HPC Tools 2024, Exercise 4 solutions

Bruno Oliveira Cattelan, Laurent Chôné

Problem 1

a)

In this first problem, we apply optimisation strategies to the following loop (example in C++):

```
for ( int idx = 0 ; idx < maxIdx ; idx++) {
    if ( idx < 500) {
        arrayA[idx] = 4.0 * arrayB[idx] + arrayB[idx + 1];
    }
    else {
        arrayA[idx] = 4.0 * arrayB[idx + 1] + arrayB[idx];
    }
}
```

The foremost issue we notice is conditional `if` statement inside the loop. Since the condition is a threshold in `idx`, the loop can be split in two instead (case "no-if"). We can try to optimise further by manually unrolling the loop, giving e.g. the following:

```
for ( int idx = 0 ; idx < 497 ; idx += 4) {
    arrayA[idx] = 4.0 * arrayB[idx] + arrayB[idx + 1];
    arrayA[idx+1] = 4.0 * arrayB[idx+1] + arrayB[idx + 2];
    arrayA[idx+2] = 4.0 * arrayB[idx+2] + arrayB[idx + 3];
    arrayA[idx+3] = 4.0 * arrayB[idx+3] + arrayB[idx + 4];
}
for ( int idx = 500 ; idx < maxIdx - 3 ; idx += 4) {
    ...
}
```

Timing the program yields the statistics plotted below. Unfortunately, while we have obtained a speedup of 1.17 and 1.31 respectively when using O0 optimisation, our attempts fail to beat O3 optimisation, causing a speeddown of 0.97 and 0.91.

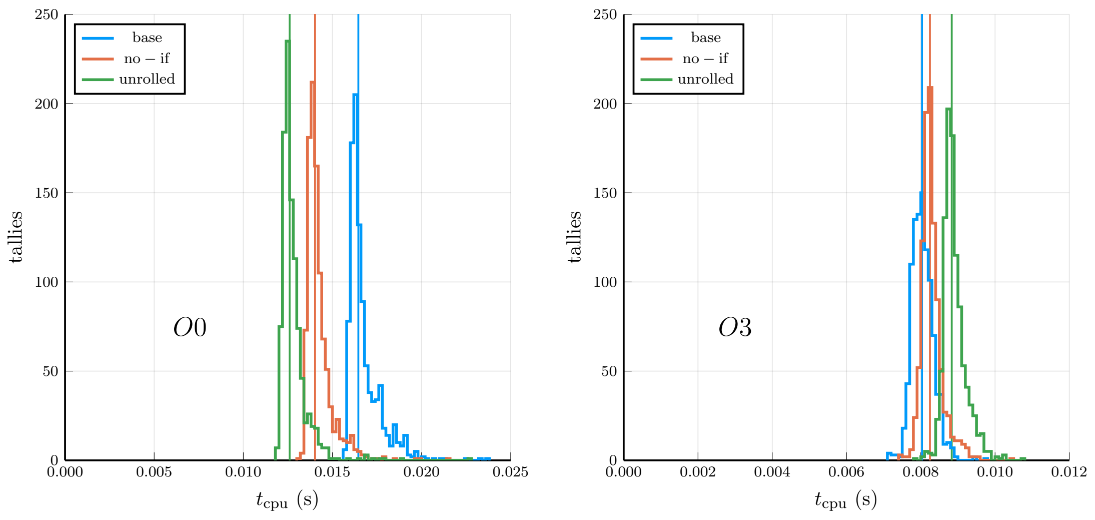

Figure 1: CPU time statistics over 1000 runs of the base and optimised loops, with O0 and O3 optimisation. Vertical lines indicate the median.

b)

In this second problem, we seek to optimise a loop accessing a 2d array. As 2d arrays are not a standard feature of C or C++, let us consider how this can be achieved in practice.

The most reliable way is to flatten the multi-dimensional array into a 1d array, and rely on indexing:

```
    a_ij = a[idxI + sizeI * idxJ] //Row major
    a_ij = a[sizeJ * idxI + idxJ] //Column major
```

As our baseline case, we implement the given loop using row-major indexing (which may be considered more intuitive, although this is subjective), while the optimised case uses column-major indexing.

The reason behind this choice in the context of this problem is the following: using row-major indexing results on discrete memory access, since the inner loop is on `idxJ`, causing us to jump within the 1d array. Using column-major indexing, we access elements of the 1d in order, resulting in much more efficient contiguous access.

In fact, contrary to problem a), here we obtain a modest 1.09 speedup using O0 optimisation, but a considerable 6.51 speedup using O3.

