Lyon Kee

keel@oregonstate.edu

Project #4: Vectorized Array Multiplication and Reduction using SSE

Key snippets of code

```
122 void
  1 NonSimdMul( float *A, float *B, float *C, int n )
  2 {
  3     #pragma omp parallal for
  4     for (int i = 0; i < n; ++i) {
  5         C[i] = A[i] * B[i];
  6     }
  7 }
  8
  9 float
 10 NonSimdMulSum( float *A, float *B, int n )
 11 {
 12     float sum = 0;
 13     #pragma omp parallel for shared(A, B, C), reduction(+:sum)
 14     for (int i = 0; i < n; ++i) {
 15         sum += C[i] = A[i] * B[i];
 16     }
 17     return sum;
 18 }
```

Line 122: We run a loop to loop through array A and B and multiply it into C, we utilize OpenMP for loop to use multiple threads, in this configuration, we are using 2 threads as it provided us the best nonSimd performance which leads the smallest speedup between the OMP for loop and Simd. This is due to the fact that the cpu only has 2 threads available.

Line 132: We run the same for loop, then sum it with OpenMP reduction to provide us with the sum.

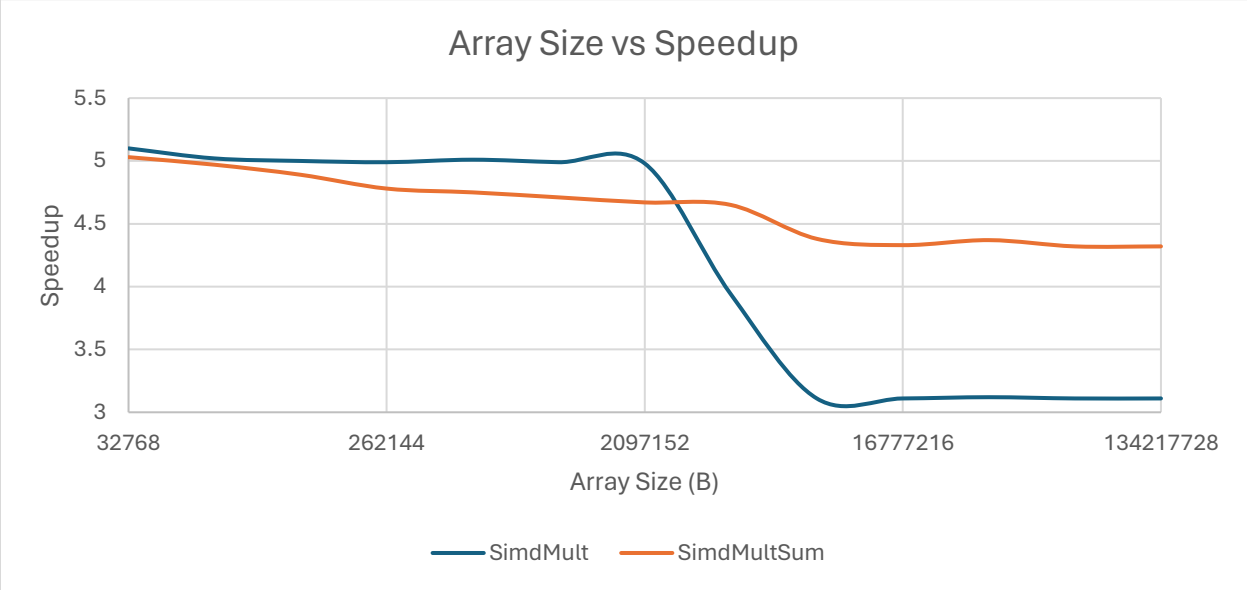
Tables of data

ARRAY_SIZE	NonSimdMul	SimdMul	SpeedUp	NonSimdSum	SimdSum	SpeedUp
32768	278.7	1420.99	5.1	300.57	1513.11	5.03
65536	278.14	1394.95	5.02	304.74	1513.32	4.97
131072	278.63	1392.75	5	310.01	1515.2	4.89
262144	277.71	1386.16	4.99	316.29	1512.75	4.78
524288	277.01	1388.14	5.01	317.59	1508.44	4.75
1048576	276.9	1381.6	4.99	318.81	1500.18	4.71
2097152	276.5	1376.77	4.98	319.04	1490.91	4.67
4194304	274.56	1081.77	3.94	317.11	1474.92	4.65
8388608	274.49	854.13	3.11	316	1385.18	4.38
16777216	274.02	853.37	3.11	315.25	1365.79	4.33
33554432	274.44	856.62	3.12	315.15	1376.46	4.37

67108864	274.39	852.62	3.11	315.15	1361.76	4.32
134217728	274.34	851.89	3.11	315.37	1363.49	4.32

Graphs of data

Graph of Array Size vs Speedup



We see a dip at about 2M to 8M, this is likely due to it reaching a point where memory is no longer being fetched from cache making it grab memory from outside of L1 cache. There is a small dip from 4M to 8M, this is because of the same issue, but because we are doing the sum, it is likely that it takes much longer time than the mult therefore it is hard to observe the mult dip through the reduction.

