Programming for performance - Assignment 3

Abhay Kumar Dwivedi
22111001

September 26, 2022

System description :

Model name : Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
Architecture : x86_64
Processor Frequency : 3.20 GHz
Core : 6
L1d cache size (Data cache) : 192 KiB
L1i cache size (Instruction cache) : 192 KiB
L2 cache size : 1.5 MB
L3 cache size : 12 MB

g++ version : 9.4.0

Solution - 1

1a :

According to the given distance vectors, Direction matrix would be

$$\begin{matrix} i & j & k & l \end{matrix}$$
$$\begin{bmatrix} + & = & - & + \\ + & - & = & + \\ = & = & + & - \\ = & + & = & - \end{bmatrix}$$

Since there is no column with all $=$ therefore we can not parallelize the outermost loop. Now, i and l column has most $+$ in them. We can't make l as the outermost loop as it contains -. Hence we will chose i column and eliminate it and all the rows where i has $+$.

We will get,

$$\begin{matrix} j & k & l \end{matrix}$$
$$\begin{bmatrix} = & + & - \\ + & = & - \end{bmatrix}$$

Now, we can reverse the l loop.

$$\begin{matrix} j & k & l \end{matrix}$$
$$\begin{bmatrix} = & + & + \\ + & = & + \end{bmatrix}$$

Now we can move the l loop to the outermost position.

$$\begin{matrix} l & j & k \end{matrix}$$
$$\begin{bmatrix} + & = & + \\ + & + & = \end{bmatrix}$$

Now we can run l loop sequentially as it carry the dependency and remove all the rows where l contains $+$.
Since both the rows in l contain $+$, hence everything will be removed from the matrix and the rest of the variables j and k can be parallelized.

Answer : j and k can be parallelized.

1b :

Since there are four level nesting, Hence number of permutation $= 4! = 24$ permutations.
Any permutation of the distance vector where first non 0 value is positive is a valid permutation.

List of permutations and whether they are valid or not :

Starting from i :
ijkl – Original permutation, **hence valid**
ijlk – The third distance vector will become (0, 0, -1, 1) which is invalid
ikjl – **valid** as all the distance vectors are valid
iklj – The fourth distance vector will become (0, 0, -1, 1) which is invalid
iljk – The third distance vector will become (0, -1, 0, 1) which is invalid
ilkj – The third distance vector will become (0, -1, 1, 0) which is invalid

All the permutation starting with j (jikl, jilk, jkil, jkli, jlik, jlki) will be invalid as for the second distance vector, value corresponding to j is -1.

All the permutation starting with k (kijl, kilj, kjli, kjil, klij, klji) will be invalid as for the first distance vector, value corresponding to k is -1.

All the permutation starting with l (lijk, likj, ljik, ljki, lkij, lkji) will be invalid as for the third and fourth distance vector, value corresponding to l is -1.

Answer : Total number of valid permutation : **2** (including the original permutation).

1c :

Condition for unroll and jam : The loops which you want to unroll and jam, when moved to the innermost position gives out a valid permutation.

Loop l : Innermost loop can always be unrolled and jammed as they produce an already jammed code.

$$\begin{array}{c} \begin{matrix} i & j & k & l \end{matrix} \\ \begin{bmatrix} + & = & - & + \\ + & - & = & + \\ = & = & + & - \\ = & + & = & - \end{bmatrix} \end{array}$$

Loop k : Moving k at innermost position produces an invalid permutation as the third row in direction matrix would have - as the first non zero entry.

Loop j : Moving j at innermost position produces an invalid permutation as the fourth row in direction matrix would have - as the first non zero entry.

Loop i : Moving i at innermost position produces an invalid permutation as the first and second row in direction matrix would have - as the first non zero entry.

Answer : Loop l can be unrolled and jammed.

1d :

According to the given distance vectors, Direction matrix would be

$$\begin{array}{c} \begin{matrix} i & j & k & l \end{matrix} \\ \begin{bmatrix} + & = & - & + \\ + & - & = & + \\ = & = & + & - \\ = & + & = & - \end{bmatrix} \end{array}$$

Case - 1 : 4-D tiling (ijkl) :
Since j, k, l contains - in it, hence the permutation of ijkl would not be valid hence 4-D tiling is not possible.

Case - 2 : 3-D tiling (ijk, jkl, ijl, ikl) :
ijk is not valid as j and k as - in it, similarly ijl and ikl are not valid. Coming to jkl, permutation ljk and lkj is valid, Hence 3-D tiling is not valid.

