18645 How To Write Fast Code

Mini-Project 1

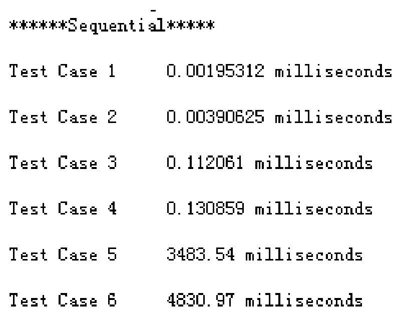
Team 003

Andrew ID	Name
siqiaow	Siqiao Wu
chul1	Chu Lin
zchai	Zijie Chai

Part 1 Matrix_mul

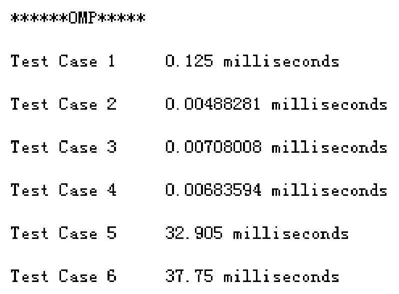
1. Task 0

(1) Runtime for sequential implementation:



```
******Sequential*****

Test Case 1     0.00195312 milliseconds

Test Case 2     0.00390625 milliseconds

Test Case 3     0.112061 milliseconds

Test Case 4     0.130859 milliseconds

Test Case 5     3483.54 milliseconds

Test Case 6     4830.97 milliseconds
```

(2) Runtime for OpenMP implementation:



```
******OMP*****

Test Case 1      0.125 milliseconds

Test Case 2      0.00488281 milliseconds

Test Case 3      0.00708008 milliseconds

Test Case 4      0.00683594 milliseconds

Test Case 5      32.905 milliseconds

Test Case 6      37.75 milliseconds
```

2. Techniques To Speed up

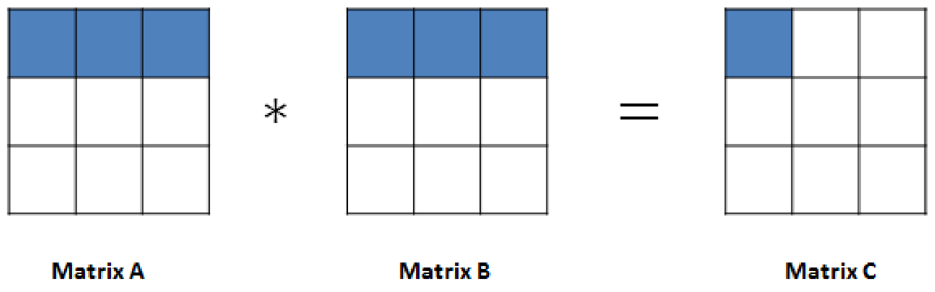
(1) Cache Optimization

a. Optimization Goal:

The hardware bing optimized toward is cache. The specification of the hardware we optimize for is reducing conflict cache misses.

b. Optimization process:

To reduce cache miss, we try to consider about data. That is to do the matrix transposition for second matrix. The accessing order of the second matrix will be like following figure:



By accessing both matrix row by row, we can remove the conflict cache misses when getting data from the second matrix. And because the platform we use is equipped with associative cache. So when reading rows of the second matrix from the DRAM we do not need to worry about that the rows of the first matrix will be replaced.

c. Performance before optimization:

3 Gflops

d. Performance after optimization:

8. Gflops

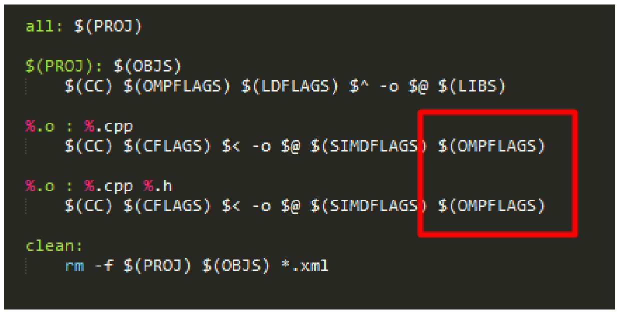
(2) Using Multicores

a. Optimization Goal:

