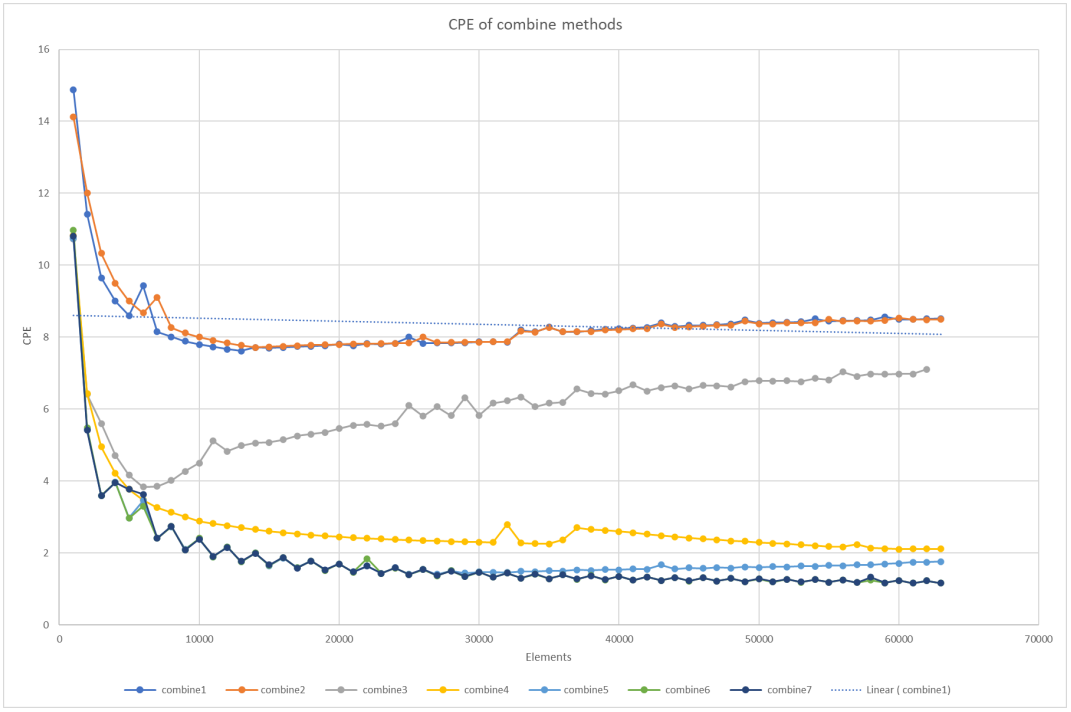
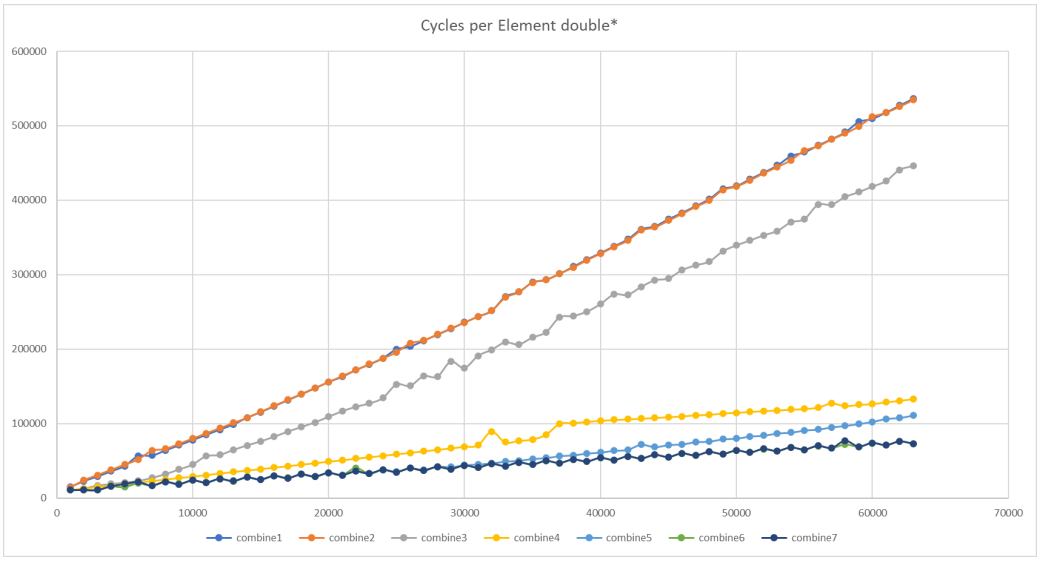
Evan Lang
Yang Lu
EC 527: Lab 2

