Project 2

FEBRUARY 25

High Performance Computing
Authored by: Kayhan Gavahi
Instructor: Dr. Dingwen Tao

THE UNIVERSITY OF ALABAMA

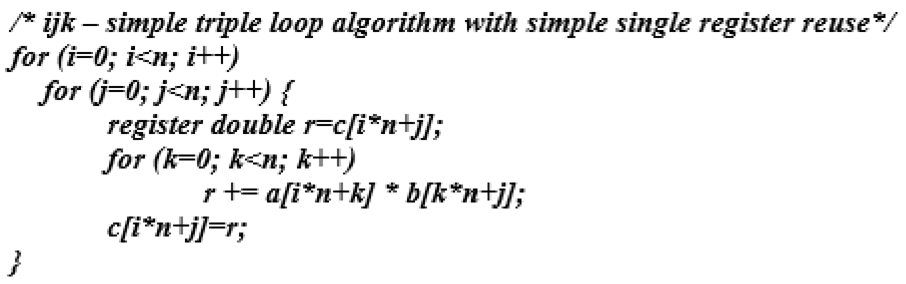
Part #1.

When matrix-matrix multiplication is performed using the simple triple-loop algorithm with single register reuse, there are 6 versions of the algorithm (ijk, ikj, jik, jki, kij, kji). Calculate the number of read cache misses for each element in each matrix for each version of the algorithm when the sizes of the matrices are 10000X10000 and 10X10 respectively. What is the percentage of read cache miss for each algorithm?

Solution:

First, lets solve the problem for n=10:

The data cache has 60 lines and each line can contain 10 doubles. Therefore, we can fetch all the values in each row of the matrices into the cache in each attempt. Now lets consider the ijk algorithm (the code is below):



```
/* ijk – simple triple loop algorithm with simple single register reuse*/
for (i=0; i<n; i++)
   for (j=0; j<n; j++) {
         register double r=c[i*n+j];
         for (k=0; k<n; k++)
                r += a[i*n+k] * b[k*n+j];
         c[i*n+j]=r;
}
```

Ijk and jik move row-wise for A and col-wise for B. But since our cache storage is big enough, only the first attempts for the first columns of A, B and C will be missed and this is due to the fact that the cache is empty at the start of the program (Cold Miss). Meaning:

A[0][0], A[1][0], A[2][0], …, A[10][0] $\rightarrow$ cold miss (10 misses for A in total)
B[0][0], B[1][0], B[2][0], …, B[10][0] $\rightarrow$ cold miss (10 misses for B in total)
C[0][0], C[1][0], C[2][0], …, C[10][0] $\rightarrow$ cold miss (10 misses for C in total)

The total number of calls for all the elements in A, B and C are n^3, n^3 and n^2, respectively.

Meaning the miss rate will be:

$$miss\ rate = \frac{10+10+10}{n^3+n^3+n^2} = \frac{30}{2100} = 0.0143$$

For kij and ikj, the A matrix does not appear in the most inner loop and they move row-wise for both B and C. But again, there will be cache misses for each first attempts of each row of each matrix and they will be cold missed. The miss rate will be same for these algorithms.

Jki and kji move col-wise for both A and C, but again, the cache is big enough for storing all the rows of A, B and C together (10+10+10=30<60) and only the first attempts will be cold missed. Thus, same misses and miss rate will occur for these algorithms too.

n=10000:

ijk:

To better understand what will happen if the cache size is not big enough to store all the data, I considered a smaller case and tracked each code to get the formula. In my case n=5 and the cache can hold 4 lines, each of each storing 3 doubles. As we can see in table 1, A0 and B0 will be cold missed. After that since we are moving row-wise on A, A1 and A2 will be still in the cache (cache can store 3 doubles in each line). But for B, B0, B5, B10, B15 and B20 (the first column) will be missed. Matrix C is not the most-inner loop and therefore it will be called only on 5, 10, …, iterations (colored in yellow) and every time it is not in the cache and will be missed.

