CS610 - Programming for Performance
2020-2021-I: Assignment 2

Name: Sarthak Singhal   Roll No.: 170635

1. Problem 1

(a) Assume a flow dependence between $A(i, i-j)$ and $A(i, i-j-1)$.
$\therefore I_0 = I_0 + \Delta I$ and $I_0 - J_0 = I_0 + \Delta I - J_0 - \Delta J - 1$
$\Rightarrow \Delta I = 0, \Delta J = -1$
Hence, there is a loop carried anti dependence $(0, 1)$ between $A(i, i-j)$ and $A(i, i-j-1)$.

(b) Assume a flow dependence between $A(i, i-j)$ and $A(i-1, i-j)$.
$\therefore I_0 = I_0 + \Delta I - 1$ and $I_0 - J_0 = I_0 + \Delta I - J_0 - \Delta J$
$\Rightarrow \Delta I = 1, \Delta J = 1$
Hence, there is a loop carried flow dependence $(1, 1)$ between $A(i, i-j)$ and $A(i-1, i-j)$.

(c) Assume a flow dependence between $A(i, i-j)$ and $A(i+1, i-j+1)$.
$\therefore I_0 = I_0 + \Delta I + 1$ and $I_0 - J_0 = I_0 + \Delta I - J_0 - \Delta J + 1$
$\Rightarrow \Delta I = -1, \Delta J = 0$
Hence, there is a loop carried anti dependence $(1, 0)$ between $A(i, i-j)$ and $A(i+1, i-j+1)$.

2. Problem 2

Data dependences: $(1, 0, -1, 1), (1, -1, 0, 1)$ and $(0, 1, 0, -2)$

(a) Loop unrolling does not change the order of dynamic instances of the statements of the program. So, any of the dependence does not changes. Hence, loop unrolling is valid for all loops i.e. t, i, j, k.

(b) In any permutation t has to be the first loop as dependences for i, j and k are negative in atleast one of the dependences. Taking t as the first loop we don't have to consider first 2 given dependences for further analysis as they are carried by loop t.
Now if we take i as the second loop then third given dependence$(0, 1, 0, -2)$ is carried by loop i, so order of next 2 loops doesn't matter. So (t, i, j, k) and (t, i, k, j) are valid permutations.
We can't take k as the second loop as then the third dependence will be invalid. So only left option is j as the second loop. Now for third dependence to be valid i has to be before k. So only valid permutation in this case is (t, j, i, k)
All valid permutations = $(t, i, j, k), (t, i, k, j), (t, j, i, k)$

(c) Necessary condition for tiling of contiguous loops to be valid is that they should be fully permutable. So we'll look over tilings involving different number of loops for all the valid permutations.

i. $4D$ tiling is not valid as all the permutations are not valid

ii. Taking first valid permutation (t, i, j, k), $3D$ tiling is not valid for tij as $ijtk$ is not valid permutation. $3D$ tiling is also not valid for ijk as $tkji$ is not valid permutation.
Taking second valid permutation (t, i, k, j), $3D$ tiling is not valid for tik as $iktj$ is not

valid permutation. $3D$ tiling is also not valid for ikj as $tkji$ is not valid permutation. Taking third valid permutation (t, j, i, k), $3D$ tiling is not valid for tji as $ijtk$ is not valid permutation. $3D$ tiling is also not valid for jik as $tkji$ is not valid permutation.

iii. Similarly we can look for different $2D$ tilings. The ones which are valid are ij and jk. As when we permute ij then $tjik$ is valid permutation. Similarly, when we permute jk then $tikj$ is valid permutation.

iv. $1D$ tiling is always valid, so all the 4 loops can be used for $1D$ tiling.

(d) Taking any of the 3 valid permutations, the first 2 dependences are carried by loop t, so it can't be parallelized.
For permutations (t, i, j, k) and (t, i, k, j) the third dependence is carried by loop i. So we can parallelize loops j and k.
For permutation (t, j, i, k) the third dependence is carried by loop i. So we can parallelize loops j and k.
So, we can parallelize loops j and k.

