MITOCW | MIT6_172_F10_lec08_300k-mp4

PROFESSOR: Good, we're going to take a detour today into the realm of algorithms. So when you're trying to make code go fast, of course, there's no holds barred. You can use whatever you need to in order to make it go fast. Today we're going to talk a little bit in a more principled way about the memory hierarchy.

And to do that we're going to introduce what we call the ideal-cache model. So as you know most caches are hacked together to try to provide something that will cache well while still making it easy to build and fast to build. The ideal-cache model is a pretty nice beast if we had them.

It's got a two-level hierarchy. It's got a cache that has m bytes that are organized in to b byte cache-lines. So each block is b bytes. And it's fully associative. So you recall, that means that any line can go anywhere in cache.

And probably the most impressive aspect of an ideal-cache is that it has an optimal omniscient replacement algorithm. So what it does, is it figures out when it needs to kick something out of cache, it says, what is the absolutely best thing you could possibly kick out of cache. And it does that one. Looking into the future if need be.

It says, oh is this going to be accessed a lot in the future? I think I'll keep this one. Let's throw out this one. I know it's never going to be used again. So it has that omniscient character to it.

The performance measures we're going to look at in this model, the first one is what we call the work. And that's just the ordinary serial running time if you just ran the code on one processor and counted up essentially how many processor instructions you would do. That's essentially the work.

The second measure, which is the one that's much more interesting, is cache misses. So the work has to do with the processor. The cache misses has to do with what moves between these two levels of memory.

So in this case, what we're interested in doing is, how often do I try to access something. It's not in the cache. I have to go to main memory and bring it back into cache. And so that's what we'll be counting in this model.

So it's reasonable to ask how reasonable ideal caches are. In particular the assumption of omniscient replacement, that's pretty powerful stuff. Well it turns out there's a great lemma due to Slater and Tarjan that says essentially the following.

Suppose that you have an algorithm that incurs q cache misses on an ideal cache of size n. So you ran the algorithm on your machine, you had a cache of size n. Then, if instead of having an ideal cache, you have a fully associative cache of size two m and use the least recently used replacement policy.

So you always, whenever you're kicking something out of cache, you kick out the thing that has been touched the longest ago in the past. Then it incurs at most 2Q cache misses. So what that says is that LRU is to with it constant factors essentially the same as optimal. Really quite a remarkable result.

Who's taking 6046? You've just seen this, right? Yeah, OK. Just seen this result in 6046. See, I do talk to my colleagues occasionally.

So then something about how this is proved. And what's important here is that really it just says, OK, Yeah you could dither on the constants, but basically whether you choose LRU or choose ideal cache with the omniscient replacement, asymptotically you're not going to be off at all.

So for most asymptotic analyses, you can assume optimal or LRU replacement as convenient. And the typical way that you do convenience is if you're looking at upper bounds. So you're trying to show that a particular algorithm is good, then what you do is you assume optimal replacement.

If you're trying to show that some algorithm is bad, then what you do is assume that it's LRU to get a lower bound. Because then you can reason more easily about what's actually in memory. Because you just say, oh we'll just keep the least recently used one. So you tend to use the two for upper bounds and lower bounds.

Now, the way this relates to software engineering is as follows. If you're developing a really fast algorithm, it's going to start from a theoretically sound algorithm. And from that then you have to engineer for detailed performance.

So you have to take into account things like real caches are not fully associative. That loads and stores, for example, have different cost with respect to bandwidth and latency. So whether you miss some a load or miss on a store, there's a different impact. But these are all the tuning.

And as you know, those constant factors can sometimes add up to dramatic numbers, orders of magnitude. And so it's important to do that software engineering. But starting from a good theoretical basis means that you actually have an algorithm that is going to work well across a large variety of real situations.

Now, there's one other assumption we tend to make when we're dealing with ideal caches, and that's called the tall-cache assumption. So what the tall-cache assumption says, is that I you have at least as many lines of cache, essentially, in your cache, as you have bytes in the line. So it says the cache is tall.

In other words, this dimension here is bigger than this dimension here. And in particular, you want that to be true for where we have some constant here of slop that we can throw in. Yes, question.

AUDIENCE: Does that [INAUDIBLE] associatively make the cache shorter here.