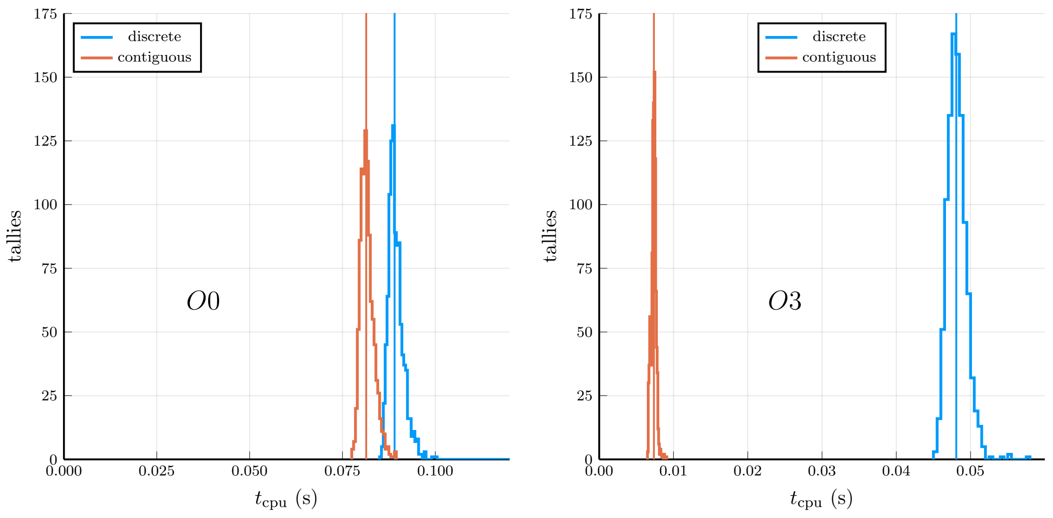

Figure 2: CPU time statistics over 1000 runs of the base and optimised loops, with O0 and O3 optimisation. Vertical lines indicate the median.

Problem 2

In this problem we wish to implement the matrix product $C = AB$, which we can write as:

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

Resulting in a nested loop of the form:

```
std::size_t idxI, idxJ, idxK;
for ( idxI = 0 ; idxI < size ; idxI++ )
{
    for ( idxJ = 0 ; idxJ < size ; idxJ++ )
    {
        for ( idxK = 0 ; idxK < size ; idxK++ )
        {
            arrayC[idxI, idxJ] += arrayA[idxI, idxK]  * arrayB[idxK,
                idxJ];
        }
    }
}
```

As before, the indexing of the array class can be done in row or column major, which we compare. Further, we try optimising by storing the dot product to a variable instead of accessing directly the element c_{ij} every time.

We find that, with O0 optimisation, the different layout are within 1% of CPU time, whereas the temporary dot product yields a speedup of 1.36. With O3 optimisation however, all three are very similar, with the column-major yielding a 1.03 speedup and dot product 1.04.

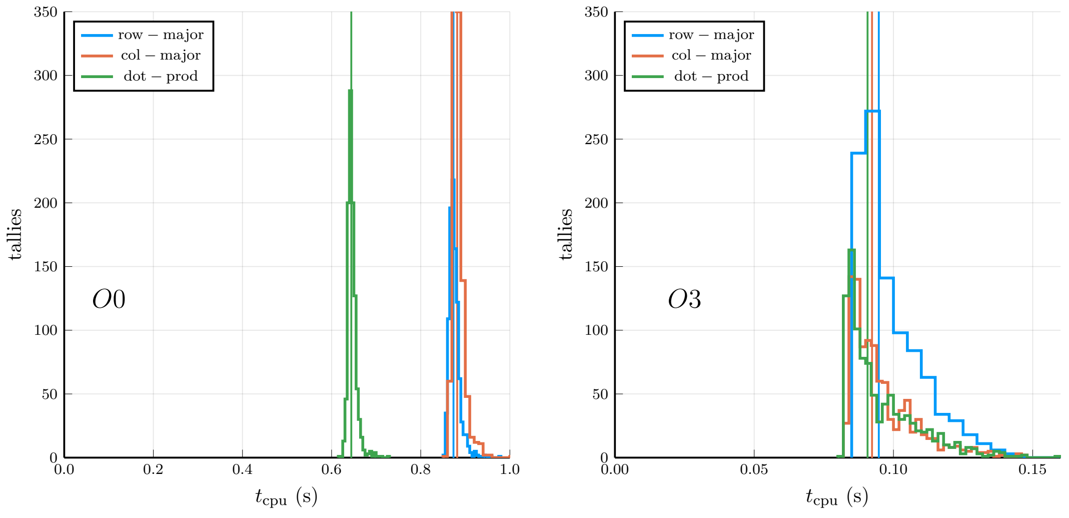

Figure 3: CPU time statistics over 1000 runs of the base and optimised loops, with O0 and O3 optimisation. Vertical lines indicate the median.

