Assignment 3

Ayush Kumar (170195)

7 October 2020

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

Solutions

Please note that all benchmarking for this assignment was performed on the Intel DevCloud using Intel's OneAPI. The specifications of the Linux system of the compute node on which the calculations were performed are listed below. Also, the versions of GCC and ICC compilers used were **gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1 18.04.1)** and **icc version 2021.1 Beta (gcc version 7.4.0 compatibility)**. For all evaluations that we performed in this assignment, we took the median runtime of 5 consecutive runs of the program.

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:	1200.480
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
NUMA node0 CPU(s):	0-5,12-17
NUMA node1 CPU(s):	6-11,18-23

Table 1: System Specifications

1. First we will benchmark using the GCC compiler and then using the ICC compiler and later on we will compare both the results as well as reason about our findings.

   GCC Compiler:

   (a) Running **the unoptimized reference version of the code gives a runtime of 0.77698 seconds**. To optimize this, we apply various loop transformations such as **tiling**, **loop interchange**, **loop splitting** and different combinations of these.

- **Tiling**
  Tiling of the two loops was first performed with different sizes to get the optimum tile size. The following table (Table 2) and the graph (Figure 1) illustrates the speedups obtained using Tiling. Clearly from the table it can be inferred that **512 and 1024 tile sizes give the optimum performance**. Note that these sizes are system dependent (in particular, they depend on the cache sizes). Hence for this problem, we will focus our analyses on tiles of similar size.

Tiling Size ($N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
2x2	0.5829	1.3330
4x4	0.4419	1.7583
8x8	0.5289	1.4692
16x16	0.4870	1.5956
32x32	0.4640	1.6745
64x64	0.4487	1.7317
128x128	0.4112	1.8895
256x256	0.3922	1.9810
512x512	**0.3850**	**2.0183**
1024x1024	**0.3836**	**2.0258**
2048x2048	0.5168	1.5034

Table 2: GCC: Runtimes obtained using Tiling

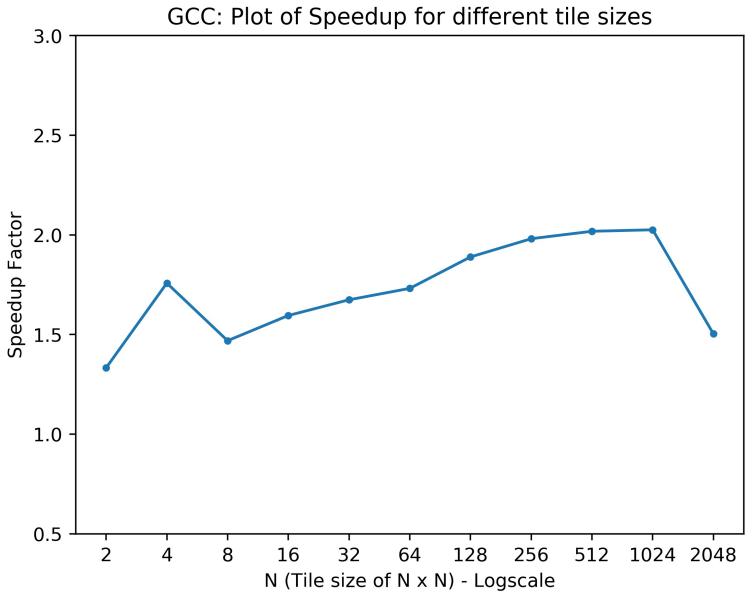

Figure 1: GCC: Plot of Speedup vs Tile Size

- **Loop Interchange**
  Next we measure the performance obtained by **interchanging the i and j loops**. This gives a **runtime of 0.71410 seconds (speedup factor of 1.0881)**. This is expected since the $-O2$ flag in GCC doesn't enable auto-vectorization, hence only interchanging the loops does not improve the runtime by much. The minor improvement observed may be due to slightly better temporal/spatial locality.

- **Loop Splitting and Interchange**
  Next, we apply **Loop splitting and interchange of the i and j loops**. We first split the two statements into two different loops of their own, and for the second loop we swap the iteration order of i and j. The code now looks like this,

```
void optimized(double** A, double* x, double* y_opt, double* z_opt) {
  int i, j;
  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];
    }
  }
}
```

This results in a **runtime of 0.12814 seconds (speedup factor 6.0635)**. The significant improvement is due to better spatial locality in accessing the matrix A. Since A is accessed in row major order in both the loops, there are much more cache hits.

- **Tilling with Loop Splitting and Interchange**
  We now combine tiling with loop splitting and interchange in the hope of getting more speedups. The following table (1a) shows the speedups in this case. Though the speedups are significant, they aren't better than the one we obtained with just loop splitting and interchange. One reason can be the overheads caused by the extra loops due to tiling.

Tiling Size ($N \times N$)	Runtime (in seconds)	Speedup Factor
256x256	0.1880	4.1333
512x512	0.1560	4.9790
1024x1024	0.1410	5.5121
2048x2048	**0.1332**	**5.8319**

Table 3: GCC: Runtimes obtained using Tiling with Loop Splitting & Interchange

Note that we did not perform loop unrolling/jamming as it did not bring in any significant improvement in the runtime.

