CS610 - Programming for Performance
2020-2021-I: Assignment 3

Name: Sarthak Singhal Roll No.: 170635

System Specifications

- Model name: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
- CPU MHz: 1200.416
- CPU max MHz: 3700.0000
- CPU min MHz: 1200.0000
- L1d cache: 32KB
- L2 cache: 1024KB
- L3 cache: 19712KB
- Sockets: 2
- Cores per socket: 6
- Threads per core: 2
- ∴ Logical cores: 24
- gcc version: 7.4.0
- icc version: 2021.1 Beta (gcc version 7.4.0 compatibility)

1. Problem 1

Compilation and execution instructions:

For gcc:

- g++ -O2 -o problem1 problem1.cpp -mavx
- ./problem1 ⟨block_size⟩

For icc:

- icc -O2 -o problem1 problem1.cpp
- ./problem1 ⟨block_size⟩

Explanation and results:

If we work out the dependences, then the loop carried ones are (*, 0) for $y[j]$ and $z[j]$ separately. So, the distinct distance vector for loop carried dependence is only (1, 0). Hence, we can interchange the loops without any worry as (0, 1) is a valid dependence. Also loop distribution is possible as there is no dependence from statement with $z[j]$ to statement with $y[j]$.

Let's denote the statement with $y[j]$ as the first statement and the statement with $z[j]$ as the second statement. For the first statement, there is good spatial locality for array A and good temporal locality for array x. For the second statement, there is good temporal locality for

array x but array A is accessed with larger strides. So, I tried to improve the locality for array A by loop distribution followed by a loop interchange for the second nested loop(i.e. the nested loop contanining array z). We can apply loop distribution and interchange without worrying about dependences which is shown above in the dependency analysis. This version of the code is in the function named *optimized1*. On average this decreased the time of execution from 0.77901 sec to 0.12962 sec i.e. speedup of 6 with gcc and from 0.65892 sec to 0.08374 sec i.e. speedup of 7.868 with icc.

Now array A had good spatial locality for both the nested loops. Array x had good temporal locality for the first nested loop. Array z had good temporal locality for the second nested loop. Array y and array x(for second nested loop) have to be fetched again into the cache when outer loop's iteration changes as they are too large to fit in the L1 cache. So, to improve the locality I tried 2D blocking for both the perfect nested loops. I also tried unrolling the innermost loop to increase parallelism and also at the same time decrease the number of loop checks and updates.

The code is general w.r.t block size which is given as a command line argument. The function *optimized2* implements blocking without unrolling. The functions *optimized3*, *optimized4*, *optimized5*, *optimized6* implement blocking with different unrolling factors of 2,4,8,16 respectively. I tried with different block sizes of 2,4,8,16,32,64. The execution time(in sec) for these experiments with gcc is given below:

Blocking Factor(right) Unrolling Factor(down)	**2**	**4**	**8**	**16**	**32**	**64**
1	0.15517	0.16698	0.17646	0.18866	0.21290	0.38240
2	0.23073	0.15172	0.14158	0.13284	0.12879	0.30479
4	0.28089	0.11921	0.11991	0.11919	0.12509	0.27067
8	0.29057	0.17266	0.11972	0.11880	0.22872	0.31817
16	0.28078	0.16275	0.13665	0.11871	0.20842	0.26853

The best variant with gcc is with block size = 16 and unrolling factor also = 16 giving speedup of 0.77901/0.11871 = 6.56. So, we have reached from speedup of 6 to 6.56 using blocking and unrolling. I tried same experiments with icc but it performed worse with blocking. It may be possibly due to the fact that icc automatically vectorizes on using O2 flag. So using blocking is maybe making it difficult for icc to auto vectorize the code leading to worse performance. So, best of icc I could get is speedup of 7.868 as given above.

The best version of gcc is given by function *optimized6* with block size = 16 which is given as command line argument. The best version of icc is given by function *optimized1* with any arbitrary block size as a command line argument as this function doesn't implement blocking.

