Homework 3
High Performance Computing

Evan Toler

Apr. 6, 2020

1. **Approximating Special Functions Using Taylor Series & Vectorization.** Special functions like trigonometric functions can be expensive to evaluate on current processor architectures which are optimized for floating-point multiplications and additions. In this assignment, we will try to optimize evaluation of $\sin(x)$ for $x \in [-\pi/4, \pi/4]$ by replacing the builtin scalar function in C/C++ with a vectorized Taylor series approximation,

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots$$

The source file `fast-sin.cpp` in the homework repository contains the following functions to evaluate $\{\sin(x_0), \sin(x_1), \sin(x_2), \sin(x_3)\}$ for different $x_0, \ldots, x_3$:

- `sin4_reference()`: is implemented using the builtin C/C++ function.
- `sin4_taylor()`: evaluates a truncated Taylor series expansion accurate to about 12-digits.
- `sin4_intrin()`: evaluates only the first two terms in the Taylor series expansion (3-digit accuracy) and is vectorized using SSE and AVX intrinsics.
- `sin4_vec()`: evaluates only the first two terms in the Taylor series expansion (3-digit accuracy) and is vectorized using the Vec class.

Your task is to improve the accuracy to 12-digits for any one vectorized version by adding more terms to the Taylor series expansion. Depending on the instruction set supported by the processor you are using, you can choose to do this for either the SSE part of the function `sin4_intrin()` or the AVX part of the function `sin4_intrin()` or for the function `sin4_vec()`.

Solution: We modified both the intrinsic function and vector class implementations by adding additional Taylor series terms. Each method has a different syntax, but after compiling and executing the new code, we see that the errors are identical, at about 10^{-12}.

Intrinsic functions have the fastest runtime on my computer architecture, while the vector class is an order of magnitude slower.

For the extra credit, we can extend the Taylor series to any angle $x \in \mathbb{R}$ as follows. Using the periodicity of sin and the identity, $\sin(-x) = -\sin(x)$, we may restrict ourselves to the interval $x \in [0, 2\pi]$. We describe how to handle each subinterval of this domain.

If, $x \in [0, \pi/4]$, we may apply the Taylor series as usual. If $x \in [\pi/4, \pi/2]$, we use the identity

$$\sin(\theta + \frac{\pi}{2}) = \cos\theta = \sqrt{1 - \sin^2\theta},$$

where $x = \theta + \pi/2$ and $\theta \in [-\pi/4, 0]$ is in the admissible Taylor series range. We use the symmetry of sin about $\pi/2$ (i.e., $\sin(\pi/2 + \theta) = \sin(\pi/2 - \theta)$) to calculate $\sin(x)$ for $x \in [\pi/2, 3\pi/4]$.

For $x \in [3\pi/4, \pi]$ we use the identity $\sin x = \sin(\pi - x)$, so that $\pi - x$ is in the admissible range. Finally, for $x \in [\pi, 2\pi]$, the same identity yields $\sin x = -\sin(x - \pi)$, and we can apply one of the previous cases.

We implement this method for the function `sin4_taylor()` and recover the expected error of 10^{-12} for any input vector $x \in \mathbb{R}^n$. However, I could not figure out how to implement this efficiently using the `Vec` class without checking cases on each vector element individually, and this destroys the speedup of vectorizing the code.

2. **Parallel Scan in OpenMP.** This is an example where the shared memory parallel version of an algorithm requires some thinking beyond parallelizing for-loops. We aim at parallelizing a scan-operation with OpenMP (a serial version is provided in the homework repo). Given a (long) vector/array $v \in \mathbb{R}^n$, a scan outputs another vector/array $w \in \mathbb{R}^n$ of the same size with entries

$$w_k = \sum_{i=1}^{k} v_i \text{ for } k = 1, \dots, n.$$

To parallelize the scan operation with OpenMP using p threads, we split the vector into p parts $[v_{k(j)}, v_{k(j+1)-1}]$, $j = 1, \dots, p$, where $k(1) = 1$ and $k(p+1) = n+1$ of (approximately) equal length. Now, each thread computes the scan locally and in parallel, neglecting the contributions from the other threads. Every but the first local scan thus computes results that are off by a constant, namely the sums obtains by all the threads with lower number. For instance, all the results obtained by the the r-th thread are off by

$$\sum_{i=1}^{k(r)-1} v_i = s_1 + \cdots + s_{r-1}$$

which can easily be computed as the sum of the partial sums $s_1, \dots, s_{r-1}$ computed by threads with numbers smaller than r. This correction can be done in serial.

- Parallelize the provided serial code. Run it with different thread numbers and report the architecture you run it on, the number of cores of the processor and the time it takes.

Solution: I ran this code on my laptop through Courant's "crunchy6" server. My laptop architecture is an Intel Core i7-6500U CPU @ 2.50GHz and the crunchy6 server is a Four AMD Opteron 6272 (2.1 GHz) with 64 cores. The compiler is `g++` version 4.8.5.

We first implemented the parallel scan through a task structure. A serial for loop creates a task for each of the p sections in the array, and then the team of threads performs the local scans in parallel. This implementation performed reasonably well (and with no error), but it ran in about the same time as the serial code.

Instead, our timings in the table below are based on a different approach. We use parallel for loops to iterate over the p sections of the array. It was a nontrivial task to set up the problem in an admissible format for parallel loops, but runtimes were nearly always better than the serial code and the task-based code.

However, the parallelization does not scale particularly strongly for this implementation, as the speedup stalls at around 2x for any number of threads greater than 16 on this computer architecture.

Number of threads	Runtime (s)	Speedup
1 (serial)	8.911483	1.00x
2	5.572102	1.59x
4	5.191279	1.71x
8	4.733514	1.88x
16	4.441658	2.00x
32	4.487883	1.98x
64	4.396989	2.02x

Table 1: Runtime comparisons for the parallel scan. Timings are based on a random array of length $N = 10^9$.