Varsha Singh
EC527, Herbordt
Lab 2
Due February 12th

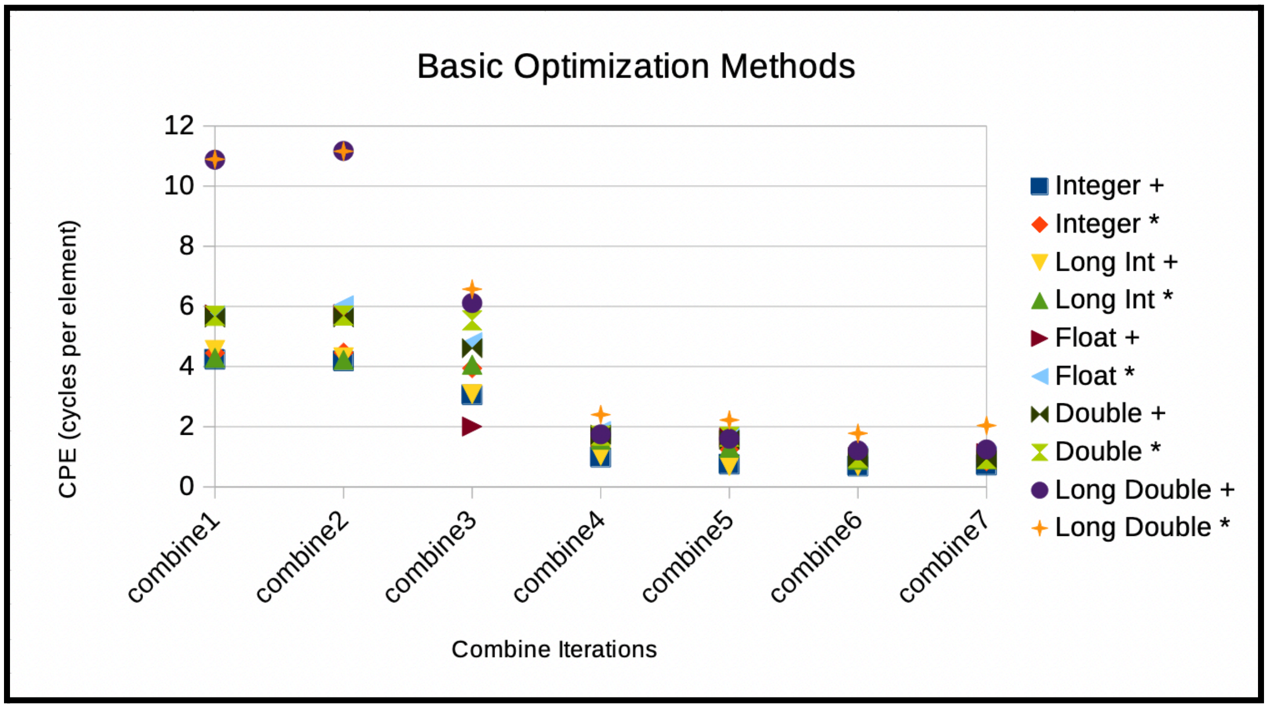
Lab 2: EC527

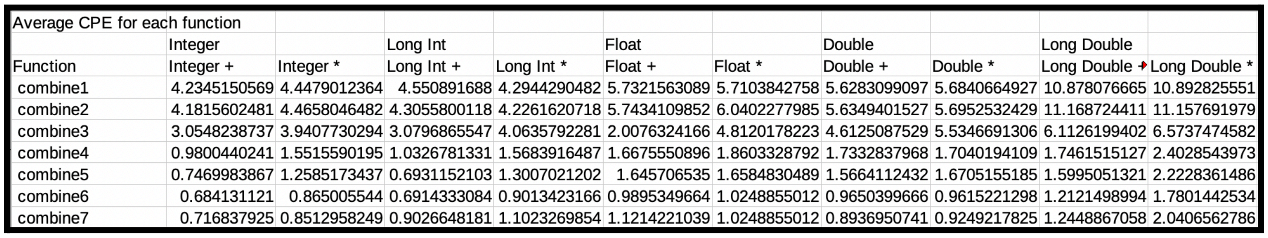
Varsha Singh

Part 1: Experiment with basic optimization methods as presented in B&O 5.4-5.10

1a. Compile (using -O1, but also try -O0 once if you're curious) and run the code.
Each of the main test loops chooses the vector lengths to test by evaluating the polynomial Ax2+Bx+C for values of x from 0 to NUM_TESTS-1. Adjust the coefficients A, B and C to make it cover a larger range of sizes; you can also increase NUM_TESTS to get more data points. However, make sure the largest size will fit in the level 1 cache (32K bytes). In other words, when x=NUM_TESTS, make sure that (Ax2+Bx+C) × sizeof(data_t) is less than 32K bytes.
Find the CPEs of the seven options (combine1 through combine7). Vary the data types and operations to see which of these change CPE.
Make some tables and graphs, like the examples seen in the textbook (e.g., section 5.9.1, pages 514 and 515). You do not need to do all combinations, just pick some you think are important. If you see differences from the result shown in the book, tell what you see and try to explain the differences.

My results are generally significantly better than those in the textbook, there is only a feasible comparison to combine7. I think that the main contributing factor as to why this happens is that there are likely major hardware differences from when this book was created compared to the computers used in this class. The processor architecture will have varied, and there can be more optimizations in a modern CPU that will lead to better benchmarking performances. Additionally, there is a possibility that the power-saving mode for which we have the wakeup_delay parameter to wait out the lowered frequency, could have not been accounted for in the textbook. I believe that the likelihood of this occurring is significantly low, given the reviewing process for textbooks, but this can help explain the