(b) Finally, for the best loop transformation we obtained in (a) (Loop Splitting and Interchange), we **vectorize the inner loops using intrinsics**. We used xmmintrinsics with 128-bit wide vector registers which can accommodate 2 double precision floating point numbers at a time. **Runtime obtained was 0.08503 seconds (speedup factor 9.1377)**. As expected, since GCC doesn't auto-vectorize the code, vectorizing using intrinsics results in a significant improvement. The vectorized code for the most optimized version is shown below.

```
void optimized(double** A, double* x, double* y_opt, double* z_opt) {
  int i, j;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j += 2) {
      __m128d r1, r2, r3, r4;
      r1 = _mm_loadu_pd(&y_opt[j]);
      r2 = _mm_loadu_pd(&A[i][j]);
      r3 = _mm_set1_pd(x[i]);
      r4 = _mm_add_pd(r1, _mm_mul_pd(r2, r3));
      _mm_store_pd(&y_opt[j], r4);
    }
  }
  for (j = 0; j < N; j++) {
    for (i = 0; i < N; i += 2) {
      __m128d r1, r2, r3, r4;
      r1 = _mm_loadu_pd(&x[i]);
      r2 = _mm_loadu_pd(&A[j][i]);
      r3 = _mm_mul_pd(r1, r2);
      r4 = _mm_hadd_pd(r3, r3);
```

```
      z_opt[j] = z_opt[j] + _mm_cvtsd_f64(r4);
    }
  }
}
```

Finally, the analyses were repeated with the ICC compiler.

ICC Compiler:

(a) The **runtime of the unoptimized reference version of the code was 0.65544 seconds**. We again apply the same methods as we had done with GCC.

- **Tiling**
  The runtimes obtained by Tiling are shown in Table 4 and Figure 2. The results are similar to the one obtained with GCC. Once again **tile sizes 512 and 1024 turn out to be the optimum**. The **speedup of almost 3 times is greater than the speedup of 2 obtained with GCC**.

Tiling Size ($N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
2x2	0.3992	1.6417
4x4	0.2905	2.2561
8x8	0.2656	2.4680
16x16	0.3240	2.0232
32x32	0.2903	2.2581
64x64	0.2793	2.3463
128x128	0.2476	2.6467
256x256	0.2367	2.7694
512x512	**0.2189**	**2.9945**
1024x1024	**0.2186**	**2.9984**
2048x2048	0.4083	1.6051

Table 4: ICC: Runtimes obtained using Tiling

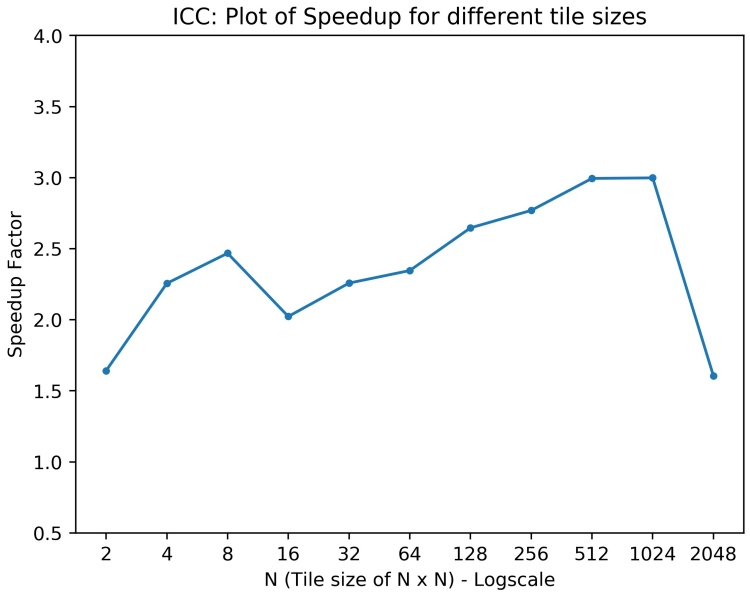

Figure 2: ICC: Plot of Speedup vs Tile Size

- **Loop Interchange**
  Interchanging loops i and j result in a runtime of 0.07803 seconds (speedup factor

8.3998). This is surprising since loop interchange itself did not give much improvement with GCC. The apparent reason for such a significant speedup is auto-vectorization by ICC. Note that GCC only performs auto-vectorization when supplied with the flags $-O3$ or $-ftree-vectorize$ which is why the runtime was much larger with GCC.

- **Loop Splitting and Interchange**
  Runtime of 0.06073 seconds (speedup factor 10.7927) was obtained.

- **Tiling with Loop Splitting and Interchange**
  We can see that **performance improves even more** when using loop splitting & interchange is done along with tiling. The reason is probably better cache temporal/spatial locality.

