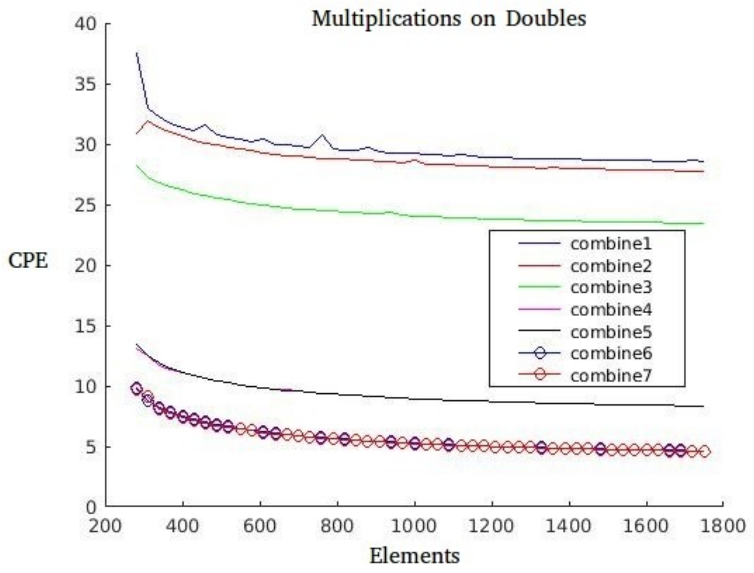
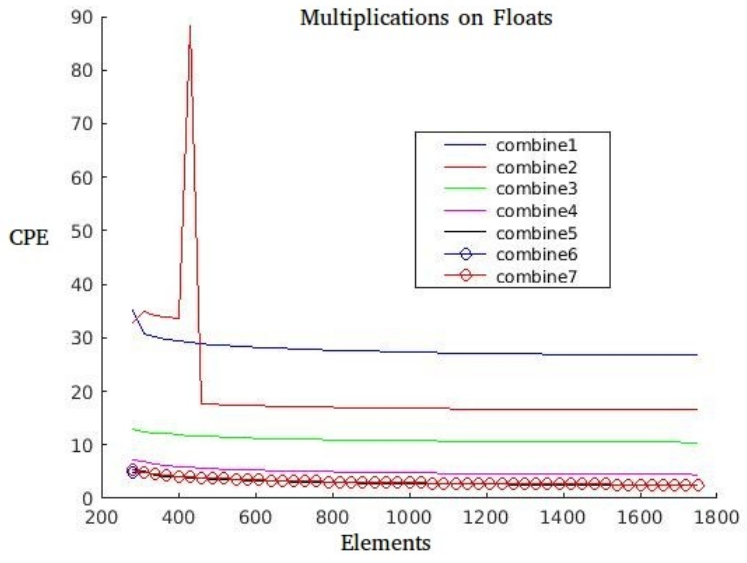
John S. Burke
Robert Munafo
EC527 – Professor Herbordt

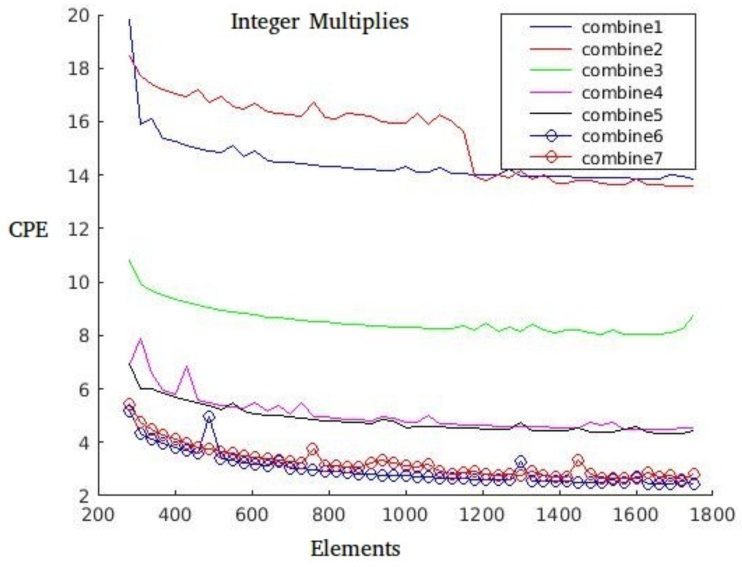
Lab 2

Part 1 (basic optimisation methods)