To compare the three products, AB, A^TB and AB^T, we choose the fastest implementation above, and modify it as needed (swapping indices). Now we find that AB^T is the fastest in both optimisation cases, as it results in the most favourable access of the data (contiguous for both A and B), yielding a speedup of 1.05 with O0 and 1.12 with O3. On the other hand A^TB results in discrete access for both A and B, which causes a noticeable performance cost (0.91 with O0, and 0.65 with O3).

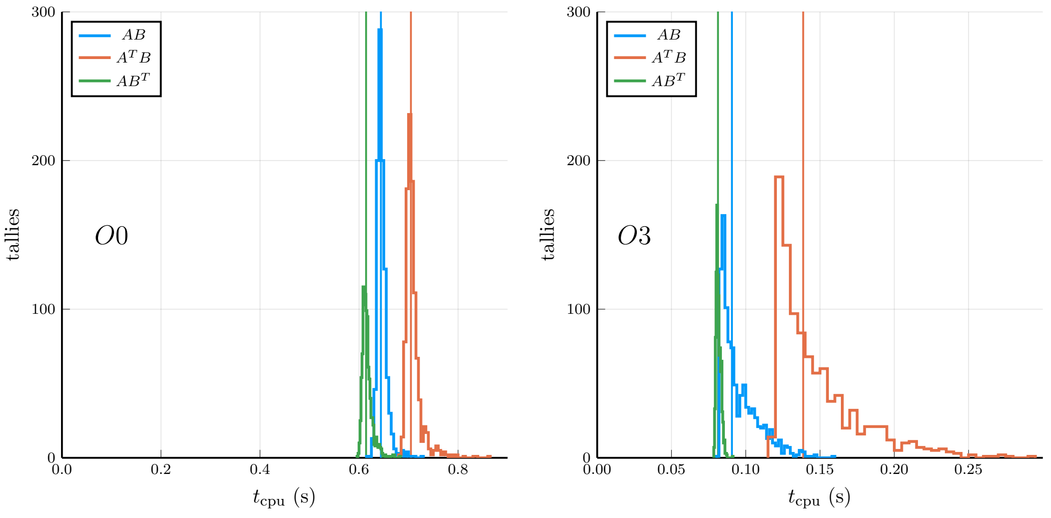

Figure 4: CPU time statistics over 1000 runs of the base and optimised loops, with O0 and O3 optimisation. Vertical lines indicate the median.

Problem 3

Loop unrolling is a classic optimization method. The idea is to explicitly expand the loops of, for example, a for loop. It can be seen as a trade-off between speed and size, as you increase the binary size by literally adding new lines of code previously covered by the loop.

It CAN have multiple benefits to a program. For example, no risk of missing a branch prediction. Similarly, there is a reduction in the overall instructions as the loop computation is reduced. Finally, if the lines are independent of each other they can be trivially parallelized.

So, why don't we always do it? First of all, compilers can decide when it will be beneficial to do the loop unrolling usually much better than we can. This is why in the Figure we can see that when using -O3 there is no benefit on loop unrolling. Furthermore, we increase our binary size. For our cases it usually does not matter, but for example embedded systems binary size can be quite important. Finally, we can potentially reduce the effectiveness of the cache, increasing its misses. It is also worth mentioning that in our case here the loop is also a power of 2, just like the loop unrolls. Now, if that was not the case more care would have been needed to make sure the array size would be respected.

```
filename="roll_"
for i in $(seq 0 9)
     do
        iter=$((2**$i))
        echo "#include <stdlib.h>" > $filename$i".c"
        echo "#include <stdio.h>" >> $filename$i".c"
        echo "#include <time.h>" >> $filename$i".c"
        echo "#include <math.h>" >> $filename$i".c"

        echo "#define N 1048576" >> $filename$i".c"
        echo "int a[N];" >> $filename$i".c"

        echo "void main() {" >> $filename$i".c"
        echo "int i,j;" >> $filename$i".c"
        echo "clock_t t1,t2;" >> $filename$i".c"
        echo "double cputime;" >> $filename$i".c"

        echo "t1=clock();" >> $filename$i".c"


        echo "for (i=0;i<N;i=i+"$iter"){" >> $filename$i".c"
        for j in $(seq 1 $iter)
            do
                echo "a[i+"$j"-1]=(i)/2;" >> $filename$i".c"
            done
        echo "}" >> $filename$i".c"
        echo "t2=clock();" >> $filename$i".c"

        echo "double sum = 0;" >> $filename$i".c"
        echo "for (i=0;i<N;i+=1) sum +=a[i];" >> $filename$i".c"

        echo "printf(\"M: %f \",sum);" >> $filename$i".c"
        echo "printf(\"\n\");" >> $filename$i".c"

        echo "cputime=(double)(t2-t1)/CLOCKS_PER_SEC;" >> $filename$i".c"
        echo "printf(\"%f\n\",cputime);" >> $filename$i".c"

        echo "}" >> $filename$i".c"

 done
}
```