Tiling Size ($N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
256x256	0.0706	9.2904
512x512	0.0637	10.2959
1024x1024	0.0583	11.2425
2048x2048	**0.0545**	**12.0286**

Table 5: ICC: Runtimes obtained using Tiling with Loop Splitting & Interchange

(b) Finally, once again we use intrinsics to vectorize the most optimized version obtained using loop transformations in part (a) (Tiling with Loop Splitting and Interchange).

```
void optimized(double** A, double* x, double* y_opt, double* z_opt) {
  int i, j, it, jt;
  const int block_size = 2048;
  for(it = 0; it < N; it += block_size) {
    for (jt = 0; jt < N; jt += block_size) {
      for (i = it; i < it+block_size; i++) {
        for (j = jt; j < jt+block_size; j += 2) {
          __m128d r1, r2, r3, r4;
          r1 = _mm_loadu_pd(&y_opt[j]);
          r2 = _mm_loadu_pd(&A[i][j]);
          r3 = _mm_set1_pd(x[i]);
          r4 = _mm_add_pd(r1, _mm_mul_pd(r2, r3));
          _mm_store_pd(&y_opt[j], r4);
        }
      }
      for (j = jt; j < jt+block_size; j++) {
        for (i = it; i < it+block_size; i += 2) {
          __m128d r1, r2, r3, r4;
          r1 = _mm_loadu_pd(&x[i]);
          r2 = _mm_loadu_pd(&A[j][i]);
          r3 = _mm_mul_pd(r1, r2);
          r4 = _mm_hadd_pd(r3, r3);
          z_opt[j] = z_opt[j] + _mm_cvtsd_f64(r4);
        }
      }
    }
  }
}
```

The final **vectorized code resulted in a runtime of 0.07673 seconds (speedup factor 8.5421)**. Note that **vectorizing the code with intrinsics reduces the speedup from 12.0286 to 8.5421**. The most likely cause of this is that, the version of code without intrinsics was auto-vectorized by the ICC compiler which generated more efficient machine code (perhaps using better hardware support) than was generated using the intrinsic code written by me.

2. **GCC Compiler**

(a) A dry run of the **reference version of the code gave a runtime of 0.62117 seconds**.

- **Tiling**
  In this case, we performed 3D tiling with different tile sizes and even though speedups were obtained in some cases, the only **significant speedup was for tile size 8. The runtime obtained was 0.4205 (speedup factor 1.4771)**. The results are shown in Table 6 and Figure 3

Tiling Size ($N \times N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
2x2x2	0.7193	0.8636
4x4x4	0.4755	1.3063
8x8x8	0.4205	1.4771
16x16x16	0.4477	1.3873
32x32x32	0.4735	1.3119
64x64x64	0.5049	1.2302
128x128x128	0.5596	1.1100
256x256x256	0.5846	1.0626
512x512x512	0.6086	1.0206

Table 6: GCC: Runtimes obtained using Tiling

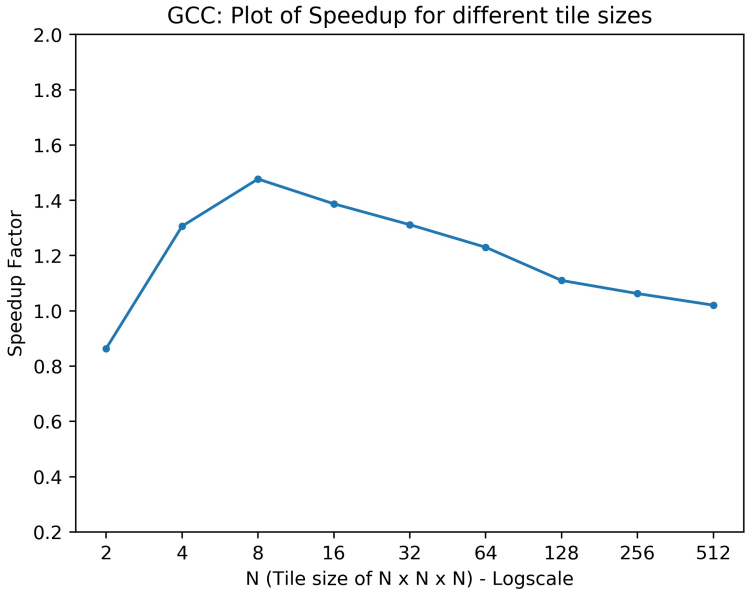

Figure 3: GCC: Plot of Speedup vs Tile Size

- **Loop Permutation**
  Permutations of the i, j and k loops were applied but **either none or negative speedup was observed**. The runtimes obtained for kij, kji, jki, jik and ikj were 0.65568, 0.76914, 0.68140, 0.70771 and 0.74218 seconds respectively.

- **Tiling with Loop Permutation and Unrolling/Jamming**
  Finally, loop unrolling/jamming was performed along with tiling and loop permutation. For all loop permutations, it was observed that unrolling/jamming loop j 4 times yielded the best performance (Unrolling/Jamming 8 times was also tried but the performance improvement was either insignificant or none). **All results shown below are for loop j unrolled/jammed 4 times**. The ijk **variant of the loop with tiling size of 32x32x32 and loop j unrolled/jammed 4 times gave the best performance. The runtime obtained was 0.19134 seconds (speedup factor 3.2464)**. The jik variant also had performance very close to this. All combinations that we tried and their results on different tile sizes are tabularized below. The code has also been shown at the end.

