From previous slides

Assignment 3: Achieve Peak

Due: Thursday 09/20/18, Multiplication Factor: 1.0

You should attempt to get as close to theoretical floating point peak as you can using `gcc` on on a 1.503 Ghz UltraSPARC III. To get to the machine log into etl as normal, and then issue `ssh sparc`, and use the name and password given to you originally for etl (if you have lost this password, let me know & I can remind).

Your code can count only floating point computations, but the computations need not be useful (i.e., you do not have to achieve peak doing something like `ddot`). Because your computations may not be used, you will probably need to use the tricks we have discussed to ensure that your results are not optimized away. You should examine the generated assembly to see that all computations in your loop are actually present in the assembly to ensure that this hasn't happened (note that getting better than peak is one of the more obvious signs this has occurred). You should use your knowledge of architecture to choose the appropriate computations to perform, as well as how many will be needed in a instruction window. You should not be afraid of very heavy loop unrollings. The timing framework takes one optional parameter, `mflop`, which is used as in previous timings (you must repeat your basic computational loop enough times to perform at least mflop megaflops).

On `sparc`, I have provided you with a framework for your tuning, which you can get via (after creating going to a local directory to hold the files):

```
cp /home/whaley/HPO_F18/asg3/* .
```

You should examine the full build framework. Note that `make` will build the whole thing, while `make GetMflop.s` will spit out the assembly so you can make sure the compiler isn't removing your flops. My own code is presently achieving just under 95% of peak, but I may be able to improve this further.

All of your changes must be confined to `GetMflop.c`; I have provided you with skeleton routine to start from. You should put your instruction inside the loop, and then set `lflops` to indicate how many flops are performed there.

You will uplo the complete `GetMflop.c` to canvas before midnight by the due date, and I expect it to work with the provided framework. Your baseline grade for this assignment will be the % you achieve compared to the fastest available implementation *when the flops you actually perform are used to correctly calculate your achieved flop rate.*

The grading will be done with the provided compiler flags, but for your understanding you may want to answer: What affect do differing compiler flags have, and is higher optimization always better? What affect does heavy unrolling have on compile time, and does this also vary with flags?