Homework 3

Question 1

*In this problem, you will utilize the IEEE 754 format and evaluate the performance implications of using floats versus doubles in a computation.*

1. *Compute f(x) = sin (x) using a Taylor series expansion. You select the number of terms you want to compute (but at least 10 terms). Compute sin(x) for 4 different values, though be careful not to use too large a value. Generate two versions of your code, first defining x and sin(x) to use floats (SP), and second, defining them as doubles (DP). Discuss any differences you find in your results for f(x). You should provide an in-depth discussion on the results you get and the reasons for any differences.*

For this part of the assignment the included script “Q1a.c” has been implemented. This script computes the sin(x) of four values wit NTERMS number of terms for both SP and DP and compares the average computing time between the two formats.

In “Q1a.c” we implemented the computation of the factorial with two functions (one for SP and one for DP) and compute the powers of x with the “pow” function included in the “math.h” library. This results in the computation of the entire factorial and power at each iteration of the sum, which is not the most efficient. For this reason, a second version named “Q1a\_eff.c” has been implemented. In this version, two global variables are declared before the loop, one for the factorial and one for the power, and they are updated by two multiplications in each iteration.

As indicated in a discussion with a classmate in Piazza (Zehuan Liu), he outlined that “x86-64 supports both SSE and SSE2. So, in terms of float-point addition, subtraction and multiplication, SP and DP should perform similarly and sometimes DP even out-performs SP. Then the main drag should come from float-point division. Assuming we have N terms, in implementation 1, the number of float-point multiplications is proportion to N^2, while in implementation 2, the number of float-point multiplications is proportional to N. Both implementations have N float-point divisions, i.e., float-point division takes more weight in implementation 2 than in implementation 1”. For this reason we expect the differences between SP and DP to be more discernable in the second implementation, “Q1a\_eff.c”.

Figure 1 shows the average computing time for SP and DP precision, both for the regular and efficient implementations. Each correspond to the average time required to compute the sin(x) for x= 10, 20, 30 and 40 degrees.

On the one hand, we see the effect of only updating the power and factorial terms inside the loop instead of computing entirely each time. On the other hand, however, we observe that despite what was mentioned before, no significant differences are reported between SP and DP.

1. *Explore the benefits of compiling on the Discovery cluster with floating point vector extensions (e.g., AVX). Use the single-precision code from part (a). First run on a node on Discovery that does not support AVX-512. Then run on a node that supports AVX-512 and report on the performance benefits. Additional information is provided on AVX support on Discovery.*

For this section of the assignment the code included in the file “Q1b.c” has been used. We also provide the two assembly files for nodes with and without support for AVX, “Q1b.s” and “Q1b\_avx.s” respectively. Similar to the previous section we analyze the performance of the SP computation of Sin(x) for x = 10, 20, 30 and 40 degrees. Averaged times are displayed in Figure 2. From Figures 2a and 2b we observe that for small number of terms not using AVX instructions performs better than using them, probably due to the overhead added. However, computing more than 1e6 terms results into a consistent performance improve around the 15%. This result is further confirmed when increasing the number of terms linearly, as shown in Figures 2c and 2d.

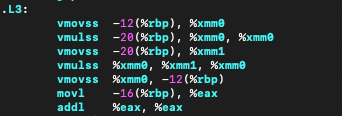
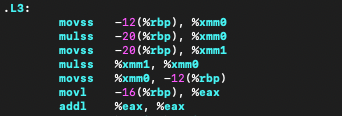
a) b)

c) d)

Figure 2: Performance comparison of SP Sin(x) between with and without AVX-512

1. *Continuing with part (b), generate an assembly listing (using the -S flag) and identify 2 different AVX instructions that the compiler generated, explaining their operation.*

Comparing the two assembly files for “Q1b.c”, “Q1b.s” and “Q1b\_avx.s” respectively, we can see the AVX instructions being used. Figure 3 shows a fraction of the code of both assembly files.

1. b)

Figure 3: Assembly code with AVX support (a) and without id (b)

For example, we observe that the instructions “*movss”* and *“mulss”* , which are in charge of moving or merging scalar single precision floating point values and multiplying scalar single precision floating point values respectively, have been compiled with their AVX counter parts: “*vmovss”* and *“vmulss”.*

1. *Provide both IEEE 754 single and double precision representations for the following numbers: 2.1, 6300, and -1.044.*

The code used to compute the fraction bits is included in file “Q1d.c”. The sign and exponent bits have been computed manually.

|  |  |  |
| --- | --- | --- |
| **Value** | **SP** | **DP** |
| 2.1 | 0100 0000 0000 0110 0110 0110 0110 0110 | 0100 0000 0000 0000 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 |
| 6300 | 0100 0101 1000 0000 0000 0000 0000 0000 | 0100 0000 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 |
| -1.044 | 1011 1111 1000 0101 1010 0001 1100 1010 | 1011 1111 1111 0000 1011 0100 0011 1001 0101 1000 0000 0000 0000 0000 0000 0000 |