large differences as well. Additionally, a TA (Sora) brought up during office hours that there was an issue with the calculation of elements and number of cycles, as both are multiplied by the macro, OUTER_LOOP. While the trend of the line shall remain the same, the factor of values are actually 2000x larger.





Average CPE for each function										
	Integer		Long Int		Float		Double		Long Double	
Function	Integer +	Integer *	Long Int +	Long Int *	Float +	Float *	Double +	Double *	Long Double +	Long Double *
combine1	4.2345150569	4.4479012364	4.550891688	4.2944290482	5.7321563089	5.7103842758	5.6283099097	5.6840664927	10.878076665	10.892825551
combine2	4.1815602481	4.4658046482	4.3055800118	4.2261620718	5.7434109852	6.0402277985	5.6349401527	5.6952532429	11.168724411	11.157691979
combine3	3.0548238737	3.9407730294	3.0796865547	4.0635792281	2.0076324166	4.8120178223	4.6125087529	5.5346691306	6.1126199402	6.5737474582
combine4	0.9800440241	1.5515590195	1.0326781331	1.5683916487	1.6675550896	1.8603328792	1.7332837968	1.7040194109	1.7461515127	2.4028543973
combine5	0.7469983867	1.2585173437	0.6931152103	1.3007021202	1.645706535	1.6584830489	1.5664112432	1.6705155185	1.5995051321	2.2228361486
combine6	0.684131121	0.865005544	0.6914333084	0.9013423166	0.9895349664	1.0248855012	0.9650399666	0.9615221298	1.2121498994	1.7801442534
combine7	0.716837925	0.8512958249	0.9026648181	1.1023269854	1.1214221039	1.0248855012	0.8936950741	0.9249217825	1.2448867058	2.0406562786

1b. In this part you only need to generate data for data type double and operation + (addition). The existing test code does unrolling, but only by a factor of 2. Task: modify the code to unroll by factors up to 10 using the method seen in combine5. Graph your CPE results as shown in Figure 5.22. The plain "non-unrolled" loop (combine4) can be plotted as "unroll factor = 1". (You don't need to do all of the intermediate values (3-10).) Does performance get better or worse as the unrolling factor increases? Why might this happen?

