Assignment 3

Aniket Sanghi (170110)

10th October 2020

I pledge on my honor that I have not given or received any unauthorized assistance.

System and Architecture Information (Used DevCloud)

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list: 0-23
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Stepping:              4
CPU MHz:               1201.063
CPU max MHz:           3700.0000
CPU min MHz:           1200.0000
BogoMIPS:              6800.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              19712K
Memory:                198GB
```

Compilers Information

```
g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0

icc (ICC) 2021.1 Beta 20200827
```

All the Problems have been tested in this system only and the speed-ups were obtained in the same.

1 Solution 1

1.1 Compilation and execution instruction

For GCC

```
g++ -O2 -march=native -o problem1 problem1.cpp
./problem1
```

For ICC

```
icc -O2 -o problem1_icc problem1.cpp
./problem1_icc
```

1.2 Problems with given reference version

- Temporal locality of y_ref and z_ref unused
- Conflicts between both A accesses
- Column-wise access of A leading to cache misses

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    y_ref[j] = y_ref[j] + A[i][j] * x[i];
    z_ref[j] = z_ref[j] + A[j][i] * x[i];
  }
}
```

1.3 Proposed Optimised version with justifications

No dependencies in the loop hence all valid loop transformations are allowed.

Loop Split
Eliminate conflict misses between A's accesses

Loop Permutation in 2nd loop
Reduce cache misses because of temporal locality of z_opt and spacial locality of A

Inner-Loop Unrolling in second loop
Reduces overhead caused due to loop comparisons

Outer-Loop Unrolling and Jamming
Reduces overhead caused due to loop comparisons

Speed Ups
GCC - 6X (0.78sec to 0.13sec)
ICC - 11X (0.66sec to 0.06sec)

```
for (i = 0; i < N; i+=2) {
  for (j = 0; j < N; j++) {
    y_opt[j] = y_opt[j] + A[i][j] * x[i]
                        + A[i+1][j] * x[i+1];
  }
}
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i+=8) {
    z_opt[j] = z_opt[j] + A[j][i] * x[i]
                        + A[j][i+1] * x[i+1]
                        + A[j][i+2] * x[i+2]
                        + A[j][i+3] * x[i+3]
                        + A[j][i+4] * x[i+4]
                        + A[j][i+5] * x[i+5]
                        + A[j][i+6] * x[i+6]
                        + A[j][i+7] * x[i+7];
  }
}
```

This has been vectorised with intrinsics. The Speed Up obtained after applying intrinsics are
GCC 8.2X (0.73sec to 0.09sec)
ICC 8.25X (0.66sec to 0.08sec)
The speed up for ICC degraded, the most plausible explanation being that we used SSE intrinsics instead of AVX and others while ICC did use that previously. Also due to increased size, inlining / permutation benefits that ICC took also diminished.
The detailed improvements for individual optimisations are given in section 1.5

1.4 Proposed Optimised version 2 that worked best with ICC

Loop Split
Eliminate conflict misses between A's accesses

Loop Permutation in 2nd loop
Reduce cache misses because of temporal locality of z_opt and spacial locality of A

Inner-Loop Unrolling in second loop
Reduces overhead caused due to loop comparisons

Loop Tiling in first loop
Helps with Spacial locality for A, x and temporal locality for y_opt

Speed Ups
GCC - 5X (0.74sec to 0.15sec)
ICC - 31X (0.62sec to 0.02sec)

```
int BLOCK_SIZE = 16;
for (i = 0; i < N; i+=BLOCK_SIZE) {
  for (j = 0; j < N; ++j) {
    for(i2 = i; i2 < i+BLOCK_SIZE; ++i2) {
      y_opt[j] = y_opt[j] + A[i2][j] * x[i2];
    }
  }
}
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i+=8) {
    z_opt[j] = z_opt[j] + A[j][i] * x[i]
                        + A[j][i+1] * x[i+1]
                        + A[j][i+2] * x[i+2]
                        + A[j][i+3] * x[i+3]
                        + A[j][i+4] * x[i+4]
                        + A[j][i+5] * x[i+5]
                        + A[j][i+6] * x[i+6]
                        + A[j][i+7] * x[i+7];
  }
}
```

1.5 Various tested optimisations with improvements and justifications

This section contains individual optimisations and their corresponding speedups with proper justification and exploration results. All these codes can be found commented in the code.