– **kij**

Tiling Size ($N \times N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
4x4x4	0.4377	1.4193
8x8x8	0.3815	1.6284
16x16x16	0.4101	1.5146
32x32x32	0.3802	1.6336
64x64x64	0.3769	1.6482
128x128x128	**0.3663**	**1.6957**
256x256x256	0.3741	1.6603
512x512x512	0.4909	1.2654

Table 7: GCC: Runtimes for Tiling with kij permutation, j unrolled/jammed 4 times

– **kji**

Tiling Size ($N \times N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
4x4x4	0.3632	1.7102
8x8x8	0.3188	1.9484
16x16x16	0.3198	1.9424
32x32x32	**0.3083**	**2.0148**
64x64x64	0.3231	1.9228
128x128x128	0.3175	1.9563
256x256x256	0.3393	1.8307
512x512x512	0.4293	1.4470

Table 8: GCC: Runtimes for Tiling with kji permutation, j unrolled/jammed 4 times

– **jki**

Tiling Size ($N \times N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
4x4x4	0.3580	1.7350
8x8x8	0.3055	2.0334
16x16x16	0.3167	1.9614
32x32x32	0.3024	2.0543
64x64x64	0.2938	2.1140
128x128x128	**0.2936**	**2.1159**
256x256x256	0.2960	2.0984
512x512x512	0.3370	1.8435

Table 9: GCC: Runtimes for Tiling with jki permutation, j unrolled/jammed 4 times

– **jik**

Tiling Size ($N \times N \times N$)	**Runtime (in seconds)**	**Speedup Factor**
4x4x4	0.2862	2.1707
8x8x8	0.2250	2.7603
16x16x16	0.2247	2.7639
32x32x32	0.1989	3.1233
64x64x64	**0.1926**	**3.2247**
128x128x128	0.1979	3.1380
256x256x256	0.2129	2.9181
512x512x512	0.2622	2.3689

Table 10: GCC: Runtimes for Tiling with jik permutation, j unrolled/jammed 4 times

– **ikj**

Tiling Size ($N \times N \times N$)	Runtime (in seconds)	Speedup Factor
4x4x4	0.4361	1.4245
8x8x8	0.3954	1.5710
16x16x16	**0.3570**	**1.7398**
32x32x32	0.3653	1.7004
64x64x64	0.3606	1.7227
128x128x128	0.3581	1.7347
256x256x256	0.3915	1.5868
512x512x512	0.5595	1.1102

Table 11: GCC: Runtimes for Tiling with ikj permutation, j unrolled/jammed 4 times

– **ijk**

Tiling Size ($N \times N \times N$)	Runtime (in seconds)	Speedup Factor
4x4x4	0.2791	2.2259
8x8x8	0.2178	2.8522
16x16x16	0.2038	3.0485
32x32x32	**0.1913**	**3.2464**
64x64x64	0.1935	3.2100
128x128x128	0.1957	3.1736
256x256x256	0.2242	2.7704
512x512x512	0.2737	2.2696

Table 12: GCC: Runtimes for Tiling with ijk permutation, j unrolled/jammed 4 times

```
// ijk variant
void optimized(double** A, double** B, double** C) {
  int i, j, k, it, jt, kt, i_max, j_max, k_max, kt_max;
  const int block_size = 32;
  for (it = 0; it < N; it += block_size) {
    for (jt = 0; jt < N; jt += block_size) {
      for (kt = 0; kt < it+block_size; kt += block_size) {
        for (i = it; i < it+block_size; i++) {
          for (j = jt; j < jt+block_size; j += 4) {
            double tmp = 0, tmp1 = 0, tmp2 = 0, tmp3 = 0;
            for (k = kt; k < min(i+1, kt+block_size); k++) {
              // C[i][j] += A[k][i] * B[j][k];
              double tmp4 = A[k][i];
              tmp += tmp4 * B[j][k];
              tmp1 += tmp4 * B[j+1][k];
              tmp2 += tmp4 * B[j+2][k];
              tmp3 += tmp4 * B[j+3][k];
            }
            C[i][j] += tmp;
            C[i][j+1] += tmp1;
            C[i][j+2] += tmp2;
            C[i][j+3] += tmp3;
          }
        }
      }
    }
  }
}
```

(b) **We now vectorize the optimal code** for the previous part (Tiling with tile size 32x32x32 for loop permutation ijk with j unrolled/jammed 4 times) **using intrinsics**. This gave us a minor performance

improvement. The reason is again that GCC fails to vectorize the code which we have performed using intrinsics. **The final runtime obtained was 0.16479 seconds (speedup factor 3.7695)**. This was a **speedup of 1.1611 over the optimal version without using intrinsics**. The final code using vector instructions is shown below.

```
// ijk variant using intrinsics
void optimized(double** A, double** B, double** C) {
  int i, j, k, it, jt, kt, i_max, j_max, k_max, kt_max;
  const int block_size = 32;
  for (it = 0; it < N; it += block_size) {
    for (jt = 0; jt < N; jt += block_size) {
      for (kt = 0; kt < it+block_size; kt += block_size) {
        for (i = it; i < it+block_size; i++) {
          for (j = jt; j < jt+block_size; j += 4) {
            __m128d tmp = {0, 0}, tmp1 = {0, 0};
            for (k = kt; k+1 < min(i+1, kt+block_size); k += 2) {
              // vectorized version
              __m128d r4 = {A[k][i], A[k+1][i]};
              __m128d r = _mm_loadu_pd(&B[j][k]);
              __m128d r1 = _mm_loadu_pd(&B[j+1][k]);
              __m128d r2 = _mm_loadu_pd(&B[j+2][k]);
              __m128d r3 = _mm_loadu_pd(&B[j+3][k]);
              r = _mm_mul_pd(r4, r);
              r1 = _mm_mul_pd(r4, r1);
              r2 = _mm_mul_pd(r4, r2);
              r3 = _mm_mul_pd(r4, r3);
              r = _mm_hadd_pd(r, r1);
              r1 = _mm_hadd_pd(r2, r3);
              tmp = _mm_add_pd(tmp, r);
              tmp1 = _mm_add_pd(tmp1, r1);
            }
            if (k < min(i+1, kt+block_size)) {
              double tmp4 = A[k][i];
              __m128d r = {tmp4 * B[j][k], tmp4 * B[j+1][k]};
              __m128d r1 = {tmp4 * B[j+2][k], tmp4 * B[j+3][k]};
              tmp = _mm_add_pd(tmp, r);
              tmp1 = _mm_add_pd(tmp1, r1);
            }
            __m128d r = _mm_loadu_pd(&C[i][j]);
            __m128d r1 = _mm_loadu_pd(&C[i][j+2]);
            r = _mm_add_pd(r, tmp);
            r1 = _mm_add_pd(r1, tmp1);
            _mm_store_pd(&C[i][j], r);
            _mm_store_pd(&C[i][j+2], r1);
          }
        }
      }
    }
  }
}
```