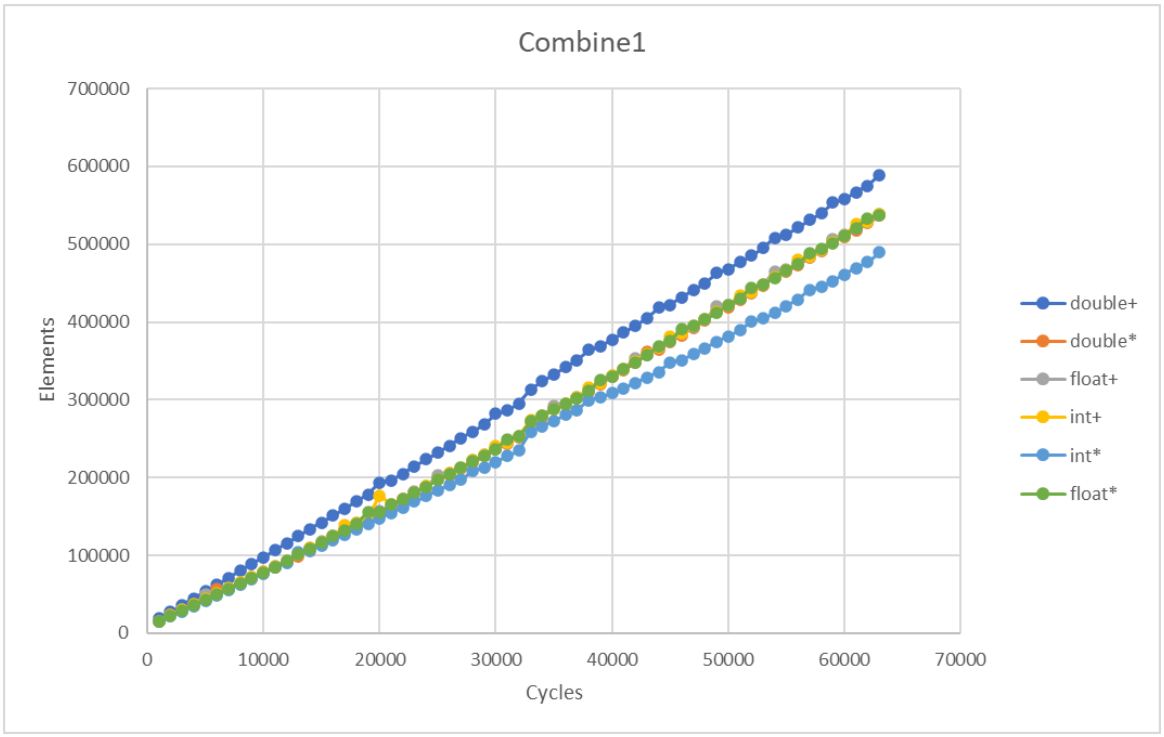
1. *Tables located in lab2q1a.xlsx*

a. 

