PHYS 5210: Computational Methods in Astrophysics, Assignment 1

Set Sept 27th due Oct 12th

Notes: Questions that begin with "Research" may require you to look up auxiliary information outside of the lectures notes in class. To make this process easier, I have ensured that all the information you need can be found rapidly by Internet searches.

Q1. (Research) (a) The Cray supercomputers of the 1970s and 80s were vector-pipeline computers. These machines had sophisticated, and expensive, memory subsystems that allow very high bandwidths (for the technically interested, they are built using SRAM rather than DRAM). Each CPU has vector registers which perform the same function as general registers on scalar CPU. The vector CPU then operates on a pair of registers by pipelining the operations through the floating point calculation unit. For example, suppose you want to add two vectors **A** and **B**. The elements of the vectors (a_i and b_i) will be added together with all the addition operations being pipelined through the floating point unit. Classify the nature of these computers (there are many websites detailing their operation if you wish to know more) under Flynn's taxonomy and explain your answer in detail (HINT: pipelining is the key here and wikipedia is misleading). (**3 marks**)
(b) Modern Intel and AMD CPUs are equipped with "streaming SIMD extensions" known by the acronym SSE. Are these true SIMD operations under the Flynn's taxonomy definition? Explain your answer in the context of how the SSE instructions work. (**2 marks**)

Q2. (a) If $b = b_0 b_1 ... b_{n-1}$ is the n-bit representation of the integer x then we can write x as a sum over the digits of b as follows:

$$x = \sum_k b_k 2^{n-1-k}.$$

I have deliberately left off the limits on k - you need to figure them out. The exponent on the 2 involves minus signs because the left most b_k corresponds to b_0 which must multiply 2^{n-1}. Two's complement representations are derived from the notion of arithmetic on a repeating number line. As was discussed in the lecture, the value of 2^n is mapped back to zero (although the system has effectively overflowed in this case), *i.e.* $1111....1111_2+1=10000....0000_2$. For n-bits of precision, so that the length of the number line is 2^n, we call the resulting arithmetic system 2^n *cyclic arithmetic*.

Show using proof by induction (*i.e.* show that the result is true for $n = 1$, and if it assumed true for $n = k$ then it can be shown to be true for $n = k + 1$) that

$$2^n = 1 + \sum_{i=0}^{n-1} 2^i$$

(**2 marks**).
(b) The above result is the first step in showing why addition of numbers defined using two's complement leads to 2^n cyclic arithmetic. The next step is to give a mathematical function equivalent to the NOT operator, *i.e.* define an $f(b)$ (for any bit b) such that $f(1) = 0$ and $f(0) = 1$. This can be achieved by a very simple operator - what is it? (**1 mark**).
(c) Using the above results show that for any input number x, by applying the operations involved in defining two's complement (the NOT and then add 1) we derive a number that is equivalent to

$-x$ under 2^n cyclic arithmetic. (**4 marks**)

Q3. (a) As we discussed in the lecture, floating point arithmetic is not associative, so that $(A+B)+C \neq A+(B+C)$. Demonstrate this using A=1234.565, B=45.68044, C=0.0003 (you may assume, for simplicity, a decimal floating-point system that carries 7 significant figures, and ensure you use appropriate rounding at each step of the calculations). Also test whether these particular values are distributive or not, *i.e.* if $(A+B) \times C = A \times C + B \times C$ and again show all the steps. (**2 marks**)

(b) What is the largest *integer* that can be represented *exactly* in the IEEE 754 floating-point standard at single precision? By "exactly" I mean a number that if you subtract 1 from it, you will still get the correct answer (*i.e.* there will be no rounding). What about at double precision? (HINT: Think about the spacing of floating point numbers on a number line, also how much precision do you have in the mantissa? Note, the answer isn't simply the largest number that you can write in single precision floating point format.) (**2 marks**)

The IEEE 754 normalization step is usually discussed slightly differently from the method we examined in class. The fraction part $f = 0.d_1d_2...d_t$ is defined by

$$m \times 2^{e+1}/2^t = \pm 2^e(1+f)$$

where we require $f < 1$ as our normalization. The number of bits set aside for storage in single precision will be 1 bit for the sign, 8 bits for the exponent and 23 bits for the fractional part (32 bits total). The logical layout of the bits is

[sign bit][bits of exponent][bits of fraction]

Given this normalization assumption, we can apply the following steps to determine the floating point representation of a decimal number x, in IEEE 754:

- The sign bit is given by 1 if x is negative, zero otherwise
- You can find the exponent, e, by successively testing values of e until $x/2^e = (1+f) = 1.d_1d_2...d_t$
- Once the value of e is found the binary representation will be biased with 127 (*i.e.* the stored value will be the binary representation of e-127)
- Finding the fractional part is a matter of summing different powers of 2^{-n}, until you match the decimal value. The logical storage order is left to right with the leftmost digit correspond to 2^{-1}.

(c) Work out the IEEE 754 single precision representation of -0.1328125_{10}. What is this in hexadecimal? Give both the big endian and little endian representations. (**4 marks**)

Q4. (Research) William Kahan is seen by many as the "grandfather" of floating point analysis. In the 1980s he helped develop a simple program named *paranoia* that tests how well the floating point rounding works on a given computer.

(a) Download the single precision floating point version of paranoia from netlib (`http://www.netlib.org/paranoia/`). Compile this code using gfortran (with no optimization) on the server castor.smu.ca (Stephen is setting-up accounts for you, you may also need to login via ssh from crux). Report whether the compiled program completed without failures (copy and paste the summary ouput statement).
(b) Now turn up the optimization by compiling with gfortran by adding the `-O3` compilation flag. Again report whether the test works or fails.
(c) Now add the `-ffast-math` compilation flag - what happens?
(d) Next instead of using gfortran, use the intel compiler ifort without any optimizations. Report your result.
(e) Turn on optmization on the intel compiler, but you should decide which flag to use yourself by looking at the manual. Report which compiler you used and what happened.
(f) Write a short (one paragraph) discussion of what you think these results imply. (**6 marks total**)