ICC Compiler

(a) The **reference version of the code gave a runtime of 0.54457 seconds**.

- **Tiling**
  The results for tiling is similar as that obtained with GCC. **Best runtime of 0.30728 seconds (speedup factor 1.7755) was obtained with a tile size of 128x128x128**.

Tiling Size ($N \times N \times N$)	Runtime (in seconds)	Speedup Factor
2x2x2	1.1242	0.4853
4x4x4	0.6008	0.9080
8x8x8	0.4407	1.2378
16x16x16	0.3577	1.5253
32x32x32	0.3241	1.6831
64x64x64	0.3124	1.7464
128x128x128	**0.3073**	**1.7755**
256x256x256	0.3157	1.7282
512x512x512	0.4134	1.3197

Table 13: ICC: Runtimes obtained using Tiling

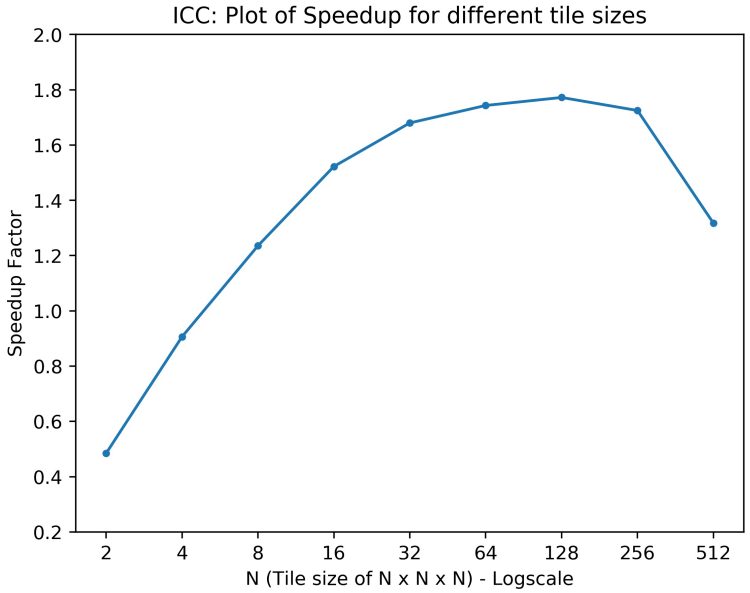

Figure 4: ICC: Plot of Speedup vs Tile Size

- **Loop Permutation**
  Different Loop permutations were applied but none gave decent results. The runtimes obtained on permutations kij, kji, jki, jik and ikj of the loop were 0.46360, 0.65898, 0.57467, 0.54914 and 0.53980 seconds. **Only the kij variant gives a reasonable speedup of 1.1747**.

- **Tiling with Loop Permutation and Unrolling/Jamming**
  Similar to the observations with GCC, the best performing loop variants were ijk and jik with loop j unrolled 4 times. Overall, **best performance was obtained on tile size 128x128x128 and loop permutation ijk with j loop unrolled/jammed 4 times. The runtime obtained was 0.15541 seconds (speedup factor 3.5105)**. The performance of the jik variant is also very close to this. The results are summarized in the following tables.

  – **jik**

Tiling Size ($N \times N \times N$)	Runtime (in seconds)	Speedup Factor
4x4x4	0.4650	1.1734
8x8x8	0.2642	2.0647
16x16x16	0.2147	2.5413
32x32x32	0.1790	3.0486
64x64x64	0.1625	3.3571
128x128x128	**0.1556**	**3.5071**
256x256x256	0.1674	3.2591
512x512x512	0.1970	2.7693

Table 14: ICC: Runtimes for Tiling with jik permutation, j unrolled/jammed 4 times

– **ijk**

Tiling Size ($N \times N \times N$)	Runtime (in seconds)	Speedup Factor
4x4x4	0.3948	1.3818
8x8x8	0.2575	2.1186
16x16x16	0.2044	2.6685
32x32x32	0.1724	3.1640
64x64x64	0.1701	3.2068
128x128x128	**0.1554**	**3.5105**
256x256x256	0.1790	3.0479
512x512x512	0.2356	2.3161

Table 15: ICC: Runtimes for Tiling with ijk permutation, j unrolled/jammed 4 times