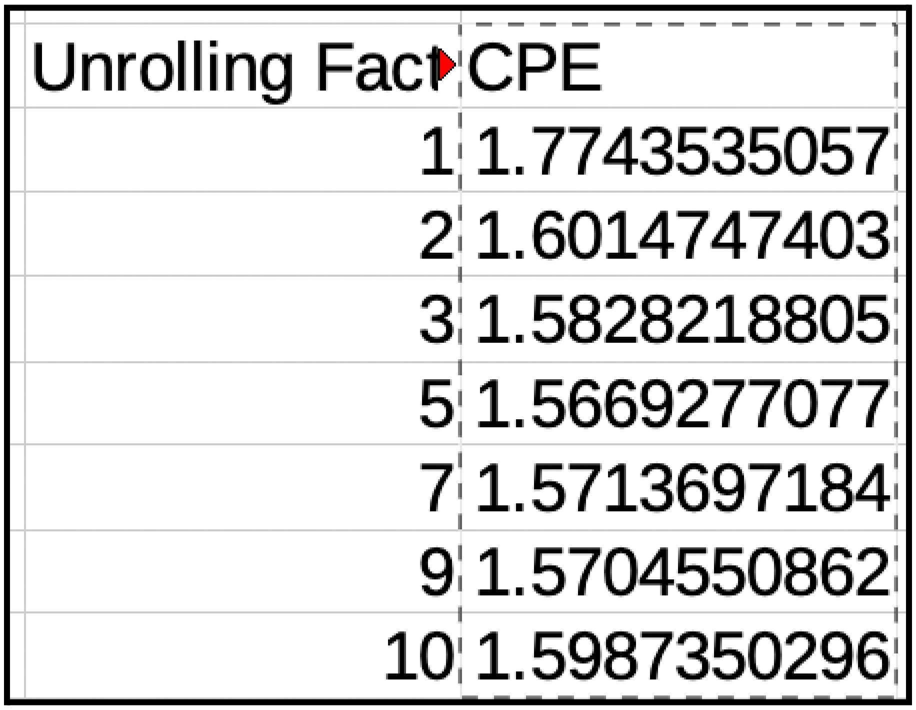
I tested the following unroll values and calculated the average CPE for each method:

Unit for the following: cycles per element

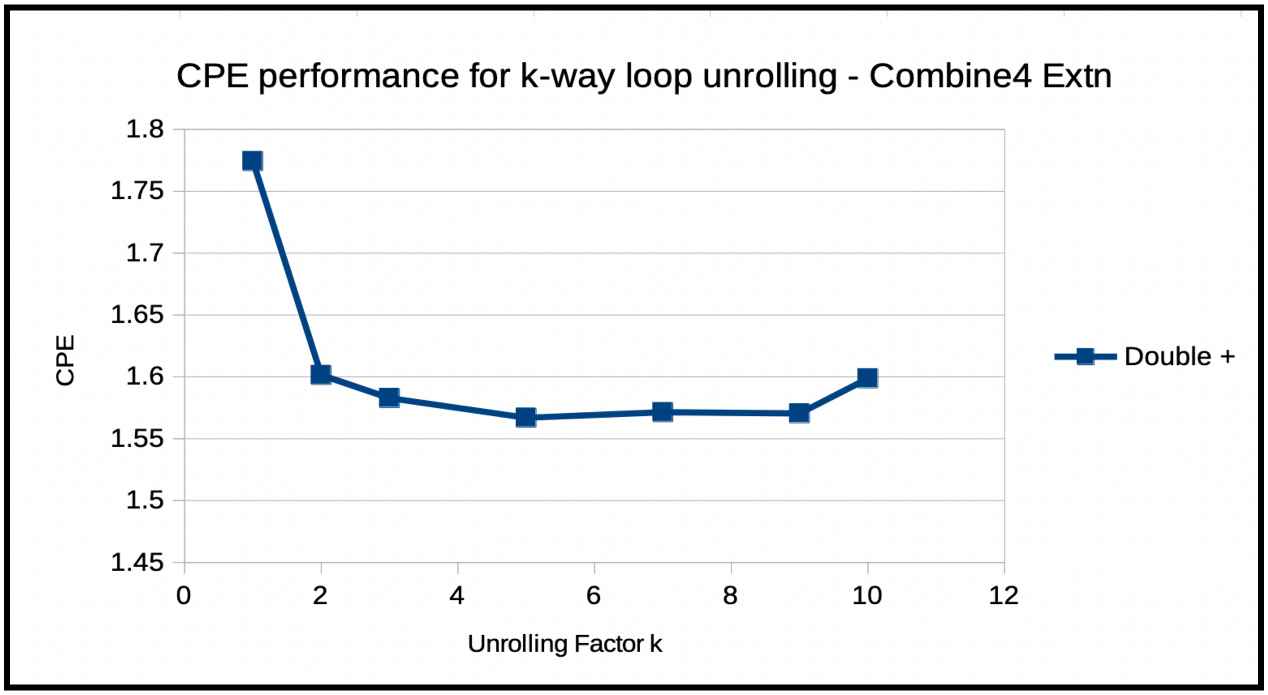
- 1 (combine 4): 1.77435350569681

- 2 (combine 5): 1.60147474032506
- 3: 1.58282188054957
- 5: 1.56692770771267
- 7: 1.57136971838641
- 9: 1.5704550862352
- 10: 1.59873502956717

Based on these values, it would seem that an unroll factor of 5 provides the most optimal CPE, with the CPE increasing as you deviate further from this value in either direction. I believe that this might be the case due to a balance between unrolling, as well as not having too large of a 'lookahead' with the unrolling such that it won't result in a predictive branch failing, and then spending more time accounting for the last few elements with a for loop with an unroll factor of 1. Having this value be too large will result in more unroll factors of one following, and too few will have more loops. Regardless, in this experiment it became clear that the unroll factor of 1 is the worst performing version, and for this instance, unrolling helps optimize the code.

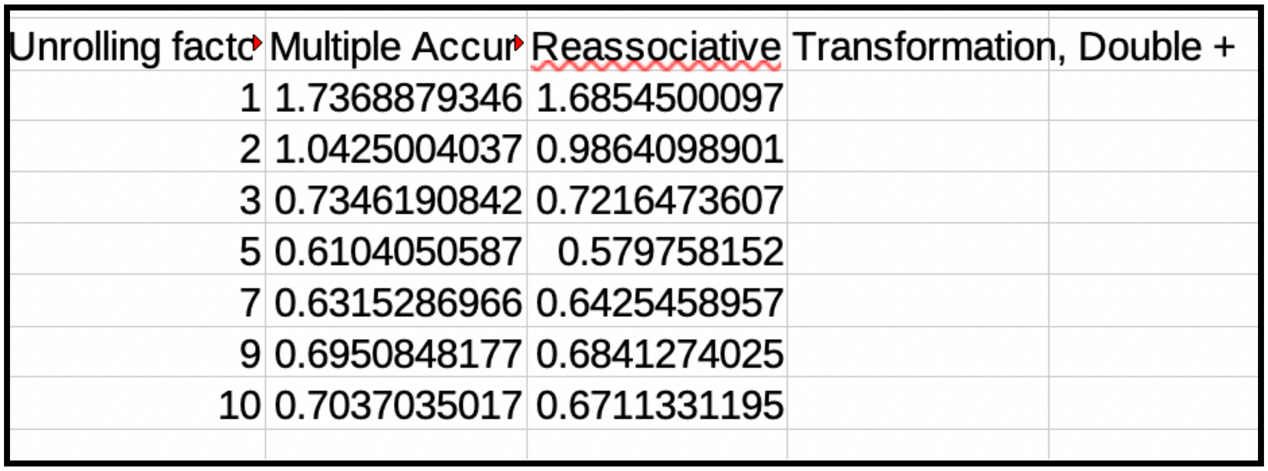


Unrolling Fact	CPE
1	1.7743535057
2	1.6014747403
3	1.5828218805
5	1.5669277077
7	1.5713697184
9	1.5704550862
10	1.5987350296

