PA5 Write-Up

2019-11563

자유전공학부

김지원

Brief explanation :

First, apply transpose to the second matrix to enable cache-friendly access since accessing elements row-wise is usually better due to the way matrices are stored in memory.

Also, all the commented out ‘matmul’ functions are different trials with different optimization techniques combined. Starting from line 66 is the ‘blocking + loop unrolling’ optimization. In this function, the second matrix is transposed first. After within the for loop, loop unrolling is applied where the block size is 32 \* 256 (bsize \* bsize\_k). ‘i’ block and ‘j’ block size is 32 and ‘k’ block has size 256. The block size is determined after several repeated trials of comparing performance. The final block size is the one where the performance was the best. The loop is unrolled by 8 where the redundant line is exposed within in the loop 8 times to cut down loop overhead by allowing the processor to execute multiple operations in parallel. And after the last k loop, the sum is stored in output[ i \* N + j].

The last function (avx submitted function) should add ‘Makefile adjustment : add ‘-mavx512f’ to c\_flags’. However, it was not able to be compiled in the hardware lab computer, so the above function was submitted. However, to briefly explain, the function uses all the same optimization techniques (transpose, blocking, and loop unrolling) optimization and has the overall similar structure. The only difference is the context in the for loop, where instead of loop unrolling part, avx instructions are used to allow more parallel execution within the instruction itself.

First, all the necessary registers are declared before any function procedure (\_\_m512i type). Next, transpose is applied to the second matrix followed by for loop to allow blocking. Blocking is applied with the same block size (32 \* 256). Inside the for loop (the last loop for loop unrolling by 16), the procedure works like this:

First, the matrix are loaded to registers (temp\_A, temp\_B) using SIMD instruction ‘\_mm\_512\_loadu\_si512’. This instruction loads 512 bits (16 integers) at once. After, the multiplication of temp\_A and temp\_B are calculated using ‘\_mm12\_mullo\_epi32’ which multiplies 16 integers within temp\_A and temp\_B in parallel and saves the result in sum1.

The sum within the k loop is stored in register ‘sum2’ by adding the calculated sum1 to with the original sum2 value which was initialized as 0 (16 0’s) before the loop start (\_mm512\_add\_epi32). After iterating through the whole loop, sum2 is stored in memory called ‘temp\_sum’ by using ‘\_mm512\_storeu\_si512’ function. Saving sum2 in memory is needed since we need to add the divided 16 parts (16 integers) of the 512-bit wide temp\_sum, and this can only be done once it is saved in memory and called back by accessing each part (as temp\_sum[0], temp\_sum[1]… ). This is done outside the for loop and all the 16 parts which are the results of parallel multiplication instruction are being added to ‘sum’. Finally, the sum is stored in output.