The CPE graphed over elements for double*

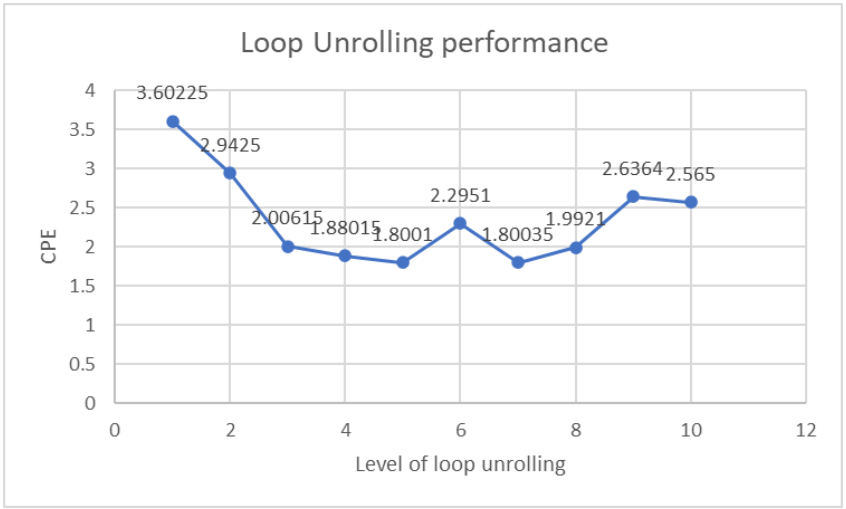


Cycles graphed over elements for double. Displays the linear scaling discussed in the lab*



*Combine 1 linear graphs for CPE. Int * had the best performance, and generally the * operations performed better than the + operations. The only outlier is float, in which both operations had the same result. The performance per shown from worst to best as double -> float -> int. The more optimized combine methods like 4-7 displayed similar performance no matter the type and operation and so were not included. Tables with their performance can be found in Lab1Q1a.xlsx*

b.



Loop unrolling was performed on double* data type and operand. The loop unrolling increases performance up until an unroll factor of 8, with 6 being a significant outlier. Starting at an unroll factor of 8 our performance begins to significantly degrade. This may be due to the unroll factor being so great it now

creates a guaranteed memory miss at least once per loop call (now that the unrolled loop is asking for 8 different elements at once).

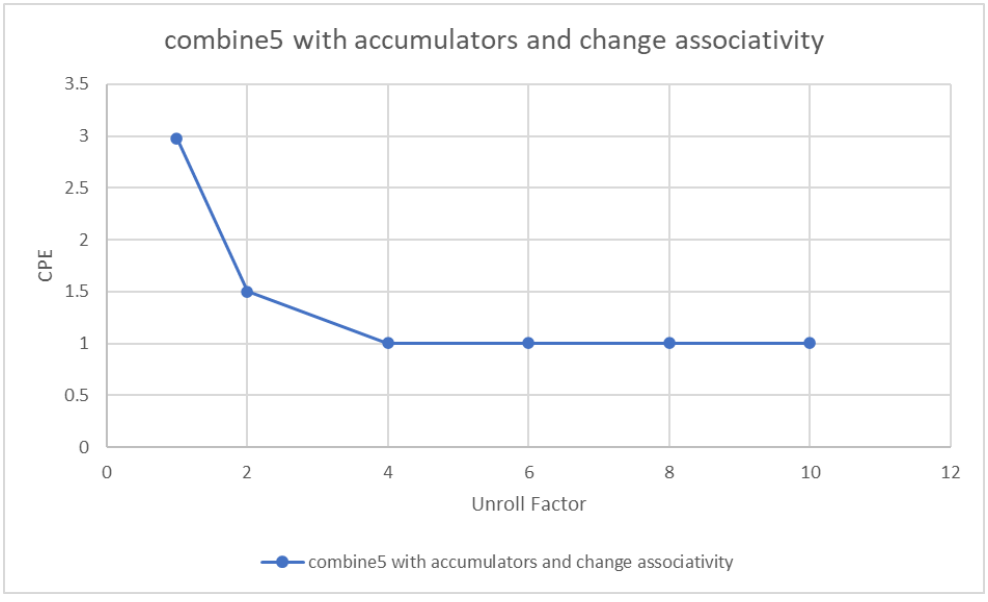
```
void combine5(array_ptr v, data_t *dest)
{
  long int i;
  long int length = get_array_length(v);
  long int limit = length - 1;
  data_t *data = get_array_start(v);
  data_t acc = IDENT;

  /* Combine two elements at a time */
  for (i = 0; i < limit; i+=10) {
    acc = (((((((((acc OP data[i]) OP data[i+1]) OP data[i+2]) OP data[i+3]) OP data[i+4]) OP data[i+5]) OP data[i+6]) OP data[i+7]) OP data[i+8]) OP data[i+9];
  }

  /* Finish remaining elements */
  for (; i < length; i++) {
    acc = acc OP data[i];
  }
  *dest = acc;
}
```

combine5 with a unroll factor of 10. This code is not preserved due to part c.

c.

