HPC_Project 2
Performance Optimization via Cache Reuse

Sihuan Li 861194722

Part1

Answer:

1.1 Large matrix (10000*10000)

1.1.1 ijk and jik:

For matrix A, the read happens in a continuous order of a row. Each element will be read 10000 times but only 1/10 of them will have cache misses. Specifically, the elements who have 10000 cache misses are:

$$a_{ij} \text{ where}$$
$$i \in \{x \in N | x \leq 9999\}$$
$$j \in \{x \in N | x \bmod 10 = 0, x \leq 9990\} \qquad \textbf{(1.1.1)}$$

The rest of the elements in matrix A don't have cache misses. Or their cache misses are 0.

For matrix B, the read happens in a continuous order of a column. But the elements taking a free ride to the cache will not be used for the next time when it is being read cause the matrix is so large and the cache is much smaller. So, each element in B will have cache misses. Each element will be read for 10000 times. So, each element in B will have 10000 cache misses.

For matrix C, though it is read continuously in row, the inner loop has made the cache remove Cij's cache status. So, each element in C will have cache miss. As for the number of cache misses for each element, because each element in C will only be read once, it will be 1 for each element.

In conclusion, some specific elements (given above) in A will have 10000 cache misses and the other, 0. Each element in B will have 10000 cache misses. Each element in C will have 1 cache miss.

Furthermore, we can compute the miss rate:

$$Miss\ rate = \frac{10000^3 + 10000^3 * 0.1 + 1 * 10000^2}{2 * 10000^3 + 10000^2} = \frac{11001}{20001} \approx \frac{11}{20}$$

$$= 55\%$$

A		B	C	Miss rate
a_{ij} (1.1.1)	Rest			
10000	0	10000	1	55%

Table 1.1.1 cache misses and miss rate of ijk, jik

1.1.2 ikj and kij

The only difference with 1.1.1 is that we fix the element in A here which make the reading order of B and C are both continuous in row. But we also use the analysis method as we use in 1.1.1. and we get the conclusion as below.

For matrix A, each element in A will have 1 cache miss. For matrix B and C, b_{ij}, c_{ij} will have 10000 cache misses if i,j satisfy the condition (1.1.1). The rest element of B and C will have no cache miss.

Furthermore, we can compute the miss rate:

$$Miss\ rate = \frac{2 * 0.1 * 10000^3 + 10000^2}{2 * 10000^3 + 10000^2} = \frac{2001}{20001} \approx \frac{1}{10} = 10\%$$

A	B		C		Miss rate
	b_{ij} (1.1.1)	Rest	c_{ij} (1.1.1)	Rest	
1	10000	0	10000	0	10%

Table 1.1.2 cache misses and miss rate of ikj and kij

1.1.3 jki and kji

This will make the reading order of matrix A and C continuous in column. Using the same analysis as 1.1.1, we get the conclusion below.

For matrix A and C, each element of A and C will have 10000 cache misses.

For matrix B, each element of B will have 1 cache miss.

Furthermore, we can compute the miss rate:

$$Miss\ rate = \frac{2 * 10000^3 + 10000^2}{2 * 10000^3 + 10000^2} = 100\%$$

A	B	C	Miss rate
10000	1	10000	100%

Table 1.1.3 cache misses and miss rate of jki and kji

1.2 Small matrix (10*10)

Analysis:

If the matrices are 10*10, totally, there will be 300 doubles in A, B and C. With a cache whose capacity is 60*10 doubles, all the elements can be held in the cache.

1.2.1 ijk and jik

Based on the analysis above, we know that only 10 elements of A, B and C will have 1 cache miss. The other elements will have no cache miss. We define $I = \{0,1,2,3,4,5,6,7,8,9\}$. We know that $a_{i0}, b_{i0}, c_{i0}\ i \in I$ will have 1 cache miss.

Furthermore, we can compute the miss rate:

$$Miss\ rate = \frac{10 + 10 + 10}{2 * 10^3 + 100} = \frac{1}{70} = 1.43\%$$

A		B		C		Miss rate
$a_{i0}, i \in I$	rest	$b_{i0}, i \in I$	rest	$c_{i0}, i \in I$	rest	
1	0	1	0	1	0	1.43%

Table 1.2.1 cache misses and miss rate of ijk and jik

1.2.2 ikj and kij

The only difference is that matrix A will be read once which means A is handled in the outer loop. But the analysis is the same since the cache is big enough to hold all the elements of A, B and C. So we also get the table as Table 1.2.1. The miss rate is the same as well, 1.43%.

