cb4904, Caraline Bruzinski

HPC
Assignment 2

March 5, 2023

1. Finding Memory bugs using valgrind.

First piece of code "val_test01_solved":

The first issue with this code was a memory leak. There wasn't enough memory allocated and the program was attempting to write one additional integer (size 4) which wasnt allocated. I added another integer to the malloc and this solved all issues.

Second piece of code "val_test02_solved":

I'm assuming that the desired output is what is printed to the screen. Therefore, if the expected output is that the remaining indices in x should be set to zero, then set them to zero by default at declaration. I referred to the solution described here (https://stackoverflow.com/questions/39909411/what-is-the-default-value-of-an-array-in-c), and added an open parenthesis after the new declaration of x.

2. Matrix-matrix multiplication.

Processor Information
Model: MacBook Air
Chip: Apple M1
Cores: 8 (4 performance, 4 efficiency)
Memory: 8 GB
M1 chip uses ARM, and stores data in 64 bits

First, I measured performance between different loop orderings of the matrix multiplication scheme. As the code performs the calculations on various matrix sizes (dimensions 16...2000), I computed the averages for each attribute after the complete run, per each loop ordering. The average values for each loop order are listed in the table below:

Loop Order (No Blocking):

Loop Order	Avg Time/s	Avg GFlop/s	Avg GB/s
JPI	0.412	12.051	192.813
PJI	0.670	8.530	136.487
IPJ	5.782	2.496	39.940
PIJ	5.728	2.404	38.469
IJP	2.359	2.245	35.913
JIP	2.628	2.173	34.762

The results are sorted by GFLOP/s, and the loop ordering j, p, i has the best performance for the greatest number of GFLOP/s and highest bandwidth. Notice that the best two runs loop over i last. Although the performance between using j last and using p last are similar, the worst runs loop over p last. The reason for the difference in performance is likely due to the method of reading the data to and from the cache, and the amount of data able to be reused in the computation.

As the code is running, when i is the innermost loop, this means that j and p are fixed when we increment over i. My processor will load a chunk of C into the cache (along with the data around it in the entire cache line). In this way, the processor can keep the C cache line in the cache and reuse it for the next element in the row of C.

This is very different if we write p to be the innermost loop. If p is the innermost loop, that means that when we are iterating over p, that i and j are fixed. This time the processor won't be able to keep B in cache, as B is accessed by columns, each cache line load only takes advantage of one element. This has lower reuse and is more costly to access the memory each time.

Blocking and Block Size:

Next, I implemented a blocking scheme over j. I experimented with various block sizes to find the optimal block size for my architecture. The way that the code is written, dimensions of the generated matrices are generated by the block size itself, so I did not always have a one-to-one comparison of flop rates with same matrix dimension. In order to gauge the performance for large matrices, I took the average flop rate for all iterations where the matrix dimensions were between 1,500 and 2,000. GFlop/s performance results at various block sizes are listed in the table below:

Effect of Blocking on code performance:

Block Size	Avg GFlop/s	Avg GB/s
4	13.733	219.722
8	13.555	216.883
12	12.302	196.839
16	11.892	190.267
20	11.397	182.358
24	11.233	179.735
28	10.932	174.919
32	10.491	167.863
36	10.891	174.254
40	10.267	164.275
44	10.516	168.252
48	10.082	161.308
52	9.896	158.332
56	10.157	162.526
60	6.276	100.420
64	6.297	100.756
68	6.212	99.401

From the table above, we observe that greatest performance in GFlop/s and GB/s is when the block size is small. Namely, when BLOCK_SIZE=4 is when I observe the best performance for my architecture. I was really surprised by how small the best block size was. Other classmates (using other machines) reported that larger block sizes as being most performant. At this point I'm considering if the M1 chip operates well on very little data when it performs matrix-matrix multiplication. I was suspicious that I didn't see the same lift as my classmates. Therefore, to double check my results, I also tried various block sizes with using other loop orders, but I observed worse performance than I did with my previously established loop order (as written in the blocking documentation, and submitted in my code).
Previously, the code without blocking sustained an average performance of 12.051 GFlop/s and 192.81 GB/s. However, now with the inclusion of blocking (using block size of 4), the performance is now 13.733 GFlop/s and 219.722 GB/s. This is an increase in performance by 13.96%. We can say that blocking has made an improvement in the performance of the code.

Parallelization:

For my Mac M1, I was not able to successfully compile openMP with my ARM architecture. From the documentation I read online, it may not be fully supported for M1 yet. Instead, I ported my repository to run on CIMS, snappy3, in order to compare the performance of my code before and after adding openMP.

The architecture for snappy3 on CIMS is:
Server: snappy3.cims.nyu.edu
Architecture: x86_64
CPU: Two Intel Xeon E5-2680 (2.80 GHz) (20 cores)
Memory: 128GB

I ran the blocked code on the snappy3 server both before and after adding parallelism. The complete run output, as well as the average results, are listed below:

Effect of openMP on code performance:

	Avg Time	Avg GFlop/s	Avg GB/s
no parallelism	0.815	6.411	102.589
with parallelism	2.357	40.043	640.675

The non-parallelized code runs with a lower average Flop rate than on my local Mac M1. Even so, the performance of the code running on CIMS significantly improved after adding parallelism. Also notice that the average time for all the runs was slightly greater than the average runtime without parallelism. This is fully attributed to the first iteration of the code with parallelism which took 94 seconds to run. This may be due to a setup time required by the processor to establish all of the threads needed to parallelize the code.

The performance difference by adding parallelism made the code run over six faster than the non-parallelized version and is a significant advantage to add to the matrix multiply code. See full output below:

Non-Parallelized code:

Dimension	Time	Gflop/s	GB/s
4	1.064538	1.878749	30.059965
52	0.284038	7.041325	112.661001
100	0.261746	7.648605	122.377453
148	0.298004	6.722802	107.564656
196	0.286005	7.002842	112.045275
244	0.280929	7.135943	114.174886
292	0.280373	7.281569	116.504898
340	0.276129	7.401623	118.425751
388	0.281638	7.466296	119.460521
436	0.286818	7.513192	120.210857
484	0.269222	7.580482	121.28749
532	0.27886	7.559199	120.946959
580	0.314914	7.434845	118.957319
628	0.336677	7.35639	117.702062
676	0.348444	7.092439	113.478865
724	0.327415	6.954513	111.272044
772	0.424303	6.506181	104.098766
820	0.343219	6.425835	102.813213

868	0.416062	6.287236	100.59566
916	0.491729	6.252018	100.032182
964	0.572063	6.263921	100.222644
1012	0.330631	6.269401	100.31027
1060	0.3807	6.256969	100.111369
1108	0.437488	6.218444	99.494998
1156	0.50078	6.169573	98.71307
1204	0.564335	6.18545	98.967116
1252	0.633855	6.192299	99.076708
1300	0.707604	6.209673	99.354702
1348	0.785456	6.237018	99.792227
1396	0.873335	6.230244	99.683852
1444	0.96242	6.257004	100.112007
1492	1.062957	6.249138	99.986158
1540	1.171535	6.234997	99.759905
1588	1.291331	6.202167	99.234631
1636	1.413715	6.194665	99.114606
1684	1.609905	5.932742	94.923846
1732	1.792208	5.798088	92.769383
1780	2.009148	5.614065	89.825021
1828	2.200164	5.552685	88.842937
1876	2.379719	5.548841	88.781441
1924	2.591126	5.497384	87.958128
1972	2.819207	5.440308	87.044914

Parallelized code:

Dimension	Time	Gflop/s	GB/s
4	94.344438	0.021199	0.339183
52	0.225561	8.866795	141.868333
100	0.091753	21.819249	349.105726
148	0.069648	28.764886	460.234324
196	0.064522	31.040864	496.64891
244	0.051755	38.733728	619.732592
292	0.056751	35.973	575.56209
340	0.059924	34.105957	545.68993
388	0.054689	38.448877	615.175232
436	0.060259	35.760767	572.166601
484	0.051408	39.696705	635.127644
532	0.055318	38.104173	609.649435
580	0.062625	37.384919	598.143896
628	0.061413	40.318032	645.081984
676	0.064039	38.590324	617.439507
724	0.056743	40.127675	642.036018
772	0.071196	38.772941	620.353395
820	0.059547	37.036672	592.580581
868	0.065639	39.850648	637.595113
916	0.076381	40.248806	643.975892
964	0.083091	43.11847	689.890508
1012	0.054232	38.220353	611.50836
1060	0.06235	38.202199	611.219551
1108	0.067143	40.516015	648.240815
1156	0.076305	40.489351	647.824469
1204	0.080875	43.160832	690.568114
1252	0.090361	43.435564	694.957317
1300	0.117513	37.390411	598.240056
1348	0.108474	45.161029	722.572499
1396	0.117183	46.431636	742.902396
1444	0.123226	48.868086	781.885601

1492	0.135585	48.991561	783.861443
1540	0.163745	44.608708	713.73672
1588	0.154154	51.954287	831.265379
1636	0.171566	51.043959	816.700499
1684	0.181761	52.546664	840.739526
1732	0.209827	49.52284	792.361048
1780	0.21587	52.250949	836.012786
1828	0.244748	49.915065	798.637
1876	0.26552	49.730897	795.689584
1924	0.273296	52.120354	833.920842
1972	0.304042	50.444507	807.110489

Peak Flop Rate Percentage:

The greatest flop rate attained with the parallelized code was 52.54 GFlop/s (0.05254 TFlop/s) on snappy3. Snappy3's Dual Xeon E5-2680 is noted to have a peak performance of 0.35 Tflop/s (https://cvw.cac.cornell.edu/MostOfMIC/xeon-vs-xeonphi). By this, our best run only achieved 15% of the peak flop rate for the processor.

