Jesse Thoren

May 8, 2017

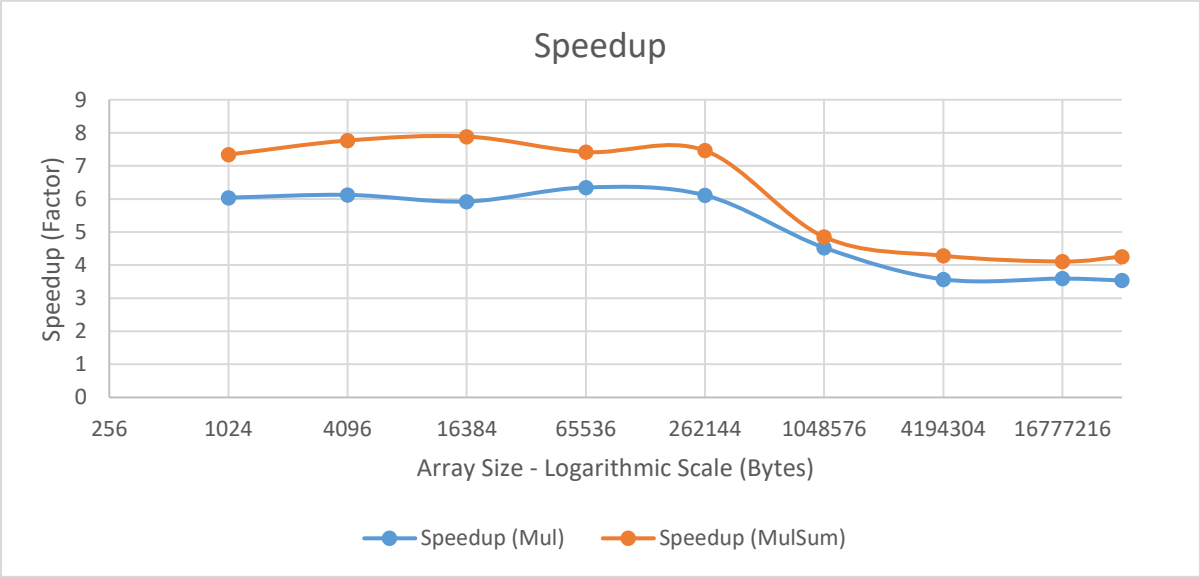
CS 475 – Parallel Programming

Project 5 – Vectorized Array Multiplication and Reduction using SSE

1. My code was run on Flip on May 8, 2017. Uptime reports that the load average is less than 1.
2. Below is the table of the data collected from running my code:

ArraySize	Speedup (Mul)	Speedup (MulSum)	SimdMul	NSimdMul	SimdMulSum	NSimdMulSum
1024	6.039559925	7.341515313	774.03	128.16	966.07	131.59
4096	6.123316965	7.76609913	773.13	126.26	1026.29	132.15
16384	5.923052615	7.887522769	749.74	126.58	1039.26	131.76
65536	6.348144113	7.418871053	815.8	128.51	984.41	132.69
262144	6.111268159	7.462259161	778.27	127.35	987.63	132.35
1048576	4.528392605	4.853016713	1092.52	241.26	1228.25	253.09
4194304	3.56937549	4.28327439	819.6	229.62	993.12	231.86
16777216	3.592368465	4.109242651	854.84	237.96	989.67	240.84
33554432	3.536689742	4.257502109	842.97	238.35	1059.82	248.93

Below is the graph produced by the first 3 columns of my data. Note that the x-axis is presented with a logarithmic scale for better readability.



3. In both the Array multiplication (plotted in blue) and the array multiplication with reduction (plotted in orange), we see an efficient speedup value that tapers off around an array size of 1 megabyte. It appears that the data tends to level off around a speedup of 4 times.

   Furthermore, it appears that the reduction example consistently produces a greater speedup than array multiplication on its own.

4. The observation that reduction is consistently producing a greater speedup than array multiplication seems to be consistent regardless of the arraysize chosen (at least between 1 KB and 32 MB). This data was chosen from the run with maximum performance over the course of 100 runs where the maximum performance for the run did not exceed 1.2x the average run performance, so it doesn't seem like this is happening because the runs were particularly lucky either.

5. I was initially surprised at two things from the data: 1) Speedup managed to get a speedup factor of greater than 4, even though we were only simd'ing 4 floats at a time, and 2) the speedup value makes a nose-dive around 1 MB arraysize.

   After considering why these might be the case, I'm inclined to think that the very large speedup values noted in observation 1 are occurring due to the simd code explicitly executing assembly, while the non-simd code was using normal array multiplication in c, so the assembly code may be much more efficient. The dropoff in speedup values occurring in observation 2 is most likely because the entirety of the arrays cannot be stored in a cache file close to the cpu, so there is decreased performance due to having to fetch different cache lines from further away from the cpu.

6. As I noted in question 5, a speedup of greater than 4 seems to be occurring in the array multiplication example because the simd code is written in assembly instructions, so it may be inherently more efficient than the code in the non-simd loop. A speedup of less than 4 could occur because there is some overhead common to both of the code examples (constant time that has to be spent fetching cache), and the simd code has some overhead with setting up the simd operation and interpreting the result. Effectively, the entire code is not parallelizable, so we know that the theoretical 4x speedup from the parallel simd operation should result in a total of less than a 4x speedup because there is a non-parallelizable section of the code.

7. This is basically the same explanation as in number 6… greater speedup can occur because the simd code is more efficient than the non-simd code, less speedup can occur because of cache fetching and non-parallelizable sections of the code, in addition to additional overhead for the set up and interpretation of the simd section.

   Furthermore, I'm guessing the reduction code has higher speedup values than the array multiplication because the reduction is happening all at once, rather than alternating between addition and subtraction in the non-simd code.