CS 610 Semester 2024–2025-I
Assignment 3

Divyansh
(210355)

23rd October 2024

Note: *The makefile for each the problems is in their respective directory.*

1 Problem 1

1.1 Solution

First, I performed loop tranformation from the `ijk` version to `kij` version. This conversion fascilitated the comfort of putting all the floating point numbers from a `128` or `256` register to different values if `C` array. Otherwise, we needed to do horizontal sum of the fat registers in order to get the sum of element-wise product.

For example:

```
// k, i loops...
__m128 row = _mm_load_ps1(&A[i][k]); //copies of a[i,k]
for(int j = 0; j<N; j += stride){
    __m128 col = _mm_load_ps(&B[k][j]);  //j,j+1,j+2,j+3

    __m128 temp = _mm_load_ps(&C[i][j]);
    temp = _mm_add_ps(temp, _mm_mul_ps(row, col));
    _mm_store_ps(&C[i][j],temp);
}
```

Similar approach for `AVX2` version as well.

1.2 Performance Comparison

1. **Sequential Version**:
   `Matmul seq time:  991 ms`

2. **Unaligned Version**:
   `Matmul SSE4 (unaligned) time:  216 ms`
   `Matmul AVX2 (unaligned) time:  163 ms`

3. **Aligned Version**:
   `Matmul SSE4 time:  210 ms`

```
Matmul AVX2 time:  141 ms
```

Remarks: In general, vectorization improves the performance of the code significantly (irrespective of being aligned or not). Further, Aligned matrix has much more impact on the performance of `AVX2` as compared to that of `SSE4`.

2 Problem 2

2.1 Solution

As discussed in class, where we performed prefix sum for the `128` bits register and then added the offset to get the running prefix sum, here intel does provide `left-shift` operation for AVX2 so I wrote my own inline function for that using element permutation and masking. Which looks like this:

```
inline __m256i  _mm256_sll(__m256i var, int x){
    uint8_t mask = 0b11111111;
    mask = mask << x;
    __m256i zero = _mm256_setzero_si256();

    __m256i shift_mask;
    switch(x){
        case 4: shift_mask = _mm256_set_epi32(3,2,1,0,0,0,0,0);
                break;
        case 2: shift_mask = _mm256_set_epi32(5,4,3,2,1,0,0,0);
                break;
        default: shift_mask = _mm256_set_epi32(6,5,4,3,2,1,0,0);
    }
    __m256i temp = _mm256_permutevar8x32_epi32(var,shift_mask);
    return _mm256_blend_epi32(zero, temp, mask);
}
```

Now, using this function we perform the prefix sum of `256` bit register with similar approach as for the `SSE` version requiring `3 shifts` and `3 adds`.

2.2 Performance comparison for $N = 2^{30}$

1. `Serial version:  1073741824 time:  787523`
2. `OMP version:  1073741824 time:  752790`(varies frequently)
3. `SSE version:  1073741824 time:  738819`
4. `AVX2 version:  1073741824 time:  727176`(this is also sometimes slower and sometimes faster as compared to `SSE` version)

Remarks: Even after multiple runs on multiple KD-systems I wasn't able to find a fixed order among the performance of all the versions. It was varying frequently and I don't have any credible reason for that as well.

3 Problem 4

3.1 Performance comparison on CSEWS17

1. `Sequential version: 397 secs`
2. `Loop transformation version: 188 secs` **(2.1x speedup)**
3. `OMP version: 46 secs` **(8.7x speedup)**

3.2 Optimization Description & Remarks in Part(i)

My main idea for this problem was to perform `LICM` since there were lots of repetitive calculation occuring inside the inner-most loop. So, the optimizations are as follows:

1. **One level LICM**: So, in the innermost loop all the q_i's are being assigned as per the loop iteration variables. We can shift the calculations involving the upper `9` loops to one level up so that those calculations can be reused in the inner most loop rather than calculation again and again. This transformation results in appoximately **2x speed-up**.
2. **Two level LICM**: Very similar to the above approach we can shift the calculations involving the upper `8` loop iteration variables up by one level - further reducing the recalculation of values. This has very small effect on the performance resulting in final speep-up of **2.1x**

Unfruitfull Optimization Attempts

1. All level LICM: From the above two optimization, the one pretty obvious thought is that why not we just move the calculation to all the specific loops where the respective iteration variable changes in a hope that it will lead to maximum reuse of calculated values. However, My results where not as expected and I believe the reason is that in order to do so we need to have two many temporary variables leading to more register pressure, large number of instructions, load & store operations.

3.3 Optimization Description & Remarks in Part(ii)

There are two thing to tackle in this problem. First one is clearly how to parallelize using multiple threads and second one is ensuring the same write order in the output file and it would have been in a sequential execution of the program on a single core.
For the first part, I have create a parallel region on the outermost loop with number threads being equal to the number of iterations in the outermost loop(`s1` in the given code file). Also, I made all the x_i's and q_i's private so that all the threads don't have to use the same memory for their execution. For the correct order of writes in the file I created a matrix where each thread store the satisfying x_i's in the separate vector of doubles. Then finally once all the threads have done their job, we eventually write the values from that matrix into the file in the expected order. This lead to approximate speedup of **8.7x to 9x** on diferent runs as compared to sequential execution.
What we can also do here is that we can apply this parallelization on top of the optimized version from **part(i)**. However, that results in little to no performance boost probably because of the overhead of all the private variables which come into existence in the previous part.