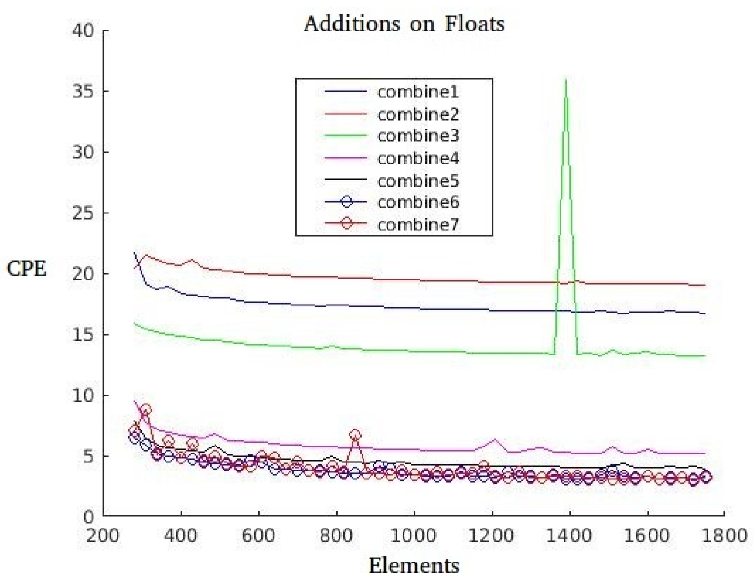
1.a. CPE of various types and operations

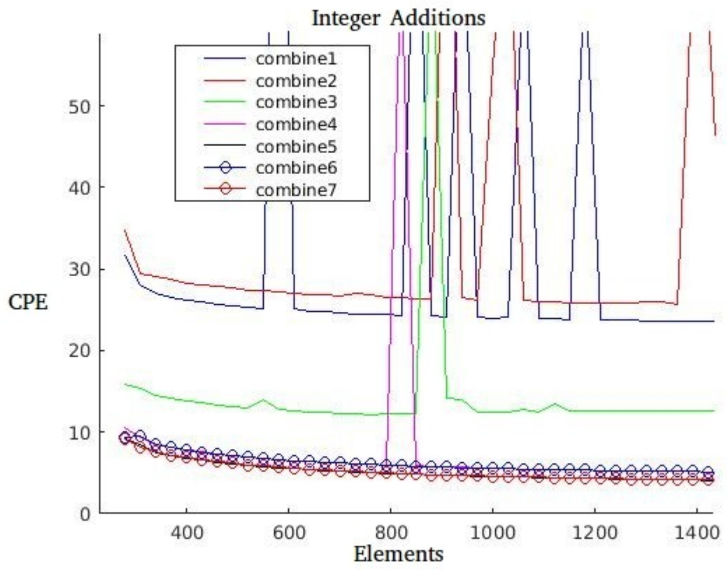






The three plots here show the CPE of the various combine methods over multiplication with various data types. Addition is done on the next page along with the overall CPE tableau.