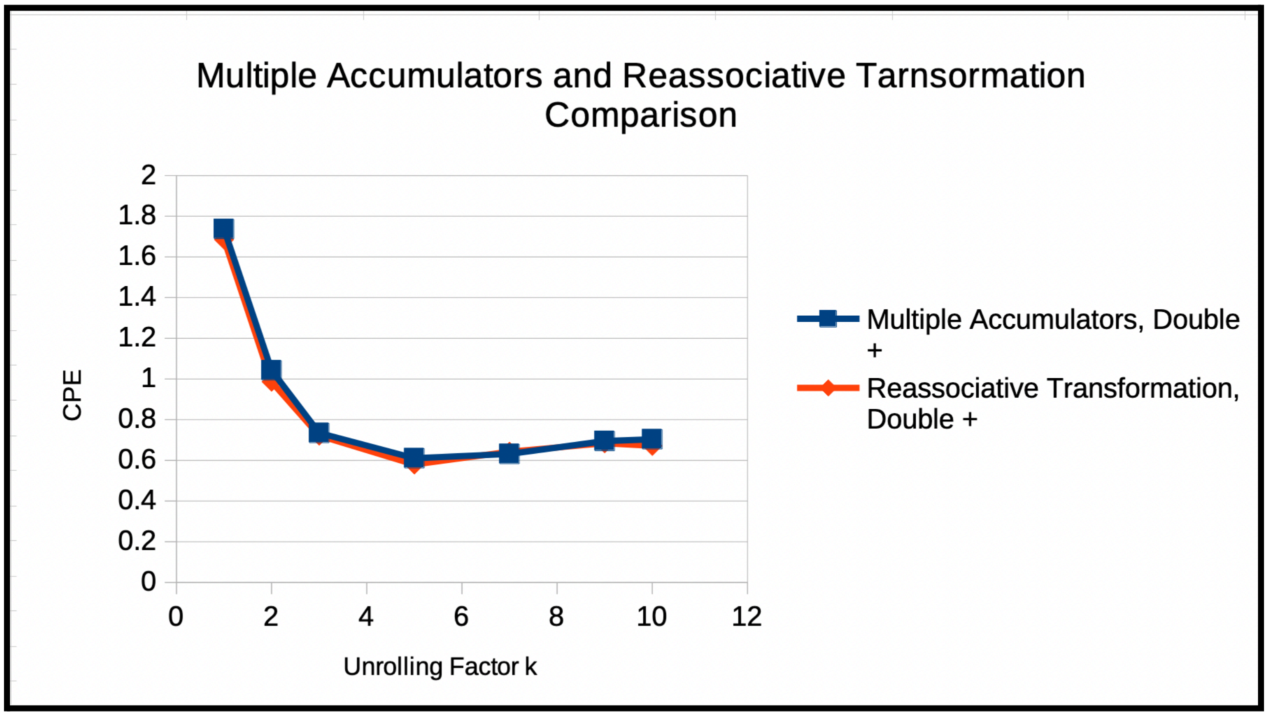


1c. In this part (again) you only need to generate data for data type double and operation + (addition). Task: Add versions of the two parallelization methods (multiple accumulators seen in combine6 and reassociative transformation seen in combine7) to the code you modified for part 1b. Plot your results and give a graph of CPE versus unroll factor.

The way I approached this problem was to take my version of the Combine5 code that I expanded for k-unrolling and make two versions of derivatives, one utilizing Combine6 and the other utilizing Combine7. The Code for the Combine5 originals were overwritten for 5 & 6, but I generated new code to write Combine7. As a result, there are no more Combine5 factors of loop unrolling from part 1b.



Unrolling facto	Multiple Accur	Reassociative Transformation, Double +		
1	1.7368879346	1.6854500097		
2	1.0425004037	0.9864098901		
3	0.7346190842	0.7216473607		
5	0.6104050587	0.579758152		
7	0.6315286966	0.6425458957		
9	0.6950848177	0.6841274025		
10	0.7037035017	0.6711331195		