Now for part(b) I implemented the best version of gcc(i.e. function *optimized6*) with intrinsics using 256 bit registers(AVX) and using the same block size(i.e. 16) that was giving the best performance earlier. This is implemented in the function named *intrinsics*. The execution time for reference function using gcc was 0.75675 sec, for *optimized6* was 0.12337 sec and for the intrinsic version was 0.08682 sec. So the speedup for intrinsic version w.r.t reference version was 0.75675/0.08682 = 8.716. We can see the speedup increased by using intrinsics from 6.56 to 8.716. Also, running the same with icc gave a speedup of 0.63040/0.08816 = 7.15, which is little worse than what icc could do itself.

Note that all the functions are present in the code but the function calls for only the best ones are not commented i.e. the rest function calls are commented, but the function definitions are present. Like calls to *reference*, *optimized1*, *optimized6*, *intrinsics* are not commented in the int main() function.

2. Problem 2

Compilation and execution instructions:

For gcc:

- g++ -O2 -o problem2 problem2.cpp -mavx
- ./problem2 ⟨block_size⟩

For icc:

- icc -O2 -o problem2 problem2.cpp
- ./problem2 ⟨block_size⟩

Explanation and results:

If we work out the dependences, then the loop carried ones are (0, 0, *) for $C[i][j]$. So, the distinct distance vector for loop carried dependence is only (0, 0, 1). Hence, we can permute the loops without any worry as all the permutations are valid. We can also use any type of blocking as there is no negative value in the distance vector, so we can permute in any way.

I checked that in kij variant, C and A will have good locality but B will have bad locality. Similarly for kji variant, A will have good locality but B and C will have bad locality. Similarly, I found these observations for all the permutations. Only kij, jki, and ijk variant had 1 array access with bad locality and the rest had 2 array accesses with bad locality. So, I used only kij, jki, and ijk variants for my analysis.

kij, jki, and ijk versions of the code are in the functions named *optimized1*, *optimized2*, and *optimized3* respectively. Note that the ijk version is the same as the given reference version. On running these 3 versions with both gcc and icc, the jki version was performing the worst, so I used only kij and ijk for the further analysis.

To improve the temporal and spatial locality, I also tried 3D blocking. I also tried unrolling the innermost loop to increase parallelism and also at the same time decrease the number of loop checks and updates.

The code is general w.r.t block size which is given as a command line argument. The function *optimized4* implements ijk version with blocking and without unrolling. The function *optimized5* implements kij version with blocking and without unrolling. The functions *optimized6*, *optimized7*, *optimized8*, *optimized9* implement blocking with different unrolling factors of 2,4,8,16 respectively for ijk version. The functions *optimized10*, *optimized11*, *optimized12*, *optimized13* implement blocking with different unrolling factors of 2,4,8,16 respectively for kij version. I tried with different block sizes of 2,4,8,16,32,64. The execution time(in sec) for these experiments with gcc for just the ijk version is given below:(the reference time is 0.61202 sec)

Blocking Factor(right) Unrolling Factor(down)	**2**	**4**	**8**	**16**	**32**	**64**
1	0.83356	0.48215	0.41602	0.43750	0.46523	0.49170
2	1.33005	0.59656	0.44861	0.44497	0.46050	0.49280
4	1.56592	0.58600	0.43653	0.42937	0.45901	0.49084
8	1.55687	0.75699	0.43524	0.43007	0.45742	0.48868
16	1.57868	0.75693	0.50034	0.42642	0.45526	0.48240

The best variant with gcc is with block size = 8 and unrolling factor = 1 giving speedup of 0.61202/0.41602 = 1.47. The best variant of ijk version is better than the best variant of kij version. The best variant for kij version was giving speedup of 0.61202/0.48407 = 1.26. I tried

the same experiments with icc. The icc gave best result on block size = 64 and unrolling factor = 1 with ijk version giving speedup of 0.60911/0.32362 = 1.88.

The best version of gcc is given by function *optimized4* with block size = 8 which is given as command line argument. The best version of icc is also given by the function *optimized4* with block size = 64 which is given as command line argument.