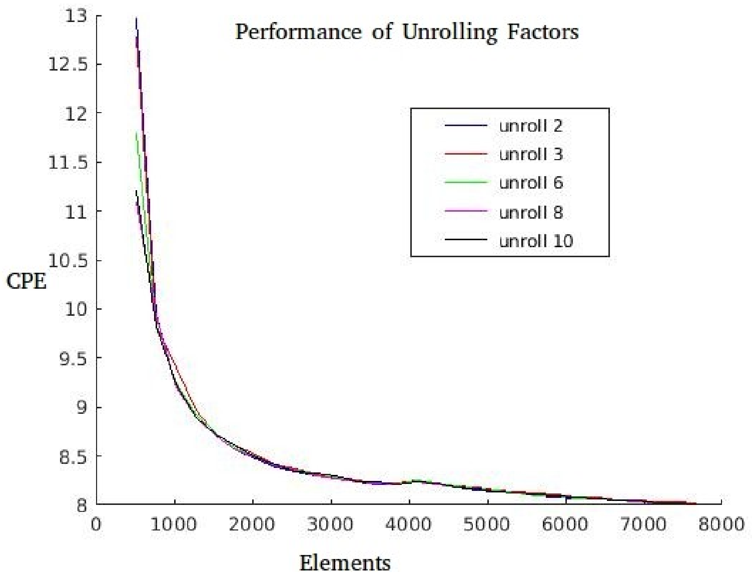




		Integer	Integer	FP	FP	FP
Function	Method	+	*	+	F*	D*
Combine 1	Max Data Abstraction	21.45	25.72	27.11	16.41	47.12
Combine 2	Simple Code Motion	23.34	29.03	30.87	18.99	27.78
Combine 3	Direct Access, Reduce Procedure Calls	11.74	14.87	21.43	19.55	23.45
Combine 4	Basic Accumulator, remove mem. Refs.	4.53	8.27	8.29	8.28	8.34
Combine 5	Unroll loop by 2	3.59	8.40	6.39	4.57	8.34
Combine 6	Unroll by 2, 2 accumulators	4.53	4.53	4.84	4.71	4.62
Combine 7	Unroll by 2, Reassociate	3.61	4.78	4.95	4.71	4.61

Our numbers are slightly higher than B&O in general.

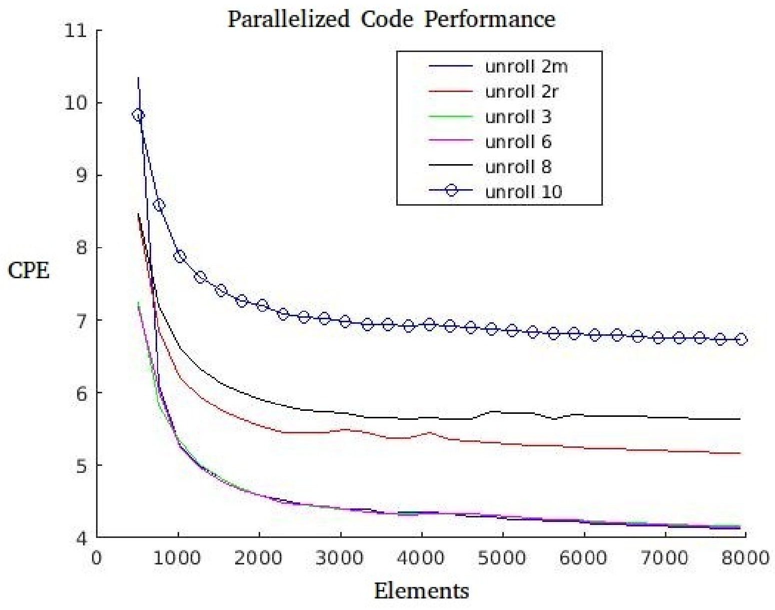
Part 1.b Loop Unrolling Evaluation



Overall in our tests we saw no net benefit to unrolling loops; as seen in the graph, they largely behave the same. The Haswell control and datapath (featuring prefetch, branch prediction, speculative execution, and out-of-order execution) effectively minimise any differences we might expect. In older processors with simpler datapaths, one would expect the unrolled loops to run faster because of a reduction in compare and conditional branch operations. In really old processors (on which Robert has done this sort of exercise) the instruction cache was so small that extreme unrolling was ineffective as the whole loop would not fit in instruction cache.

We expect that these tests would show an benefit from unrolling on smaller/lower-power RISC chips (e.g. ARM) since they frequently have simpler or no branch prediction and related resources.

