Spring 2023: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 2, due Mar 5, 2023

Handing in your homework: To hand in your homework, create a Git repository on Github (if for some reason you don't want to use Github, you can use Bitbucket or Gitlab). If you choose your repository to be private, please give me, Cai and Utkarsh read access to the repo (Georg's username on all platforms is `georgst`, Cai's Github username is `caim-d`, and Utkarsh's Github username is `khandu-utkarsh`. If you use another platform than Github, make the repo public or use email invites). The repository should contain the source files, as well as a simple Makefile (we will talk about Makefiles in class). To hand in your homework, please provide the location of your repo through Brightspace (I'll open an assignment there.) Generate a `hw2` directory in this repo, which contains all the source code and a short `.txt` or LaTeX file that answers the questions/reports timings from this assignment. Alternatively, you can hand in a sheet with the answers/timings that also specifies the location of your repo through Brighspace. To check if your code runs, we will type the following commands[1]:

```
git clone YOURPATH/YOURREPO.git
cd YOURREPO/hw2/
make
./MMult1
./val_test01_solved
./val_test02_solved
./fast-sin
```

The git repository https://github.com/NYU-HPC23/homework2.git contains the matrix-matrix multiplication code you can start with, the valgrind examples and the sine example needed for this homework.

1. **Finding Memory bugs using valgrind.** The homework repository contains two simple programs that contain bugs. Use valgrind to find these bugs and fix them.[2] Add a short comment to the code describing what was wrong and how you fixed the problem. Add the solutions to your repository using the naming convention `val_test01_solved.cpp`, `val_test02_solved.cpp`, and use the Makefile to compile the example problems.

2. **Optimizing matrix-matrix multiplication.** In this homework you will optimize the matrix-matrix multiplication code from the last homework using blocking. As background reading, I suggest the Wikipedia entry on matrix tiling [3]. This increases the computational intensity (i.e., the ratio of flops per access to the slow memory) and thus speed up the implementation substantially. The code you can start with, along with further instructions are in the source file `MMult1.cpp`. Specifying what machine you run on, hand in timings for various matrix sizes obtained with the blocked version and the OpenMP version of the code.

3. **Approximating Special Functions Using Taylor Series & Vectorization.** Special functions like trigonometric functions can be expensive to evaluate on current processor architectures which

[1] Since we will partially automize this, make sure this will work for the code you hand in.

[2] When you compile, do not forget to add the `-g` flag, which allows pointing back from the compiled files to specific lines of the source code.

[3] https://en.wikipedia.org/wiki/Loop_nest_optimization

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 4 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()`.

4. **Pipelining and Optimization. Choose one of the following two tasks to work on.** For both problems, specify the processor you are using.

   (a) Follow the steps to optimize an inner product computation using Pipelining from Section 1.7.3 in the text book by V. Eijkhout (read the short corresponding section on pipelining). Make a comparison of the performance you get for the various approaches that are outlined. Try unrolling by factors of 4 (additionally to 2) also. Report the run time for vectors with sizes that do not fit into cache and try to explain the differences.

   (b) Run and try to explain the following codes from the https://github.com/NYU-HPC23/lecture4.git directory, report and try to explain the behavior.

   - `compute.cpp`: Report timings and study the latency of other functions such as sqrt, sin, cos (as outlined at the bottom of the file). Try to run with different compiler optimization flags and report timings.
   - `compute-vec.cpp` Read the code and report timings for the vectorized code. Try to explain the different timings (it's OK if these explanations aren't fully correct).
   - `compute-vec-pipe.cpp` Measure and report the performance of the code for different M as outlined in the comment at the bottom of the file. Summarize your observations.