Conclusion:

All the elements in matrix C will be missed upon memory request: C[i][j]=1 → $n^2 = 10^8$

For Matrix A: A[i][0], A[i][10], A[i][20], …, A[i][9990]=1 → $1000 \times n^2 = 10^{11}$

For matrix B: always miss: B[k][j]=1 → $n^2 \times n = 10^{12}$

Total memory requests: $n^3 + n^3 + n^2$

$$miss\ rate = \frac{n^2 + 1000 \times n^2 + n^3}{n^3 + n^3 + n^2} = 0.55$$

Jik:

The same thing will happen for this algorithm since the most-inner matrices are the same and we are still going row-wise for A and col-wise for B. Using the same procedure on my small case, proves this and the miss rate will be the same as ijk.

$$miss\ rate = \frac{n^2 + 1000 \times n^2 + n^3}{n^3 + n^3 + n^2} = 0.55$$

				a	b	c
iteration	i	j	k	in+k	kn+j	in+j
1	0	0	0	0	0	0
2	0	0	1	1	5	0
3	0	0	2	2	10	0
4	0	0	3	3	15	0
5	0	0	4	4	20	0
6	0	1	0	0	1	1
7	0	1	1	1	6	1
8	0	1	2	2	11	1
9	0	1	3	3	16	1
10	0	1	4	4	21	1
11	0	2	0	0	2	2
12	0	2	1	1	7	2
13	0	2	2	2	12	2
14	0	2	3	3	17	2
15	0	2	4	4	22	2
16	0	3	0	0	3	3
17	0	3	1	1	8	3
18	0	3	2	2	13	3
19	0	3	3	3	18	3
20	0	3	4	4	23	3
21	0	4	0	0	4	4
22	0	4	1	1	9	4
23	0	4	2	2	14	4
24	0	4	3	3	19	4
25	0	4	4	4	24	4

kij:

for this algorithm B and C are both row-wise and therefore will be missed every 10 columns. for A (which is fixed) the cache will be always full the B and C elements and therefore all the elements will be missed upon request:

For Matrix B: B[i][0], B[i][10], B[i][20], …, B[i][9990]=1 → $1000 \times n^2 = 10^{11}$

For Matrix C: C[i][0], C[i][10], C[i][20], …, C[i][9990]=1 → $1000 \times n^2 = 10^{11}$

For Matrix A: A[i][j]=1 → $n^2 = 10^8$

$$miss\ rate = \frac{1000 \times n^2 + 1000 \times n^2 + n^2}{n^3 + n^3 + n^2} = 0.10$$

ikj:

same as kij:

$$miss\ rate = \frac{1000 \times n^2 + 1000 \times n^2 + n^2}{n^3 + n^3 + n^2} = 0.10$$

jki and **kji:**

A and C are col-wised → all the elements will be missed upon request. B is fixed always missed too.

$$miss\ rate = \ 1$$

Part #2.

If matrices are partitioned into block matrices with each block being a 10 by 10 matrix, then the matrix-matrix multiplication can be performed using one of the 6 blocked version algorithms (ijk, ikj, jik, jki, kij, kji). Assume the multiplication of two blocks in the inner three loops uses the same loop order as the three outer loops in the blocked version algorithms. Calculate the number of read cache misses for each element in each matrix for each version of the blocked algorithm when the size of the matrices is 10000. What is the percentage of read cache miss for each algorithm?

Solution:

I used the same strategy to derive the formulas for this part. Table 2 shows some sample approach. As we can see by using blocks Matrix C will not be always missed and the number of misses for A and B has also be reduced.

ijk & **jik:**

For this part we should consider the loop iteration and storing in cache for blocks.

Since the cache is big enough to have 3 blocks in the cache therefore in each block we have (For A and B) $B^2/10$ misses. Each row will have n/B blocks. Thus:

For A and B:

$$total\ misses = \frac{nB}{10} \times \left(\frac{n}{B}\right)^2 = \frac{n^3}{10B} = 10^{10}$$

For C:

C[i][0B], C[i][10B], C[i][20B],… will be missed therefore:

$$total\ misses = \frac{B^2}{10} \times \left(\frac{n}{B}\right)^2 = \frac{n^2}{10} = 10^7$$

$$miss\ rate = \frac{2 \times 10^{10} + 10^{7}}{n^3 + n^3 + n^2} = 0.01$$

								a	b	c
n		i	j	k	i1	j1	k1	i1n+k1	k1n+j1	i1n+j1
5		0	0	0	0	0	0	0	0	0
5		0	0	0	0	0	1	1	5	0
5		0	0	0	0	1	0	0	1	1
5		0	0	0	0	1	1	1	6	1
5		0	0	0	1	0	0	5	0	5
5		0	0	0	1	0	1	6	5	5
5		0	0	0	1	1	0	5	1	6
5		0	0	0	1	1	1	6	6	6
5		0	0	2	0	0	2	2	10	0
5		0	0	2	0	0	3	3	15	0
5		0	0	2	0	1	2	2	11	1
5		0	0	2	0	1	3	3	16	1
5		0	0	2	1	0	2	7	10	5
5		0	0	2	1	0	3	8	15	5
5		0	0	2	1	1	2	7	11	6
5		0	0	2	1	1	3	8	16	6

Other algorithms:

The same thing will happen for other algorithms too, since the whole block can fit into the cache and the case like part 1 (in which the cache size was greater than the matrices) will happen. The only difference will be in the total number of misses. For example, for kij and ikj since A is not in the inner loop the total miss for A will be 10^7. For jki and kji B is not in the inner loop and the total misses for B will be 10^7. But the total miss rate will be the same. This shows how using efficient cache blocking could enhance the performance.

Part #3.

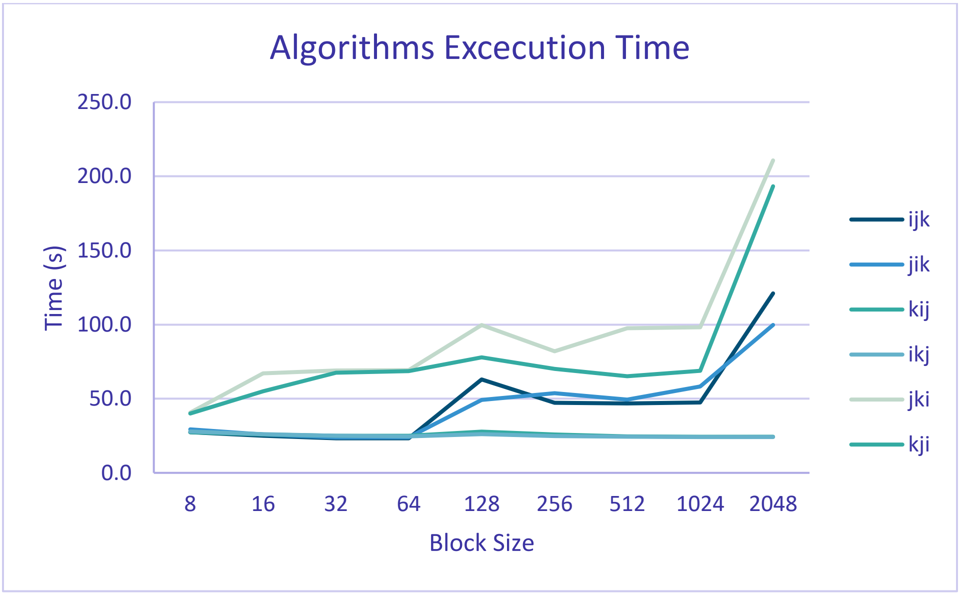
Implement the algorithms in part (1) and (2). Report your execution time on our Pantarhei cluster. Adjust the block size from 10 to other numbers to see what the optimal block size is. Compile your code using the default compiler (gcc-7.3.0) on Pantarhei without optimization tag. Compare and analyze the performance of your codes for n=2048. Please always verify the correctness of your code.