Now for part(b) I implemented the best version of gcc(i.e. function *optimized4*) with intrinsics using 256 bit registers(AVX) and using the same block size(i.e. 8) that was giving the best performance earlier. This is implemented in the function named *intrinsics*. The execution time for reference function using gcc was 0.60988 sec, for *optimized4* was 0.41561 sec and for the intrinsic version was 0.38420 sec. So the speedup for intrinsic version w.r.t reference version was 0.60988/0.38420 = 1.58. We can see the speedup increased by using intrinsics from 1.47 to 1.58. Also, running the same with icc gave a speedup of 0.53594/0.44174 = 1.21, which is less than the previous version as for icc the optimal block size was 64, but here we are just considering the best version of gcc which required block size = 8.

Note that all the functions are present in the code but the function calls for only the best ones are not commented i.e. the rest function calls are commented, but the function definitions are present. Like calls to *reference*, *optimized4*, *intrinsics* are not commented in the int main() function.

3. Problem 3

Compilation and execution instructions:

- g++ -O2 -fopenmp -o problem3 problem3.cpp
- ./problem3 ⟨num_threads⟩ ⟨block_size⟩

Explanation and results:

Let's first find the dependences but only loop carried ones. The output dependence between $X[i][j+1]$ and $X[i][j+1]$ has distance vector (1, 0, 0). Dependences between $X[i][j+1]$ and $X[i-1][j+1]$ are: (0,1,0) flow, (1,1,0) flow and (1,-1,0) anti. Dependences between LHS $X[i][j+1]$ and RHS $X[i][j+1]$ are: (1,0,0) flow and (1,0,0) anti. So, the distance vector we need to worry about is only (1, -1, 0) as it is the only one that contains a negative element which affects the validity of the loop permutations. So using this dependence only permutations that are valid are jki, kij and kji. Also, only loop j can be parallelized as both k and i carry dependences. Also, 3D blocking can't be done as all the permutations are not valid.

I implemented jki, kij and kji versions of the code and parallelized loop j using OpenMP setting the desired data scoping parameters. jki, kij, and kji versions of the code are in the functions named *omp_version1*, *omp_version2*, and *omp_version3* respectively. I tried varying the number of threads initially. All the 3 versions were getting better performance on increasing the number of threads. As the number of logical cores on the machine I was running the code was 24, I kept the number of threads = 24 for the further analysis. For now the speedups for jki version was 1.01882/0.755194 = 1.349, for kij version was 1.01882/6.77603 = 0.15, for kji version was 1.01882/0.868018 = 1.173. The performance degradation for kij version is due to that j is the innermost loop so thread creation frequently can lead to a large overhead. Note that the number of threads is the first command line argument given to the executable. So from now, while executing give the first argument to the executable = 24.

To improve the temporal and spatial locality, I also tried 2D blocking of loops i and j. The code is general w.r.t block size which is given as second command line argument. The functions *omp_version4*,*omp_version5*,*omp_version6* implement jki, kij, kji version with blocking respectively. I tried with different block sizes of 2,4,8,16,32,64. The jki version performed the best among the 3, so I kept just the jki version for the further analysis. The best speedup obtained with jki version was for block size = 16. The speedup was 1.01713/0.195765 = 5.195.

I also tried unrolling the loops i and j separately to increase the parallelism and also at the same time decrease the number of loop checks and updates. The functions *omp_version7*, *omp_version8*, *omp_version9*, *omp_version10* implement blocking with different unrolling factors of 2,4,8,16 respectively with unrolling of loop i. The functions *omp_version11*, *omp_version12*, *omp_version13*, *omp_version14* implement blocking with different unrolling factors of 2,4,8,16 respectively with unrolling of loop j. I tried with different block sizes of 2,4,8,16,32,64.

The best variant is with block size = 32 and unrolling factor = 16 and unrolling loop i, giving speedup of 1.05842/0.144042 = 7.34. Unrolling loop i gave better results than unrolling loop j.

The best version is given by function *omp_version10* with number of threads = 24 and block size = 32 which are given as command line arguments first and second respectively.

Note that all the functions are present in the code but the function calls for only the best ones are not commented i.e. the rest function calls are commented, but the function definitions are present. Like calls to *reference*, *omp_version10* are not commented in the int main() function.