Use multiple threads to run one code to achieve parallelism. The hardware being optimized toward is cpu. The specification we are optimizing for is multiple cores of the cpu.

b. Optimization process:

Use OpenMP library. By modify the make file to enable openMP library.



```
all: $(PROJ)

$(PROJ): $(OBJS)
	$(CC) $(OMPFLAGS) $(LDFLAGS) $^ -o $@ $(LIBS)

%.o : %.cpp
	$(CC) $(CFLAGS) $< -o $@ $(SIMDFLAGS) $(OMPFLAGS)

%.o : %.cpp %.h
	$(CC) $(CFLAGS) $< -o $@ $(SIMDFLAGS) $(OMPFLAGS)

clean:
	rm -f $(PROJ) $(OBJS) *.xml
```

c. Performance before optimization:

3.8 Gflops

d. Performance after optimization:

e. 16 Gflops

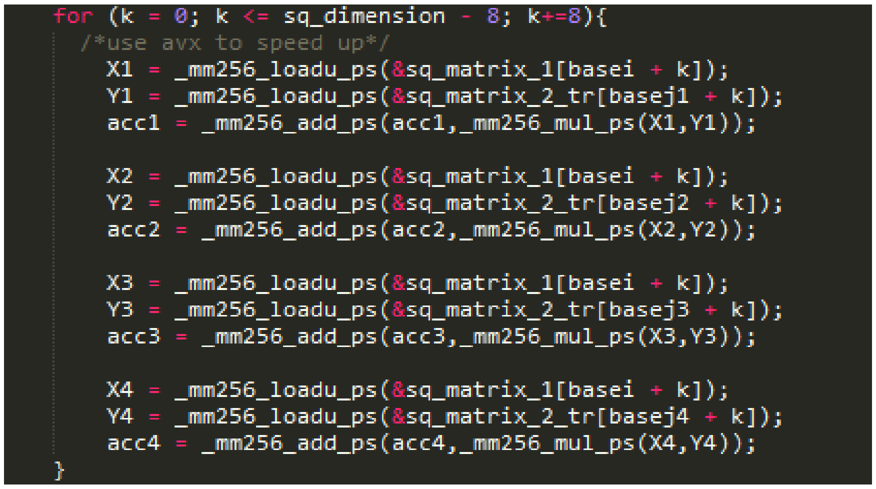
(3) SIMD

a. Optimization Goal:

The hardware being optimized is cpu by using SIMD(Single Instruction Multiple Data). And the specification we are optimizing for is the instruction set of processor. The function of this optimization is trying to calculate multiple pairs of data at the same time when executing one instruction.

b. Optimization process:

Use AVX instruction set. In each loop we can handle 16 data (8 in matrix A and 8 in matrix B), so that there might have an 8x speed up. The example code is presented below:



```
for (k = 0; k <= sq_dimension - 8; k+=8){
  /*use avx to speed up*/
    X1 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
    Y1 = _mm256_loadu_ps(&sq_matrix_2_tr[basej1 + k]);
    acc1 = _mm256_add_ps(acc1,_mm256_mul_ps(X1,Y1));

    X2 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
    Y2 = _mm256_loadu_ps(&sq_matrix_2_tr[basej2 + k]);
    acc2 = _mm256_add_ps(acc2,_mm256_mul_ps(X2,Y2));

    X3 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
    Y3 = _mm256_loadu_ps(&sq_matrix_2_tr[basej3 + k]);
    acc3 = _mm256_add_ps(acc3,_mm256_mul_ps(X3,Y3));

    X4 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
    Y4 = _mm256_loadu_ps(&sq_matrix_2_tr[basej4 + k]);
    acc4 = _mm256_add_ps(acc4,_mm256_mul_ps(X4,Y4));
}
```

c. Performance before optimization:

