Using matrices of dimensions **A=(500,800) and B=(800, 400)**. **n=720,000**
The following are the times (in seconds) of each algorithm averaged over **100** iterations.

	-O0	-O3
ijk	0.331601	0.100169
jki	0.569263	0.177466
ikj (not required)	0.460295	0.0311214

Using matrices of dimensions **A=(1000,2000) and B=(2000, 800)**. **n=3,600,000**
The following are the times of each algorithm averaged over **20** iterations.

	-O0	-O3
ijk	3.51625	1.38942
jki	6.5717	6.64096
ikj (not required)	4.56366	0.302171

Analysis
C++ is a row-major language, so contiguous rows entries are stored in adjacent memory locations, but contiguous columns entries are likely to exist in nonadjacent memory locations. As such, accessing a different record in the same row makes it likely to be cached and accessing a different record in the same column makes it unlikely to already be cached.
For a matrix[*i*][*j*], every time we increment *i*, we are going to a new column and may have a higher potential for a miss in our cache.

We can see that the *ijk* algorithm runs faster than the *jki* algorithm. It comes down to how many times we reference a different column. With each different column access, we increase the chances of a miss in our cache. Let's compare the two (assuming matrix A=mXn, B=pXq):

ijk Algorithm*:*
```
for i={0,1,...,m-1}
    for j={0,1,…,q-1}
        int sum = 0
        for k={0,1,...,n-1}
            sum += a[ i ][ k ] * b[ k ][ j ]
        c[ i ][ j ] = sum
```

We can simply count how many times we change columns.

A: m	(i is the outermost loop, so it only runs m times)
B: n*m*q	(k is the innermost loop, so it gets reset every time either outer loop resets)
C: m	(i is the outermost loop, so it only runs m times)
Total: 2m + mnq	(sum of all our column accesses)

jki Algorithm*:*

```
for j={0,1,...,q-1}
    for k={0,1,…,n-1}
        r = b[ k ][ j ]
        for i={0,1,...,m-1}
            c[ i ][ j ] += a[ i ][ k ] * r
```

A: mnq	(i is the innermost loop, so it gets changed the most)
B: qn	(k is the middle loop, so it's reset only when the outer loop increments)
C: mnq	(i is the innermost loop, so it gets changed the most)
Total: 2mnq + qn	(sum of all our column accesses)

Since each time we change the column, we are increasing the chance of a cache miss, we will compare the number of times we change the columns. The *ijk* algorithm has (2m+mnp) column changes, while the *jki* algorithm has (2mnq+qn) column changes. For square matrices, we would get *ijk*=(2m+mmm) compared to *jki*=(2mmm+mm). Doubling a cubic term is going to lead to a much higher number of cache misses. This makes sense with our data that shows that the runtime is doubled (sometimes more than double) due to these extra cache misses.

Why is optimization much faster?

The optimizer is likely parallelizing the for-loops because the runtime is significantly reduced. In the best case scenario, the optimized code is up to 12 times faster, and in the worst case, it's nearly equal. However, this big difference in runtime deltas can also be attributed to randomness (relative to my program) in scheduling on the CPU. The optimizer may also be doing things like inlining functions, which moves the overhead from runtime, to compile time.

Note:

It is possible to find some matrix dimensions (m,n,p,q) that make it so *jki* is favorable, but in general, that algorithm is much less efficient when using a row major language.