1.5.1 Loop Interchange

Justification
This will improve temporal locality
for $y\_opt[j]$ & $z\_opt[j]$

Speed Up
GCC - 1.1X (0.74sec to 0.70sec)
ICC - 8X (0.64sec to 0.08sec)

```
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    y_opt[j] = y_opt[j] + A[i][j] * x[i];
    z_opt[j] = z_opt[j] + A[j][i] * x[i];
  }
}
```

Explanation – Optimisation report for ICC (using flag -qopt-report) showed inlining of the function and interchange of loop (1, 2, 3) to (2, 3, 1) suggesting that it performed loop permutation, making the outermost Niter loop (which was detected since function got inlined) as the innermost loop promoting temporal locality even further. That seems to be the most plausible reason for such high speed up with ICC as compared to GCC apart from vectorisation which will also be one of the reason as with icc -O2 optimisations include vectorisation

1.5.2 Loop Interchange + Tiling

Justification
This will improve temporal locality
for $y\_opt[j]$ & $z\_opt[j]$ and
Spacial locality for 2D Array A

Speed Up
GCC - 1.75X (0.74sec to 0.40sec)
ICC - 3X (0.65sec to 0.22sec)

BLOCK_SIZE = 256 seemed nice

```
for (j = 0; j < N; j+=BLOCK_SIZE) {
  for (i = 0; i < N; i+=BLOCK_SIZE) {
    for(j2 = j; j2 < j+BLOCK_SIZE; ++j2) {
      for(i2 = i; i2 < i+BLOCK_SIZE; ++i2) {
        y_opt[j2] = y_opt[j2] + A[i2][j2] * x[i2];
        z_opt[j2] = z_opt[j2] + A[j2][i2] * x[i2];
      }
    }
  }
}
```

Explanation – Spacial locality for A must have given Speed ups in GCC and ICC but we can see ICC performance degraded with respect to above one. Seeing into the optimisation report again (flag -qopt-report) revealed that this time it **didn't perform inlining and loop permutation** probably because of complex loop nest which was the expected reason for high speedUp in above scenario

1.5.3 Loop Interchange + Loop Split + Loop Unrolling

Justification
This will improve temporal locality for $y\_opt[j]$ & $z\_opt[j]$ and Spacial locality for 2D Array A by eliminating conflicts between them

Speed Up
GCC - 1.12X (0.74sec to 0.68sec)
ICC - 30X (0.65sec to 0.02sec)

```
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    y_opt[j] = y_opt[j] + A[i][j] * x[i];
  }
}
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i+=8) {
    z_opt[j] = z_opt[j] + A[j][i] * x[i]
                        + A[j][i+1] * x[i+1]
                        .....
                        + A[j][i+7] * x[i+7];
  }
}
```

Explanation – Loop split eliminated the conflicts the row-wise and column-wise access of A were having in the above case. Unrolling also further increased the speedUp (8 seemed optimal, further increment degraded performance). But this didn't improve much performance with GCC, bottleneck being above loop as it has column-wise access of A.
Tremendous increment of ICC is surprising. Upon checking the compiler optimisation report, it showed that it inlined the function and also permuted all 3 loop nest (as we saw in above too). Added to it it vectorised the internal loop as well.

1.5.4 Loop Split + Loop Interchange in 1

Justification
This will improve temporal locality for $z\_opt[j]$ and Spacial locality for 2D Array A more than above case as both are independent row-wise access now, also they won't clash with each other now

Speed Up
GCC - 5.3X (0.74sec to 0.14sec)
ICC - 11X (0.65sec to 0.06sec)

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    y_opt[j] = y_opt[j] + A[i][j] * x[i];
  }
}
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    z_opt[j] = z_opt[j] + A[j][i] * x[i];
  }
}
```

Explanation – In the previous scenario both accesses to A were conflicting with each other as one was row-wise and other column-wise. But here since both are row-accesses and conflicts also nullified, This is making much better use of Cache and Spacial Locality. Huge increment of ICC is again due to inlining and permutation with 3rd outer loop as evident from -qopt-report. It did incline code here but did permutation only for 2nd loop and not first here maybe because the compiler couldn't detect that optimisation here.

2 Solution 2

2.1 Compilation and execution instruction

For GCC

```
g++ -O2 -o problem2 problem2.cpp
./problem2
```

For ICC

```
icc -O2 -o problem2_icc problem2.cpp
./problem2_icc
```

2.2 Problems with given reference version

- Conflict misses due to clash between array accesses
- Column-wise access of A leading to cache misses
- Permutation order (i, j, k) is best though

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    for (k = 0; k < i + 1; k++) {
      C[i][j] += A[k][i] * B[j][k];
    }
  }
}
```

2.3 Proposed Optimised version with justifications

No dependencies in the loop hence all valid loop transformations are allowed.