16 Gflops

d. **Performance after optimization:**

89 Gflops

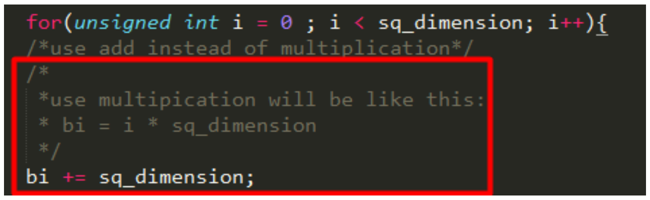
(4) Other optimization methods

<1> Remove expensive operations:

a. **Optimization goal:**

Try to remove the multiplication operation in accessing data of step of matrix transposition.

b. **Implementation of our code:**



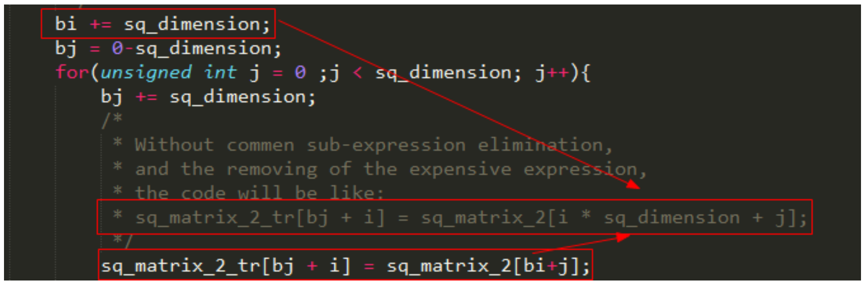
```
for(unsigned int i = 0 ; i < sq_dimension; i++){
/*use add instead of multiplication*/
/*
 *use multipication will be like this:
 * bi = i * sq_dimension
 */
bi += sq_dimension;
```

<2> Common sub-expression elimination:

a. **Optimization goal:**

For some variables that will be use multiple times in one loop with the same value, we can calculate the value of it outside the loop to reduce the operations in one loop.

b. **Implementation of our code:**



```
    bi += sq_dimension;
    bj = 0-sq_dimension;
    for(unsigned int j = 0 ;j < sq_dimension; j++){
        bj += sq_dimension;
        /*
         * Without commen sub-expression elimination,
         * and the removing of the expensive expression,
         * the code will be like:
         * sq_matrix_2_tr[bj + i] = sq_matrix_2[i * sq_dimension + j];
         */
        sq_matrix_2_tr[bj + i] = sq_matrix_2[bi+j];
```

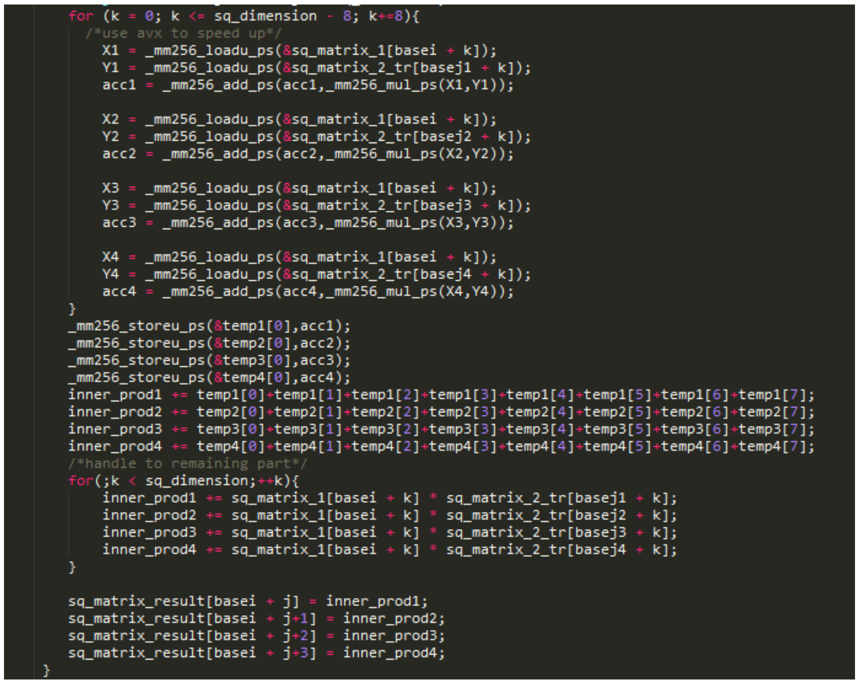
<3> loop unrolling:

a. **Optimization goal:**

Try to decrease the number of the iteration by doing more calculations in one loop.

b. **Implementation of our code:**

We use 4-way loop unrolling with avx so that for one matrix we can handle 4 lines in second loop and 8 data in third loop in one time:



```
    for (k = 0; k <= sq_dimension - 8; k+=8){
      /*use avx to speed up*/
        X1 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
        Y1 = _mm256_loadu_ps(&sq_matrix_2_tr[basej1 + k]);
        acc1 = _mm256_add_ps(acc1,_mm256_mul_ps(X1,Y1));

        X2 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
        Y2 = _mm256_loadu_ps(&sq_matrix_2_tr[basej2 + k]);
        acc2 = _mm256_add_ps(acc2,_mm256_mul_ps(X2,Y2));

        X3 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
        Y3 = _mm256_loadu_ps(&sq_matrix_2_tr[basej3 + k]);
        acc3 = _mm256_add_ps(acc3,_mm256_mul_ps(X3,Y3));

        X4 = _mm256_loadu_ps(&sq_matrix_1[basei + k]);
        Y4 = _mm256_loadu_ps(&sq_matrix_2_tr[basej4 + k]);
        acc4 = _mm256_add_ps(acc4,_mm256_mul_ps(X4,Y4));
    }
    _mm256_storeu_ps(&temp1[0],acc1);
    _mm256_storeu_ps(&temp2[0],acc2);
    _mm256_storeu_ps(&temp3[0],acc3);
    _mm256_storeu_ps(&temp4[0],acc4);
    inner_prod1 += temp1[0]+temp1[1]+temp1[2]+temp1[3]+temp1[4]+temp1[5]+temp1[6]+temp1[7];
    inner_prod2 += temp2[0]+temp2[1]+temp2[2]+temp2[3]+temp2[4]+temp2[5]+temp2[6]+temp2[7];
    inner_prod3 += temp3[0]+temp3[1]+temp3[2]+temp3[3]+temp3[4]+temp3[5]+temp3[6]+temp3[7];
    inner_prod4 += temp4[0]+temp4[1]+temp4[2]+temp4[3]+temp4[4]+temp4[5]+temp4[6]+temp4[7];
    /*handle to remaining part*/
    for(;k < sq_dimension;++k){
        inner_prod1 += sq_matrix_1[basei + k] * sq_matrix_2_tr[basej1 + k];
        inner_prod2 += sq_matrix_1[basei + k] * sq_matrix_2_tr[basej2 + k];
        inner_prod3 += sq_matrix_1[basei + k] * sq_matrix_2_tr[basej3 + k];
        inner_prod4 += sq_matrix_1[basei + k] * sq_matrix_2_tr[basej4 + k];
    }

    sq_matrix_result[basei + j] = inner_prod1;
    sq_matrix_result[basei + j+1] = inner_prod2;
    sq_matrix_result[basei + j+2] = inner_prod3;
    sq_matrix_result[basei + j+3] = inner_prod4;
}
```

c. **Performance before optimization:**

89 Gflops

d. **Performance after optimization:**

112 Gflops

3. Expected Speed up

(1) Using SIMD:

a. The speed up is:

8x

b. Reason: because for every instruction it can handle 8 pairs of data from two matrix in an ideal situation.