Solution:

For part 1, the code has been implemented in the ***Project2_Part3_simple.c*** source code. ***part3_NoBlocking*** and ***part3_NoBlocking.425.out*** are the complied version using the default compiler of Pantarhei and the output file after sending the compile version as a job, respectively. The same files were created and implemented in the ***Project2_Part3_Blocked.c***, ***part3_Blocked*** and ***part3_Blocked.428.out***. The correctness of each algorithm has been checked and the resulted maximum difference is reported at the end of the running part of each algorithm in the output file. Table 3 shows the results;

algo	miss rate	No Blocking	8	16	32	64	128	256	512	1024	2048
ijk	55%	95.6	27.4	25.0	23.3	23.3	63.1	47.2	46.8	47.5	121.0
jik	55%	90.3	29.3	26.0	23.9	24.0	49.2	53.7	49.4	58.3	99.8
kij	10%	24.8	27.5	25.9	25.1	25.0	27.8	25.8	24.6	24.3	24.4
ikj	10%	25.0	27.9	26.0	24.9	24.5	26.1	24.8	24.3	24.1	24.2
jki	100%	177.2	40.6	67.1	69.0	69.3	99.7	81.9	97.6	98.2	210.7
kji	100%	173.3	40.0	54.9	67.6	68.6	77.8	70.2	65.1	68.8	193.3

For the No Blocking part, the worst performance, as was expected according to the miss rates, goes to jki and kji. But for the blocked version, the performance increases to a point and decreases after that which shows there is an optimum block size. The results for different block values are close and running the code different times will result is different winners. Optimum block size is 64 for this run but it will change if you run it again and will iterate between 32 and 64 (why??)



Part #4.

Improve your implementation by using both cache blocking and register blocking at the same time. Optimize your block sizes. Compile your code using both the default compiler and gcc-5.4.0 with different optimization flags (-O0, -O1, -02, and -O3) respectively. Compare and analyze the performance of your codes for n=2048. Highlight the best performance you achieved. Please always verify the correctness of your code. Note that you can use "module swap gnu7/7.3.0 gnu/5.4.0" to replace the default compiler by gcc-5.4.0.

Solution:

For this part the source code could be found in the part4 folder with the name ***Project2_Part4.c.*** In this code first, the simple ijk algorithm was used just as a proof for correctness checking. After that for each algorithm, 2×2 cache blocking was implemented. For ijk and jik 12 registers and for the other 4, 8 register were used. Various compiled version of the source code using different compilers and optimization flags were created and sent as jobs. The results are contained in the ***p4_540_O0.482.out*** files. ***p4*** stands for part4, ***540*** means using gcc 5.4.0 and ***482*** is the job number associated by the Parantahei cluster. Other results were sampled using the same approach and naming.

	gcc 7.3.0-O0								
	8	16	32	64	128	256	512	1024	2048
ijk	16.47	12.66	11.73	11.77	25.23	19.24	19.12	19.12	50.45
jik	16.36	12.63	11.56	11.64	19.81	19.55	19.65	21.42	38.44
kij	13.60	12.79	12.56	13.69	13.85	12.75	12.37	12.26	12.16
ikj	13.37	12.74	12.43	12.31	12.47	12.22	12.14	12.05	12.19
jki	21.44	15.16	14.09	13.98	34.43	27.10	31.56	34.27	71.60
kji	21.38	15.14	14.11	13.71	27.85	26.91	30.21	40.44	74.36

	gcc 7.3.0-O1								
	8	16	32	64	128	256	512	1024	2048
ijk	7.00	4.78	3.94	3.98	10.67	8.09	7.67	7.83	22.50
jik	6.98	5.20	4.15	4.22	8.26	8.34	8.24	16.86	19.41
kij	4.67	4.06	4.09	4.99	5.15	4.46	4.12	4.10	4.13
ikj	4.59	4.09	3.95	3.95	4.17	3.97	3.85	3.85	4.04
jki	10.53	10.71	10.25	9.85	14.27	13.68	14.58	34.90	43.50
kji	10.25	9.91	9.63	9.58	13.66	13.95	15.55	20.77	31.75

	gcc 7.3.0-O2								
	8	16	32	64	128	256	512	1024	2048
ijk	6.15	4.17	3.39	3.49	10.33	7.72	7.53	7.63	23.36
jik	6.73	4.97	3.89	3.97	8.66	8.51	8.24	9.96	19.51
kij	3.69	2.76	2.80	3.53	3.66	3.07	2.87	2.71	3.26
ikj	3.89	3.04	2.74	2.73	3.30	2.82	2.63	2.58	3.08
jki	10.48	10.82	10.09	9.81	14.33	13.70	13.78	18.14	43.30
kji	9.35	9.67	9.63	9.47	13.84	14.05	14.39	14.65	40.80

