Homework – 3

1.
```
for i = 2,N-1
  for j = 2,N-1
   A(i,j) = A(i-1,j+1)+A(i+1,j+1)
  end for
end for
```

   (a) Data dependence, type and distance vector:

   Two pairs are to be tested for flow/anti dependence:

   [A(i,j), A(i-1,j+1)] : iw = ir -1, jw = jr + 1 => ir – iw = 1, jr – jw = -1 i.e read follows write, or flow dependence (1,-1)

   [A(i,j), A(i+1,j+1)]: iw = ir +1, jw = jr + 1 => iw – ir = 1, jw – jr = 1 i.e read precedes write, or anti-dependence (1,1)

   (b) Valid permutations:
   The loops here cannot be permuted because interchange of 'i' and 'j' would cause the dependence vector to become lexicographically negative. Hence loop interchange is not valid.

   (c) Loop unrolling:
   Only the inner loop can be unrolled. The outer loop cannot be unrolled for the same reason as above.

   (d) Tiling:
   Tiling a set of loops is only valid when the loops are fully permutable. We have seen that the above loops are not permutable. Hence tiling is not possible.

   (e) Parallel Loops:
   For the j^{th} loop to be parallel it is required that for each dependence vector, the i^{th} component is nonzero.
   Here the dependence vectors are (1, -1) and (1, 1).
   Hence the j^{th} loop is parallel.

2.
```
for t = 1,nt
  for i = 2,N-1
    for j = 2,N-1
     A(i,j) = A(i-1,j+1)+A(i+1,j+1)
    end for
  end for
end do
```

   (a) Data dependence, type and distance vector:

Two pairs are to be tested for flow/anti dependence:

[A(k,i,j), A(k, i-1,j+1)] : kw = kr, iw = ir -1, jw = jr + 1 => kr – kw = 0, ir – iw = 1, jr – jw = -1 i.e read follows write, or flow dependence (0, 1,-1)

[A(i,j), A(i+1,j+1)]: kw = kr, iw = ir +1, jw = jr + 1 => kw – kr =0, iw – ir = 1, jw – jr = 1 i.e read precedes write, or anti-dependence (0,1,1)

(b) Valid permutations:
The valid loop permutations are the one that are lexicographically positive. With this rule in mind we can say that the permutations ***kij, ikj, ijk***.

(c) Loop unrolling:
Inner loop unrolling is always possible. For the outer loops unrolling, the permutation that moves the unrolled loop innermost needs to be valid. According to this we can see that only unrolling of the **k** loop is valid. The **i** loop cannot be unrolled. Because when the 'i' loop is shifted to the innermost, then the dependence vector become lex negative.

(d) Tiling:
Tiling a set of loops is only valid when the loops are fully permutable. We have seen that the above loops are not permutable. Hence tiling is not possible.

(e) Parallel Loops:
For the j^{th} loop to be parallel it is required that for each dependence vector, the i^{th} component is nonzero.
Here the dependence vectors are (0, 1, -1) and (0, 1, 1).
Hence the j^{th} loop is parallel.

Similarly we can conclude that the i^{th} loop is not parallel because the k^{th} is not non-zero.

Only the j^{th} loop is parallel.

3. Optimization of performance of the code:

```
double A[N][N][N], C[N][N], B[N][N][N];
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        for (l=0; l<N; l++)
          C[i][j] += A[l][i][k]*B[k][l][j];
```

The optimized code for the above algorithm reduces to:

```
for (l=0; l<N; l++)
  for (k=0; k<N; k++)
    for (i=0; i<N; i+=4)
      for (j=0; j<N; j+=4)
      {
        CC[i][j] += A[l][i][k]*B[k][l][j];
        CC[i][j+1] += A[l][i][k]*B[k][l][j+1];
        CC[i][j+2] += A[l][i][k]*B[k][l][j+2];
        CC[i][j+3] += A[l][i][k]*B[k][l][j+3];

        CC[i+1][j] += A[l][i+1][k]*B[k][l][j];
        CC[i+1][j+1] += A[l][i+1][k]*B[k][l][j+1];
        CC[i+1][j+2] += A[l][i+1][k]*B[k][l][j+2];
        CC[i+1][j+3] += A[l][i+1][k]*B[k][l][j+3];

        CC[i+2][j] += A[l][i+2][k]*B[k][l][j];
        CC[i+2][j+1] += A[l][i+2][k]*B[k][l][j+1];
        CC[i+2][j+2] += A[l][i+2][k]*B[k][l][j+2];
        CC[i+2][j+3] += A[l][i+2][k]*B[k][l][j+3];

        CC[i+3][j] += A[l][i+3][k]*B[k][l][j];
        CC[i+3][j+1] += A[l][i+3][k]*B[k][l][j+1];
        CC[i+3][j+2] += A[l][i+3][k]*B[k][l][j+2];
        CC[i+3][j+3] += A[l][i+3][k]*B[k][l][j+3];
       }
```

We arrive at this by permuting the loops and subsequently unrolling the inner most 2 loops by 4. We find through experimentation that 4 way unrolling is the efficient level of unrolling for both the loops.

The results achieved are as below:

Tensor Size = 64
Base-TensorMult: 109.1 MFLOPS; Time = 0.308 sec;
Test-TensorMult: 1748.9 MFLOPS; Time = 0.019 sec;
No differences found between base and test versions

Which is almost a 16x increase in performance.

4. Optimization of the code:

```
  for (i=0; i<1024; i++)
    for (j=0; j<1024; j++)
      B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];
```

```
for (i=0; i<1024; i++)
  for (j=0; j<1024; j++)
        x[i] = x[i] + beta* B[j][i]*y[j];

for (i=0; i<1024; i++)
   x[i] = x[i] + z[i];

for (i=0; i<1024; i++)
  for (j=0; j<1024; j++)
    w[i] = w[i] + alpha* B[i][j]*x[j];
```

We notice that there is an opportunity to exploit spatial locality in the second loop structure by using loop interchange.

Hence the optimized code in as below:

```
  for (i=0; i<1024; i++)
    for (j=0; j<1024; j++)
      B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];


  for (j=0; j<1024; j++)
     for (i=0; i<1024; i++)
        x[i] = x[i] + beta* B[j][i]*y[j];

for (i=0; i<1024; i++)
   x[i] = x[i] + z[i];

for (i=0; i<1024; i++)
  for (j=0; j<1024; j++)
    w[i] = w[i] + alpha* B[i][j]*x[j];
```

Using this optimized code we get the following increase in performance.

Matrix Size = 1024
Base: 286.1 MFLOPS; Time = 3.665 sec; wref[n/2-1] = 443506304.000000;
Test: 1177.3 MFLOPS; Time = 0.891 sec; w[n/2-1] = 443506304.000000;
No differences found between base and test versions

Which is more than 4x increase in performance.