CS610: Programming For Performance

Assignment 3
Devesh Shukla (200322)

October 23, 2024

Problem 1

The following table shows the average values of execution times (in milliseconds) for matrix-matrix multiplication using various kinds of intrinsic instructions:

N	Scalar	SSE4 (unaligned)	SSE4 (aligned)	AVX2 (unaligned)	AVX2 (aligned)
512	137	31	31	22	15
1024	1045	265	267	223	139
2048	10493	2819	3081	2487	1804

Clearly, we can see that the vectorised versions perform significantly better than the scalar version of the program. Also, the AVX2 versions perform much better than the SSE4 variants.
Moreover, the unaligned version of SSE4 performs at par with the aligned version. This may be because there is a cost of checking if the memory address from where the aligned load needs to be performed is actually 16-byte aligned or not. This overhead slows down the SSE4 aligned variant as compared to the unaligned variant as modern day processors perform reasonably well even for some unaligned memory loads and stores.
For AVX2 instructions, the aligned version easily outperforms the corresponding unaligned version as expected. This is because the runtime checks needed to be done in this case are fewer than SSE4 versions as one AVX2 vector operation operates on twice the memory operated upon by SSE4 instructions. So, the aligned loads ensure a smaller execution time for these instructions.

Problem 2

The following table summarises the average running times (in microseconds) for various versions of the inclusive prefix sum function :

N	Sequential	OMP	SSE4	AVX2
2^{25}	15297	15378	15533	15308
2^{28}	125042	118053	120122	117136
2^{30}	592927	474282	472460	467622

Clearly, AVX2 version performs the best, while OMP and SSE4 variants perform almost similarly and the sequential version is the slowest among all. The separation among the variants emerges more clearly as N is increased but the run times still show significant variability in different runs of the program.

Problem 4

Workstation used : CSEWS166

Part (i)

The given program has a 10-level deep loop nest. All the loops iterate for the same number of times. Moreover, the computations performed are symmetric w.r.t each loop (without considering the nested loops), so loop permutations don't provide any reasonable speedup in this case as all loops are similar to each other. Loop tiling also doesn't help in this program as the program does not have spatial locality.
In the program's code, we can observe that the innermost loop computes the variables q1, q2, ..., q10 each time using 10 multiplications each time. 9 out of these 10 multiplications can be computed before entering the innermost loop as these don't depend on any variable of the innermost loop. So, we can shift these computations to outside the innermost loop. Continuing in a similar manner, we can keep moving more of these multiplications to an outermost level to reduce these multiplication operations being performed repetitively. So, LICM (loop invariant code motion) seems to be a feasible optimisation for the given program to avoid redundant multiplications at the cost of increase in memory used by introducing more variables at all levels of the loop nest.
Average execution time for sequential program, $t_{seq} = 293.896$ seconds
Average execution time for optimised program, $t_{opt} = 104.942$ seconds

Observed speedup $= \frac{t_{seq}}{t_{opt}} = 2.800$

Part (ii)

Average execution time for sequential program, $t_{seq} = 293.896$ seconds
Average execution time for OpenMP program, $t_{omp} = 30.157$ seconds

Observed speedup $= \frac{t_{seq}}{t_{omp}} = 9.745$