Spring 2022: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 3 (due Apr. 4, 2022)

Anav Prasad, ap7152

Handing in your homework: Hand in your homework as for the previous homework assignment (git repo with Makefile), answering the questions by adding a text or a LaTeX file into your repo. The git repository https://github.com/pehersto/HPCSpring2022 contains the code you can build on for this homework.

1. **Pitch your final project.** Summarize your current plan for the final project in a PDF document and send to me and Melody and Cai via email. We assume you have already talked to us about your project ideas when this homework is due and when you have sent the project description via email. Detail *what* you are planning to do, and with *whom* you will be cooperating. It is important that you call out 4-5 concrete tasks in your project description. We will request frequent updates during the rest of the semester on the progress you are making on these tasks.

2. **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()`.

Extra credit: develop an efficient way to evaluate the function outside of the interval $x \in [-\pi/4, \pi/4]$ using symmetries. Explain your idea in words and implement it for the function `sin4_taylor()` and for any one vectorized version. Hint: $e^{i\theta} = \cos\theta + i\sin\theta$ and $e^{i(\theta+\pi/2)} = ie^{i\theta}$.

Solution:
Output:

```
Reference time: 17.0904
Taylor time:    4.4074      Error: 6.928125e-12
Intrin time:    0.5798      Error: 2.454130e-03
Vector time:    0.9102      Error: 6.928125e-12
```

Extra Credit:
First of all, any angle given beyond the $[-\pi/4, \pi/4]$ range is first reduced to that range by repeatedly using the following transformations:

$$
\begin{aligned}
sin(\theta) &= -cos(\pi/2+\theta)\\
cos(\theta) &= sin(\pi/2+\theta)\\
sin(\theta) &= cos(-\pi/2+\theta)\\
cos(\theta) &= -sin(-\pi/2+\theta)
\end{aligned}
$$

The above is done in the `angle_transform()` function.
Of course, due to the use of the above transformations, we may end up needing to compute the cosine of an angle in lieu of the sine. As such, for that express purpose, the function `cos4_vector()` has been written. Also, some code has been added to `sin4_taylor()` function for the same purpose.
But, since angles are sent in batches of 4 to the vectorized functions and any recomputed angle (which was originally beyond the given $[-\pi/4, \pi/4]$ bound) might require its cosine computation instead of sine, a vectorized version of code seems to be impossible for recomputed angles.

3. **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 $\boldsymbol{v} \in \mathbb{R}^n$, a scan outputs another vector/array $\boldsymbol{w} \in \mathbb{R}^n$ 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 $s_1, \ldots, s_{r-1}$ 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:
Processor: NYU Compute Server (snappy1.cims.nyu.edu)
CPU: Two Intel Xeon E5-2680 (2.80 GHz)
Number of Cores: 20

Time taken:

- Number of threads: 2
  Output:

  ```
  sequential-scan = 0.461476s
  parallel-scan   = 0.324193s
  error = 0
  ```

- Number of threads: 4
  Output:

  ```
  sequential-scan = 0.462048s
  parallel-scan   = 0.220988s
  error = 0
  ```

- Number of threads: 8
  Output:

  ```
  sequential-scan = 0.263884s
  parallel-scan   = 0.131912s
  error = 0
  ```

- Number of threads: 10
  Output:

  ```
  sequential-scan = 0.455564s
  parallel-scan   = 0.106301s
  error = 0
  ```

- Number of threads: 20
  Output:

  ```
  sequential-scan = 0.461901s
  parallel-scan   = 0.075205s
  error = 0
  ```