(b) Again, we vectorize the optimal code (loop permutation ijk with tile size 128x128x128 and loop j unrolled/jammed 4 times) using intrinsics. The vector code remains the same as for GCC and only the tile-size changes to 128x128x128. **The final runtime obtained was 0.16066 seconds (speedup factor 3.3958).** There is **no performance improvement at all** in comparison to the optimal code without intrinsics. **The reason is that ICC auto-vectorizes the code while GCC doesn't. This is why, using intrinsics gave a small improvement of 1.1611 with GCC but no improvement with ICC.**

3. Let's first find the dependences in the loop nest. Using the **Delta Test method**, we analyze each pair of array references (one of which is a write to a memory location) for the two loop vectors (K, I, J) and $(K + \Delta K, I + \Delta I, J + \Delta J)$.

- **X[i, j+1]** and **X[i-1, j+1]**

$$I = I + \Delta I - 1$$
$$J + 1 = J + \Delta J + 1$$

Solving them simultaneously, we get $\Delta I = 1, \Delta J = 0$ while ΔK can take any value. We have 3 cases,

- $\Delta K > 0$. We get a flow-dependence with dependence vector $(\delta k, 1, 0)$ and direction vector $(+, +, 0)$ where $0 < \delta k \leq ITER - 1$.
- $\Delta K = 0$. We get a flow-dependence with dependence vector $(0, 1, 0)$ and direction vector $(0, +, 0)$.
- $\Delta K < 0$. We get an anti-dependence with dependence vector $(\delta k, -1, 0)$ and direction vector $(+, -, 0)$ where $0 < \delta k \leq ITER - 1$.

- **X[i, j+1]** and **X[i, j+1]**

$$I = I + \Delta I$$
$$J + 1 = J + \Delta J + 1$$

Solving them simultaneously, we get $\Delta I = 0, \Delta J = 0$ while ΔK can take any value. We have 3 cases,

- $\Delta K > 0$. We get a flow-dependence with dependence vector $(\delta k, 0, 0)$ and direction vector $(+, 0, 0)$ where $0 \leq \delta k \leq ITER - 1$.
- $\Delta K = 0$. We get an anti-dependence with dependence vector $(0, 0, 0)$ and direction vector $(0, 0, 0)$.
- $\Delta K < 0$. We get an anti-dependence with dependence vector $(\delta k, 0, 0)$ and direction vector $(+, 0, 0)$ where $0 \leq \delta k \leq ITER - 1$.

Finally, we get the dependence matrix,

$$\begin{bmatrix} + & + & 0 \\ 0 & + & 0 \\ + & - & 0 \\ + & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

From the dependence matrix, it is clear that **only kij, kji and jki are valid permutations of the loop nest**. In all these permutations **only the j loop can be parallelized because the i and k loop carry dependences**. Hence we will attempt to parallelize the j loop in each of these permutations. We will also try tiling and unrolling/jamming to improve performance.

The **unoptimized reference version of the code runs in 1.03030 seconds**.

We will now parallelize the j loop with different number of threads. The intel devcloud provides a maximum of 24 threads. But note that performance might not scale linearly with the number of threads used, due to thread creation overheads. We report the speedup factors with different number of threads in the following tables and the corresponding figures shown below. The code used for parallelization has been provided as well.

- **Parallelize loop j in kij variant**
  We see that the **performance worsens as we increase the number of threads**. This goes hand in hand with the phrase "parallelization of outer loops". Loop j is the innermost loop and innermost loops should be vectorized and not parallelized. This is because the threads become idle and go to sleep after 1 full iteration of the outer i loop. In the next iteration of i, the threads are woken up again and work is re-distributed among them. This happens in each iteration of i which creates overheads.

Number of Threads	Runtime (in seconds)	Speedup Factor
2	1.7007	0.6058
4	**1.6731**	**0.6158**
6	1.8479	0.5576
8	2.0474	0.5032
10	2.2631	0.4553
12	2.4754	0.4162
14	2.6292	0.3919
16	2.7269	0.3778
18	2.9469	0.3496
20	3.1155	0.3307
22	3.3001	0.3122
24	3.3819	0.3046