1.2.3 jki and kji

We can also give the result as Table 1.2.1. The miss rate is also the same, 1.43%

Part 2

2.1 ijk and jik

Based on the analysis and result in Part1, we it's easier to analyze Part2. For each block in matrix A, the first element of each row will have 1 cache miss. Totally, each block will be read for $\frac{10000}{10} = 1000$ times. So the element in A whose index satisfy condition (1.1.1) will have 1000 cache misses. And the rest of the elements will have no cache miss. For matrix B, it also has the same cache miss situation as matrix A does. For matrix C, when blocks of A and B move to the next iteration, the element of C will stay in the cache. So it will have only 1 cache miss for the element of C whose index satisfy condition (1.1.1) and no cache miss for the rest of the elements in C. So, we get the result.

Furthermore, we can compute the miss rate:

$$Miss\ rate = \frac{2 * 10000^2 * 0.1 * 1000 + 10000^2 * 0.1}{2 * 10000^3 + 10000^2 * 1000}$$

$$= \frac{2001}{210000} \approx \frac{2}{210} = 0.95\%$$

A		B		C		Missrate
a_{ij} (1.1.1)	rest	b_{ij} (1.1.1)	rest	c_{ij} (1.1.1)	rest	0.95%
1000	0	1000	0	1	0	

Table 2.1 cache miss and miss rate using block of ijk and jik

2.2 ikj and kij

The only difference with 2.1 is that we fix a block in A this time. In 2.1, we fix a block in C. This will exchange the miss situation of A and C with each other. B will stay the same. Furthermore, the miss rate will be the same with that in 2.1. So, we can conclude it below.

A		B		C		Missrate
a_{ij} (1.1.1)	rest	b_{ij} (1.1.1)	rest	c_{ij} (1.1.1)	rest	0.95%
1	0	1000	0	1000	0	

Table 2.2 cache miss and miss rate using block of ikj and kij

2.3 jki and kji

The only difference with 2.1 is that we fix a block in B this time. Using the same analysis method, we can give the result below.

A		B		C		Missrate
a_{ij} (1.1.1)	rest	b_{ij} (1.1.1)	rest	c_{ij} (1.1.1)	rest	0.95%
1000	0	1	0	1000	0	

Table 2.3 cache miss and miss rate using block of jki and kji

Part 3

3.1 Run time report

We run the program “part3-1.c” and “part3-2.c” using default GCC compiler whose version is 4.4.7 without any compiling parameter. The matrix dimension is 2048. The execution time is measured below.

Loop order	Time (s)	Error
ijk	284.760000	0.000000e+00
jik	287.940000	0.000000e+00
ikj	130.430000	0.000000e+00
kij	109.980000	0.000000e+00
jki	364.350000	0.000000e+00
kji	359.850000	0.000000e+00

Table 3.1.1 Execution time and error without cache block

3.1.1 Analysis:

We can see from Table 3.1.1 that the experiment results are consistent with the miss rate I computed in Part1, 55% for ijk and jik, 10% for ikj and kij, 100% for jki and kji. The naïve version of ijk can have a more than 2 times faster by reducing the miss rate from 55% to 10%. But the bad version such as jki and kji will lead to worse performance which increases the miss rate from 55% to

100%.

Loop order	Time (s)	Error
Ijk(naive)	229.110000	
ijk	111.010000	0.000000e+00
jik	97.440000	0.000000e+00
ikj	123.130000	0.000000e+00
kij	113.010000	0.000000e+00
jki	143.400000	0.000000e+00
kji	153.510000	0.000000e+00

Table 3.1.2 Execution time and error with cache block size of 10*10

Note that since 2048 is not a multiple of 10, we choose 2050 here. But we use the naïve ijk version as the standard to check the speedup of block size of 10.

3.1.2 Analysis:

Though we use a 2050*2050 matrix, we still speedup the program by using cache block. We see from Table 3.1.2 that each version has a better performance than the version without cache block does. The best version will be up to 2.5 times faster. Another phenomenon is that with cache block, the different versions have less performance difference with each other. This is because all of the 6 versions have a miss rate, 0.95%, when we use cache block.

