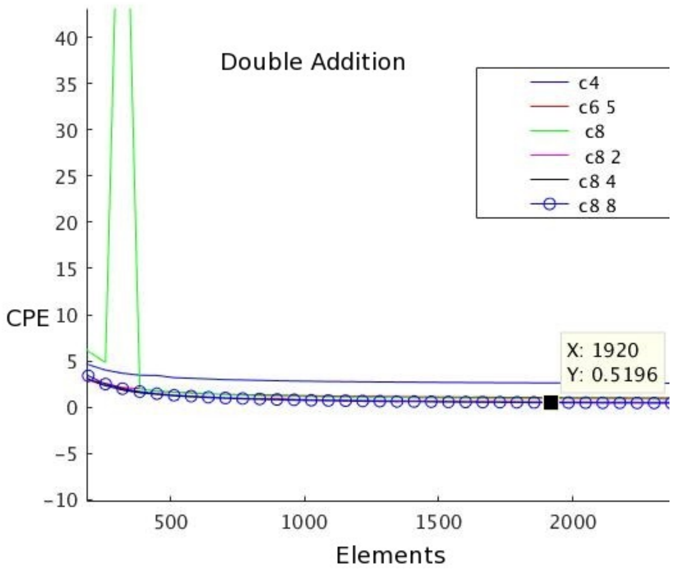
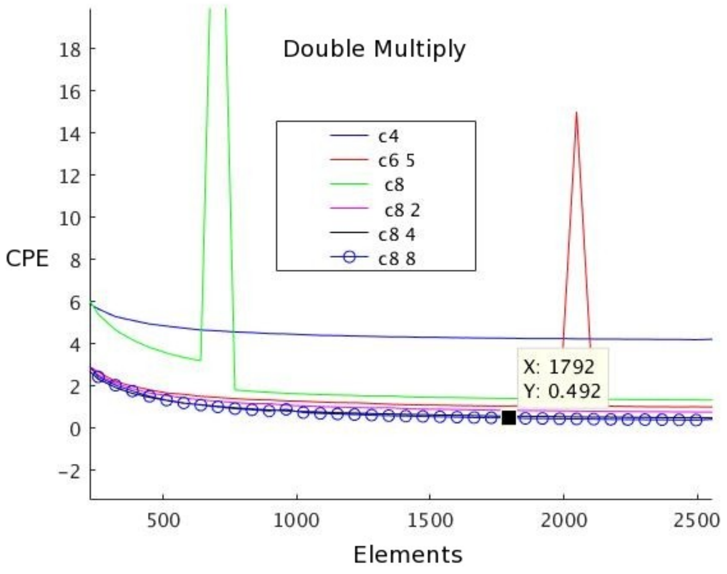
Robert Munafo
John S. Burke
EC 527