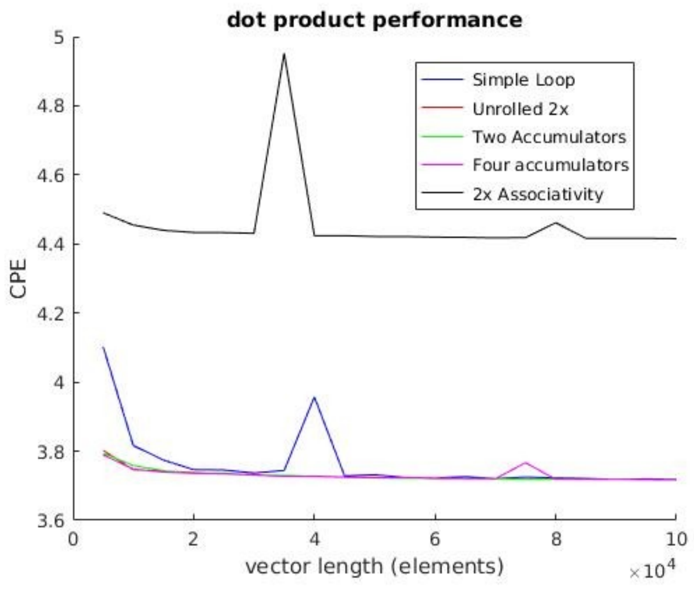
Part 1.c Parallelization (multiple accumulators; reassociation)



Parallelization showed a notably better effect on performance. Here, 2m refers to an unroll factor of two with multiple accumulators, while 2r is using reassociation. Here the factors beyond 6 of unrolling show a net negative effect, with 8 and 10 performing the worst. This could be a result of cache associativity. Unroll factors of 2 and 3 are the best, because, in this case, they make the best use of the multiplier being able to handle distinct multiplies on sequential cycles.

Part 2. Dot product

Code for the dot product tests is in `test_dot.c`. We tested the direct implementation without any attempted improvements; a 2x unrolled version; 2x parallel (dual accumulators); 4x parallel (four accumulators); and 2x associativity. All ran at a bit under 4 CPE, except the 2x associative version which was slightly slower.

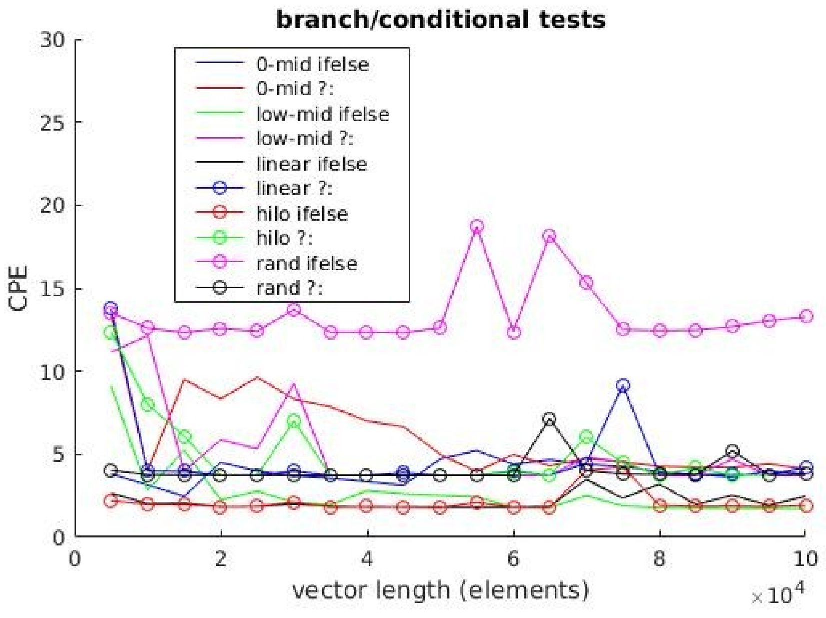


All five tests had the same memory bandwidth requirements: two loads and one store per element. This seems to be the only reason why the tests would all run at about the same speed.