(e)
```
int i,j,t,k;
for(t = 0; t < 2048; t++){
    for(i = t; i < 1024; i++){
        for(k = 1; k < i; k++){
            for(j = max(t, k+1); j < i; j++){
                S(t,i,j,k);
            }
        }
    }
}
```

3. Problem 3

(a) Let's define some notations for simplifying further analysis:
LHS $A[i][j] = \textcircled{1}$
$A[i-1][j] = \textcircled{2}$
RHS $A[i][j] = \textcircled{3}$
$A[i+1][j] = \textcircled{4}$
$A[i][j-1] = \textcircled{5}$
$A[i][j+1] = \textcircled{6}$

- Assume a flow dependence between $\textcircled{1}$ and $\textcircled{2}$.
$\therefore I_0 = I_0 + \Delta I - 1$ and $J_0 = J_0 + \Delta J$
$\Rightarrow \Delta I = 1, \Delta J = 0$
Hence, direction vector $= (*, 1, 0)$ which means flow-loop carried $(1, 1, 0)$, flow-loop carried $(0, 1, 0)$ and anti-loop carried $(1, -1, 0)$.

- Assume a flow dependence between $\textcircled{1}$ and $\textcircled{3}$.
$\therefore I_0 = I_0 + \Delta I$ and $J_0 = J_0 + \Delta J$
$\Rightarrow \Delta I = 0, \Delta J = 0$
Hence, direction vector $= (*, 0, 0)$ which means flow-loop carried $(1, 0, 0)$, anti-loop carried $(1, 0, 0)$ and anti-loop independent $(0, 0, 0)$.

- Assume a flow dependence between $\textcircled{1}$ and $\textcircled{4}$.
$\therefore I_0 = I_0 + \Delta I + 1$ and $J_0 = J_0 + \Delta J$
$\Rightarrow \Delta I = -1, \Delta J = 0$

Hence, direction vector $= (*, -1, 0)$ which means flow-loop carried $(1, -1, 0)$, anti-loop carried $(1, 1, 0)$ and anti-loop carried $(0, 1, 0)$.

- Assume a flow dependence between $\textcircled{1}$ and $\textcircled{5}$.
  $\therefore I_0 = I_0 + \Delta I$ and $J_0 = J_0 + \Delta J - 1$
  $\Rightarrow \Delta I = 0, \Delta J = 1$
  Hence, direction vector $= (*, 0, 1)$ which means flow-loop carried $(1, 0, 1)$, flow-loop carried $(0, 0, 1)$ and anti-loop carried $(1, 0, -1)$.

- Assume a flow dependence between $\textcircled{1}$ and $\textcircled{6}$.
  $\therefore I_0 = I_0 + \Delta I$ and $J_0 = J_0 + \Delta J + 1$
  $\Rightarrow \Delta I = 0, \Delta J = -1$
  Hence, direction vector $= (*, 0, -1)$ which means flow-loop carried $(1, 0, -1)$, anti-loop carried $(1, 0, 1)$ and anti-loop carried $(0, 0, 1)$.

- Assume a output dependence between $\textcircled{1}$ and $\textcircled{1}$.
  $\therefore I_0 = I_0 + \Delta I$ and $J_0 = J_0 + \Delta J$
  $\Rightarrow \Delta I = 0, \Delta J = 0$
  Hence, direction vector $= (*, 0, 0)$ which means output-loop carried $(1, 0, 0)$.

Summarizing the distinct loop carried direction vectors: $(1, 1, 0)$, $(0, 1, 0)$, $(1, -1, 0)$, $(1, 0, 0)$, $(1, 0, 1)$, $(0, 0, 1)$, $(1, 0, -1)$

(b) For finding valid loop permutations we only need to worry about those dependences which have atleast one negative sign. So, we just have to worry about $(1, -1, 0)$ and $(1, 0, -1)$. Now as i is negative in first dependence and j is negative in second dependence, we can't start the permutation with either of i or j. So, first element of the permutation is fixed i.e. t. Now as both of them are carried by t the order of i and j don't matter. So we can permute them in all possible ways.
All valid permutations $= (t, i, j), (t, j, i)$

(c) Loop unrolling does not change the order of dynamic instances of the statements of the program. So, any of the dependence does not changes. Hence, loop unrolling is valid for all loops i.e. t, i, j.

(d) Necessary condition for tiling of contiguous loops to be valid is that they should be fully permutable. So we'll look over tilings involving different number of loops for all the valid permutations.

   i. $3D$ tiling is not valid as all the permutations are not valid. E.g. ijt is not valid

   ii. Taking first valid permutation (t, i, j), $2D$ tiling is not valid for ti as itj is not valid permutation. $2D$ tiling is valid for ij as tji is a valid permutation.
   Taking second valid permutation (t, j, i), $2D$ tiling is not valid for tj as jti is not valid permutation. $2D$ tiling is valid for ji as tij is a valid permutation.

   iii. $1D$ tiling is always valid, so all the 3 loops can be used for $1D$ tiling.

