1.

For the final project, I am working with Guilherme Albertini and Mayukh Ghosh. We are going to parallelizing Stochastic Gradient Descent using the Hogwild! algorithm using both

OpenMP and CUDA. Then we are going to compare runtime and scaling using the OpenMP version against the CUDA implementation.

Planned Tasks:

1) Implement serial SGD.

2) Implement Hogwild! SGD using CUDA

a) Distribute each batch of training data over the SM's.

3) Experiment with different sizes of training data and mini batches.

4) Analyze total execution time and scalability of CUDA version compared to the serial version.

a) For CUDA, also analyze time used for each part, like cudaMalloc vs cudaMemcpy vs the

GPU computation time.

5) Explore new ways of parallelization.

2.

I choose to improve the accuracy to 12-digits for the AVX part of the function sin4_intrin().

Reference time: 16.9668

Taylor time: 3.2785 Error: 6.928125e-12

Intrin time: 0.8983 Error: 6.928125e-12

For Extra credit:

I have developed a way to evaluate the function outside of the interval [-pi/4,pi/4] using symmetries. Combining the two Euler's formulas by setting angle t as (t+pi/2), we have e^(i*(t+pi/2)) = i*e^(i*t). Then based on the hint, we have cos(t+pi/2) + i*sin(t+pi/2) = i*cost - sint) since i^2 = -1. Equating the complex parts, we get sin(t+pi/2) = cost. Equating the real parts, we get -sint = cos(t+pi/2). Solving for cost for the real parts, we can get cost = -sin(t-pi/2). Now, equating the cost, we can get sin(t+pi/2) = -sin(t-pi/2). Because sin(-t) = -sint, we get sin(t+pi/2) = sin(pi/2-t). Then, solve for sint, we get sint = sin(pi-t). Based on these formulas and the graphs of both sint and cost, we can see that same-length-intervals that are pi away from interval [-pi/4,pi/4] of sin have the opposite value (-sint) like [3pi/4,5pi/4], and same-length-intervals that are 2pi length away from interval [-pi/4,pi/4] of sin have the same value such as

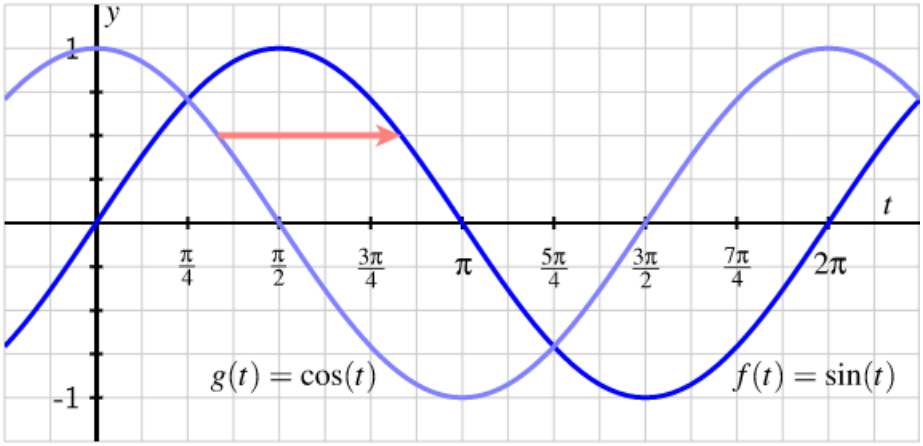
[7pi/4,9pi/4]. Now we have covered half of the whole intervals but leaving gaps like [pi/4,3pi/4]. These gaps can be filled with shifted cos functions since sin(t+pi/2) = cost. As a result, starting from interval [-pi/4,pi/4], we have [-pi/4,pi/4] as sint, [pi/4,3pi/4] as cost, [3pi/4,5pi/4] as -sint, [5pi/4,7pi/4] as -cost which completes a cycle. We can then normalize all the intervals into this cycle using only the values computed in [-pi/4,pi/4] of sin.

My implementation for extra credit is in functions sin4_taylor_extra and sin4_intrin_extra. The cos(t) is computed using Pythagorean identity, sin^2(t)+cos^2(t)=1. I tested my implementation with interval [-2pi, 2pi] and get very small error (6.928125e^-12). Both taylor and intrin functions slow down a lot since there are extra computations to determine which part of the cycle the given t is in, but they are still faster than the reference.

Reference time: 33.5075

Taylor Extra time: 18.5721

Intrin Extra time: 15.2720



3.

I am using CIMS snappy2 machine. It has Intel Xeon E5-2680 2.80 GHz 20 cores CPU.

Number of threads	Sequential scan (s)	Parallel scan (s)
1	0.241076	0.435242
2	0.274208	0.232850
4	0.249587	0.130539
8	0.288354	0.077138
16	0.237537	0.069676
20	0.248797	0.069550

32	0.238356	0.078144
64	0.246834	0.074130

The speedup reaches maximum at about 20 threads.