1.

a)Attempted transformations

All the parallel for is acted on j loop. We can’t put ‘parallel for’ on i or k loop, because the dependence. All the transformations are based on this parallel.

I tried permutation ikj, and that will help a lot. So the further transformations are based on ikj. Then I tried tiled (IT-256), tiled(IT-8),tiled(KT-4),tiled(KT-8),tiled(KT-256), tiled(IT-256,JT-256), tiled(IT-256,JT-256,KT-256),tiled(IT-128,KT-128), tiled(IT-128,KT-256),tiled(IT-256,KT-256),tiled(IT-512,KT-512), tiled unroll(IT-256,KT-256,unroll-2j), tiled(JT-256,KT-512),tiled(JT-256,KT-256),unroll(2j), unroll(2i).

b) How performance varied with the different transformations.

for convenience, I just show the performance for number of threads=1,2,4,8,10.

For gcc:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| threads | 1 | 2 | 4 | 8 | 10 |
| base | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| Just parallel | 0.2 | 0.3 | 0.5 | 0.7 | 0.7 |
| ikj | 1.9 | 2.3 | 2.5 | 2.3 | 2.2 |
| tiled(IT-8) | 1.9 | 2.3 | 2.3 | 2.4 | 2.3 |
| tiled(IT-256) | 1.9 | 2.3 | 2.6 | 2.3 | 2.8 |
| tiled(KT-4) | 2 | 2.5 | 2.8 | 2.9 | 2.6 |
| tiled(KT-8) | 2.1 | 2.6 | 3.2 | 3.3 | 3.1 |
| tiled(KT-256) | 2.2 | 2.7 | 3.2 | 3.1 | 3.3 |
| tiled(IT-256,JT-256) | 0.9 | 0.6 | 0.6 | 0.5 | 0.5 |
| tiled(IJK-256) | 1.1 | 0.7 | 0.6 | 0.5 | 0.5 |
| tiled(IT-128,KT-128) | 2.2 | 2.8 | 3.5 | 3.5 | 3.3 |
| tiled(IT-128,KT-256) | 2.2 | 2.8 | 3.5 | 3.5 | 3.1 |
| tiled(IT-256,KT-256) | 2.2 | 2.8 | 3.5 | 3.6 | 3.1 |
| tiled(IT-512,KT-512) | 2.2 | 2.8 | 3.5 | 3.7 | 3.4 |
| tiledUnrol(IK-256,2j) | 1.4 | 2 | 2.8 | 3.5 | 3.4 |
| tiled(JT-256,KT-512) | 1.1 | 0.7 | 0.8 | 0.6 | 0.5 |
| tiled(JT-256,KT-256) | 1.1 | 0.7 | 0.8 | 0.5 | 0.5 |
| Unroll-2i | 1.7 | 2.7 | 4.3 | 4 | 3.1 |
| Unroll-2j | 1.3 | 1.8 | 2.3 | 2.6 | 2.1 |

From the table above, we can conclude that unroll i is a good idea to optimize. And tile I,K is fine but not J, because we do “parallel for” for j loop.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| threads | 1 | 2 | 4 | 8 | 10 |
| base | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| Just parallel | 0.2 | 0.3 | 0.6 | 0.7 | 0.7 |
| ikj | 2.7 | 2.3 | 2.3 | 2.1 | 1.9 |
| tiled(IT-8) | 2 | 2.7 | 2.6 | 2.5 | 2.1 |
| tiled(IT-256) | 2.7 | 2.3 | 2.4 | 2 | 1.8 |
| tiled(KT-4) | 3.1 | 4.3 | 3 | 2.3 | 2 |
| tiled(KT-8) | 3.4 | 3.7 | 3.1 | 2.8 | 2.5 |
| tiled(KT-256) | 3.6 | 4.4 | 3.5 | 3.3 | 2.8 |
| tiled(IT-256,JT-256) | 0.6 | 0.5 | 0.5 | 0.3 | 0.3 |
| tiled(IJK-256) | 3 | 0.8 | 0.5 | 0.5 | 0.4 |
| tiled(IT-128,KT-128) | 3.7 | 3.3 | 3.4 | 3.4 | 3 |
| tiled(IT-128,KT-256) | 3.7 | 3.4 | 3.5 | 2.8 | 2.1 |
| tiled(IT-256,KT-256) | 3.6 | 3.5 | 3.7 | 2.4 | 2.3 |
| tiled(IT-512,KT-512) | 2.7 | 3.4 | 3.6 | 3.3 | 2.8 |
| tiledUnrol(IK-256,2j) | 2.3 | 2.6 | 3 | 3.1 | 2.9 |
| tiled(JT-256,KT-512) | 3 | 0.7 | 0.5 | 0.4 | 0.4 |
| tiled(JT-256,KT-256) | 2.9 | 0.7 | 0.5 | 0.4 | 0.3 |
| Unroll-2i | 2.6 | 3.9 | 4.7 | 3.7 | 3.3 |
| Unroll-2j | 1.7 | 2.1 | 2.2 | 1.9 | 1.8 |