(2) Using OpenMP:

a. The speed up is:

8x

b. Reason:

Because the platform we use has 8 virtual cores, that means cpu can execute 8 threads at the same in an ideal situation.

(3) Using Loop unrolling:

a. The speed up is:

4x

b. Reason:

The iteration number of loop will be decrease to 1/4 of the original loop by using loop unrolling.

(4) The total expected speed up:

8x * 8x *4x = 256x

4. Observed Speed up

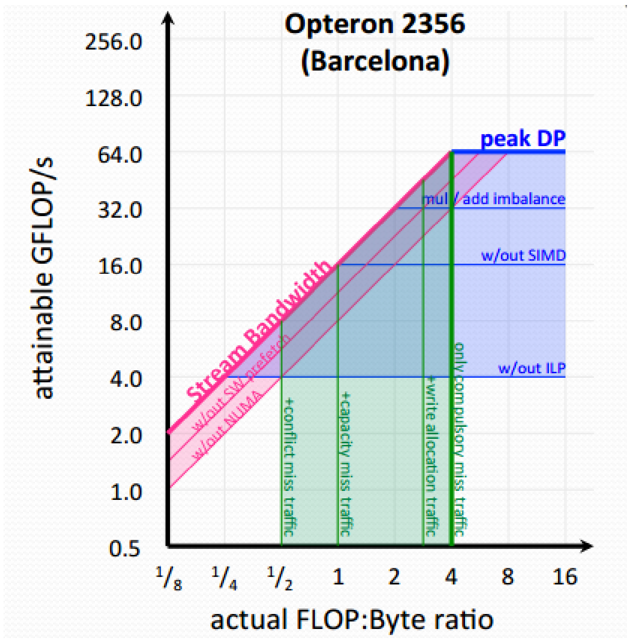
(1) Performance before optimization: 3 Gflops

(2) Performance before optimization: 110 Gflops

(3) Speed up: 112 / 3 ≈ 37.3

5. Explanation For The Difference Between Expected And Observed Speed Up

Our explanation of the difference between the expected and observed speed up is according to the model of roofline. Although we use different platform, the things that may affect performance are the same. The picture comes from the slides of our course.



(1) Mul/add imbalance

a. Reason:

In matrix_mul application, there are a lot of operation of multiplication and addition. Because the time to execute mul instruction spends more time than the adding instruction, which makes AVX, that is SIMD, cannot be fully used. So, here the speed up will be cut off.

b. Decrement of speed up:

   256x / 2 = 128x.

(2) Capacity miss traffic

a. Reason:

   The dimension of the matrix may be very large. One row of a matrix may be larger than the cache line, which may cause the capacity traffic.

b. Decrement of speed up:

   128x / 2 = 64x.

(3) Write allocation traffic

a. Reason:

   When updating the element of the result matrix my cause the allocation traffic.

b. Decrement of speed up:

   According to the figure from the slides, after the decrement the speed up will be larger than the half of the original speed up. And the overhead results from the thread context switching and the latency of the data transfer should be considered. So we may get the actual speed up as what we observed. So we can get:

   37.3x > 64x / 2 = 32x.

Part 2 K-means

1. Task 0

(1) Runtime for sequential implementation before optimization:

Kmeans01.dat: 0.0167sec

Kmeans02.dat: 0.4314sec

Kmeans03.dat: 22.1744sec

Kmeans04.dat: 166.1230sec

(2) Runtime for Openmp implementation before optimization:

Kmeans01.dat: 0.0006sec

Kmeans02.dat: 0.0130sec

Kmeans03.dat: 0.6656sec

Kmeans04.dat: 5.5084sec

(3) The configuration of the K-means algorithm being tested:

	numObjs	numCoords	numClusters	threshold
Kmeans01.dat	351	34	32	0.0010
Kmeans02.dat	7089	4	32	0.0010
Kmeans03.dat	191681	22	32	0.0010
kmeans04.dat	488565	8	32	0.0010