PROFESSOR: Yes, so this is basically assuming everything is ideal. We're going to go back. When you engineer things, you have to deal with the fact that things aren't ideal.

But usually that's just a little bit of a tweak on the actual ideal algorithm. And for many programs, the ideal algorithm you don't actually have to tweak at all to get a good practical algorithm. So here is just saying the cache should be tall.

Now, just as an example, if we look at the machines that we're using, the cache-line length is 64 bytes. The L1 cache size is 32 kilobytes. And of course, for L1 it's is 32 kilobytes and for L2 and L3, it's even bigger. It's even taller. Because they also have 64k line. So this is a fairly reasonable assumption to make, that you have more lines in your cache then essentially the length, the number of items you can put on a cache line.

Now why is this an important assumption? So what's wrong with short caches? So we're going to look at, surprise, surprise, matrix multiplication. Which, by the end of this class you will learn more algorithms than matrix multiplication. But it is a good one to illustrate things.

So the idea here is, suppose that you have an n by n matrix here. And you don't have this tall-cache assumption. So where your cache is short. You have a lot of bytes in a line, but very few lines.

Then even if the size of your matrix fits, in principle, in the cache. In other words, n squared is less than m by more than a constant amount. So you'd say, Oh gee, that ought to fit in. If you have a short cache it doesn't necessarily fit in because your length n here is going to be shorter than the number of bytes on a line.

However, if you have a tall cache, it's always the case that if the matrix size is smaller than the cache size by a certain amount, then the matrix will fit in the cache. OK, question?

AUDIENCE: Why wouldn't you fit more than one row per cache line?

PROFESSOR: Well the issue is you may not have control over the way this is laid out. So, for example, if this is row-major order, and this is a submatrix of a much bigger matrix, then you may not have the freedom to be using these. But if you have the tall-cache assumption, then any section you pull out is going to fit. As long as the data fits mathematically in the cache, it will fit practically in the cache if you have the tall-cache assumption.

Whereas if it's short, you basically end up with the cache lines being long and you not having any flexibility as to where the data goes. So this is sort of a-- So any questions about that before we get into the use of this? We're going to see the use of this and where it comes up.

So one of the things is that, if it does fit in, then it takes, at most, size of the matrix divided by the cache line size misses to load it in. So this is linear time in the cache world. Linear time says, you should only take one cache fault for every line of cache. And so that's what you'll have here if you have a tall cache. You'll have n squared over b cache misses to load in n square data. And that's good. OK, good.

So let's take on the problem of multiplying matrices. We're going to look at square matrices because they're easier to think about than rectangular ones. But almost everything I say today will relate to rectangular matrices as well. And it we'll generalize beyond matrices as we'll see next time.

So here's a typical code for multiplying matrices. It's not the most efficient code in the world, but it's good enough to illustrate what I want to show you. So the first thing is, what is the work of this algorithm? This is, by the way, the softball question. What's the work? So the work, remember, is just if you're analyzing it just like processor forget about caches and so forth.

AUDIENCE: n cubed.

PROFESSOR: n cubed, right. Because there's a triply nested loop going up to n and you're doing constant work in the middle. So it's n times n times 1. n cubed work. That was easy.

Now let's analyze caches. So we're going to look at row major. I'm only going to illustrate the cache lines on this side because B is where all the action is.

So we're going to analyze two cases when the matrix doesn't fit in the cache. If the matrix fits in the cache, then there's nothing to analyze, at some level. So we're going to look at the cases where the matrix doesn't fit in the cache.

And the first one is going to be when the side of the matrix is bigger than m over b. So remember, m over b is the height of our cache, the number of lines in our cache. So let's assume for this, now I have a choice of assuming optimal omniscient replacement or assuming LRU. Since I want to show this is bad, I'm going to assume LRU.

Could somebody please close the back door there? Because it's we're getting some noise in from out there. Thank you.

So let's assume LRU. So what happens in the code is basically I go across a row of A, while I go down a column of B. And now if I'm using LRU, what's happening?

I read in this cache block and this one, then this one et cetera. And if n is bigger than M/B and I'm using least recently used, by the time I get down to the bottom here, what's happened the first cache line? First cache block? It's out of there. It's out of there if I used LRU.