We'll point out that is arithmetic intensity were the limiting factor, these tests would suggest that the CPU can perform four FP operation-streams in parallel without a loss of throughput to any of them; but this processor in fact has only two. (See Haswell datapath illustration at www.anandtech.com/show/ 6355/intels-haswell-architecture/8 and note that all FP ops are in Port 1 and Port 2)

Part 3. Branch and Conditional Move

The code for these tests is in `test_branch.c`.



The chart is a bit dense, so we'll list them here (in the same order as in the chart legend):

	CPE	*cond. type*	*description of data pattern(s)*
zeros	4.0	ifelse	[0,0,0,0,0,0,0,...] vs. [5,5,5,5,5,5,5,...]
	4.3	? :	[0,0,0,0,0,0,0,...] vs. [5,5,5,5,5,5,5,...]
const	1.8	ifelse	[1,1,1,1,1,1,1,...] vs. [5,5,5,5,5,5,5,...]
	3.7	? :	[1,1,1,1,1,1,1,...] vs. [5,5,5,5,5,5,5,...]
ramp	1.9	ifelse	[1,2,3,4,5,6,7,...] vs. [5,5,5,5,5,5,5,...]
	3.8	? :	[1,2,3,4,5,6,7,...] vs. [5,5,5,5,5,5,5,...]
hilo	1.9	ifelse	[1,9,1,9,1,9,1,...] vs. [5,5,5,5,5,5,5,...]
	3.7	? :	[1,9,1,9,1,9,1,...] vs. [5,5,5,5,5,5,5,...]
rand	13.0	ifelse	[<random> , ...] vs. [5,5,5,5,5,5,5,...]
	3.7	? :	[<random> , ...] vs. [5,5,5,5,5,5,5,...]

There were ten tests, using two types of "elementwise MAX" on vector data. The two types are:

ifelse	`if a > b { c = a; } else { c = b; }`
? :	`c = (a > b) ? a : b;`

The generated code differs in substantial ways. In the "ifelse" version, only one assignment statement is executed, in addition to compare and branch instructions. In the "? :" version, both assignment statements are executed (to temporary registers).

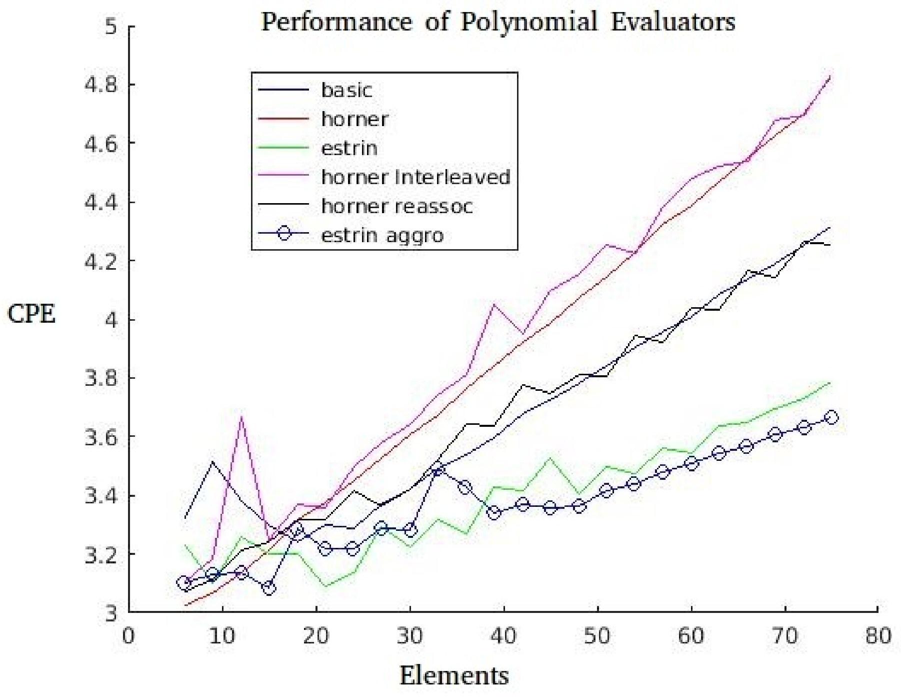
Either version might run faster, depending on details. If the CPU's branch predictor can always predict correctly, the compare and branch instructions in the "ifelse" version should have little or no impact on overall performance; while the "? :" version might run slower because both assignments (one of which always goes unused) are being run each time through the loop.