3.2 Changing block size

In this section, we use a 2048 matrix to see the trends when we change block size. Since cache block will make the difference of different versions less like we said in 3.1.2. We only choose the version of ijk with cache block to find the optimal block size.

size	naive	4	8	16	32	64	128	256
time	289.45	160.84	114.07	95.49	89.42	86.32	86.39	128.23
error		0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 3.2 execution time with changing block size of ijk version

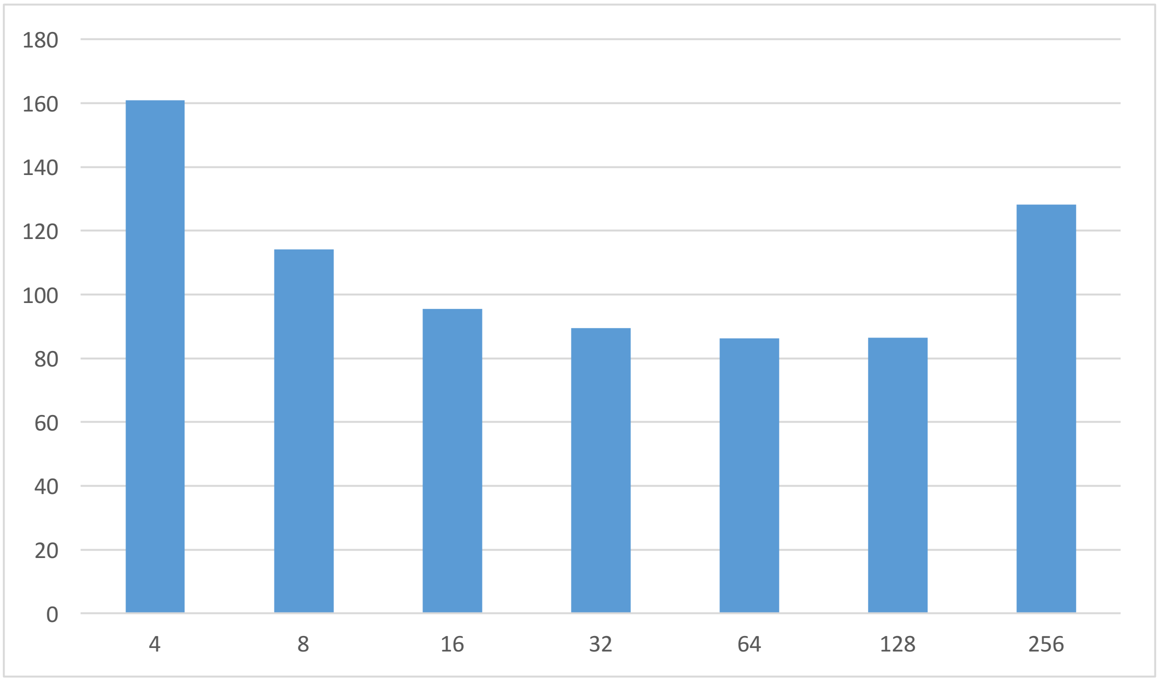

Figure 3.2 execution time with changing block size of ijk version

Results and analysis:

From Table 3.2 and Figure 3.2, we can see that the optimal block size is 64. It's natural we could have this kind of trend. When the block size is small, it will be more like the naïve version. But when the block size is too large, the benefit of blocking will be gone. It will be more like a large matrix without block. So the trend displayed in Figure 3.2 is expected. Note that we also check the correctness when we change block size. It gives perfect accuracy with the naïve one.

Part 4

In this part, we use cache block together with register block to speedup our program. As mentioned in section 3.2, the optimal cache block size is 64. As for the register block size, we use 2*2 here. The matrix is still 2048*2048. Note that since cache block will flatten the performance difference of different loop order, we can choose only 1 order, ijk. We get tables below.

4.1 Results

Gcc4.4.7	Naïve (ijk)	Cache block (64)		Rgst block (2)	
	Time	Time	Error*10^{-8}	Time	Error*10^{-8}
O0	449.89	82.48	0.0	41.99	4.097819
O1	317.56	32.87		9.14	4.284084
O2	372.00	31.11		8.67	4.190952
O3	341.00	28.78		8.01	4.377216

Table 4.1.1 Execution time and error with GCC-4.4.7

Gcc4.7.2	Naïve (ijk)	Cache block (64)		Rgst block (2)	
	Time	Time	Error*10^{-8}	Time	Error*10^{-8}
O0	455.77	96.46	0.0	42.42	4.190952
O1	407.01	29.92		8.27	4.284084
O2	301.54	29.16		**8.22**	4.656613
O3	125.99	14.99		8.24	4.470348

Table 4.1.2 Execution time and error with GCC-4.7.2

4.2 Analysis

4.2.1 Performance only using cache block

In each individual Table 4.1.1 or Table 4.1.2, we see that using 64*64 cache block, we can reduce the execution time of ijk naïve program from 450 seconds to 80 seconds around without compiler optimization. It's nearly 6 times faster. We gain the performance improvement cause we use blocks trying to avoid cache misses. This is consistent with the miss rates we compute in Part1 and Part2.

