Adam Wright

wrighada

CS-475

Project 4

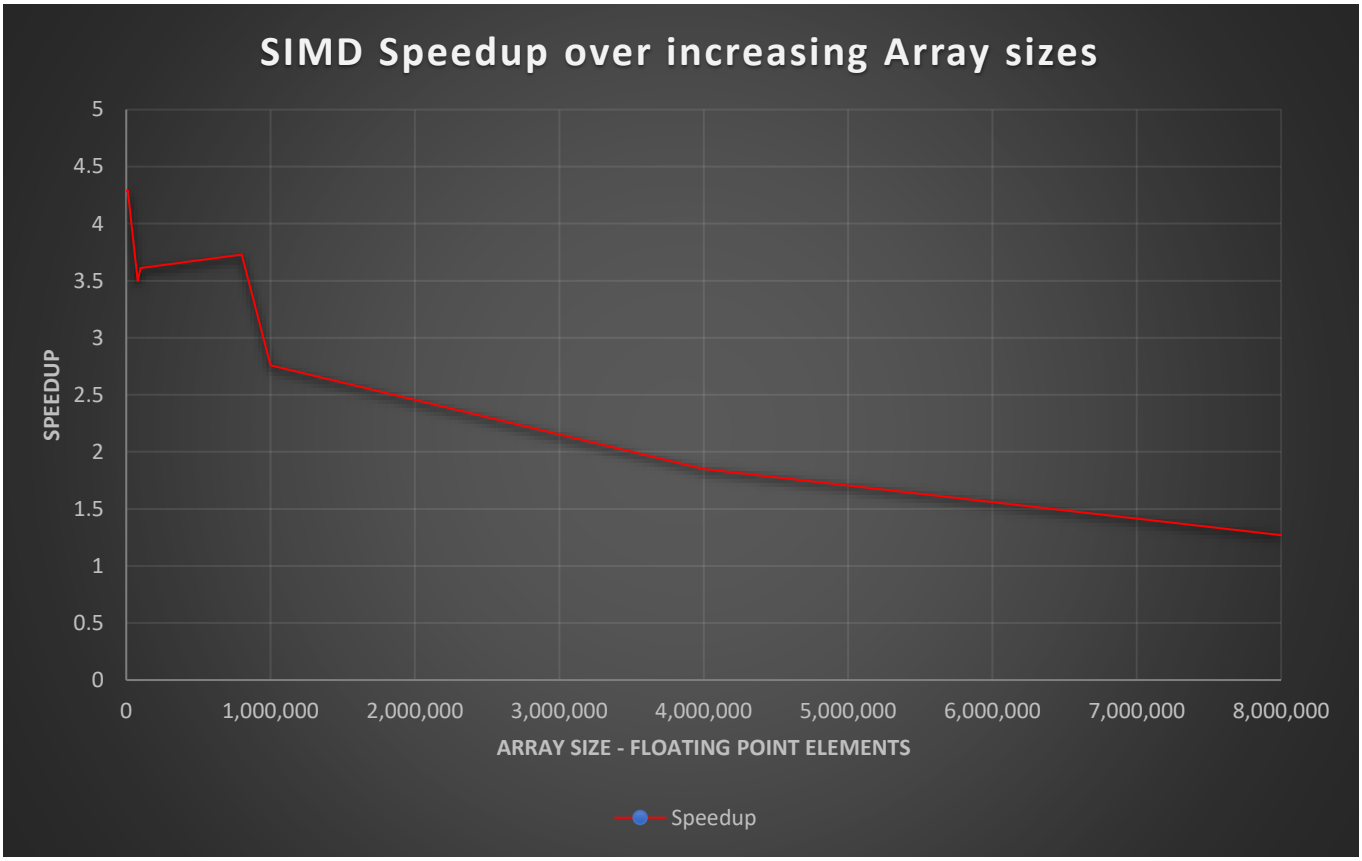
1. What machine did I run this on?

I ran the simulation sucessfully on OSU's Flip server with one five and fifteen minute load averages of 3.66, 3.80, and 3.73 respectively. I originally attempted to run the tests on my local Windows machine, but I didn't get any resultant speedup from the tests.