i-Loop Unrolling and Jamming
Reduces overhead due to loop comparisons
Taking benefits of Spacial locality of A now (reducing cache misses)

j-Loop Unrolling and Jamming
Reduces overhead due to loop comparisons
Taking benefits of Spacial locality of B now (reducing cache misses)

j-Loop Peeling
To safeguard from *if* comparisons in k-loop peel the loop for the k = i+1 case

Speed Ups
GCC - 3X (0.62sec to 0.20sec)
ICC - 3.3X (0.54sec to 0.16sec)

This has been vectorised with intrinsics. The Speed Up obtained after applying intrinsics are
GCC 4X (0.62sec to 0.16sec)
ICC 3.3X (0.54sec to 0.16sec)
No change with ICC as ICC vectorises even with -O2 flag so it already vectorised without explicit intrinsics as well

```
for (i = 0; i < N; i+=2) {
  for (j = 0; j < N; j+=4) {
    double t1 = C[i][j], t2 = C[i][j+1];
    double t3 = C[i][j+2], t4 = C[i][j+3];
    double t11 = C[i+1][j], t21 = C[i+1][j+1];
    double t31 = C[i+1][j+2], t41 = C[i+1][j+3];
    for (k = 0; k < i + 1; k++) {
      t1 += A[k][i] * B[j][k];
      t2 += A[k][i] * B[j+1][k];
      t3 += A[k][i] * B[j+2][k];
      t4 += A[k][i] * B[j+3][k];
      t11 += A[k][i+1] * B[j][k];
      t21 += A[k][i+1] * B[j+1][k];
      t31 += A[k][i+1] * B[j+2][k];
      t41 += A[k][i+1] * B[j+3][k];
    }
    t11 += A[i+1][i+1] * B[j][i+1];
    t21 += A[i+1][i+1] * B[j+1][i+1];
    t31 += A[i+1][i+1] * B[j+2][i+1];
    t41 += A[i+1][i+1] * B[j+3][i+1];

    // Update all C's from ti's here
  }
}
```

Other optimisations carried out as experiments have been commented in the code with the speed ups obtained.

2.4 Various tested optimisations with improvements and justifications

2.4.1 i-j Loop Tiling

Expected speed-up due to spacial locality benefits for arrays but got very minute increment
GCC - 1.04X
ICC - 1.3X

2.4.2 Loop permutations

Tried various loop permutations but didn't get any improvement. The given loop permutation appeared to be best. Others degraded performance except (k, i, j) permutation which almost gave same time values as reference version.

2.4.3 Loop Unrolling of j-loop with Jamming

Expected speed-up due to better spacial locality and decreased number of loop comparison computations
GCC - 2.4X
ICC - 2.5X

3 Solution 3

3.1 Compilation and execution instruction

For GCC

```
g++ -O2 -fopenmp -o problem3 problem3.cpp
./problem3
```

3.2 Pre-processing and information

- Only j-th loop can be parallelised as others carry dependences
- Only (k i j), (k j i), (j k i) loop permutations are valid

3.3 Proposed Optimised/Parallel version with justifications

Parallelised j-loop when it was innermost degraded performance. The normal (j k i) permutation takes 9sec as compared to 1sec of (k i j) permutation but after parallelising with OpenMP and making use of multiple cores and some transformation, The time of (j, k, i) version got down to 0.16 on devCloud
Speed Up - 6.5X (1.1sec to 0.16sec) on devCloud

Loop Unrolling + Jamming
Helping with Spacial locality for all accesses of A

Loop Tiling
Blocking did help with accesses

OMP Parallelism
Parallelised the outermost tiled j-loop. After tiling individual outer j loop is equivalent to 16 sized chunk. This came out to be the optimal block-size and unrolling degree (4) for best performance with OpenMP parallel version. Tried with manual schedule and other clauses but default gave best results.

```
int BS = 16;
#pragma omp parallel for
for (j = 0; j < (N - 1); j+=BS) {
  for (k = 0; k < ITER; k++) {
    for (i = 1; i < N; i+=BS) {
      for(j2 = j; j2 < min(j+BS, N-1); j2+=4) {
        for(i2 = i; i2 < min(i+BS, N); i2++) {
          A[i2][j2 + 1] = A[i2 - 1][j2 + 1] + A[i2][j2 + 1];
          A[i2][j2 + 2] = A[i2 - 1][j2 + 2] + A[i2][j2 + 2];
          A[i2][j2 + 3] = A[i2 - 1][j2 + 3] + A[i2][j2 + 3];
          A[i2][j2 + 4] = A[i2 - 1][j2 + 4] + A[i2][j2 + 4];
        }
      }
    }
  }
}
```