A1 - Report

Pratyush V Moorthy

December 2019

This is a brief report on the techniques, findings and inferences of the experiments done on each function analysed as part of the A1 assignment.

1 P1

On analysing the function p1.c, we can see that both y[i] and z[i] are calculated in the same nested loop. A[j][i] is accessed to calculate y[i], which is a definite cache miss for every iteration, and our first bottleneck. If we traverse the matrix A by row order, while simultaneously traversing the array y in every inner iteration, we can significantly decrease the number of cache misses without losing consistency in what the previous algorithm achieved. z[i] is computed in the same manner, but in a separate nested loop. Further, each loop is unrolled to achieve better performance.

By this method, we can observe a consistent increase in the computing speed by up to **10 times**.

The reasoning behind this is that since memory is loaded in the cache as blocks, row order traversal gives much lesser cache misses than column order traversal, which simply gives a cache miss always. The reason for choosing 8 as a factor of unrolling is that since a block is of size 64 bytes, and the double datatype is of size 8 bytes, we can say 8 values are loaded for each cache miss. So in row order traversal, we get 7 cache hits between each cache miss. So when we decrease the number of iterations accordingly, it significantly decreases the time taken by the program.

It is also interesting to note that converting array traversal to traversal with pointers slows down the program. This is because in an array, elements are loaded in the cache upholding spatial locality(as blocks), whereas in the case of pointers, only the single element is loaded. Therefore, a cache miss is incurred every time instead of once every eight iterations. This is because a pointer treats every element singularly whereas an array is treated as an ordered sequence of elements.

2 P2

On observing the function p2.c, we see that the matrix B is a symmetric matrix. The instinctive first approach would therefore be to compute only the minor triangle and equate correspondingly to the major triangle, or vice versa. We can notice that a[j][i] is always a cache miss. However, when we follow the computing minor/major triangle approach, we do not reduce the number of cache misses, but also increase the number of instructions, which is much more expensive than the number of iterations we reduce. This is therefore not the ideal approach.

We therefore simply unroll the outer loop as well as the inner loop without modifying the instruction. Any modification will have at least one data to be always a cache miss, and so, as long as we have minimal number of total instructions, we get optimal performance.

We can observe a consistent speedup of the program of a little more than **4 times**.

3 P3

The function p3.c is a nested loop with three levels of iterations, and only one instruction per iteration. Therefore, we must focus on coming up with an approach such that the number of cache misses are reduced(since number of instructions is already minimal). We can observe that both a[k][i] and b[k][j] are column order traversals of the inner-most loop, and therefore always cache misses. But if we make j as the innermost iteration variable, and k as the iteration variable of the loop in the middle, we can observe that matrices c and b are now traversed by row order, leading to lesser cache misses. We cannot make k as the iteration variable of the outermost loop as it is dependent on i.

However, we must note that loop unrolling of the outer loops is not possible due to the data dependency. Unrolling the inner loop experimentally gives slower performance than when not unrolled. Therefore we avoid loop unrolling in this case.

In this method, the program achieves a speedup of more than **20 times**.