2. Techniques To Speed up

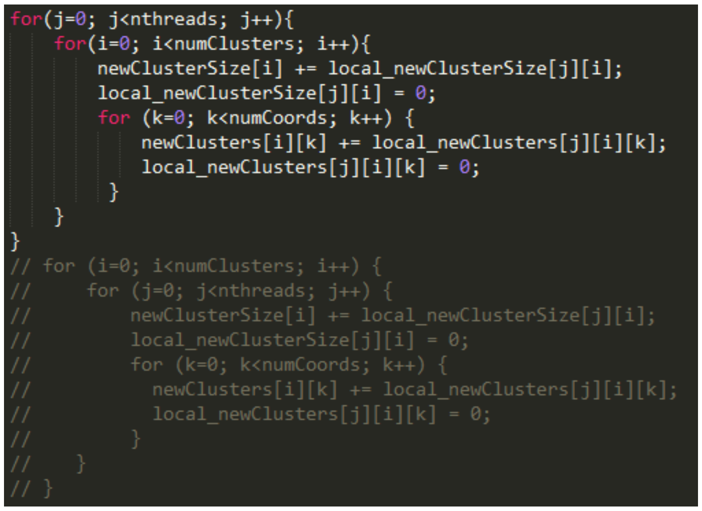
(1) Cache Optimization

a. **Optimization Goal:**

Reduce conflict cache miss.

b. **Optimization Process:**

Reorder the nested loops i, j into j, i.



```
for(j=0; j<nthreads; j++){
    for(i=0; i<numClusters; i++){
        newClusterSize[i] += local_newClusterSize[j][i];
        local_newClusterSize[j][i] = 0;
        for (k=0; k<numCoords; k++) {
            newClusters[i][k] += local_newClusters[j][i][k];
            local_newClusters[j][i][k] = 0;
        }
    }
}
// for (i=0; i<numClusters; i++) {
//     for (j=0; j<nthreads; j++) {
//         newClusterSize[i] += local_newClusterSize[j][i];
//         local_newClusterSize[j][i] = 0;
//         for (k=0; k<numCoords; k++) {
//           newClusters[i][k] += local_newClusters[j][i][k];
//           local_newClusters[j][i][k] = 0;
//         }
//     }
// }
```

Since in 2-D array, the values which have same j index are nearby, the spatial locality will achieve and so cache misses will reduce.

A[0][0]	A[0][1]	......	A[0][N]	A[1][0]	A[1][1]	A[1][2]	......	......

c. **Expected Speed up:**

L1 cache is 4 cycles and L2 cache is 12 cycles, so the expected speed-up: 12/4*0.01%+1*99.99%= 1.3

d. **Observed Speed up:**

Observation Speed-up: 1.2

e. **Explanation for the difference:**

In fact, if we review the procedure of k-means algorithm (initialization, Expectation, Maximization), the reorder is only referred to one of the procedure. So, the whore GFLOPs speed up partly.

Besides, there may be the false sharing in cache, where the variables of different threads may share the same cache set. This might make weaken performance.