Table 16: GCC: Runtimes for kij variant with j loop parallelized

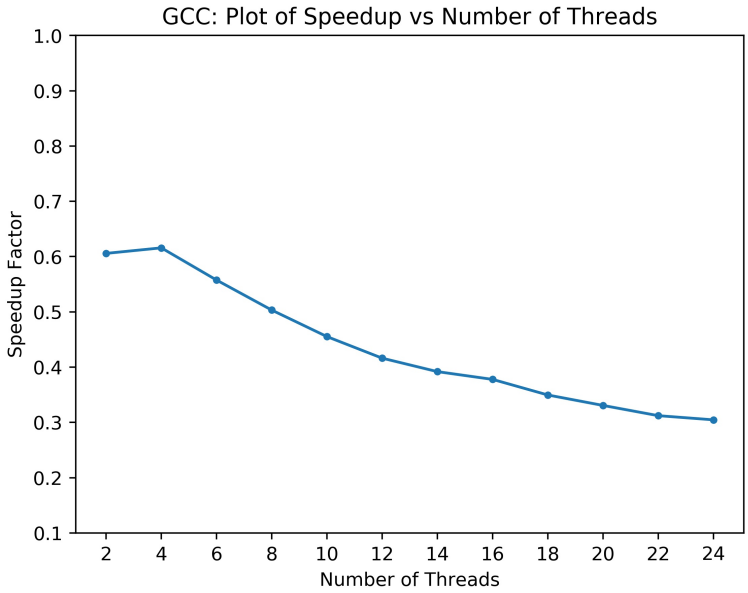

Figure 5: GCC: Plot of Speedup vs Number of Threads for kij variant

- **Parallelize loop j in kji variant**
  Again similar to the previous case, since the loop we are parallelizing is not the outermost, the **performance improvement is very low even with 24 threads**. Note that the speedup factor takes a dip as number of threads increases beyond 12. This is because the processor on which the code was tested had 12 cores. Even though each core can support 2 threads using hyper-threading, speedup won't scale linearly. In fact, the thread-creation overhead overshadows the speedup. **The best runtime obtained was 0.8163 seconds (speedup factor 1.2621) using 24 threads**.

Number of Threads	Runtime (in seconds)	Speedup Factor
2	4.7966	0.2148
4	2.5198	0.4089
6	1.7064	0.6038
8	1.3505	0.7629
10	1.1463	0.8988
12	0.9170	1.1236
14	1.3135	0.7844
16	1.2253	0.8408
18	1.1076	0.9302
20	0.9529	1.0812
22	0.9167	1.1240
24	**0.8163**	**1.2621**

Table 17: GCC: Runtimes for kji variant with j loop parallelized

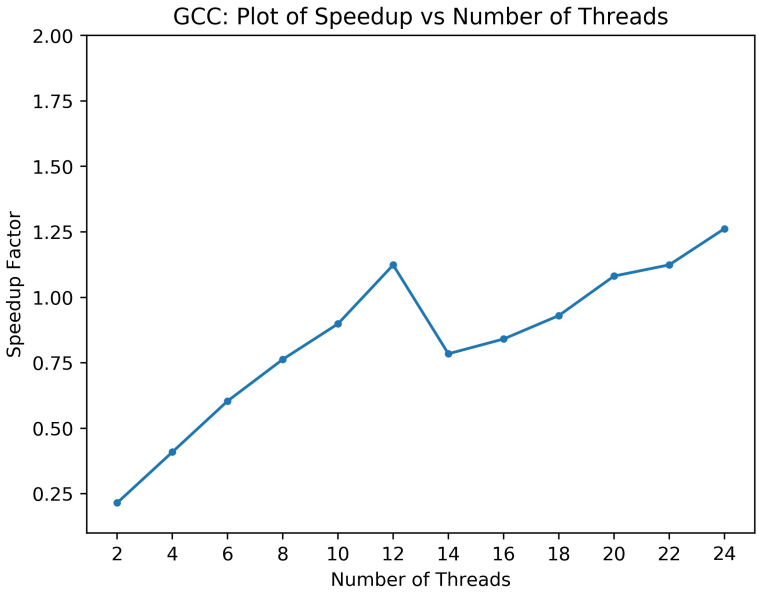

Figure 6: GCC: Plot of Speedup vs Number of Threads for kji variant

- **Parallelize loop j in jki variant**
  This time since j is the outermost loop, we are able to achieve a decent runtime of 0.6922 seconds (speedup factor 1.4885) using 12 threads. Here again, speedup takes a dip as number of threads increases beyond 12 for the same reason as described in previous case.

Number of Threads	Runtime (in seconds)	Speedup Factor
2	4.1078	0.2508
4	2.0809	0.4951
6	1.3940	0.7391
8	1.0556	0.9760
10	0.8324	1.2377
12	**0.6922**	**1.4885**
14	0.9125	1.1291
16	0.8241	1.2502
18	0.7448	1.3834
20	0.7957	1.2949
22	0.7925	1.3001
24	0.7326	1.4064

