i\cos\theta - \sin\theta \\ i^2e^{i\theta} &= -\cos\theta - i\sin\theta \\ i^3e^{i\theta} &= -i\cos\theta + \sin\theta \end{aligned}$$

Combined with the fact that $i^{-1} = -i$ we get the following result:

$$\begin{aligned} \text{if } r = 0 \quad &\text{take } \sin(\theta') \\ \text{if } r = 1 \quad &\text{take } \cos(\theta') \\ \text{if } r = 2 \quad &\text{take } -\sin(\theta') \\ \text{if } r = 3 \quad &\text{take } -\cos(\theta') \\ \text{if } r = -1 \quad &\text{take } -\cos(\theta') \\ \text{if } r = -2 \quad &\text{take } -\sin(\theta') \\ \text{if } r = -3 \quad &\text{take } \cos(\theta') \end{aligned}$$

Where r is calculated in c by

```
r = n\%4;
```

Then all that is needed the first couple of coefficients for cosine's Taylor series.

Question 3

See code for how I implemented the parallel scan. One thing my method does is assume that the number of threads evenly divides N, this means we can only take the number of threads to be either 1, 2, 4, 5, 8, 10, or 16 (my computer only has 16 threads). I felt that this was enough to see the effects of varying the number of threads. If we wanted an arbitrary number of threads we could find the remainder, r, when dividing N by the number of threads and have the first r threads sum over one more term then the other threads. I ran the code on my laptop which as am Intel Core i9-10980HK CPU which has 8 cores and 16 logical processors 3.10GHz base speed, 512 KB L1 cache, 2.0 MB L2 cache, and 16.0 MB L3 cache. Here are the results for varying different numbers of threads:

```
num_of_threads = 1
sequential-scan = 0.157096s
parallel-scan   = 0.157303s
error = 0

num_of_threads = 2
sequential-scan = 0.158580s
parallel-scan   = 0.126898s
error = 0

num_of_threads = 4
sequential-scan = 0.152992s
parallel-scan   = 0.114606s
error = 0

nun_of_threads = 5
sequential-scan = 0.154852s
parallel-scan   = 0.111547s
error = 0

num_of_threads = 8
sequential-scan = 0.155850s
parallel-scan   = 0.124572s
error = 0

num_of_threads = 10
sequential-scan = 0.153784s
parallel-scan   = 0.127744s
error = 0

num_of_threads = 16
sequential-scan = 0.154110s
```

```
parallel-scan   = 0.139279s
error = 0
```

The first thing to note is that with only 1 thread the parallel-scan is just as fast as the sequential scan as expected (slightly slower as expected do to a few if calculations and if statements). Then we note that as we increase the number of threads the time for the parallel scan decreases with the largest decrease being from going 1 thread to 2, then 2 to 4, with the 5th thread only decreasing the time by a little bit. Then the time begins increasing for thread counts of 8, 10, and 16. This makes sense because the calculations done by each thread are very quick so at some point the overhead of organizing the threads becomes more costly than the speed up gained from doing the work in parallel. Additionally, by doing the calculation in parallel we actually increase the total number of flops needed because it requires updating each threads calculation with the previous threads.

If we were to have a fixed number of threads we could potentially have even more of a speed up. At the moment I coded it to have a barrier before each thread uses the previous threads final result to update but we could have the second thread start once the first is done and have that thread start from the largest value so once it is done with that largest value the next thread can begin. This can be done by having a mutex for each thread then each thread locks their mutex until they have calculated their largest value (by first adding up all their values for A then taking the last value from previous thread) then the thread can unlock the mutex and add the previous threads result to the remaining values. I am not sure if this would be faster because each thread can begin adding the previous thread's results to its values sooner, but there would also be multiple locks which may be more overhead and thus slower than just having two barriers.