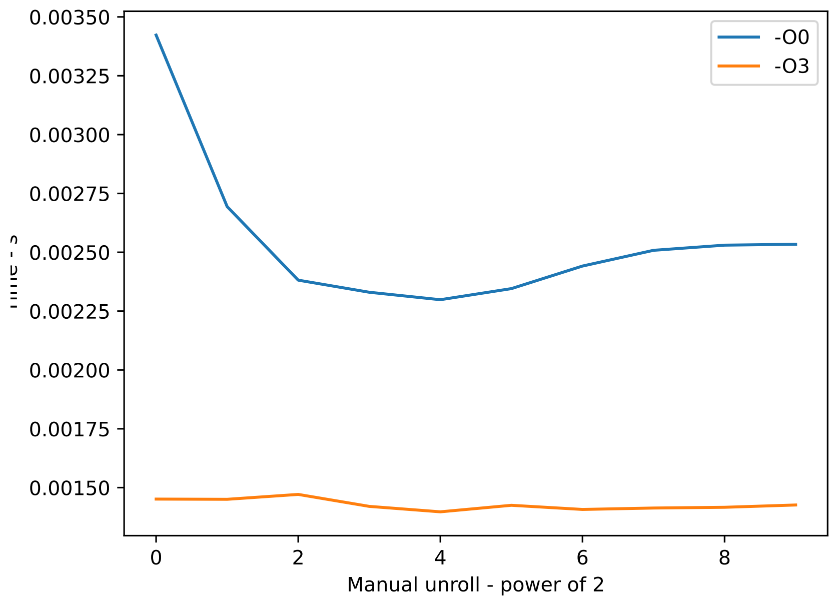

Figure 5: Average of 300 runs for different manual unrolls

Problem 4

For this exercise we are creating an example to showcase the benefits of SIMD extensions. For this I choose the size of my arrays from 32 up to 512. We see that both versions (with and without SIMD extension) behave exponentially as we increase the array size. However, we see a reduction in runtime of almost half.

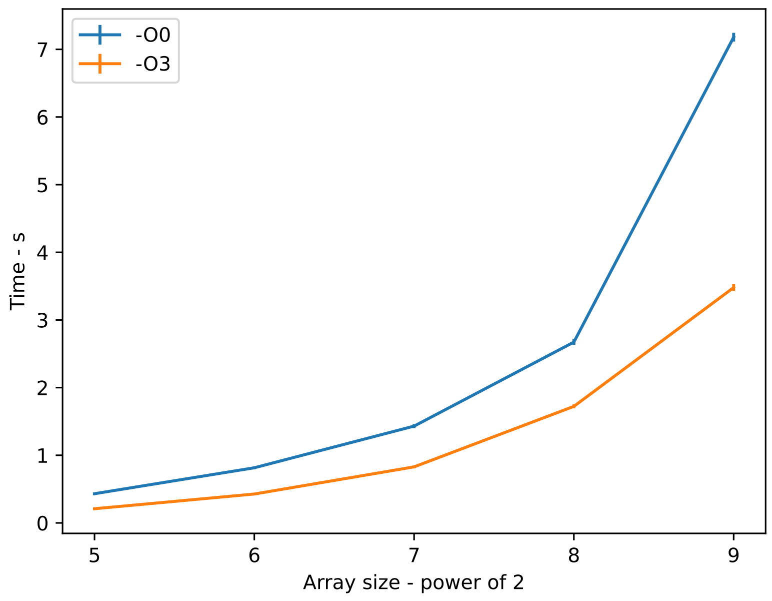

Figure 6: Average of 100 runs with and without SIMD extensions for different array sizes