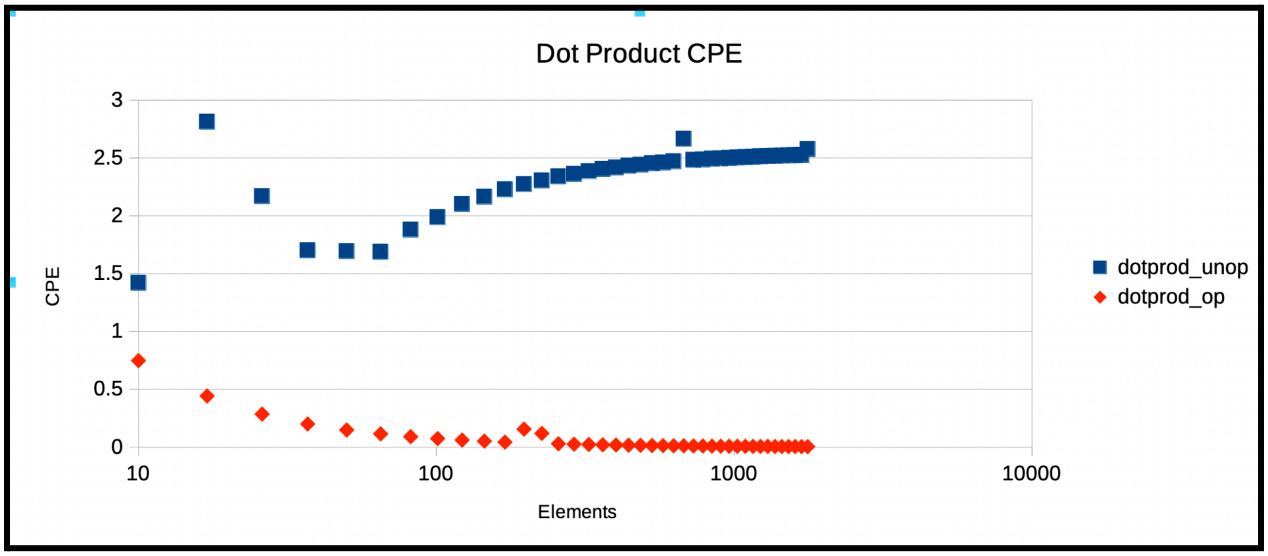


Part 2: Apply basic methods to dot product

By using the the methods in part 1b and 1c, I was able to develop a more optimized version of dot product

- The simplest version I could come up with used the get_array_length function to check for bounds, and did not store local accumulators for the data, except for one with acc. This could be made less optimal by updating *dest at each iteration, but this was not the most simplest and straightforward solution I thought of.

- My optimized version had loop unrolling 5 times and 5 accumulators. I based this off of the CPE graph for the previous part, finding that a unrolling factor of 5 seemed to be the most ideal. I created the 5 accumulators and added them at each loop iteration – the upper bound is calculated once before the loop. Afterwards, I account for the remaining elements that could be at the end of the loop, and add everything to the *dest. I used associativity to add together the data array values first before adding to the accumulator variable.