4.2.2 Performance using both cache and register block

Also, in each individual Table 4.1.1 or Table 4.1.2, we see that by using cache and register blocks together the execution time, 40 seconds, get halved than that of only using cache block, 80 seconds. In another words, we totally speedup our naïve program nearly 12 times faster. Using register block, we can use each element twice for each read and this will reduce the accessing to cache or memory. Therefore, our program gets improved.

4.2.3 Performance using compiler optimization

What we can see from Table 4.1.1 and Table 4.1.2 is that if we use compiler optimization no matter with O1, O2 or O3 and no matter the version of compiler, the speedup is significant. With only O1, we can make “Rgst block” version 5 times faster around, reducing execution time from 42 seconds to 9 seconds. If we compare it with the naïve version, we reduce the execution time from several minutes to several seconds. It’s incredible.

As for the version of compiler, the 4.7.2 version might be more reliable because the performance is increasing with higher optimization level. However, the 4.4.7 version give poorer performance when using O2 than O1 for the naïve ijk version. Another difference is that O3 optimization of gcc-4.7.2 is amazingly better than that of gcc-4.4.7. At least for naïve and cache block version, it does much better than gcc-4.4.7.

Another trend I found in the table is that the optimization level of compiler brings much less improvement for “Rgst block” version no matter it’s gcc-

4.7.2 or gcc-4.4.7. I think it’s because the “Rgst block” program we wrote has been well optimized already and there will be less chance to get it improved by using compiler.

Part5 Instructions and notes

5.1 Environment and compilers

All the work is done on Tardis cluster with one single node of AMD chips. All the programs in Part3 got compiled by gcc-4.4.7, the default compiler on Tardis, without any optimization parameters.

In Part4, we use both gcc-4.4.7 and gcc-4.7.2 to compile the same program. We also use different optimization parameters upon both of the compilers to compare the performance.

5.2 How to run

To finish the tasks in Part3, we need open the directory Sihuan_HPC_prjct2/part3.

“part3-1.c” --- Measure the execution time of the naïve 6 algorithms;

“part3-2.c” --- Measure the execution time of 6 algorithms improved by cache block;

“part3-3-findblk.c” --- Find the optimal cache block size from 4, 8, 16, 32, 64, 128, and 256.

Then, compile them below:

```
gcc –o part3-1exe part3-1.c
gcc –o part3-1exe part3-2.c
gcc –o part3-3-findblk-exe part3-3-findblk.c
```

Then, using the scripts to sub all the 3 programs:

```
qsub part3-1.job
qsub part3-2.job
qsub part3-3-findblk.job
```

Finally, we can get the result from output files by:

```
cat part3-1.out
cat part3-2.out
cat part3-3-findblk.out
```

To finish tasks in Part4, we need open the directory Sihuan_HPC_prjct1/part4.

"part4.c" --- Measure the execution time with only cache block and together with register block using different optimization level under different compiler, gcc-4.4.7 and gcc-4.7.2.

To compile the program, we need input:

```
gcc –O0 -o part4-447-O0-exe part4.c

gcc –O1 -o part4-447-O1-exe part4.c

gcc –O2 -o part4-447-O2-exe part4.c

gcc –O3 -o part4-447-O3-exe part4.c
```

```
module load gcc-4.7.2

gcc –O0 -o part4-472-O0-exe part4.c

gcc –O1 -o part4-472-O1-exe part4.c

gcc –O2 -o part4-472-O2-exe part4.c

gcc –O3 -o part4-472-O3-exe part4.c
```

Then, we use qsub to submit all the job files:

```
qsub part4-447-O0.job

qsub part4-447-O1.job

qsub part4-447-O2.job

qsub part4-447-O3.job

qsub part4-472-O0.job

qsub part4-472-O1.job
qsub part4-472-O2.job

qsub part4-472-O3.job
```

Finally, we can get the results:

```
cat part4-447-O0.job

cat part4-447-O1.job

cat part4-447-O2.job

cat part4-447-O3.job

cat part4-472-O0.job

cat part4-472-O1.job

cat part4-472-O2.job

cat part4-472-O3.job
```