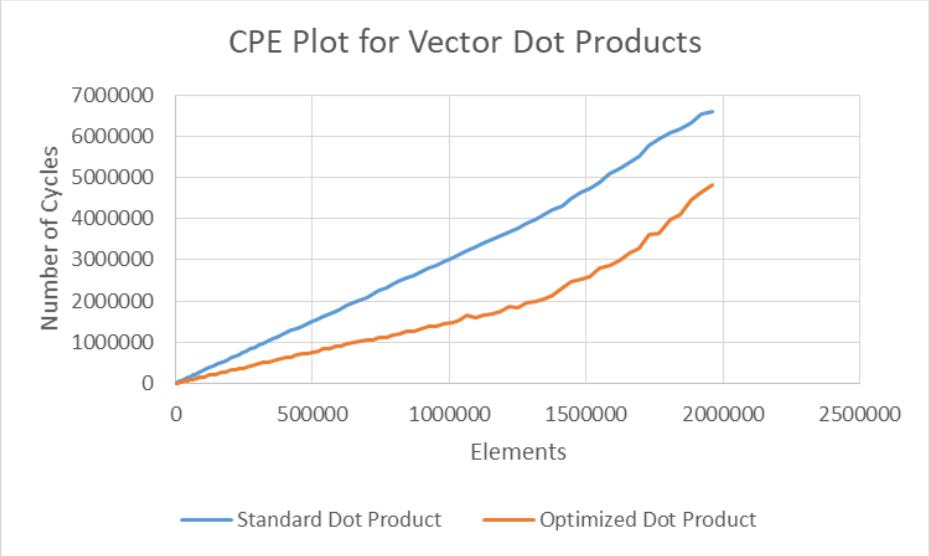
```
void combine5(array_ptr v, data_t *dest)
{
  long int i;
  long int length = get_array_length(v);
  long int limit = length - 1;
  data_t *data = get_array_start(v);
  data_t acc = IDENT;
  data_t acc1 = IDENT;

  /* Combine two elements at a time */
  for (i = 0; i < limit; i+=6) {
    acc = acc OP (data[i] OP data[i+1] OP data[i+4]);
    acc1 =  acc1 OP (data[i+2] OP data[i+3] OP data[i+5]);

  }

  /* Finish remaining elements */
  for (; i < length; i++) {
    acc = acc OP data[i];
  }
  *dest = acc OP acc1;
}
```