NOTE: The full outputs from all 26 runs of the code can be found at the public nyu link here:
https://docs.google.com/spreadsheets/d/193hxfwjqvQx4tDJsB8lLzszSDgUsmKm6dl3lQwKkjU8/edit?usp=sharing

3. Approximating Special Functions Using Taylor Series Vectorization.

For this question I chose to implement SIN4_INTRIN function.

Snappy3 on CIMS uses the AVX instruction set, for which the registers can hold and work on four 64-bit doubles at a time. As the code works in chunks of four values, it can apply the same instruction set on each of the four values at one time.

I continued to declare variables using the 256-bit type (*_mm*256) to hold the four vector elements for each term in x1...x11. Then, I applied the intrinsic multiply (*_mm*256*_mul_pd*) and then add function (*_mm*256*_add_pd*) to aggregate the results for each term with the constant set of (*_mm*256*_set*1*_pd*). The set function will set all four vector elements to the argument provided. Finally, the store function (*_mm*256*_store_pd*). My full code is in my github repository. I was successful in improving the accuracy of the function to 12 digits.

Original Error (no modifications)

```
Intrin time:    18.1620      Error: 2.454130e-03
```

Final Error (with additional terms)

```
Intrin time:    13.9083      Error: 6.928125e-12
```

4. Pipelining and Optimization.

I chose to work on part (a) for carrying out loop unrolling.

My code is available in the repository as pipelining.cpp. I first implemented the naive approach in order to get a baseline for performance results. I tried arrays with sizes $n >= 100,000,000$ with using $NREPEAT = 10$. My results for the baseline are in the table below:

Naive Implementation:

Dimension (n)	Time (s)
100,000,000	3.484
150,000,000	5.265
200,000,000	6.989
250,000,000	9.169
300,000,000	12.978

Initially, I had tried compiling my code with the -O3 optimizer, which made the compute time 0 for my large arrays, and my flop rate went to infinity. Likely the compiler is able to optimize my code in ways that I could not write. For the purpose of this assignment, I switched my compiler optimizer flag to -O0 in order to fully capture the differences which I write for pipelining and unrolling.
Next, I followed all of the steps in pipelining and unrolling by a factor of 2. The results for this are below:

Pipelining and unrolling by 2:

Dimension (n)	Time (s)
100,000,000	1.786
150,000,000	2.700
200,000,000	3.617
250,000,000	5.079
300,000,000	6.566

By carrying out pipelining and unrolling by a factor of 2 I am able to see consistently a 50% increase in the speed for the time taken to do the computation. This is a very impressive increase. Unrolling will improve performance by leveraging more of what is cached. When the processor brings in the cache line, nearby data is loaded. Additionally, pipelining takes advantage of using less clock cycles, in attempting to do separate steps on the same clock cycle if they are clearly different to the compiler. In this case, we separated the add function from the multiply function as far as possible, by moving the add function to immediately before the next multiplication step. This enables the two steps to be done during the same clock cycle, and it did improve my runtimes by half.

In order to show the effect of unrolling on performance, I also unrolled by four, results are in the table below:

Pipelining and unrolling by 4:

Dimension (n)	Time (s)
100,000,000	1.083
150,000,000	1.654
200,000,000	2.223
250,000,000	3.288
300,000,000	5.435

Again, I observed a performance increase from the baseline and from unrolling by 2. This time, computations finished an additional 9% faster than when unrolling by a factor of 2. The M1 chip appears to be very performant in the types of computation with pipelining and unrolling.

I was surprised to observe such a significant improvement from the baseline on my Macbook M1 Air. On the class discussion, other classmates (on other machines) did not see an improvement as great as mine. I suspect that the Macbook Air M1 is better at performing some tasks than others.

Earlier, in question 2 regarding matrix multiplication, other classmates (on other machines) reported greater lift than I did. It is possible that the Macbook M1 Air is not as good at performing matrix multiplication, and that it is better at these pipelined inner product operations.