Lab 3

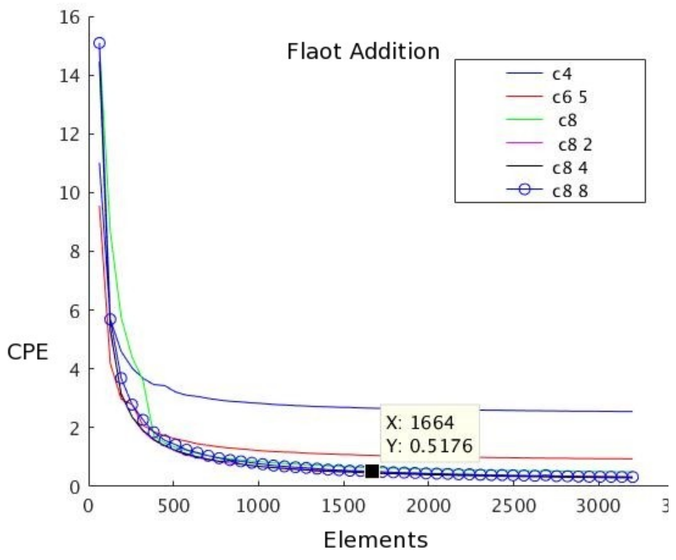
Part One

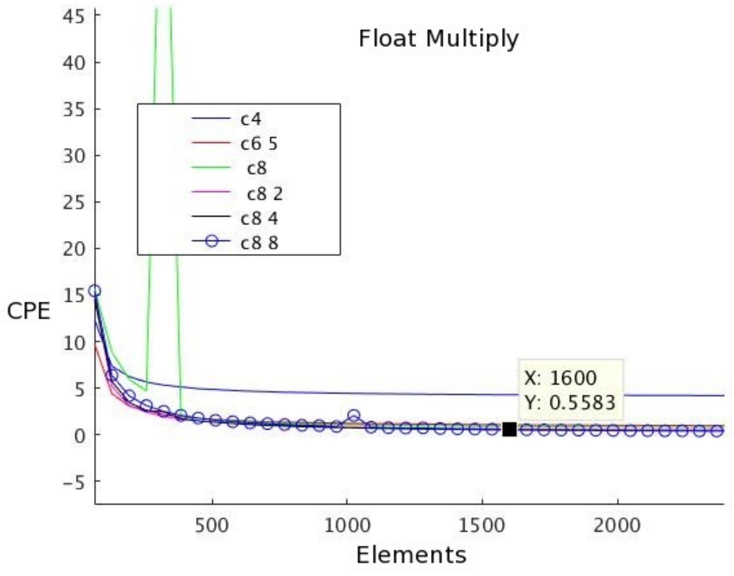
Combine



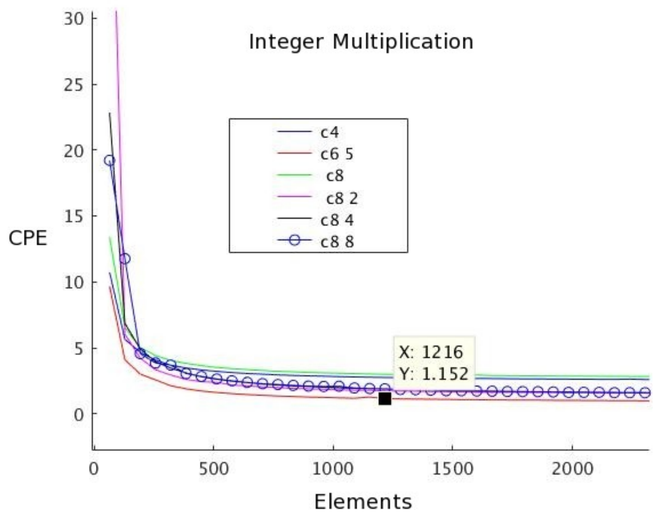


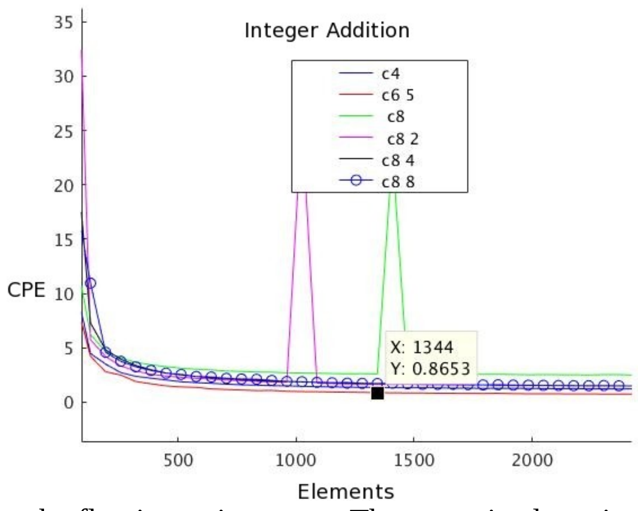
Above are the results from the combine tests on `doubles` for both addition and multiplication. As is clear, the scalar versions ("c4", "c6 5") fare at least a little worse than the vectorized versions. The later versions using GCC vector intrinsics ("c8", "c8 2", "c8 4", "c8 8") fare better because once memory is aligned nicely, we can stride on the size of the double. It allows us to make better use of underlying hardware, such as the multiplier which can take a new multiply every cycle and hold up to five at one time. This also helps with memory access concerns.





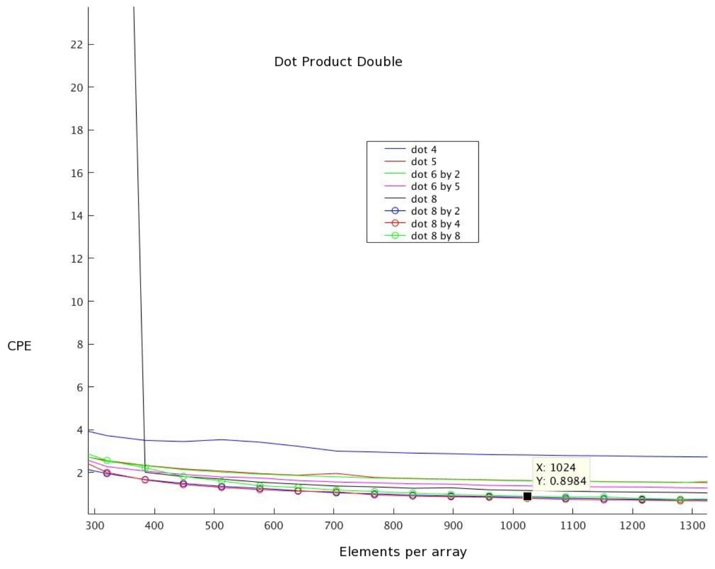
These two plots show the performance of the same six combine tests on vectors containing 8 `float`s. We see a continuance of the trends in the `double`, with the major difference being that the initial downward sloping portion at low element counts is stronger. This is a result of the `float` being smaller than the `double`. Again, the vector version with multiple accumulators has the best performance.