Combine5 with an unroll factor of six, 2 accumulators, and change associativity



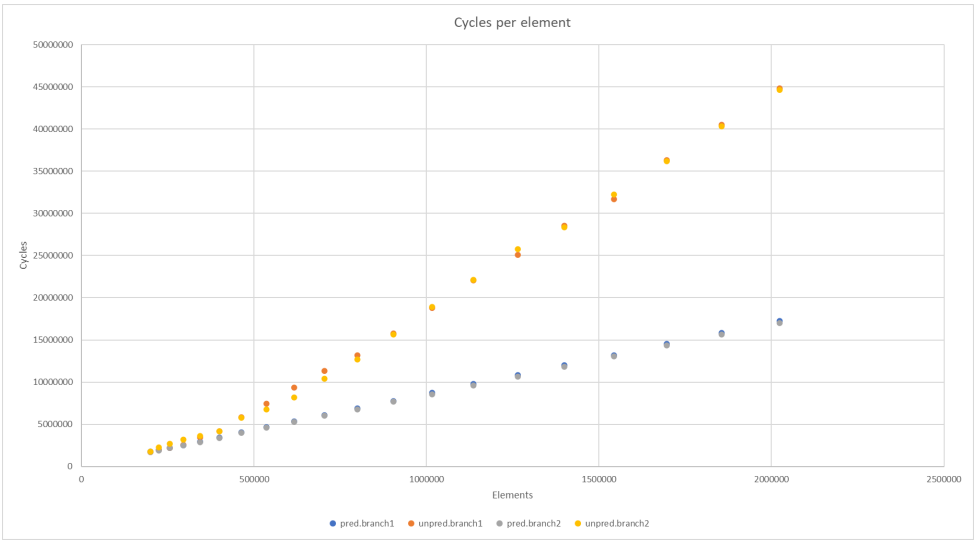
Unroll factor	*Cycles*	*CPE*
1	*19843838*	*2.978661*
2	*9984554*	*1.498732*
4	*6685588*	*1.003541*
6	*6684931*	*1.003442*
8	*6689020*	*1.004056*
10	*6685622*	*1.003546*

2. *Tables located in lab2q2.xlsx*



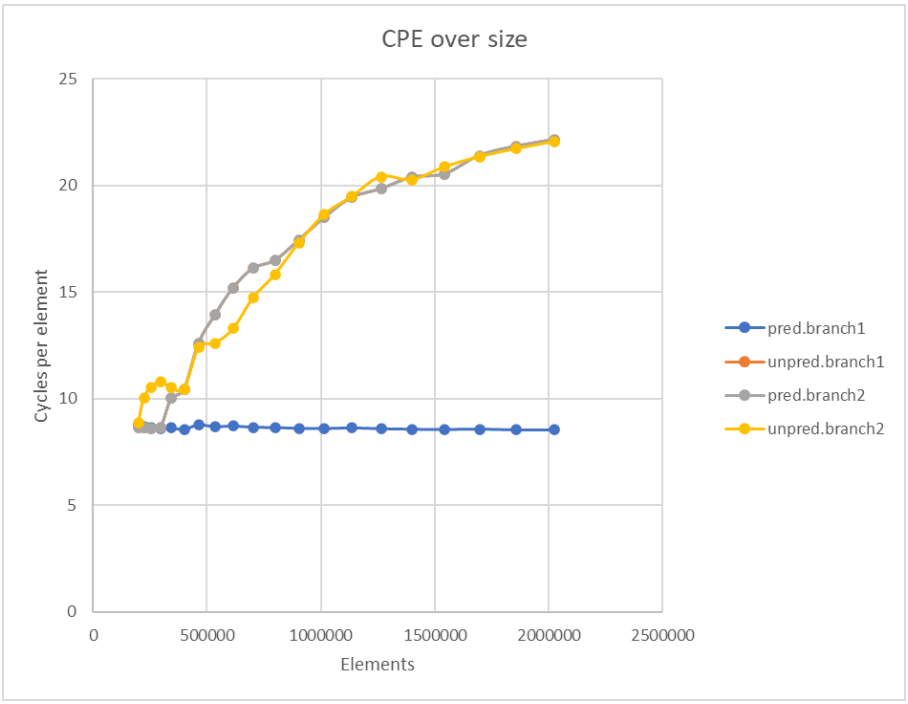
Loop unrolling and parallelism was applied to gain the optimizations shown in the graph above. Specifically a loop unrolling factor of 5 was applied concurrently with 5 nested accumulaters.

3. *3a was performed using an -O0 optimizer in order to prevent the compiler from optimizing out the effects of the unpredictable data. Using the -01 optimizer as directed by the lab results in identical data no matter how unpredictable the data in the array is. CPE, graphs and tables from the -O1 runs are not presented as the data adds no value. The results of the -O1 runs and -O0 runs can be found in Lab2Q3.xlsx*



a.