From the table above, we can conclude that tile k and tile i,k will be good. Also unrolling j for more threads is better.

c) the performance for the best version for each compiler with each of 1-12 threads

|  |  |  |
| --- | --- | --- |
| Threads | gcc | icc |
| 1 | 2.2 | 3.7 |
| 2 | 2.8 | 4.4 |
| 3 | 3.1 | 4.3 |
| 4 | 4.3 | 4.7 |
| 5 | 4.0 | 4.7 |
| 6 | 4.1 | 4.5 |
| 7 | 3.8 | 4.1 |
| 8 | 4.0 | 3.7 |
| 9 | 3.4 | 3.5 |
| 10 | 3.4 | 3.3 |
| 11 | 2.8 | 3.0 |
| 12 | 2.4 | 2.9 |

d) the code for my best version

for both gcc and icc:

#pragma omp parallel private(i,k)

// Template version is intentionally made sequential

// Make suitable changes to create parallel version

{

if (omp\_get\_thread\_num()==0 && omp\_get\_num\_threads() != nt)

printf("Warning: Actual #threads %d differs from requested number %d\n",omp\_get\_num\_threads(),nt);

//

// Test version of code; initially just contains a copy of base code

// To be modified by you to improve performance

//

//#pragma omp master

for (i=0; i<N; i+=2)

{

#pragma omp single

for(j=0;j<N;j++)

B[i][j]+=A[i][i]\*B[i][j];

for(k=i+1;k<N;k++)

#pragma omp for

for(j=0;j<N;j++)

{

B[i][j] += A[i][k]\*B[k][j];

B[i+1][j] += A[i+1][k]\*B[k][j];

}

}

} // End of parallel region

2.

a)Attempted transformations

First of all, I did the permutation klij without the parallel. That was pretty faster than the old one. Then did the parallel. The parallel is that put “parallel for” on the outer loop of k. The further transformations are based on permutation and parallel. The attempted transformations are permutation klij, parallel for klij, tile(i/2,j/2), tile(i/4,j/4), thileUnroll(i/2,j/2,2i), unroll(2i), unroll(4i), unroll(2j), unroll(4j).

b) How performance varied with the different transformations.

for convenience, I just show the performance for number of threads=1,2,4,8,12.

For gcc:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| threads | 1 | 2 | 4 | 8 | 12 |
| base | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| klij | 2.3 | 2.3 | 2.3 | 2.3 | 2.3 |
| klij-parallel | 2.3 | 3 | 4.4 | 5.9 | 5.5 |
| tile(i-2,j-2) | 1.7 | 2.1 | 3.5 | 5.5 | 6.3 |
| tile(i-4,j-4) | 1.6 | 2 | 2.6 | 4.6 | 5.8 |
| tileUnrol(i-2,j-2,2i) | 1.7 | 2.1 | 3.5 | 4.6 | 6.2 |
| unroll(2i) | 2.4 | 3.1 | 3.6 | 6.8 | 7.9 |
| unroll(4i) | 1.8 | 2.3 | 3.9 | 5.9 | 7 |
| unroll(2j) | 2.5 | 3.3 | 3.6 | 5.5 | 6.5 |
| unroll(4j) | 2.6 | 3.3 | 3.6 | 5.6 | 6.6 |

We got the best results by doing unroll(2i).