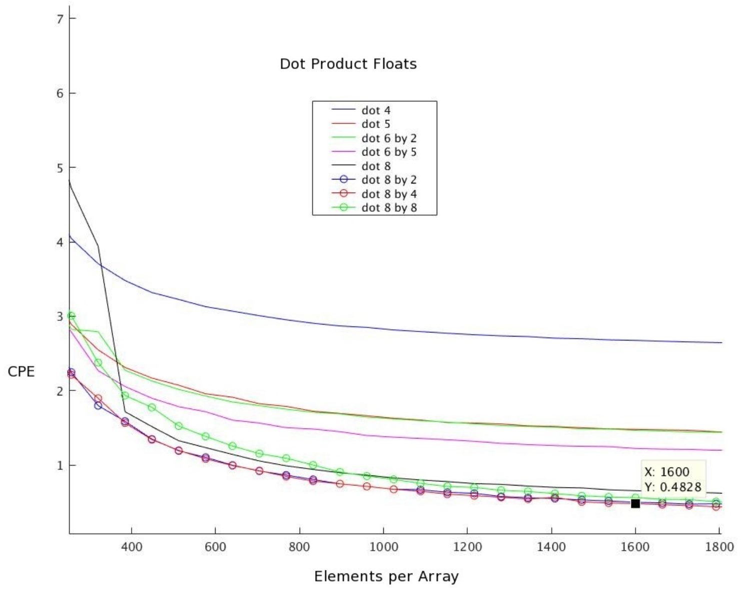




Integers show slightly different behavior from the floating point types. The vectorized versions do show good performance, but the scalar unrolled with multiple accumulators ("c6 5") fared the best. This is likely due to the fact that vectorization structures have floating point in mind for speed ups, and have underlying hardware to take advantage of when such structures are used.

Dot product



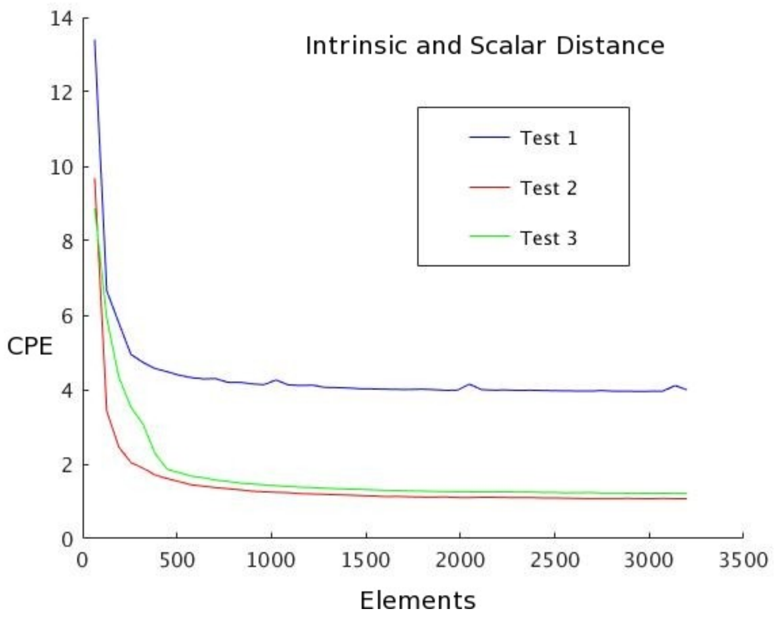


Above are the CPE plots of dot products for both `double`s and `float`s. "dot4" is the simplest scalar version; "dot5" uses associativity; "dot6" uses two accumulators; "dot6 5" uses 5 accumulators. The vector versions are “dot8” (single vector acting as accumulator), “dot8 2” (two vector accumulators), “dot8 4” and “dot8 8” (four and eight accumulators). The vectorized versions with multiple accumulators perform better than the other options, but amongst themselves have fairly similar performance. Dot Product is a great example of code that benefits from vectorizing and parallelizing because each multiply can be evaluated independently and the summation only requires that all the sub-totals be added together at the end; intermediate partial sums can be accumulated in parallel. At the same time, being able to use both the multiplication and addition units in a parallel manner over data that can be vectorized exploits the hardware naturally because multiplies can be stacked deep into the multiplier while partial sums are evaluated. This allows for the CPE metric to become extremely good (as low as 0.34 CPE for `float`, 0.6 CPE for `double`).