The number of cycles are the y-axis and the number of elements is the x-axis. As the size of the arrays increases the cycles also increase. At the largest size the performance impact of branch misprediction leads to almost a 3x growth in cycles



Both branches handled the predictable data with nearly the same performance. Below is the table of CPE values as size increases

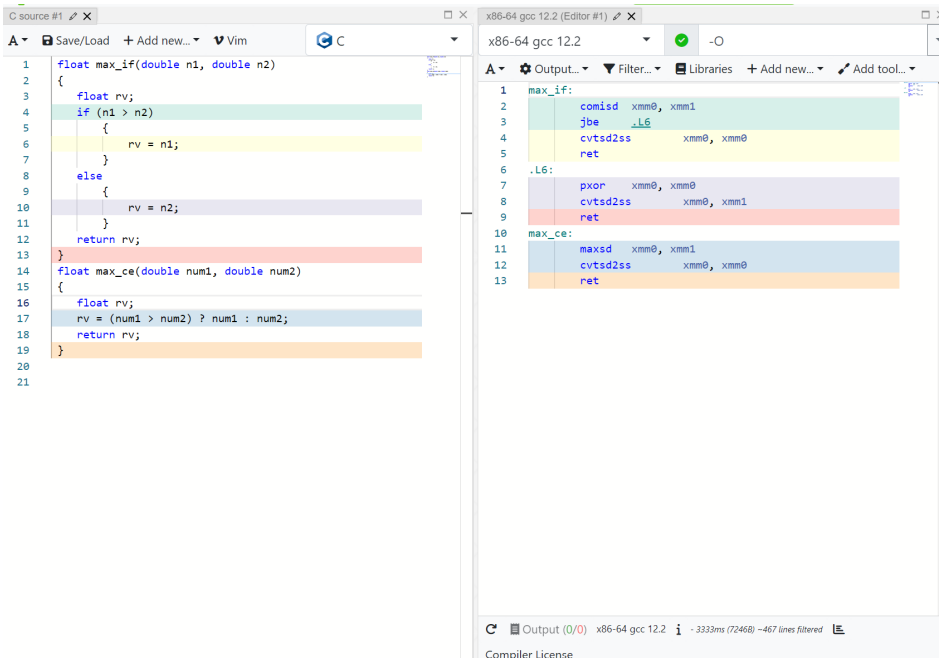
pred.branch1	*unpred.branch1*	*pred.branch2*	*unpred.branch2*
8.77626	*8.62549*	*8.66316*	*8.882745*
8.663397	*8.630228*	*8.530339*	*10.03227*
8.652375	*8.600531*	*8.558332*	*10.54392*

8.593459	*8.628679*	*8.485807*	*10.78164*
8.632346	*10.00975*	*8.398163*	*10.52563*
8.547785	*10.44383*	*8.48305*	*10.43523*
8.780653	*12.60616*	*8.624897*	*12.40099*
8.686903	*13.93688*	*8.63275*	*12.60651*
8.721529	*15.20504*	*8.5915*	*13.28181*
8.659864	*16.12193*	*8.544466*	*14.74241*
8.647275	*16.49142*	*8.501499*	*15.83212*
8.607058	*17.44146*	*8.507779*	*17.30816*
8.595599	*18.49648*	*8.434299*	*18.63784*
8.630754	*19.43057*	*8.459937*	*19.478*
8.589881	*19.83601*	*8.442852*	*20.40892*
8.564941	*20.37876*	*8.458354*	*20.26398*
8.549427	*20.53541*	*8.464001*	*20.87179*
8.572154	*21.3954*	*8.467419*	*21.34535*
8.53158	*21.83234*	*8.444842*	*21.71224*
8.535901	*22.14153*	*8.397125*	*22.05781*