Case - 3 : 2-D tiling (ij, ik, il, jk, jl, kl) :
ij, ik, il is not valid as j, k and l has - in it. jl, kl is not valid as ilkj and ijlk are not valid permutations. jk is valid 2-D permutation.

Case - 4 : 1-D tiling (i, j, k, l) :
1-D tiling is always valid.

Answer : 2-D tiling (jk) is valid, 1-D tiling is by default valid.

1e :

```
int i, j, k, l;

for(int j = 1; j < 1024; j++){
    for(int i = max(0, j+1); i < 1024; i++){
        for(int l = 1; l < min(i-2, k+1) + 1; l++){
            for(int k = max(j, l+1); k < i; k++){
                S(i, j, k, l);
            }
        }
    }
}
```

Solution - 2

	Reference Version	Version - 1	Version - 2	Version - 3	Version - 4	AVX Intrinsic
Time taken(O2)	0.410 sec	0.444 sec	0.429 sec	0.279 sec	0.108 sec	0.080 sec
Time taken(O3)	0.213 sec	0.435 sec	0.429 sec	0.277 sec	0.129 sec	0.080 sec

Methods :

Version - 1 : Distributing both inner and outer loop : Performance is comparable with the reference version, usually slower as it has not used the spatial locality of x array in the same loop.

Version - 2 : Unrolling inner loop 8 times : Performance is again comparable with the reference version, sometimes slower, sometimes faster, as there is less loop overhead in this version.

Version - 3 : Unrolling outer loop 16 times then jamming it : Performance is better than the reference version as the access A[j][i] uses spatial locality from it. It is not much better as the access A[i][j] is accessing value of a column which slows the code a bit. **Speedup : 1.46**.

Version - 4 : Distributing both inner and outer loop and interchanging i and j in the second loop and then unrolling the inner loop : Performance is much better than the reference and the previous versions as it enjoys the spatial locality of A[i][j] in first loop and A[j][i] in the second loop. **Speedup : 3.79**.

AVX Intrinsic : In AVX intrinsic I have made three vectors of type __m256d. For the first loop I have loaded y_opt, x and A then added y_opt with the multiplication of the A and x vectors and finally loaded in in y_opt. In the second loop I have loaded z_opt, x and A then added z_opt with the multiplication of x and A and finally loaded in z_opt. **Speedup : 5.12**.

Solution - 3

	Reference Version	Optimized Code
Time taken	0.922 sec	0.957 sec

Explanation : In the given loop we can not parallelize k and i loop, therefore I have parallelized the innermost loop j. Since there is overhead of creating threads at line #pragma omp parallel (which is very large as it is there in the innermost loop) the code can not give the speed up and is quite close to time of the reference version.

I have also used #pragma omp nowait to avoid the implicit barrier caused by the for loop to make code run faster.

Solution - 4

	Reference Version	Without Reduction	With reduction	Tasks
Time taken	0.0094 sec	0.0058 sec	0.0044 sec	0.0241 sec

Explanation : Here omp with and without reduction is running faster than the reference version. The omp code with task is just a bit slower than the reference version as there is overhead of creating tasks inside a loop which slows the code down.

In without reduction version, I have taken a private variable for every thread, it calculates the sum for some iterations in its private variable and then adds it to the global variable to get the final answer. To ensure synchronization I have used critical to make sure only one thread gets inside the critical section at any time. **Speedup : 1.62**.

In reduction version, I have simply used the for reduction(+ : sumVar) to use reduction given by omp. **Speedup : 2.13**.

In task version, there will be 2^{24} / 2^{10} tasks, each task will execute 2^{10} iterations and store the sum in its private variable ts. After that each thread will go inside the critical section one by one to add the sum in the global variable. I have used #pragma omp single to make sure only once these tasks are executed. Finally the overall sum is returned. Speedup is not achieved due to the overhead of making threads, ensuring synchronization and specially making tasks.

Solution - 5

	Reference Version	OpenMP	SSE4	AVX
Time taken	20	158	270	194

Explanation :

In omp version, I have implemented a loop which runs serially to calculate the prefix sum of the array source. There is a true dependency of distance 1 in the statement of the loop which can not be parallelized. After that I have copied the values of tmp array into dest array using #pragma omp parallel.

In sse4 version, I have implemented a loop which runs serially to calculate the prefix sum of the array source. There is a true dependency of distance 1 in the statement of the loop which can not be vectorized. After that I have copied the values of tmp array into dest array using SSE4 intrinsic.

In sse4 version, I have implemented a loop which runs serially to calculate the prefix sum of the array source. There is a true dependency of distance 1 in the statement of the loop which can not be vectorized. After that I have copied the values of tmp array into dest array using AVX2 intrinsic.