4. Problem 4

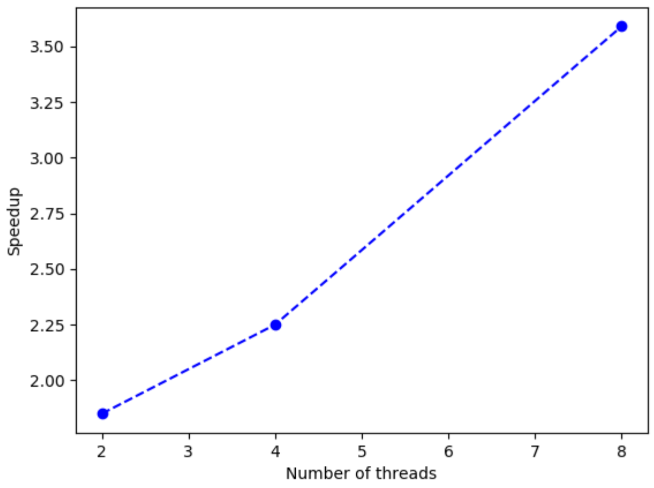
How to Compile: g++ -o problem4 problem4.cpp -pthread
How to Execute: ./problem4 ⟨number of customers⟩

5. Problem 5

How to Compile: g++ -o problem5 problem5.cpp -pthread
How to Execute: ./problem5 ⟨block_size⟩ ⟨num_threads⟩

Experiments were run on CSE IITK server.
Machine Specifications: x86_64 bit, 8 cores, L1 data cache = 32KB

(a) Avg. time for sequential implementation: 1114.73s.
Avg. time for parallel implementation with 2 threads: 602.04s , hence speedup = 1.85
Avg. time for parallel implementation with 4 threads: 493.75s , hence speedup = 2.25
Avg. time for parallel implementation with 8 threads: 310.40s , hence speedup = 3.59



The function *parallel_matmul* contains the code for parallel implementation.

(b) $2D$ and $3D$ blocking was tried with different blocking factors and different unrolling factors with unrolling the innermost loop. The table for speedups in $3D$ blocking for the sequential version for various blocking factors and unrolling factors in given below.

Blocking Factor(right) Unrolling Factor(down)	**4**	**8**	**16**	**32**	**64**
2	1.82	2.4	2.41	2.62	2.73
4	2.02	2.64	2.66	2.92	3.06
8	1.48	2.8	2.8	3.07	3.22
16	1.5	2.1	2.86	3.12	3.32
32	1.5	2.1	2.17	3.20	3.36

Each element is of size 64 bits = 8 bytes. Using the calculation $3 * B^2 * 8 \leq$cache size. Taking only L1 cache size gives optimal $B = 32$. But the experiments are giving better speedups even on $B = 64$. I think the reason can be that maybe the penalty by using L2 cache is not more than the benefit of the more spatial and temporal locality. For blocking factor = 64 increasing unrolling factor increases the speedup as the loop increments and checking for loop conditions decreases.
I also tried $2D$ tiling for jk which gave best performance for unrolling factor = 32 and blocking factor = 128. The speedup is 2.66. It is worse than the best version of $3D$ tiling as maybe it does not exploit the spatial and temporal locality better.
So, the best variant for sequential is with $3D$ tiling with blocking factor = 64 and unrolling factor = 32 which gives speedup of 3.36. Note that it is comparable with the best variant of parallel version with no optimization which had speedup of 3.59.

The functions *sequential_matmul_opt_1*, *sequential_matmul_opt_2*, *sequential_matmul_opt_3*, *sequential_matmul_opt_4*, *sequential_matmul_opt_5* are the optimized versions of sequential implementation with $3D$ tiling and unrolling factors of $2, 4, 8, 16, 32$ respectively. Note that the block size and number of threads are given as command line arguments as given in the execution instructions. So the best variant for sequential implementation is the function *sequential_matmul_opt_5* and the executable would be run as: ./problem5 64 8 to generate the results of best variant of sequential version. Also the function *sequential_matmul_opt_6* contains the code for $2D$ tiling of jk with unrolling = 32.

The parallel optimized version gives the best performance for unrolling factor = 32, number of threads = 8, and blocking factor = 64 with $3D$ tiling. The speedup obtained is 12.65. Note that it is comparable to the product of speedups of the best parallel implementation without optimization(3.59) and the best optimized sequential implementation(3.36) which is 12.06.
The function *parallel_matmul_opt* contains the code for this variant. The executable would be run as: ./problem5 64 8 to generate the result.

In summary, you can run the code using: ./problem5 64 8, to get the best results for sequential optimized and parallel optimized as the blocking factor is same for both(i.e. 64) and the number of threads(i.e. 8) does not matter for the sequential optimized version.