Here's another way to compute the computational intensity for blocked matrix-matrix multiplication, which I talked about in CS 240A on January 16, 2014.

First recall the definition: The computational intensity of an algorithm is q = t/v, where t is # of basic operations (e.g. floating-point adds and multiplies) and v is # of words moved between fast and slow memory. Because moving a word of data is much slower than doing an operation on it, we want to use algorithms with high computational intensity.

Now consider the naive 3-loop algorithm for multiplying two n-by-n matrices A and B, adding the result to matrix C:

```
> for i = 1:n
>   for j = 1:n
>     for k = 1:n
>       C(i,j) += A(i,k) * B(k,j);
```

What's the computational intensity? We need to count words moved between main memory (slow memory) and cache (fast memory), which means that we need to figure out what data in cache can be reused while it's still there. We will assume that there's enough room in cache for a few matrix rows of n elements each, but there's not enough room in cache for all n^2 elements of a matrix. Here's the reuse analysis:

> C(i,j) is reused at least n times, once for each iteration of the "k" loop
> A(i,k) is reused at least n times, once for each iteration of the "j" loop
(because row i of A can stay in cache throughout the whole "k" and "j" loops)
> B(k,j) is read n times (once per iteration of the "i" loop), but all the rest of B must enter cache in between two reads of B(k,j), so we can't keep B(k,j) in cache for more than one use.

Thus, the statement in the inner loop always moves one word, B(k,j), from slow to fast memory. It does two operations, so the computational intensity is q = 2/1 = 2.

Now consider the blocked algorithm described on the slides and in class. We divide each n-by-n matrix into blocks of size b-by-b. Each matrix has n^2 elements, and has N^2 blocks where N = n/b. The blocked algorithm uses three loops over the blocks. The statement in the inner loop multiplies and adds blocks using the naive method above -- thus, if we wrote it out in detail, the blocked algorithm would actually have six nested loops. Here it is:

```
> for i = 1:N
>   for j = 1:N
>     for k = 1:N
>       (block C(i,j)) += (block A(i,k)) * (block B(k,j));
```

The key is to choose the block size b so that a few blocks of b^2 elements each can fit into cache at the same time. Then the inner "block" * and += operations can run within cache -- the blocks come into cache at the beginning of the *+= , and go out at the end of it, but there's no memory traffic during the *+=.

Here are the details of the analysis of the computational intensity of the blocked algorithm.

Ignore the cost of moving C for the moment, and focus just on A and B. The statement in the inner loop above moves 2*b^2 words of data (reading a block of A and a block of B), and does 2*b^3 arithmetic operations (multiplying and adding blocks). Thus the computational intensity for every single execution of the inner loop is exactly 2*b^3 / 2*b^2 = b. Therefore the computational intensity for the whole algorithm is exactly b as well, still ignoring the cost of moving C.

To justify ignoring C, note that each block of C moves twice (once into fast memory and once out), while each block of A and of B moves N times. Thus, if N is reasonably large, C's contribution to m is relatively small, and the intensity will still be q = b to first order.

The blocked algorithm is a big deal: we have increased the computational intensity from a fixed 2 to a quantity b that can be (roughly) as large as half the square root of the cache size. On a modern processor with multiple levels of cache, the matrix multiplication routine in the BLAS library plays the blocking game once per level of cache, so there may be as many as 12 or 15 nested loops in it!