Part 3: Force and evaluate conditional moves

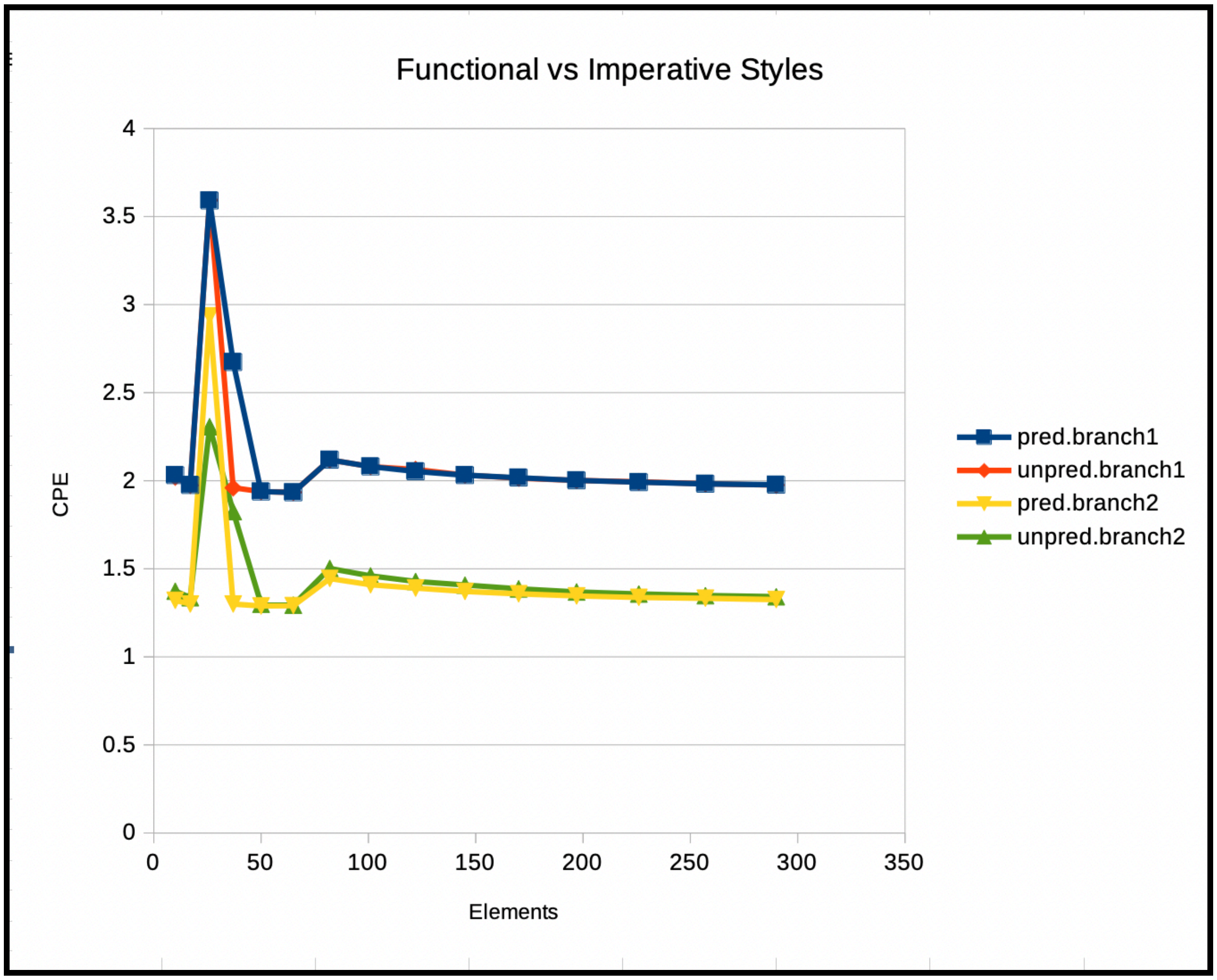
I used the same coefficients from part 1 (1, 6, 10, with 15 tests), but this was optimized for fitting a double array into cache. A float is 4 bytes, whereas a double is 8 bytes – these parameters took about 15 KiB of space, and the calculation for this is in *test_combine1-7.c*. I filled init_array_pred with a standard value, and filled init_array_unpred with random values, with the helper function fRand.

It seems like branch1 performs more or less about the same, so it is likely that the compiler is too smart. When using the godbolt compiler explorer, I saw the following:

```
max_if:
        ucomisd xmm0, xmm1
        jbe     .L6
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
.L6:
        unpcklpd        xmm1, xmm1
        cvtpd2ps        xmm0, xmm1
        ret
max_ce:
        maxsd   xmm0, xmm1
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
```

The first function uses a branch instruction (*jbe .L6*), whereas the second function uses (*maxsd*) which is also known as *return maximum scalar double precision*. This does not have predictive branches and will not have to negate previous instructions for an incorrect prediction – like max_if will have to do. When I changed all of the floats to doubles, the code became very concise and the same assembly language for both functions.

For my experiment, branch2 performed better than branch1 by around 68%. This can be attributed to the way the predictable array is initialized. All of the data in the array is set to 1, so the conditional statement will never actually be true. By choosing the greater of the two and setting it to the value, like done in branch2, there will be no attempt at prediction. Therefore, it makes sense that the CPE for branch2 is lower than the CPE for branch1.



Part 4: Quality Control

1. This lab took a lot less time than before, I think it took around 8 hours (a lot of this time was spent reading and experimenting).
2. I do not think any part took an unreasonable amount of time for what it is trying to accomplish, but part 1 definitely took me longer than the other two parts.
3. I do not believe there are any skills missing, but I do think that a better understanding of how the compiler works as well as how branches 'fall through'/'are taken' would've made part 3 made more sense.
4. Part 3 asks us to modify *init_vector_pred* and *init_vector_unpred*, but the code has *init_array_pred* and *init_array_unpred*. Additionally, when discussing with a TA (Sora) about the output, he mentioned that the size and cycles calculations are multiplied by the OUTER_LOOP macro, which is not accurate for our experimentation. This will produce the same general loop of the graph, but the number of elements is scaled to an unscaled

CPE. Apart from these, the rest of the complications come from having a faster and smarter processor.