Lab 5 Daniel Hjelm:

# Task 0:

Regularcode (serial (single-threaded) program doing some computation):

real 0m1.597s

user 0m1.297s

sys 0m0.004s

Sleepycode (calls sleep() a few times) :

real 0m6.560s

user 0m1.317s

sys 0m0.008s

Mallocycode (dynamic memory allocations):

real 0m10.191s

user 0m9.064s

sys 0m0.584s

Threadedcode (performs code using two threads):

real 0m1.585s

user 0m2.625s

sys 0m0.012s

We can see that in the Threadedcode we had more user time than real time, this is because it user counts every thread time together. To save everything into arrays are very speed-inefficient, so it should be avoided. Sleep will only affect real time, not user time.

We should focus on user time since that is the time that represent how efficient our code is.

Also, use time around 5 times and save the lowest speed, because noise will effect.

Task 1:

loop\_invariants:

No compiler optimization:

Slow:

real 0m1.565s

user 0m1.321s

sys 0m0.237s

Fast:

real 0m1.716s

user 0m0.937s

sys 0m0.389s

Compiler optimized:

Slow:

real 0m1.121s

user 0m0.442s

sys 0m0.283s

Fast:

real 0m1.146s

user 0m0.442s

sys 0m0.299s

It seems like we want to have loop invariants outside the loops, so if a variable won’t change when we loop, we should calculate it outside the loop. This seems to be fixed with compiler optimization though.

string\_loop:

No compiler optimization:

Slow:

real 0m2.390s

user 0m2.144s

sys 0m0.007s

Fast:

real 0m1.842s

user 0m1.827s

sys 0m0.005s

Compiler optimized:

Slow:

real 0m0.724s

user 0m0.470s

sys 0m0.003s

Fast:

real 0m0.188s

user 0m0.013s

sys 0m0.004s

For string\_loop the fast version used a for-loop instead of a while loop, which increased the speed a bit. This is because it does not have to do the “if” statement in while for every iteration. The optimization with fast is a lot faster.

array\_loop:

No compiler optimization:

Slow:

real 0m1.303s

user 0m1.004s

sys 0m0.007s

Fast:

real 0m1.195s

user 0m1.044s

sys 0m0.004s

Compiler optimized:

Slow:

real 0m0.191s

user 0m0.005s

sys 0m0.003s

Fast:

real 0m0.467s

user 0m0.248s

sys 0m0.003s

For array\_loop the slow version uses an array to store all the elements, which of course would make it slower. With optimization the array method seems to be faster, which feels weird.

Task -2:

If we have if(a && b) we want to put it the least likely Boolean first since if that is false we don’t have to check the other one.

Instead of checking if a variable is 0 <= i <= a (where a is not too large) we can use unsigned variables since a negative unsigned variable will give a very large positive number.

It went just a little bit faster tbh.

Slow:

real 0m0.465s

user 0m0.458s

sys 0m0.003s

Fast:

real 0m0.570s

user 0m0.418s

sys 0m0.003s

Task-3:

Norms:

0: real 0m0.167s

user 0m0.146s

sys 0m0.005s

02: real 0m0.722s

user 0m0.144s

sys 0m0.004s

03:

real 0m0.161s

user 0m0.147s

sys 0m0.005s

0fast:

real 0m0.289s

user 0m0.085s

sys 0m0.004s

DeNorms:

0: real 0m0.355s

user 0m0.145s

sys 0m0.005s

02: real 0m0.191s

user 0m0.141s

sys 0m0.003s

03:

real 0m0.192s

user 0m0.141s

sys 0m0.004s

0fast:

real 0m0.697s

user 0m0.637s

sys 0m0.005s

Wrong number for both norms.c and denorms.c when tiny 1e-50

Task-4:

Memset

Slow:

real 0m0.697s

user 0m0.637s

sys 0m0.005s

Fast:

real 0m0.227s

user 0m0.012s

sys 0m0.004s

Lookup

Slow:

real 0m0.764s

user 0m0.490s

sys 0m0.005s

Fast:

real 0m0.377s

user 0m0.231s

sys 0m0.003s

So lookup tables can be used to increase the performance. It is good to have them in an array since we can find the value with computational complexity O(1).

Strength\_functions and math\_functions look at code.

Task-5: