CPSC 524: Parallel Programming Techniques

Assignment 1

1. Information on building and running

1.1 Modules

a) Loaded Module List
1) StdEnv (S) 2) Langs/Intel/2015_update2

b) Intel C/C++ Compiler (ICC Version 15.0.2 20150121)
/gpfs/apps/hpc/Langs/Intel/2015_update2/composer_xe_2015.2.164/bin/intel64/icc

1.2 Commands

On the Grace cluster, run the ***run.sh*** script to compile and execute all the programs. You can also run ***make*** to compile them manually. And then run *exercise1a*, *exercise1b*, *exercise1c*, *exercise1d*, *exercise1_division*, *exercise2* to execute them separately.

1.3 Outputs

The output is as follows:

```
[hs746@grace2 Homework1]$ ./run.sh
====== Make ======
icc -c exercise1.c -g -O0 -fno-alias -std=c99 -o exercise1a.o
icc -o exercise1a -g -O0 -fno-alias -std=c99 exercise1a.o timing.o
icc -c exercise1.c -g -O1 -fno-alias -std=c99 -o exercise1b.o
icc -o exercise1b -g -O1 -fno-alias -std=c99 exercise1b.o timing.o
icc -c exercise1.c -g -O3 -no-vec -no-simd -fno-alias -std=c99 -o exercise1c.o
icc -o exercise1c -g -O3 -no-vec -no-simd -fno-alias -std=c99 exercise1c.o timing.o
icc -c exercise1.c -g -O3 -xHost -fno-alias -std=c99    -o exercise1d.o
icc -o exercise1d -g -O3 -xHost -fno-alias -std=c99    exercise1d.o timing.o
icc -c exercise1_division.c -g -O0 -fno-alias -std=c99 -o exercise1_division.o
icc -o exercise1_division -g -O0 -fno-alias -std=c99 exercise1_division.o timing.o
icc -c exercise2.c -g -O3 -xHost -fno-alias -std=c99    -o exercise2.o
icc -c dummy.c -g -O3 -xHost -fno-alias -std=c99    -o dummy.o
icc -o exercise2 -g -O3 -xHost -fno-alias -std=c99    exercise2.o timing.o dummy.o

====== Run Exercise 1 with Compiler Option (a) ======
Pi = 3.141593
Runtime = 5.476347
MFlops = 913.017387
====== Run Exercise 1 with Compiler Option (b) ======
Pi = 3.141593
```

Runtime = 5.418086
MFlops = 922.835103
====== Run Exercise 1 with Compiler Option (c) ======
Pi = 3.141593
Runtime = 5.407856
MFlops = 924.580834
====== Run Exercise 1 with Compiler Option (d) ======
Pi = 3.141593
Runtime = 2.740908
MFlops = 1824.212901

====== Run Exercise 1 Division Operation Latency Test ======
a = 0.840188
b = 0.394383
Runtime = 20.169607
MFlops = 99.159097

====== Run Exercise 2 ======

N	MFlops
9	1291.639664
19	2962.810368
40	4284.489139
85	4964.106083
180	5414.975069
378	5625.043547
794	5484.705519
1667	2846.375647
3502	2673.422080
7355	2503.035363
15447	1549.323018
32439	1547.405970
68122	1549.539445
143056	1543.460473
300419	1543.988285
630880	1364.923687
1324849	698.658181
2782184	690.015121
5842587	689.767211
12269432	691.187495
25765808	691.970591
54108198	693.179498
113627216	694.120695

2. Exercise 1: Division Performance

2.1 Result

Compile Option	*Runtime*	*MFlops*
a) -g -O0 -fno-alias -std=c99	5.476347	913.017387
b) -g -O1 -fno-alias -std=c99	5.418086	922.835103
c) -g -O3 -no-vec -no-simd -fno-alias -std=c99	5.407856	924.580834
d) -g -O3 -xHost -fno-alias -std=c99	2.740908	1824.212901

2.2 Analysis

The result shows that there are minor improvements among first three compile options, and the significant improvement for the last one.

The first option is using -O0, which means there will be no optimization. It if often used to make sure the application correctness and for easy debugging.

The second one is using -O1, this option enables global optimization which includes data-flow analysis, code motion, etc. When using the O1 option, the compiler's auto-vectorization functionality is disabled. Since our code doesn't really need much global optimization, this compile option didn't improve the performance.

The third one is using -O3, which performs O2 optimizations (contains basic loop optimizations) and enables more aggressive loop transformations. The O3 option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets, which should be really helpful in our case. However, we add -no-vec and -no-simd after -O3. -no-vec will turn off auto-vectorization in plain source code with no simd directives. And -no-simd in effect removes the effect of simd directives; otherwise -no-vec doesn't apply within the scope of those directives. Those two options forbade -O3's most useful optimization. In fact, if you remove these two options, the runtime can be almost as good as the last compile option.

The last one is still using -O3, but removed -no-vec and -no-simd. In addition, the -xHost option enable the processor-specific optimizations, which further improve the performance. When using this option, for loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, Intel SSE provides SIMD instructions for many arithmetic operators including addition, subtraction, multiplication, division, etc[1]. And since AVX 1.0 on the Ivy Bridge increase the floating-point vector width from 128-bit to 256-bit[2], which will lead to 2X flops. Thus, the performance is increased two times.

2.3 Latency of Division Operation

We create another C file *exercise1_division.c*, which replace the original Pi Calculation inside the for-loop with the follow:

```
for (int i = 0; i < N; i++) {
    a = a / b;
    a = b / a;
}
```

1 Intel Automatic Vectorization, https://software.intel.com/en-us/node/522569

2 Introduction to Intel AVX, https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

Where a and b are random generated double number between 0 and 1. The reason to execute two division operation in the for-loop is to prevent division losing precision. Since during the experiment, if there's only a = a / b in the for loop, the results may vary a lot among each other, thus we use this way to make sure that the divisor and dividend are always in the proper range.

The current for-loop will guarantee that for each two loop, a and b will return to their original value. Suppose $a = a_0$ and $b = b_0$, after the first loop, $a = b_0/(a_0/b_0) = b_0^2/a_0$ and b remains the same. After the second loop, $a = b_0/((b_0^2/a_0)/b_0) = a_0$ and $b = b_0$.

When $N = 1{,}000{,}000{,}000$, the runtime is 20.125756 seconds. The estimated latency is as follows:

$$latency = \frac{cycles}{instruction} = \frac{clock\_rate \times time}{loop\_num \times instruction\_per\_loop}$$
$$= \frac{2.2 \times 1{,}000{,}000{,}000 \times 20.125756}{1{,}000{,}000{,}000 \times 2} = 22.138332$$

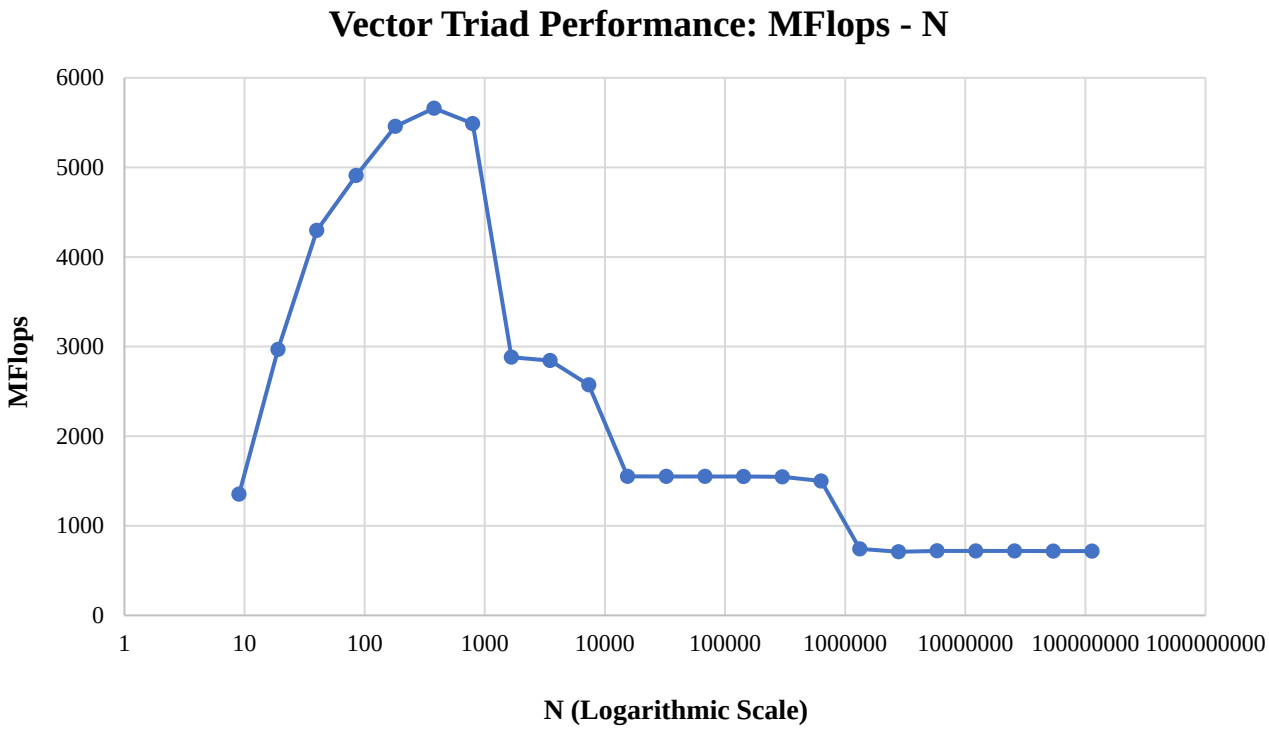
3. Exercise 2: Vector Triad Performance

3.1 Result

The result is as follows:

N	9	19	40	85	180	378	794	1667
MFlops	1352.716	2966.813	4296.538	4909.161	5457.711	5660.343	5489.176	2881.603
N	3502	7355	15447	32439	68122	143056	300419	630880
MFlops	2844.168	2572.604	1551.792	1550.17	1550.646	1549.732	1546.544	1499.366
N	1324849	2782184	5842587	12269432	25765808	54108198	113627216	
MFlops	741.9603	710.1452	720.0418	718.7717	718.5987	717.6529	717.4806	

The plot of MFlops and N is as follows:



3.2 Analysis

The initial raising stage is due to the parallel loop overhead[3]. The barrier overhead dominates when the loop lengths is short[4]. As the loop length grows, such overhead became negligible and the program reach its peak performance. After that, we can see three major drawback since then, which is caused by the cache size.

According to the CPU information on the cluster, we can know the cache size as follows:
L1 Cache = 32KB, L2 Cache = 256 KB, L3 Cache = 25MB.

The first significant decrease happened between N = 794 and N = 1667. At this time the program required four 794-length double array, which need $4 \times 794 \times 8 = 25408\text{bytes} = 24.8\text{KB}$. As the size increased to 1667, it required $4 \times 1667 \times 8 = 53344\text{bytes} = 52.1\text{KB}$, which is larger than the 32KB. Thus, we saw the first significant decrease at this point. The bandwidth when N = 794 is $5489.176 \times 12 / 1000 = 64.5\text{GB/sec (without store)}$. The reason of 12 bytes/flops is that we count 2 flops per loop, however there are actually 24 bytes for each loop, so it's equivalent to 12 bytes/loop. And if we look at it with store, there will be additional 8 bytes, which leads to 16 bytes/loop. Thus, the bandwidth will be $5489.176 \times 16 / 1000 = 87.8\text{GB/sec(with store)}$.

The second significant decrease happened between N = 7355 and N = 15447. The size required is 4×7355×8=235360bytes=229.8KB and 4×15447×8=494304bytes=482.7KB. Since it exceeded the size of L2 Cache, the performance decreased significantly. The bandwidth when N = 7355 is $2572.604 \times 12 / 1000 = 30.9\text{GB/sec(without store)}$ and $2572.604 \times 16 / 1000 = 41.2\text{GB/sec(with store)}$

The last significant decrease happened between N = 630880 and N = 1324849. The size required is $4 \times 630880 \times 8 = 20188160\text{bytes} = 19.3\text{MB}$ and $4 \times 1324849 \times 8 = 42395168\text{bytes} = 40.4\text{MB}$. Since it exceeded the size of L3 Cache, the performance decreased significantly. The bandwidth when N = 630880 is $1499.366 \times 1.5 / 1000 = 18\text{GB/sec (without store)}$ and $1499.366 \times 16 / 1000 = 24\text{GB/sec(with store)}$

3 Ingredients for good parallel performance on multicore-based systems
4 Georg Hager's Blog, https://blogs.fau.de/hager/archives/6883