This situation seems to be the case in the middle three pairs of tests, labeled "const", "ramp", and "hilo". Notably, the "hilo" version is set up so that the if-else will take alternate branch paths each time through the loop; but the ifelse version is still the fastest, because the Intel branch predictor can detect regular repeating patterns.

On the other hand, if the vector data are random or otherwise difficult to predict, the "? :" version will run faster because it does not do any unpredictable branches.

This is the case in the "rand" test, where one vector contains quasi-random data. The "ifelse" version is much slower, because a mispredicted branch requires a flush of the entire pipeline.

Part 4: Polynomial Evaluation



NOTE the vertical axis scale! The current version of MATLAB does not allow a chart's axes to be zoomed out beyond the range of the data in the chart (see bug reports online, for example, www.mathworks.com/matlabcentral/answers/310631-it-is-not-possible-to-zoom-out-beyond-the-plots-current-zoom-out-point)

Methods and Explanation

Six different methods of evaluating polynomials were implemented. Two were basic ones from B&O: a straightforward iterative approach that builds the degree of x every loop and multiplies it by the according coeffecient and a simple implementation of Horner's Scheme, which evaluates the polynomial from the highest order to the lowest and uses the successive iterations to build the degree of x by the distributive property.

The next method applied is called Estrin's Scheme. It behaves similarly to the basic method with a little unrolling and balancing multiplies and adds. In this usage, we evaluate two degrees of the polynomial at once, with one having an x term multiplied in, then multiply the sum of that by the accumulated x factor.

Another method used was an interleaved version of Horner's scheme. A small unroll so that two terms are evaluate per loop was used. We know the previous result has a coefficient that needs a multiply by x, so that is handled, then we add the next lower coefficient, multiply that sum by x again, and add the following coeffecient.

Another Horner's Scheme method attempted was trying to break up the multiply dependency by using temp variables for reassociation. Essentially, the idea was that we can look at a given iteration of Horner like this:

```
coef[i-1] + x * (coef[i] + x * result)
```

where result is an accumulator, so we can multiply in x

```
coef[i-1] + x * coef[i] + x^2 * result
```

So by precalculating x^2, we can push the two multiplies into the multiply unit one after the other to try to take advantage of the architecture, and when they are done sum them with the low order coeffecient. With some loss of precision, something like this might benefit from a fused multiply-accumulate instruction.

The final method attempted was an aggressive form of Estrin's scheme. It worked by using a precalculated x^2 value, taking three coefficients instead of two, and accumulating the degree of x by using a precalculated x^3 term.

The basic simple implementation and Horner's with reassociation worked relatively the same. Meanwhile, the Basic Horner and the Interleaved version were a little worse. The Estrin based methods performed better, with a slight edge to the aggressive version. It would seem the Estrin based methods can take advantage of balancing multiplies and adds fairly well, which helps in balancing the load on the FPU, and the use of some simple precalculation beforehand helps to speed that up at the loss of generality. The Interleaved Horner's attempt would seem to have similar properties, but because there is a data dependence there has to be waiting for previous computations to complete. Similarly for the reassociative version. Since each loop ends with result needing two adds to calculate before the first temp can use it, the dependence on order of adds and multiplies seems to be slowing down how fast it can move. This isn't as strict in the Estrin models since a coefficient can multiply, get added against the low order coefficient while the result of the third coefficient multiply is being done, and summed again when both are ready, then into a final multiply. It makes more room for the compiler or scheduler to move about instructions while waiting for a coefficient multiply to finish or the x accumulator multiply to finish from the previous cycle.