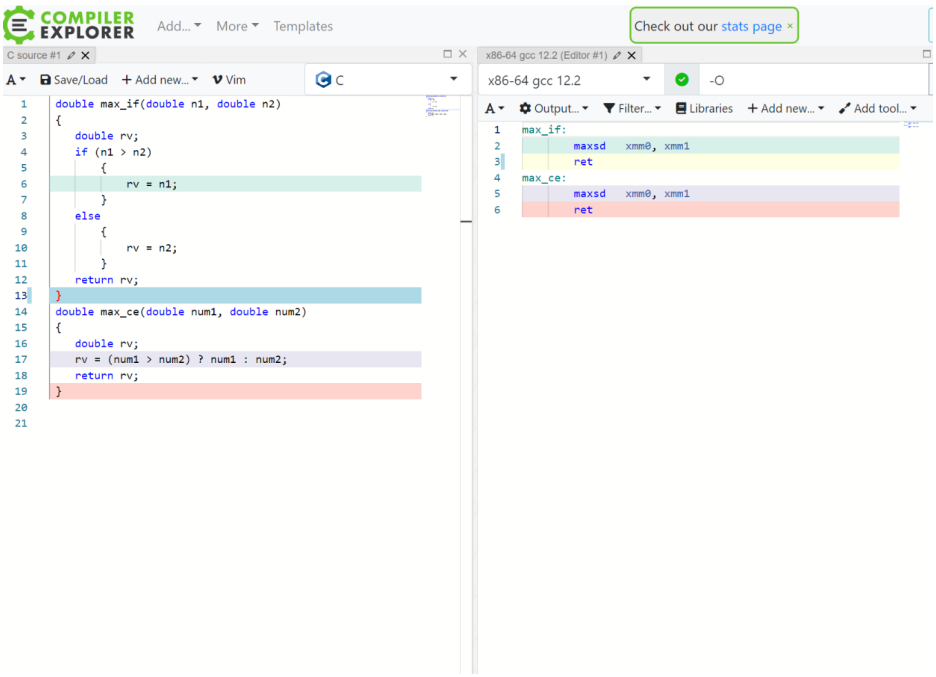
```
/* initialize an array with a "predictable" pattern */
int init_array_pred(array_ptr v, long int len)
{
  long int i;

  if (len > 0) {
    v->len = len;
    for (i = 0; i < len; i++) {
      v->data[i] =  1; /* Modify this line!! */
    }
    return 1;
  } else {
    return 0;
  }
}

//  double quasi_random = 1; /* global */

/* initialize an array with an "unpredictable" pattern */
int init_array_unpred(array_ptr v, long int len)
{
  long int i;
  double ran1,ran2;
  if (len > 0) {
    v->len = len;
    for (i = 0; i < len; i++) {
    ran1 = fRand(0,1000);
    ran2 = fRand(3,1000);
    if(ran1 > ran2){
    v->data[i] =  fRand(0,10);
     }
     else{
       ran1 = fRand(3,1000);
       ran2 = fRand(3,1000);
       if(ran2 > ran1){
     v->data[i] = fRand(0,3);
         }
     else{
       v->data[i] = fRand(7,1000000000000);
     }
     }


    }
    return 1;
   } else {
    return 0;
  }
}

data_t *get_array_start(array_ptr v)
{
  return v->data;
}
```

The altered code. The unpredictable data was created by comparing 2 random numbers. These comparisons resulted in 3 different possible numbers being added. Either a number between 0 and 3, a number between 0 and 10, or a random very large number. The randomness of this data led to a large amount of misprediction by the cpu, as there was no way to predict if the values stored in data1 would be less than, equal to, or greater than those stored in data2.

b.



Shown above is the code compiled with no changes. The max_if function utalizes a loop called out by "jbe" while the max_ce function does not use a loop but rather the maxsd instruction.



Shown above here is the code compiled when the "float" declaration is changed to a "double". Using an all double variable c code leads to both functions having the maxsd instruction which allows for comparisons between 2 registers without the need for a conditional branch instruction.

4.

a. This lab took about 6 hours.
b. No portion of this lab was unreasonable. However, problem 3a could have been clarified a bit better. The results that problem 3a asks for is not possible when a compiler optimization of -O1 is used as the compiler is changing the assembly instructions to increase efficiency. However, by using a compiler optimization of -O0 the issue mentioned above can be resolved.
c. All skills needed for this lab was covered during lecture.
d. The only issue with this lab was mentioned in 4b.