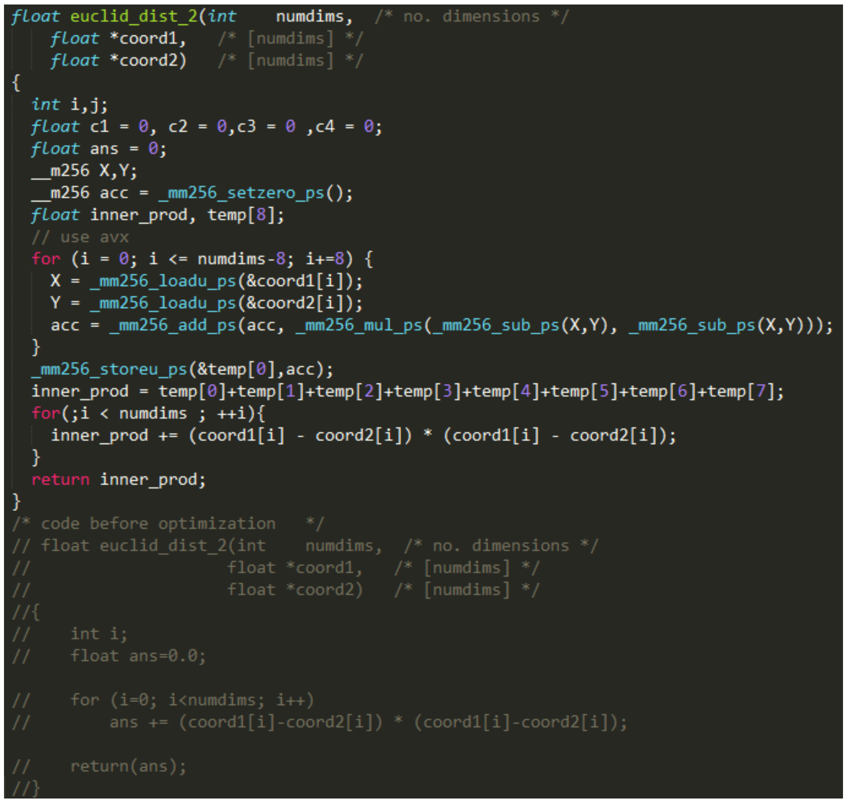
(2) SIMD

a. **Optimization Goal:**

Do the instruction-level optimization. Try to calculate multiple results for the same operation by using different data.

b. **Optimization Process:**

Use AVX instruction set. In each loop we can handle 8 data, so that there might have a 8x speed up. The example code is presented below (We optimize the euclid_dist_2):



```c
float euclid_dist_2(int    numdims,  /* no. dimensions */
    float *coord1,   /* [numdims] */
    float *coord2)   /* [numdims] */
{
  int i,j;
  float c1 = 0, c2 = 0,c3 = 0 ,c4 = 0;
  float ans = 0;
  __m256 X,Y;
  __m256 acc = _mm256_setzero_ps();
  float inner_prod, temp[8];
  // use avx
  for (i = 0; i <= numdims-8; i+=8) {
    X = _mm256_loadu_ps(&coord1[i]);
    Y = _mm256_loadu_ps(&coord2[i]);
    acc = _mm256_add_ps(acc, _mm256_mul_ps(_mm256_sub_ps(X,Y), _mm256_sub_ps(X,Y)));
  }
  _mm256_storeu_ps(&temp[0],acc);
  inner_prod = temp[0]+temp[1]+temp[2]+temp[3]+temp[4]+temp[5]+temp[6]+temp[7];
  for(;i < numdims ; ++i){
    inner_prod += (coord1[i] - coord2[i]) * (coord1[i] - coord2[i]);
  }
  return inner_prod;
}
/* code before optimization   */
// float euclid_dist_2(int    numdims,  /* no. dimensions */
//                    float *coord1,   /* [numdims] */
//                    float *coord2)   /* [numdims] */
//{
//    int i;
//    float ans=0.0;

//    for (i=0; i<numdims; i++)
//        ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);

//    return(ans);
//}
```

c. **Expected Speed up:**

Since the system do the operations in vector in length of 8, so the bytes of the data transfer per second is 7 times more than previous counterpart, so the speed-up should be 8.

d. **Observed Speed up:**

7

e. **Explanation for the difference:**

This might results from the Ceilings. Opterons have dedicated multipliers and adders. Since the code is dominated by adders and multipliers, the SIMD performance may be weakened.

(3) Using Multicore

a. **Optimization Goal:**

Use multiple threads to run one code to achieve parallelism.

b. **Optimization Process:**

Use Openmp library.

c. **Expected Speed up:**

since the thread number = 8, the speed up should be 8.

d. **Observed Speed up:**

comparison to sequential the speed-up is 8, which achieve expectation.