For icc.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| threads | 1 | 2 | 4 | 8 | 12 |
| base | 0.3 | 0.3 | 0.3 | 0.3 | 0.3 |
| klij | 2.3 | 2.9 | 2.9 | 2.9 | 2.8 |
| klij-parallel | 2.4 | 3.4 | 5.3 | 6.7 | 6.1 |
| tile(i-2,j-2) | 1.9 | 2.9 | 4.5 | 6.8 | 7.5 |
| tile(i-4,j-4) | 1.6 | 2.1 | 3.3 | 5.8 | 6.5 |
| tileUnrol(i-2,j-2,2i) | 1.5 | 1.9 | 3.1 | 5.7 | 6.1 |
| unroll(2i) | 1.8 | 2.3 | 3.6 | 6.2 | 6.5 |
| unroll(4i) | 2.1 | 2.8 | 4.4 | 6.8 | 7.6 |
| unroll(2j) | 1.8 | 2.5 | 3.6 | 5.8 | 6.4 |
| unroll(4j) | 2.2 | 3.4 | 5 | 7.9 | 8.2 |

We got the best results by doing unroll(4j).

c) the performance for the best version for each compiler with each of 1-12 threads

|  |  |  |
| --- | --- | --- |
| Threads | gcc | icc |
| 1 | 2.6 | 2.4 |
| 2 | 3.3 | 3.4 |
| 3 | 3.8 | 4.3 |
| 4 | 4.4 | 5.3 |
| 5 | 4.5 | 6.1 |
| 6 | 5.8 | 6.6 |
| 7 | 6.3 | 7.6 |
| 8 | 6.8 | 7.9 |
| 9 | 6.8 | 7.8 |
| 10 | 7.6 | 7.8 |
| 11 | 7.9 | 8.0 |
| 12 | 7.9 | 8.2 |

d) the code for my best version.

For gcc:

#pragma omp parallel private(i,j,k,l)

// Template version is intentionally made sequential

// Make suitable changes to create parallel version

{

if (omp\_get\_thread\_num()==0 && omp\_get\_num\_threads() != nt)

printf("Warning: Actual #threads %d differs from requested number %d\n",omp\_get\_num\_threads(),nt);

//

// Test version of code; initially just contains a copy of base code

// To be modified by you to improve performance

//

//#pragma omp master

#pragma omp for

for(k=0;k<N;k++)

for(l=k+1;l<N;l++)

for (i=0; i<N; i+=2)

for (j=0; j<N; j++){

C[k][l] += 0.5\*(A[l][i][j]\*B[k][i][j]+A[k][i][j]\*B[l][i][j]);

C[k][l] += 0.5\*(A[l][i+1][j]\*B[k][i+1][j]+A[k][i+1][j]\*B[l][i+1][j]);

}

} // End of parallel region

for icc:

#pragma omp parallel private(i,j,k,l)

// Template version is intentionally made sequential

// Make suitable changes to create parallel version

{

if (omp\_get\_thread\_num()==0 && omp\_get\_num\_threads() != nt)

printf("Warning: Actual #threads %d differs from requested number %d\n",omp\_get\_num\_threads(),nt);

//

// Test version of code; initially just contains a copy of base code

// To be modified by you to improve performance

//

//#pragma omp master

#pragma omp for

for(k=0;k<N;k++)

for(l=k+1;l<N;l++)

for (i=0; i<N; i++)

for (j=0; j<N; j+=4){

C[k][l] += 0.5\*(A[l][i][j]\*B[k][i][j]+A[k][i][j]\*B[l][i][j]);

C[k][l] += 0.5\*(A[l][i][j+1]\*B[k][i][j+1]+A[k][i][j+1]\*B[l][i][j+1]);

C[k][l] += 0.5\*(A[l][i][j+2]\*B[k][i][j+2]+A[k][i][j+2]\*B[l][i][j+2]);

C[k][l] += 0.5\*(A[l][i][j+3]\*B[k][i][j+3]+A[k][i][j+3]\*B[l][i][j+3]);

}

} // End of parallel region