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

Array Size	Conventional Array MegaFlops/sec	SIMD MegaFlops/sec	SIMD Speedup
1000	818.40	3,508.96	4.29
8000	821.61	3,518.30	4.28
10,000	822.32	3,523.93	4.29
80,000	773.84	2,709.12	3.50
100,000	809.64	2,921.31	3.61
800,000	793.13	2,958.83	3.73
1,000,000	704.14	1,946.43	2.76
4.000,000	713.28	1318.97	1.85
8,000,000	705.55	897.59	1.27

3. Graph of SIMD/non-SIMD speedup versus array size



4. What patterns are you seeing in the speedups?

As the array size grows it is very clear that the algorithm is able to process the values in the array more quickly than the cache or memory bus are able to provide values. Before the 10,000 element mark the cache is well equiped to provide elements to process and between 80,000 and 800,000 elements there is only a slight decline in performance to a speedup of around 3.5. At 1,000,000 elements the speedup is below three and as it apporaches 8,000,000 it is close to having no remaining speedup at all.

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

No, the speedups decline significaltly after the array size grows to larger than 1,000,000 elements. While the array size is below 10,000 elements, the performance is consistent, but between 80,000 and 1,000,000 elements there is a noticable decline. The 4,000,000 element array does still approach a doubling of the speed, but by 8,000,000 there is nearly no speedup left at all.

6. Why or why not, do you think?

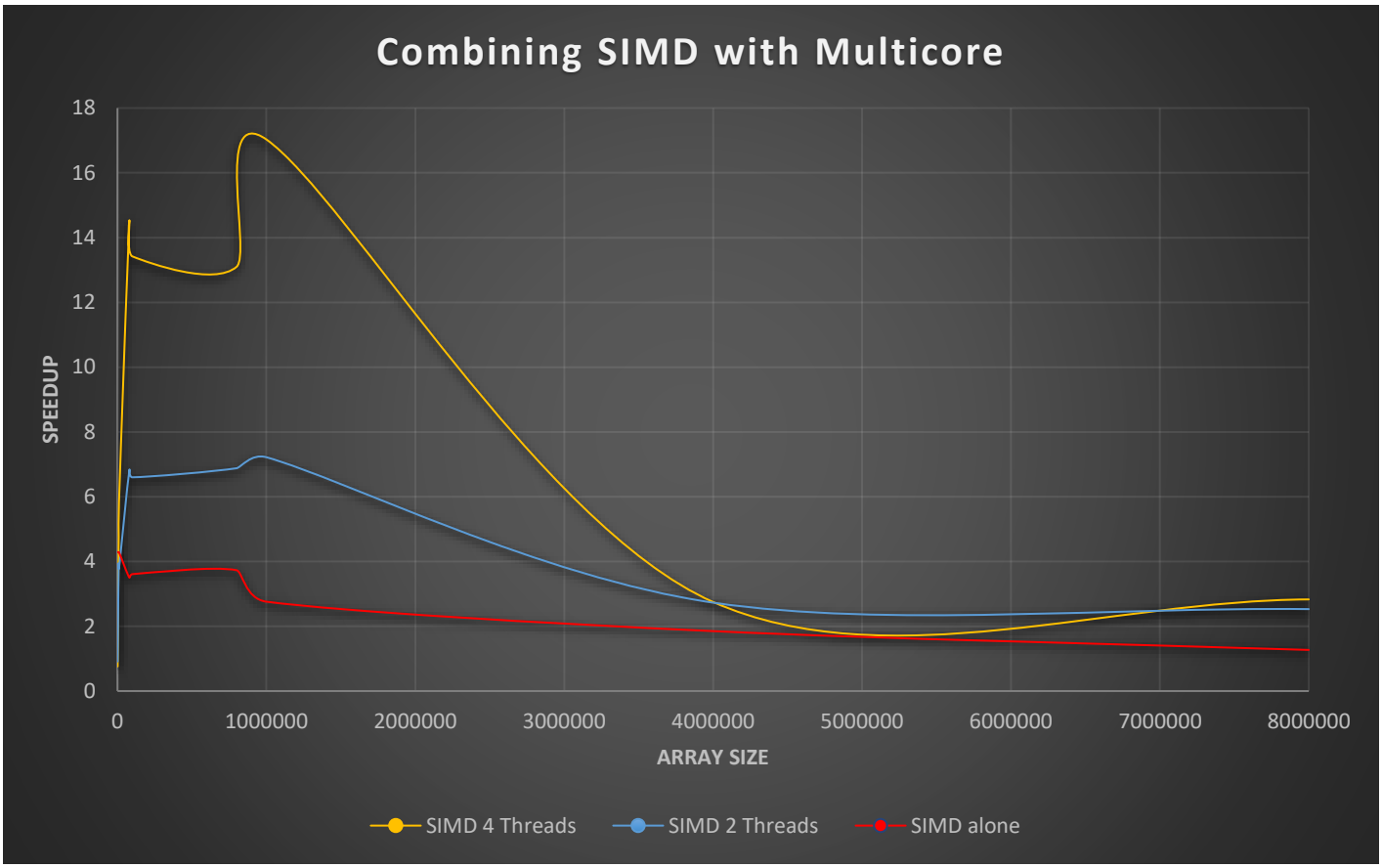
I believe that the algorithm is outrunning the supply of elements in the cache to provide elements for the registers. Once the registers need to make a round trip to RAM, the memory access negates the benefits of SIMD. This algorithm would benefit greatly from using prefetch, if the algorithm's primary purpose were to process arays of more than 10,000 elements. Prefetch would likely allow the algorithm to maintain it's 4x speedup through the entire range of array sizes in this test.

