DD2356 High Performance Computing

Yuchen Gao,Yuze Cui

April 5, 2022

Introduction

This is the report for DD2356 High Performance Computing assignment 1, the benchmark and performance modeling.

Questions

Exercise1

Q1

Calculated Performance			
n	nrows	nnz.	time/s
10	100	460	4.09e-7
100	10000	49600	4.41e-5
1000	1000000	4996000	4.41e-3
10000	100000000	499960000	0.441

Q2

Measured Performance				
n	nrows	nnz.	time/s	floating operations per second
10	100	460	1e-5	9.2e8
100	10000	49600	1.8e-4	5.6e8
1000	1000000	4996000	1.8e-2	5.6e8
10000	100000000	499960000	1.724	5.8e8

Q3

The main reason for the performance difference is mainly because of the data fetching and storing overhead .

Q4

The bandwidth			
n	nrows	nnz.	bandwidth(byte/s)
10	100	460	6.72e8
100	10000	49600	3.97e9
1000	1000000	4996000	3.99e10
10000	100000000	499960000	4.2e9

Q5

Function	Best Rate byte/s	Avg time	Min time	Max time
Copy:	2.89e10	0.005676	0.005522	0.005884
Scale:	1.73e10	0.009429	0.009244	0.009613
Add:	2.16e10	0.011257	0.011113	0.011595
Triad:	2.07e10	0.011747	0.011568	0.011962

The performance of bandwidth in Q5 is far better than the Q4, first it is relevant to the compiler optimization '-O2' we use, and there is also a bandwidth collapse in the Q4 as the size of matrix exaggerating.
There also might be the parallel execution of The stream file.

Exercise2

Q1

The CPU we used is AMD EPYC 7742 64-Core Processor. Each cache size: L1d cache:32K, L1i cache:32K, L2 cache:512K, L3 cache:16384K

Q2

The image is shown below:

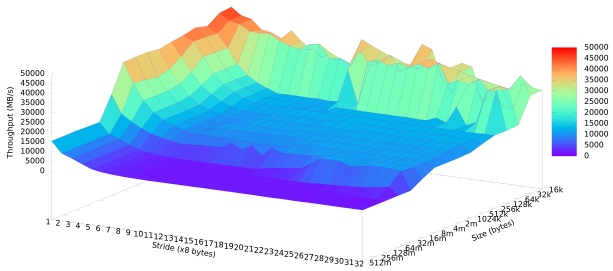

Figure 1: Memory mountain calculated on Dardel

Q3

16k-32k array size with stride=1-4 can get the highest performance. The reported highest bandwidth is 49682 MB/s.

Q4

The 128m-512m array size with stride=15-32 can get the lowest consistently lowest performance, with bandwidth around 800MB/s.

Q5

For array size larger than the cache size, cache prefetching mechanism is useful. Cache prefetching means fetching data from their original storage in main memory to a faster local memory or lower level of cache before it is needed by the processor. In hardware prefetching, there is a hardware which recognizes the next few elements that the program might need based on this stream and prefetches into the processor's cache.

Typically, the prefetching fetches the data in a stream, so the stride will affect spatial locality and the efficiency (i.e. A lot useless data been prefetched). If the hardware can do a strided prefetching, it need to calculate the next address of data to fetch, which will cost time so the performance is deteriorated.

Q6

Temporal locality: If at one point a memory location is used, and the same location will be used again in the near future. So the data stored in the cache can be reused.
Spatial locality: If the data is referenced at a time, the data nearly the location of the adjacent data is likely to be referenced. We can guess the data which will be fetched later, which can increase the performance.

Q7

Yes, because the array size can decide whether the data store will be flushed by new data in next period. If the previous data been flush, when it is referenced again, there is no temporal locality. So an increased array size will increased the temporal locality.

Q8

Yes. This is because when we change the read stride, the possibility that the data stored near the adjacent data will be fetched is changed. The stride is bigger, the nearest data is less likely to be referenced, so the spatial locality is lowered.

Q9

Using my laptop, the memory mountain is shown below:

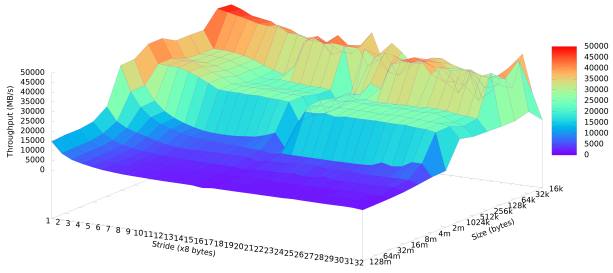

Figure 2: Memory mountain calculated on laptop

The memory mountain compiled by laptop has more spikes in the middle. The maximum bandwidth is nearly the same with Dardel's but the lowest are two times of Dardel's. And the bandwidth is obviously higher than Dardel when the array size is lower than 2M. I guess the reason is that my laptop has a bigger L1 and L2 cache: L1d cache:192KiB, L1i cache:128 KiB, L2 cache:5 MiB, L3 cache:8 MiB.

Exercise3

To switch the compiling environment:
module swap PrgEnv-cray PrgEnv-gnu

Q1

i

The average running time is 0.00000000 s.

ii

N less than 500,000,000	0.00000000 s
N=500,000,000	0.00000119 s. (Not always)
N=2,100,000,000	0.00000095 s (not always)

Q2

i

The execution time is like this because the arrays inside the loops are not used in the code, and the compiler optimized this in the execution file, '-O2'

optimization is a better way to optimize code than '-O1' or nothing, it makes the execution time shorter an size smaller. We can see in the assembly code, the instructions are implemented different ways, and we can see in the no '-O2' version, the instructions are mainly focused on the loop part of the code(line 13-17,line 20-24), however, the instructions mainly focus on the line 34-36 part, which is the timer, so the execution of the loop are ignored, since it is not used.
ii

Without '-O2' optimization: n=5000 Average execution time:0.00001407 s

Q3

i
srun -n 1 ./a.out 100
min dist = 9.54e-07,
max dist = 1.19e-06,
total time = 3.10e-06 The clock granularity is 9.54e-07s in the Dardel.
ii
The average clock granularity is 6e10-7s using the RDTSC.

Q4

For n=1000:

Minimum Running Time:

Time for Sparse Ax, nrows=1000000, nnz=4996000, T = 0.004885s

Average Running Time:

Time for Sparse Ax, nrows=1000000, nnz=4996000, T = 0.005130s

Exercise4

Q1

EVENT NAME	MSIZE = 64 Naive	MSIZE = 64 Optimized	MSIZE = 1000 Naive	MSIZE = 1000 Optimized
Elapsed time (seconds)	0.0297	0.0048	36.382873498	0.0255
Instructions per cycle	1.76	1.05	1.29	2.15
L1 cache miss ratio	14.08%	0.86%	56.30%	1.04%
L1 cache miss rate PTI	66.31	3.83	296	5.06
LLC cache miss ratio	-	-	-	-
LLC cache miss rate PTI	-	-	-	-

As the perf tool in Dardel do not provide the LLC cache monitoring service, so this is ignored from the table above, but we can make assumption that the optimization is a blocking procedure for the LLC, as the L1 is called much more often when the optimization is applied.

The time for matrices multiplication increases when the size of matrix increases. The optimization may be a good idea to minimize the execution time by optimizing the spatial locality and temporal locality, and this improves the reuse of data in the caches.

Exercise5

Q1

Ran the code on Dardel:"srun -n 1 perf stat -e cycles,instructions,L1-dcache-load-misses,L1-dcache-loads ./transpose.out"

EVENT NAME	MSIZE = 64	MSIZE = 128	MSIZE = 2048	MSIZE = 2049
Elapsed time (seconds)	0.074007558	0.188479984	29.051932283	1.183845017
Bandwidth /Rate	1.94e+04 MB/s	6.94e+03 MB/s	1.17e+02 MB/s	3.02e+03 MB/s
Instructions per cycle	1.21	1.12	0.02	0.48
L1 cache miss ratio	5.38%	18.11%	23.34%	39.46%
L1 cache miss rate PTI	25.98	146.16	2110.29	268.53
LLC cache miss ratio	-	-	-	-
LLC cache miss rate PTI	-	-	-	-

Q2

Cache size impact the performance most. Because the transpose code can increase the miss rate of cache and increase access memory if the size of matrix increases.

Q3

Using loop transformations:
Strip mining: Break a loop up into blocks of consecutive elements.So that the blocks size can be hold by L1, which will enhance the processing speed.
Loop reordering: To optimize access pattern.

Exercise6

Q1

For GCC compiler, "-ftree-vectorize" can turn on the auto vectorization, if using the optimization flag "-O3", this vectorization can be included.

Q2

From the online file, "-fopt-info-vec-all" can be used to report all details about the vectorization optimization information, which includes optimization and missed optimization report and notes.
If want to report information on which loops were or were not vectorized and why,using the flag:"-ftree-vectorizer-verbose-ftree-vectorizer-verbose=2" The user

can choose between 6 verbosity levels using the flag, which will decide how much to report.

Q3

According to the GNU online documetation, there are some special directives or command-line options:
"-ftree-vectorize-O3-maltivec-msse"/"-msse2-ffast-math-fassociative-math": enable vectorization of floating point reductions use.
"–param min-vect-loop-bound=X": A flag to limit vectorization to loops with large enough loop-bound. It prevents vectorization of loops whose vectorized loop-bound is equal or less than X.
"-ftree-slp-vectorize-O3-ftree-vectorize":Basic block vectorization

Q4

We can read from the report that these lines are not vectorized and the reasons of those:

Line	Code	Reason
18	`i = gettimeofday(&tp,&tzp);`	It is function call
19	`return((double)tp.tv_sec+(double) tp.tv_usec * 1.e-6 );`	Analysis failed with vector mode V2DI
26	`for (i = 0 ; i < MSIZE ; i++) {`	couldn't vectorize loop. Call rand() in the loop.
27	`for (j = 0 ; j < MSIZE ; j++) {`	couldn't vectorize loop. Call rand() in the loop.
28	`matrix_a[i][j] = (double) rand() / RAND_MAX;`	Call function rand()
29	`matrix_b[i][j] = (double) rand() / RAND_MAX;`	Call function rand()
39	`for (i = 0 ; i < MSIZE ; i++) {`	Multiple nested loops. Index is not consistent.
41	`for (k = 0 ; k < MSIZE ; k++) {`	outer-loop already vectorized.
52	`for (i = 0 ; i < MSIZE ; i++) {`	couldn't vectorize loop: memory index is not consistent.
54	`ave += matrix_r[i][j]/(double)(MSIZE*MSIZE);`	not vectorized: complicated access pattern.
71	`for (i = 0; i < TRIALS; i++) {`	
72	`memset(matrix_r,'0',sizeof(double)*MSIZE*MSIZE);`	Call function memset
64	`printf("1. Initializing Matrices \n");`	It is function call
66	`initialize_matrices();`	It is function call
68	`printf("2. Matrix Multiply \n");`	It is function call
69	`multiply_matrices();`	It is function call
70	`t1 = mysecond();`	It is function call
73	`multiply_matrices();`	It is function call
75	`t2 = mysecond();`	It is function call
77	`double sum = average_result();`	It is function call
78	`printf("3. Sum = %8.6f \n", sum);`	It is function call
79	`printf("4.time=%f\n",(t2-t1)/(double)TRIALS)`	It is function call

Figure 3: not vectorized code and reasons

The vectorization report for the code:

```
yuzec@uan01:~/DD2356/Assignment-I> gcc -g -O2 -ftree-vectorize -fopt-info-vec-all matrix_multiply.c -o matrix_multiply.out
matrix_multiply.c:18:7: missed: statement clobbers memory: gettimeofday (&tp, &tzp);
matrix_multiply.c:19:31: note: ***** Analysis failed with vector mode V2DI
matrix_multiply.c:19:31: note: ***** Skipping vector mode V16QI, which would repeat the analysis for V2DI
matrix_multiply.c:26:18: missed: couldn't vectorize loop
matrix_multiply.c:28:33: missed: statement clobbers memory: _1 = rand ();
matrix_multiply.c:27:20: missed: couldn't vectorize loop
matrix_multiply.c:28:33: missed: statement clobbers memory: _1 = rand ();
matrix_multiply.c:22:6: note: vectorized 0 loops in function.
matrix_multiply.c:28:33: missed: statement clobbers memory: _1 = rand ();
matrix_multiply.c:29:33: missed: statement clobbers memory: _4 = rand ();
matrix_multiply.c:33:1: note: ***** Analysis failed with vector mode V2DF
matrix_multiply.c:33:1: note: ***** Skipping vector mode V16QI, which would repeat the analysis for V2DF
matrix_multiply.c:39:18: missed: couldn't vectorize loop
matrix_multiply.c:39:18: missed: not vectorized: multiple nested loops.
matrix_multiply.c:40:20: optimized: loop vectorized using 16 byte vectors
matrix_multiply.c:41:22: missed: couldn't vectorize loop
matrix_multiply.c:41:22: missed: outer-loop already vectorized.
matrix_multiply.c:35:6: note: vectorized 1 loops in function.
matrix_multiply.c:46:1: note: ***** Analysis failed with vector mode V2DF
matrix_multiply.c:46:1: note: ***** Skipping vector mode V16QI, which would repeat the analysis for V2DF
matrix_multiply.c:52:18: missed: couldn't vectorize loop
matrix_multiply.c:54:25: missed: not vectorized: complicated access pattern.
matrix_multiply.c:53:20: optimized: loop vectorized using 16 byte vectors
matrix_multiply.c:48:8: note: vectorized 1 loops in function.
matrix_multiply.c:57:9: note: ***** Analysis failed with vector mode VOID
matrix_multiply.c:71:17: missed: couldn't vectorize loop
matrix_multiply.c:72:5: missed: statement clobbers memory: memset (&matrix_r, 48, 8000000);
matrix_multiply.c:60:5: note: vectorized 0 loops in function.
matrix_multiply.c:64:3: missed: statement clobbers memory: __builtin_puts (&"1. Initializing Matrices "[0]);
matrix_multiply.c:66:3: missed: statement clobbers memory: initialize_matrices ();
matrix_multiply.c:68:3: missed: statement clobbers memory: __builtin_puts (&"2. Matrix Multiply "[0]);
matrix_multiply.c:69:3: missed: statement clobbers memory: multiply_matrices ();
matrix_multiply.c:70:8: missed: statement clobbers memory: t1_11 = mysecond ();
matrix_multiply.c:72:5: missed: statement clobbers memory: memset (&matrix_r, 48, 8000000);
matrix_multiply.c:73:5: missed: statement clobbers memory: multiply_matrices ();
matrix_multiply.c:75:8: missed: statement clobbers memory: t2_13 = mysecond ();
matrix_multiply.c:77:16: missed: statement clobbers memory: sum_14 = average_result ();
matrix_multiply.c:78:3: missed: statement clobbers memory: printf ("3. Sum = %8.6f \n", sum_14);
matrix_multiply.c:79:3: missed: statement clobbers memory: printf ("4. time = %f\n", _2);
matrix_multiply.c:79:3: note: ***** Analysis failed with vector mode VOID
```

Figure 4: vectorization report

Bonus

i

- Time for Nbody problem when N=,500, time : T = 0.416265 /s;
- Time for Nbody problem when N=,1000, time : T = 1.707363 /s;
- Time for Nbody problem when N=,2000, time : T = 6.668668 /s;
- Time for Reduced Algorithom of Nbody problem when N=,500, time : T = 0.248409 /s;
- Time for Reduced Algorithom of Nbody problem when N=,1000, time : T = 1.220337 /s;

- Time for Reduced Algorithom of Nbody problem when N=,2000, time : T = 4.827926 /s;

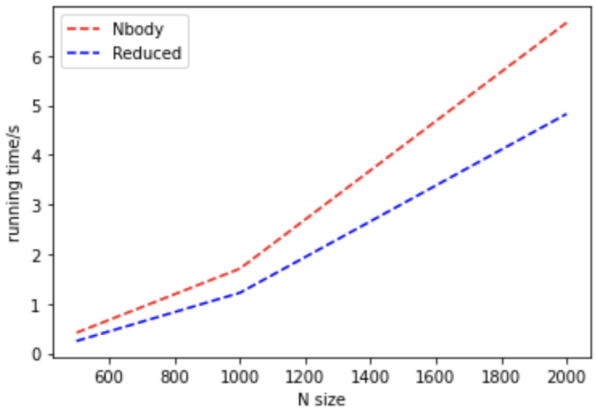

Figure 5: Performance plot.

The Reduced algorithm performs better than the simple one, and we can tell from the graph that when the N size is larger than 1000, there is a turning point where the running time increases faster than before.

- Time for Nbody problem when N=500 0.01% of all L1-dcache misses
- Time for Nbody problem when N=1000 0.03% of all L1-dcache missess
- Time for Nbody problem when N=2000 0.60% of all L1-dcache misses
- Time for Reduced Algorithom of Nbody problem when N=500 0.02% of all L1-dcache misses
- Time for Reduced Algorithom of Nbody problem when N=1000 0.27% of all L1-dcache misses
- Time for Reduced Algorithom of Nbody problem when N=2000 0.69% of all L1-dcache misses

The Simple algorithom performs better when we consider the cache reuse. As there are force_qk needs to be loaded in the procedure, and it consumes data loading.

ii

- **Performance model for neglecting the overhead:**

$$sum\_time = num\_of\_operations * 1/f \quad (1)$$

$$num\_of\_perations = n\_steps * ((N * ((N-1) * 18)) + (N * 10)) \quad (2)$$

- **Performance model for considering the overhead**

$$data\_move\_size = n\_steps * ((N * ((N-1) * 16 * sizeof(double))) + (N * 10 * sizeof(double))) \quad (3)$$

$$sum\_time = data\_move\_size * 1/b\_w + num\_of\_perations * 1/f \quad (4)$$

(f denotes the frequency of the computer, and the b_w denotes the bandwidth of the loading operation in computer)

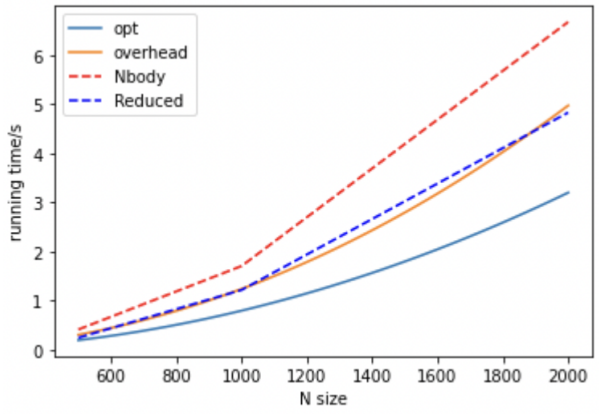

Figure 6: Comparison

The graph above illustrates the comparison of the performance model and the practical execution time, the 'opt' is the optimal time without considering the data movement. The 'overhead' is considering the data movement. The 'Nbody' and 'Reduced' are the practical performance of the two algorithms.
As we can tell from the graph, the performance model we give is the simple algorithm's model, and the one with overhead fits better. However, there is still a large gap between the reality and the model, since we consider the operations only take one cycle exactly to be executed. Also, the loading time varies when the different levels of cache are taken into consideration as the size of matrix increases.