	gcc 7.3.0-O3								
	8	16	32	64	128	256	512	1024	2048
ijk	5.83	4.00	3.35	3.41	9.51	7.39	7.04	7.10	16.17
jik	6.17	4.76	3.81	3.85	7.66	7.75	7.82	9.12	20.64
kij	3.63	2.69	2.73	3.38	3.58	3.17	2.80	2.77	3.40
ikj	3.55	2.76	2.61	2.64	3.10	2.74	2.58	2.58	2.91
jki	10.30	10.55	9.99	9.67	13.81	13.24	13.44	28.58	33.66
kji	9.09	9.63	9.50	9.46	13.34	13.62	14.51	17.70	56.56

	gcc 5.4.0-O0								
	8	16	32	64	128	256	512	1024	2048
ijk	15.51	12.05	11.17	11.29	23.55	18.57	18.46	18.61	48.89
jik	15.74	12.35	11.35	11.47	19.21	19.17	19.12	27.71	36.91
kij	13.79	12.86	12.64	13.51	13.50	12.62	12.39	12.22	12.26
ikj	13.58	12.78	12.42	12.36	12.61	12.23	12.09	12.27	12.19
jki	20.93	15.08	14.11	13.96	29.40	25.69	25.65	27.75	59.62
kji	20.58	15.08	14.07	13.81	26.79	28.07	31.42	34.37	59.54

	gcc 5.4.0-O1								
	8	16	32	64	128	256	512	1024	2048
ijk	7.19	4.88	4.06	4.10	12.54	7.76	7.70	7.91	30.47
jik	7.02	5.23	4.15	4.14	8.08	7.98	8.06	8.77	23.06
kij	4.48	3.98	4.03	4.89	5.03	4.34	3.96	4.08	4.06
ikj	4.65	4.06	3.90	3.94	4.08	3.90	3.77	3.75	3.95
jki	10.49	10.65	10.02	9.74	16.70	13.88	14.85	21.35	50.34
kji	9.91	9.72	9.55	9.52	13.78	14.34	16.02	13.94	36.98

	gcc 5.4.0-O2								
	8	16	32	64	128	256	512	1024	2048
ijk	5.93	4.10	3.36	3.50	11.14	7.43	7.42	7.71	21.63
jik	6.46	4.87	3.85	3.91	8.41	7.97	8.03	14.68	21.48
kij	3.75	2.81	2.82	3.57	3.73	3.19	2.78	2.77	3.67
ikj	3.77	2.90	2.69	2.71	3.19	2.77	2.59	2.58	3.20
jki	10.08	10.53	10.00	9.78	14.53	13.57	13.77	19.87	35.72
kji	8.75	9.46	9.69	9.46	13.46	13.66	15.67	14.74	28.65

	gcc 5.4.0-O3								
	8	16	32	64	128	256	512	1024	2048
ijk	5.98	4.06	3.36	3.46	9.95	7.52	7.29	7.27	17.64
jik	6.30	4.83	3.88	3.86	7.85	7.86	7.91	8.38	17.12
kij	3.75	2.81	2.81	3.47	3.56	3.08	2.79	2.66	3.08
ikj	4.37	2.28	1.86	1.75	2.44	2.04	1.72	**1.67**	2.21
jki	10.17	10.51	9.98	9.65	13.54	13.19	13.35	19.93	32.93
kji	8.70	9.44	9.49	9.50	13.22	13.64	14.88	13.42	39.28

ikj algorithm compiled on gcc 5.4.0 with flag -O3 and with block size of 1024 has the best performance. It reduced the execution time to 1.67 second which is incredible. This shows how using registers and writing cache friendly codes could enhance the performance of the system.

Using higher version of compilers as shown in the tables will enhance the performance. Alos, gcc 5.4.0 worked better than the default compiler.

Just out of curiosity I changed the register blocking size to 4 and used 48 registers. Since the system does not have enough space for this much of registers this will reduce the performance which was exactly what happened. The codes are provided in the 4v4 folder in the part 4 (for brevity the results tables are not shown) Of course using 3*3 register blocking will be the optimum case for using registers and will definitely improve these results.