Part Two

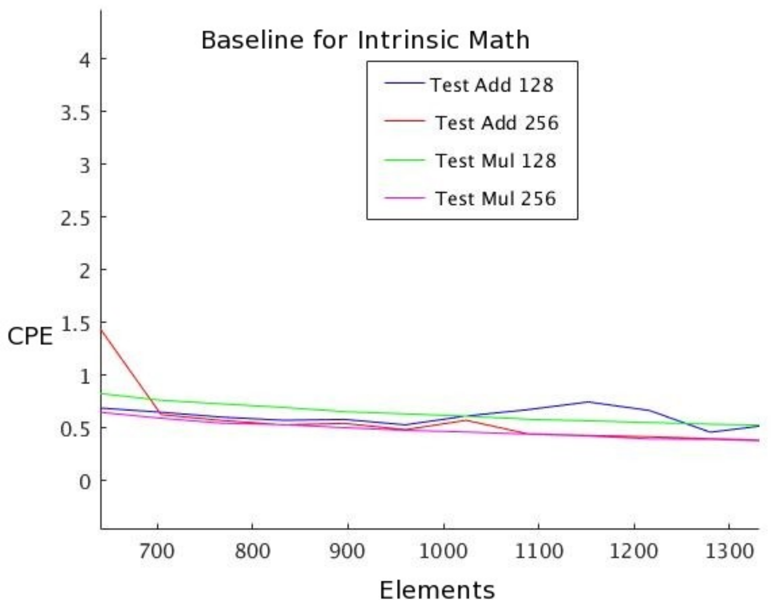
Basic Distance Calculations

This chart shows performance of the supplied `ArrayTest1/2/3`. These calculated an elementwise Pythagorean distance using scalar (“Test 1”), 128 bit SSE `_mm_mul/add/sqrt` intrinsics (“Test 2”), and 256-bit AVX `_mm256_mul/add/sqrt` bit intrinsics. The only real surprise is that 128 bit thinly outstrips 256, but this is potentially due to slightly more overhead.

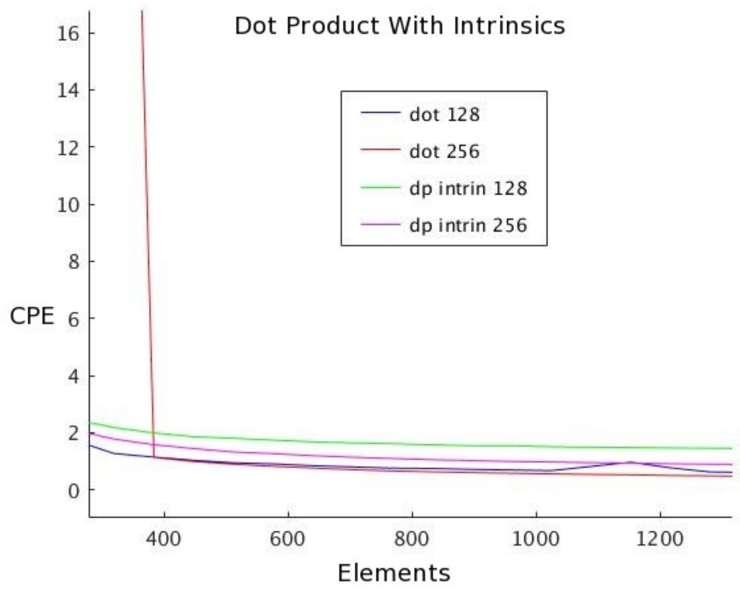


Elementwise Multiplies and Add

In this plot we show the baseline performance of the intrinsic multiplies and adds for the 128 and 256 bit intrinsics. Generally they sit at around 0.5-0.6 Cycles per Element, which is good as is. Smart use of vectorizing and parallelism can potentially push the throughput up a little bit more.