EXTRA-CREDIT:

NUMT: Number of threads --- SIZE: array size --- Arr_MFL: non-SIMD array MegaFlops

Simd_MFL: SIMD MegaFlops --- S_UP-1: single threaded speedup --- S_MT_MFL: SIMD Multicore MegaFlops --- S_UP_2: multicore and SIMD speedup

```
************************************************
**    Non-SIMD vs SIMD vs Multi-core SIMD     **
************************************************

NUMT   SIZE      Arr_MFL      Simd_MFL   S_UP-1   S_MT_MFL   S_UP_2

 1     1000       781.47       3334.60    4.27     1161.43    1.49
 1     8000       821.77       3542.97    4.31     2769.16    3.37
 1     10000      821.97       3547.80    4.32     2893.01    3.52
 1     80000      809.24       2717.56    3.36     2606.80    3.22
 1     100000     773.70       2780.99    3.59     2680.17    3.46
 1     800000     729.17       2075.40    2.85     2039.45    2.80
 1     1000000    779.08       2520.75    3.24     2647.73    3.40
 1     4000000    713.28       1318.97    1.85     1301.80    1.83
 1     8000000    706.77       1285.61    1.82     1302.12    1.84

 2     1000       780.90       3334.60    4.27      756.96    0.97
 2     8000       786.98       3373.89    4.29     2921.75    3.71
 2     10000      787.03       3394.69    4.31     3115.46    3.96
 2     80000      809.13       2805.84    3.47     5204.92    6.43
 2     100000     809.28       2839.12    3.51     5007.66    6.19
 2     800000     787.17       2964.86    3.77     5478.33    6.96
 2     1000000    743.14       2689.74    3.62     4844.05    6.52
 2     4000000    697.17       1292.38    1.85     1902.96    2.73
 2     8000000    656.93        971.06    1.48     1198.82    1.82

 3     1000       818.40       3508.96    4.29      589.97    0.72
 3     8000       821.73       3514.70    4.28     3570.21    4.34
 3     10000      786.97       3386.66    4.30     3831.37    4.87
 3     80000      773.61       2749.44    3.55     7395.23    9.56
 3     100000     773.81       2734.78    3.53     7303.37    9.44
 3     800000     453.91       1713.87    3.78     4690.06   10.33
 3     1000000    415.29       1515.80    3.65     4243.18   10.22
 3     4000000    697.27       1236.39    1.77     2128.35    3.05
 3     8000000    698.38       1244.02    1.78     1704.46    2.44

 4     1000       819.03       3508.96    4.28      571.14    0.70
 4     8000       786.73       3365.30    4.28     3420.24    4.35
 4     10000      822.57       3547.21    4.31     4221.51    5.13
 4     80000      773.21       2683.81    3.47    10743.46   13.89
 4     100000     424.56       1599.44    3.77     5974.86   14.07
 4     800000     734.29       2404.44    3.27     6760.08    9.21
 4     1000000    449.18       2767.75    6.16     7650.00   17.03
 4     4000000    701.73       1276.63    1.82     1929.37    2.75
 4     8000000    640.70       1305.79    2.04     2067.03    3.23
```



E.C. A brief discussion of what the curves are showing and why I think it is working that way.

The curves are showing that in the regions where the cache can provide a sufficient number of elements to keep the registers filled, it is possible to get close to a 16 times improvement by doing SSE SIMD with four core multi-threading. The multicore version of the implementation still suffers from the same behavior as the single core SIMD implementation, which is that above 1,000,000 array elements, the speedup becomes much smaller. The reason for this is the same as the reason for the performance degradation in the single core version which is that the cache is unable to provide contain enough values for the algorithm to remain effentient. The addition of properly implemented prefetch would allow the larger array sizes to remain performant, though at array sizes below 1,000,000 this non-prefetch implementation is a great performer.