SIMD Parallelization of the Matrix Multiplication Method

Avery Boyd

SIMD instructions take advantage of the fact that processor registers are becoming bigger while normal data type stay the same size. We can put multiple pieces of data into one register and do functions on all of the data at once instead of multiple times for each piece of data. In this study, 128-bit registers were used to hold four 32-bit floating point data types. Two kinds of ways to implement the multiplication of two matrices, A and B, are augmented to show the potential speedup when using SIMD instruction sets.

Both of these methods used three nested loops to go through A's rows, B's rows/columns, and add up the multiplication of the respective elements. Each of these methods were run many times on floating point arrays of random floats that represented square matrices. The matrices used had row sizes of powers of two starting at four and going up to 2048, i.e., 4x4 elements up to 2048x2048 elements.

The first method of implementation of the matrix multiplication is by transposing the B matrix first and then multiplying rows in A by rows in B. Using one transpose function before multiplying makes it so that the columns can be localized in memory. This allows us to not only avoid cache misses, but also gives an easier time augmenting over to SIMD instructions.

Firstly, I changed the inner most loop for compatibility with SIMD instructions so that the multiplication and addition of the elements could be done in parallel. Secondly, I unrolled the inner most loop and changed the middle loop for compatibility with SIMD instructions to run multiplications of 4 rows in parallel. **Figure 1** shows that the first conversion gave an average speedup of about 3 times across all data sets. And the second conversion gave an average speedup of about 6.5 times across all data sets except the last. The last data set contains 16 MB of data and thus overflows the cache causing a reduced speedup of about 4 times. To not give a biased comparison, I mirrored the process on the sequential version by unrolling the inner most loop 4 times and then unrolling the second loop 4 times. In **Figure 1**, we can clearly see that the SIMD instructions beat their respective unrolled counterpart in every data set.

The second way is the natural way by multiplying rows in A by columns in B. This way does not take advantage of localized the data, and thus data must be gathered from different locations in memory.

Once again, I converted the loops for SIMD compatibility and an unrolled sequential version and compared them to the sequential version. This time the performance did not increase until the 256x256 data set. In fact, as seen in **Figure 2**, the method actually slowed down about 8% until that point. This is probably because the SIMD and unrolled version require the cache to load multiple columns earlier than the sequential version would. Whereas, when the data sets become much larger than the cache, this problem is minimized due to many cache lines getting evicted anyway.

Figure 1: Speedup of Method One on Increasingly Larger Data Sets

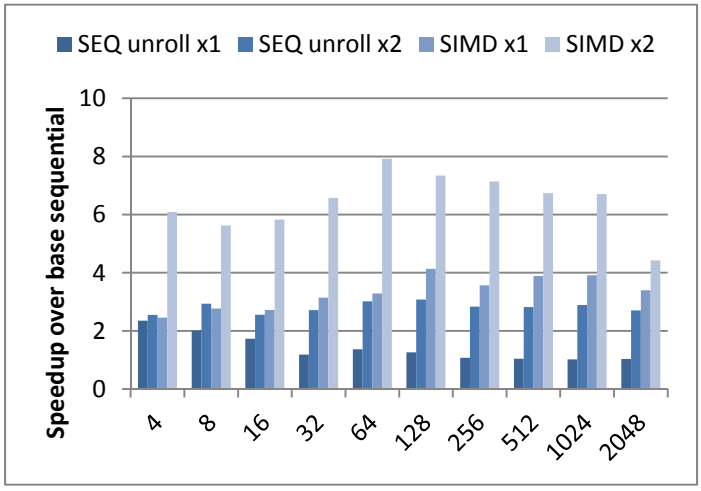

Figure 2: Speedup of Method Two on Increasingly Larger Data Sets