Conclusion

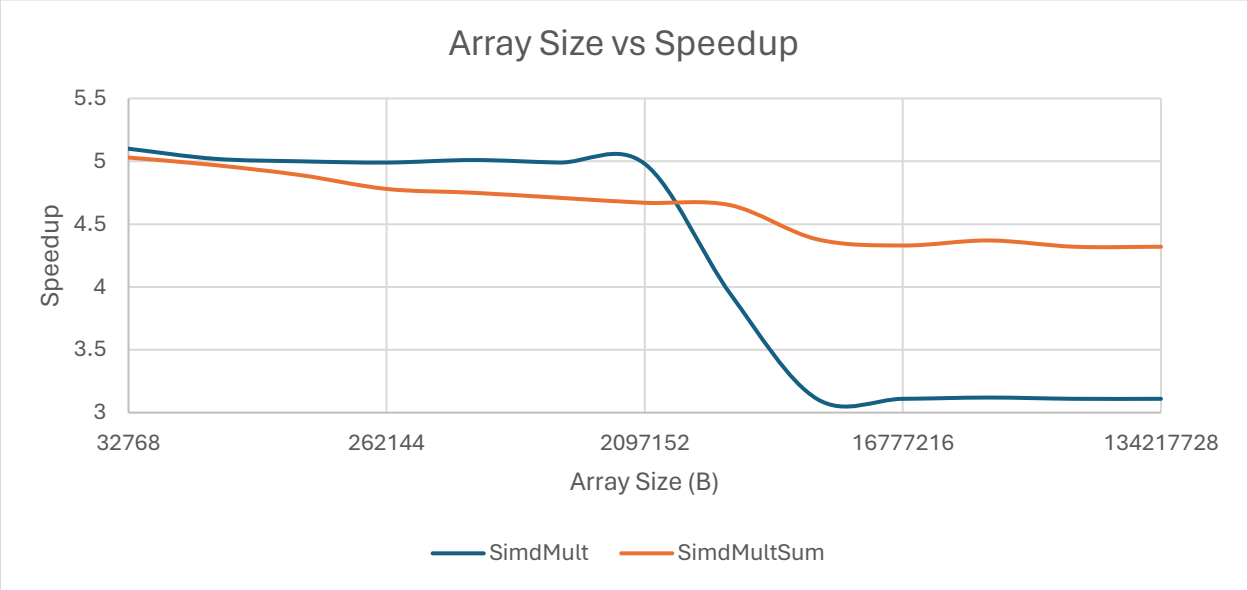
1. What machine you ran this on

I ran this on the A100s that the class has access to, therefore I get dedicated cpu time and it is very reliable.

2. Show the 2 tables of performances for each array size and the corresponding speedups

It is shown on the **Tables of data** section with both combined into one.

3. Show the graphs (or graph) of SIMD/non-SIMD speedup versus array size (either one graph with two curves, or two graphs each with one curve)



4. What patterns are you seeing in the speedups?

We see a dip at about 2M to 8M, this is likely due to it reaching a point where memory is no longer being fetched from cache making it grab memory from outside of L1 cache. There is a small dip from 4M to 8M, this is because of the same issue, but because we are doing the sum, it is likely that it takes much longer time than the mult therefore it is hard to observe the mult dip through the reduction.

5. Are they consistent across a variety of array sizes?

These patterns are consistent across array sizes. For SimdMult, there is a consistent pattern before and after 4M.

6. Why or why not, do you think?

For SimdMultSum, due to the reduction being O(log(n)) instead of O(n/parallelization factor) overwhelming the time consumption.

For SimdMult, this is likely the point where memory swapping is happening.

How did things turn out?

This turned out pretty well, I had a small mistake because I did not read the structs beforehand and assumed some things. And as always, on my own laptop, I need to change the number of threads to experiment on because I have 16 threads and the behavior is slightly different from a server due to the scheduling of OS and other applications in the background.

Why do you think they turned out that way?

The charts turned out the way that it is, is due to the nature of program and also scheduling, there is always a cost to starting threads, and if each thread's responsibility is not a lot then it is not worth it to create a thread for it. Another thing is that if each task if too large, then work load is not distributed evenly as the final task might take a very long time to finish, and at the same time, all other threads are doing nothing when the last task can be broken up and be done by other threads.

NOTE: I did extra credit, all statements prints to stderr except for results so we can use `>` to send stdout to another file and not other logging messages.