Table 18: GCC: Runtimes for jki variant with j loop parallelized

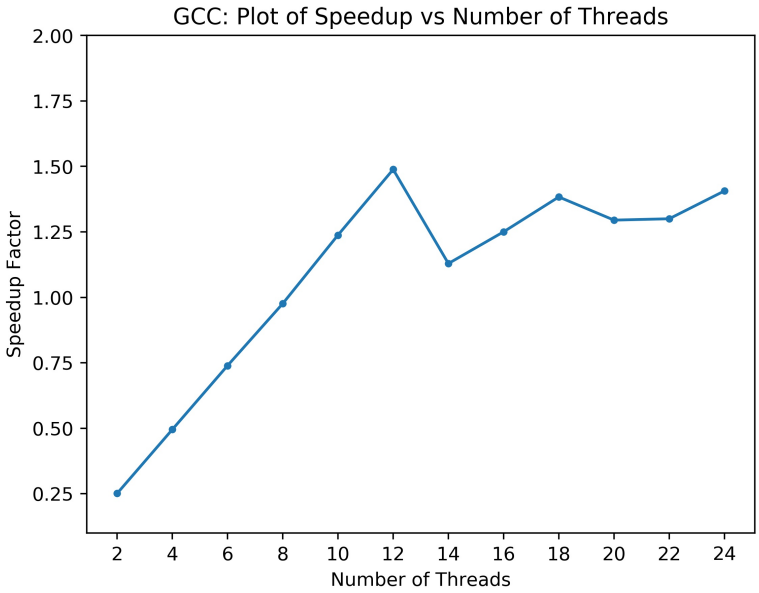

Figure 7: GCC: Plot of Speedup vs Number of Threads for jki variant

Parallelizing kji variant with Tiling
We again parallelize loop j in the kji variant, but this time we introduce tiling of loops i and j. Note that loop k cannot be tiled because the band kji is not fully permutable, while the band ji is. **Using 24 threads and a tile size of 16, we get a significantly smaller runtime of 0.2123 seconds (speedup factor 4.8537)**. The code we used for tiling and the results obtained have been shown below.

```
void omp_version(uint32_t** A) {
  int jt;
  const int n_threads = 24;
  const int block_size = 16;
  omp_set_num_threads(n_threads);
  #pragma omp parallel for
  for (jt = 0; jt < N-1; jt += block_size) {
    int i, j, k, it;
    for (k = 0; k < ITER; k++) {
      for (it = 0; it < N; it += block_size) {
        for (j = jt; j < min(jt+block_size, N-1); j++) {
          for (i = max(1, it); i < it+block_size; i++) {
            A[i][j + 1] = A[i - 1][j + 1] + A[i][j + 1];
          }
        }
      }
    }
  }
}
```

Number of Threads	Tile Size	Runtime (in seconds)	Speedup Factor
6	4x4	0.5468	1.8841
6	8x8	0.4917	2.0954
6	**16x16**	**0.4885**	**2.1091**
6	32x32	0.5015	2.0546
6	64x64	0.4979	2.0694
6	128x128	0.5423	1.8998
6	256x256	0.7731	1.3327
12	4x4	0.2807	3.6706
12	8x8	0.2549	4.0424
12	**16x16**	**0.2501**	**4.1201**
12	32x32	0.2682	3.8416
12	64x64	0.2795	3.6868
12	128x128	0.3857	2.6713
12	256x256	0.5805	1.7749
18	4x4	0.3509	2.9358
18	8x8	0.2904	3.5474
18	**16x16**	**0.2617**	**3.9373**
18	32x32	0.2625	3.9253
18	64x64	0.3019	3.4124
18	128x128	0.3583	2.8752
18	256x256	0.4806	2.1436
24	4x4	0.3552	2.9010
24	8x8	0.2336	4.4105
24	**16x16**	**0.2123**	**4.8537**
24	32x32	0.2224	4.6324
24	64x64	0.3005	3.4289
24	128x128	0.4131	2.4941
24	256x256	0.4901	2.1024

Table 19: GCC: Runtimes for Tiling loops i, j for kji variant with j loop parallelized

Parallelizing kji variant with Tiling and Unrolling/Jamming
For the best performing combination we obtained in the previous case, we tried unrolling loops i and j. Finally, **using 24 threads, tiling loops i and j with tiles of size 16x16 and unrolling/jamming loop i 8 times in the kji variant, we get a runtime of 0.1517 seconds (speedup factor 6.7907)**. This was the best performance obtained. The final best-performing code we used is provided below.

Unroll/Jam	Runtime (in seconds)	Speedup Factor
j unrolled/jammed 4 times	0.1628	6.3280
j unrolled/jammed 8 times	0.1837	5.6072
i unrolled/jammed 4 times	0.1591	6.4743
i unrolled/jammed 8 times	**0.1517**	**6.7907**

Table 20: GCC: Runtimes for unrolling/jamming loops i, j in kji variant (using 24 threads, and i, j tiled with 16x16 tiles)

```
void omp_version(uint32_t** A) {
  int jt;
  const int n_threads = 24;
  const int block_size = 16;
  omp_set_num_threads(n_threads);
  #pragma omp parallel for
  for (jt = 0; jt < N-1; jt += block_size) {
    int i, j, k, it;
    for (k = 0; k < ITER; k++) {
      for (it = 0; it < N; it += block_size) {
        for (j = jt; j < min(jt+block_size, N-1); j++) {
          if (it == 0) {
            A[1][j+1] = A[0][j+1] + A[1][j+1];
            A[2][j+1] = A[1][j+1] + A[2][j+1];
            A[3][j+1] = A[2][j+1] + A[3][j+1];
            A[4][j+1] = A[3][j+1] + A[4][j+1];
            A[5][j+1] = A[4][j+1] + A[5][j+1];
            A[6][j+1] = A[5][j+1] + A[6][j+1];
            A[7][j+1] = A[6][j+1] + A[7][j+1];
          }
          for (i = max(8, it); i < it+block_size; i += 8) {
            A[i][j+1] = A[i-1][j+1] + A[i][j+1];
            A[i+1][j+1] = A[i][j+1] + A[i+1][j+1];
            A[i+2][j+1] = A[i+1][j+1] + A[i+2][j+1];
            A[i+3][j+1] = A[i+2][j+1] + A[i+3][j+1];
            A[i+4][j+1] = A[i+3][j+1] + A[i+4][j+1];
            A[i+5][j+1] = A[i+4][j+1] + A[i+5][j+1];
            A[i+6][j+1] = A[i+5][j+1] + A[i+6][j+1];
            A[i+7][j+1] = A[i+6][j+1] + A[i+7][j+1];
          }
        }
      }
    }
  }
}
```