(4) Other optimization methods

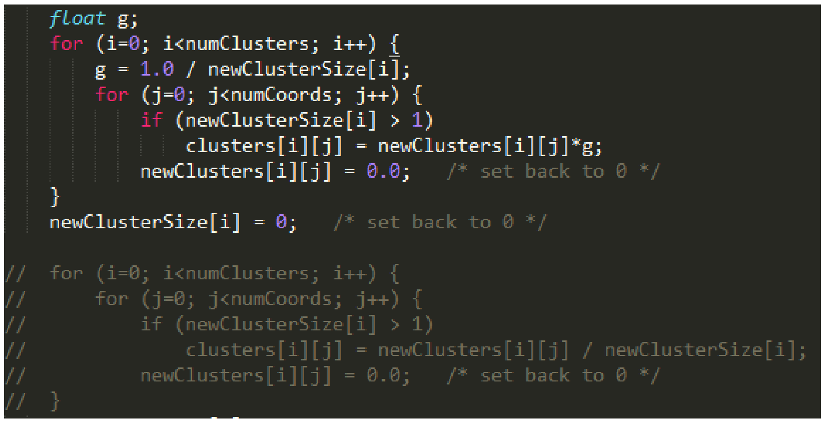
<1> Remove expensive operations:

a. **Optimization goal:**

Try to remove the multiplication operation in accessing data of step of matrix transposition.

b. **Implementation of our code:**

Since division operation is more expensive than multiplication, we use multiplication instead of division.



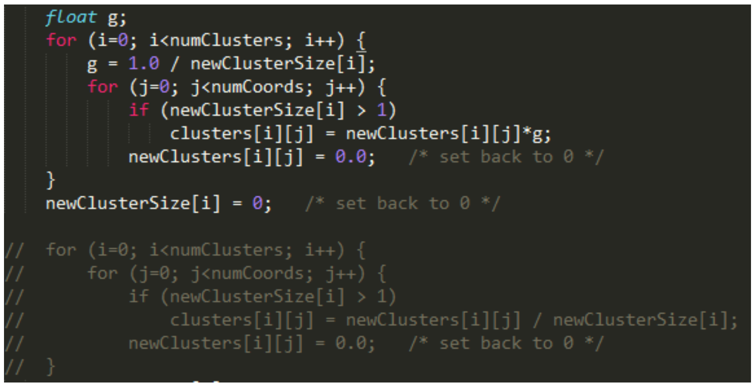
```
    float g;
    for (i=0; i<numClusters; i++) {
        g = 1.0 / newClusterSize[i];
        for (j=0; j<numCoords; j++) {
            if (newClusterSize[i] > 1)
                clusters[i][j] = newClusters[i][j]*g;
            newClusters[i][j] = 0.0;   /* set back to 0 */
    }
    newClusterSize[i] = 0;   /* set back to 0 */

//  for (i=0; i<numClusters; i++) {
//      for (j=0; j<numCoords; j++) {
//          if (newClusterSize[i] > 1)
//              clusters[i][j] = newClusters[i][j] / newClusterSize[i];
//          newClusters[i][j] = 0.0;   /* set back to 0 */
//  }
```

<2> Common sub-expression elimination:

a. **Optimization goal:**

For some variables that will be use multiple times in one loop with the same value, we can calculate the value of it outside the loop to reduce the operations in one loop.

b. **Implementation of our code:**



```
    float g;
    for (i=0; i<numClusters; i++) {
        g = 1.0 / newClusterSize[i];
        for (j=0; j<numCoords; j++) {
            if (newClusterSize[i] > 1)
                clusters[i][j] = newClusters[i][j]*g;
            newClusters[i][j] = 0.0;   /* set back to 0 */
    }
    newClusterSize[i] = 0;   /* set back to 0 */

//  for (i=0; i<numClusters; i++) {
//      for (j=0; j<numCoords; j++) {
//          if (newClusterSize[i] > 1)
//              clusters[i][j] = newClusters[i][j] / newClusterSize[i];
//          newClusters[i][j] = 0.0;   /* set back to 0 */
//  }
```

c. **Observed speed up:**

1.02