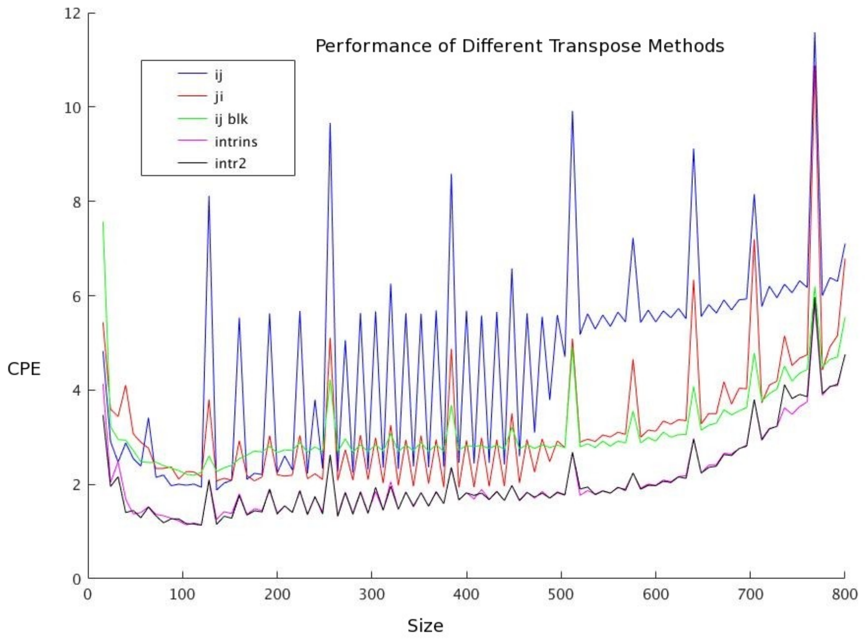
Dot Product Tests



This plot shows four versions of the dot product with intrinsics. The blue and red lines use `_m_mul/add_ps` or `_mm256_mul/add_ps` intrinsics to build up the dot product, while the other two invoke the specialized `_mm_dp_ps` and `_mm256_dp_ps` intrinsics. Oddly, the 128 bit `dp_ps` intrinsic does worse than baseline multiply or add at just over 2 CPE. Similarly, the 256 bit version is at about .73 CPE. It is good, but clearly beaten by the other two methods. The best of the four was the one built from other 256 bit intrinsics, which gets down to about 0.33 CPE when the array size is large (3200 elements), while 128 bit was around 0.5 CPE. This means the 256 bit version is making better use of the processor than the baseline multiplies and adds, which makes some sense since it can push multiple multiplies in and evaluate partial sums in a parallel manner.

Overall, intrinsics can get a good speed up, but it is comparable to the vectorized and multiple accumulator code presented in part one. It requires more careful planning as well including some specialized knowledge of the intrinsics. For example, in the best performing code here, the 128 bit intrinsic version required extra `hadd_ps` calls and a final store to extract the final true dot product. The 256 bit version required making a new sum routine ("sum_256" in the code) using several `extract/cast/movehl/shuffle` intrinsics to ensure all corner cases were covered, and it required a fair deal of research and careful planning to ensure it would function. Overall, they require more time and more careful thought, and anything that uses them extensively deserves more dedicated rigorous testing to ensure all corner cases are covered. Such measures should really only be taken when performance demands are severely consequential.

Part Three : Transpose



Once again we used `posix_memalign` to create the data blocks. For the transpose tests we only tested array sizes in multiples of four. Spikes of relative slowness align with powers of two and multiples thereof (128, 256, 384, 512, …) strongly suggesting that cache conflict misses are impacting performance when the arrays have row lengths of those values. Note that the largest arrays tested are too big even for the L3 cache.

The supplied scalar code ("ij" and "ji" on the plot) performed similarly to the transpose functions in Lab 1. We added a 4x4 blocked version ("ij_blk") to improve L1/L2 cache hit rates. In the region around 100x100 it performs at about 2.1 CPE. We found no 4x4 transpose macro for double data so we created out own using `unpacklo/hi` and `permute2f128` intrinsics. The first version ("intrins") improves performance to just about 1.1 CPE.

This improvement is not quite as good as we had hoped for, but seems reasonable. 16 units (four vectors of four doubles each) requires, at a minimum: four 32 byte wide vector loads, 8 shuffles, and four 32 byte wide stores, with all loads and stores accessing new, unique locations.

Since GCC doesn't allow for `_m256d` parameters to be passed to an inline function by reference, we tried a different version ("intr2") that invokes the intrinsics through a preprocessor macro. This version also attempts to make all pointer operations, address calculation, explicit; however, its performance is no better than "intrins".