Part1

The optimized function is

```
for(i=0; i<N; i++)
   for(j=0; j<N; j++){
     c[i][j] = 0;
     for(k=0; k<N; k++){
       c[i][j] += a[i][k] * b[k][j];
     }
   }
```

It will take the longest time for both type Long and Double, the least effective way.

The function eliminating the unnecessary memory access is

```
int i, j,k;
  for(i=0; i<N; i++)
    for(j=0; j<N; j++){
     data_t acc = 0; //use local variable to accumulate the result
      for(k=0; k<N; k++){
        acc += a[i][k] * b[k][j];
      }
      c[i][j] = acc;// assign the accumulator to c[i][j], which will eliminate the access to the memory address to elements in C for every iterations
    }
```

It is faster than the optimized one, but it is still the second least effective way for both Long and Double.

The function applying 2X1 loop unrolling is

```
int i, j,k;
  int limit = N-1;
  for(i=0; i<N; i++)
    for(j=0; j<N; j++){
      data_t acc = 0; // also use the accumulator
      for(k=0; k<limit; k+=2){ instead of go through the list one by one, it will go through two element for one time, and take two element into operation for one time, which will eliminate half of the iterations(iteration = N/2)
        acc += a[i][k] * b[k][j] + a[i][k+1] * b[k+1][j];
      }
      c[i][j] = acc;
    }
  for(; i<N; i++){to check any missing element
    for(; j<N; j++){
      data_t acc = 0;
      for(; k<N; k++){
        acc += a[i][k] * b[k][j];
      }
      c[i][j] = acc;
    }
  }
```

It is faster than eliminating memory access for both types and five N

For the function 2X2 loop unrolling, it also eliminates the iterations by taking two elements at one time. Instead of using only one accumulator(temp), I created another accumulator(temp1) to accumulate the element of k+1. Thus, it will create 2 critical paths for data flow, and 2 paths can do

the operations simultaneously. It is slightly slower than 2X1 for small size N, but quicker for large size N. **For the function 4X1 loop unrolling**, it just uses one accumulator to do the operation, but it will take 4 elements in the array for operation for one iteration which make iterations from N to N/4. Thus quicker than 2X1 and 2X2 unrolling. **For the 4X4**, it eliminates the iterations just like 4X1(from N to N/4), but it uses 4 accumulators, which creates 4 critical paths for data flow which operate simultaneously. 4X4 is quicker than 4X1 for both types of variable and 5 different Ns. **8X1** uses one accumulator which takes 8 elements of the array into operation for one iterations, which make the iterations from N to N/8. 8X1 is further quicker than 4X4. **8X8** uses 8 accumulators to take 8 different variables at one time and create 8 different critical paths for data flow that run simultaneously. Thus, faster than 8X1.

Part 2

For the case ijk and jik(Case 1), we assigned c[i][j] at the inner loop which would fix i and j, thus the missing rate of c is 0. For the a[i][k], it will access the memory with stride-1 which has good spatial locality, and the missing rate for a is 0.25. Somehow, for the b[k][j], it will access the memory with stride-N which has bad spatial locality, and misses every time(missing rate = 1). Thus the total number of miss for one iteration is 1 + 0.25 + 0 = 1.25. ijk and jki have similar performance with slight differences. ijk is slightly slower than jik.

For the case jki and kji(Case 2), in the inner loop of i, we fixed j and k, thus fixing the b[k][j], which has a missing rate 0. Somehow, for a[i][k], we fix k and increment i in the loop, which will access the memory with stride-N which has bad spatial locality, and misses every time(missing rate = 1). Same for the c[i][j] accessed with stride-N which gives a missing rate = 1. Thus the total number of misses for one iteration is 1+1+0 = 2, which is higher than the previous case(ijk and jik). For both double and long, kji is slightly quicker than jki.

For the case kij and ikj(Case 3), in the inner loop of j, we fixed k and i. Thus, fixing a[i][k], which hits every time and gives a missing rate 0. For both b[k][j] and c[i][j], we access the memory with stride-1 which has good spatial locality. Thus the missing rate for both b[k][j] and c[i][j] is 0.25. The total number of misses for one iteration is 0.25 + 0.25 + 0 = 0.5, which is the lowest in these 3 cases. For both double and long, ikj is slightly faster than kij.

Speed: Case 3 > Case 1 > Case 2