So therefore, what happens is I took a miss on every one of those. And then when I go to the second one, I take a miss on every one again. And so as I keep going through, every access to B causes a miss throughout the whole accessing of B. Now go on to the second row A and I had the same thing repeats. So therefore, the number of cache misses is order n cubed since we miss on matrix B on every access. OK, question.

AUDIENCE: I know that you said it's e. Does B push out due to conflict misses or capacity misses?

PROFESSOR: So in this case they're capacity misses that we're talking about here. So there's no conflict misses in a fully associative cache. Conflict misses occurs because of direct mapping. So there's no conflict misses in what I'm going to be talking about today. That is an extra concern that you have for real caches, not a concern when you have a fully associative cache.

AUDIENCE: [INAUDIBLE] n needs to be bigger than B?

PROFESSOR: So the number of lines to fit in my cache is m over b, right?

AUDIENCE: So can't you put multiple units of data--

PROFESSOR: Well there are multiple units of data. But notice this is row major, what's stored here is B11, B12, B13. That's stored here. The way I'm going through the access, I'm going down the columns of B. So by the time to get up to the top again, that cache block is no longer there.

And so when I access B12, assuming indexing from one or whatever, this block is no longer in cache. Because LRU would say, somewhere along here I hit the limit of my size of cache, let's say around here. Then when this one goes in, that one goes out. When the next one goes in, the next one goes out et cetera using the least recently used replacement. So my--

AUDIENCE: Spatial locality.

PROFESSOR: I'm sorry? You don't have any spatial locality here.

AUDIENCE: I'm just wondering why they can't hold units. I guess this is the question, why can't they hold multiple addresses per cache line. So why is it even pushed out? It's being pushed out [UNINTELLIGIBLE] one per cache line, right?

PROFESSOR: No, so it's getting pushed out because the cache can hold M/B blocks, right? So once it's accessed m over b blocks, if I want to access anything else, something has to go out. It's a capacity issue. I access m over b blocks, something has to go out.

LRU says, the latest thing that I accessed, well that was the first one, gets knocked out. So what happens is every one causes a miss. Even though I may access that very nearby in the future, it doesn't take advantage of that. Because LRU says knock it out.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Is it row major that's the confusion? This is the way we've been dealing with are matrices. So in row major, there's a good-- that's nice, there's no chalk here. Oh, there's a big one there, great.

Yeah, so here's B. So the order that B is stored is like this in memory. So basically we're storing these elements in this order. So it's a linear block of memory, right? And it's being stored row by row as we go through.

So actually if I do it like this, let me do this a little bit more. So the idea is that the first element is going to be here, and then we get up to n minus 1, and then we get to n minus 2 is stored here. n plus 1, n plus 2, 2n minus 1. So that's the order that they're stored in linear and memory.

Now these guys will all be on the same cache line if it's in row-major storage. So when I'm accessing B, I'm going and I'm accessing zero, then I'm accessing the thing at location n. And I'm going down like this.

At some point here I reach the limit of my cache. This is M/B. Notice it's a different B-- script B verses-- so when I get to m over b. Now all these things are sitting in cache, that's great. However, now I go to one more, and it says OK, all those things are sitting in cache, which one do I kick out? And the answer is the least recently used one. That's this guy goes out.

AUDIENCE: Do you only use one element per cache link?

PROFESSOR: And I've only used one element from each cache line at this point. Then I go to the next one and it knocks out the second one. By the time I get to the bottom and then I go up to the top to access 1 here, it's not in cache. And so it repeats the same process, missing every single time. We have a question.

AUDIENCE: Yeah,so my question is why does the cache know where each row is? To us, we draw the matrix, but the computer doesn't know it's a matrix. To the computer, its a linear array of numbers.

PROFESSOR: That's correct.

AUDIENCE: So why would it load the first couple elements in the first row, and the second column is an extended row the second time.

PROFESSOR: So the cache blocks are determined by the locality and memory.

AUDIENCE: So my assumption would be the first cache line for say--

PROFESSOR: Let's say 0 through 3.

AUDIENCE: So yeah, 0 through 3.

PROFESSOR: Let's say we have four items on that cache line.

AUDIENCE: 4 to 6.

PROFESSOR: The next one the hold 4 to 7, I think.

AUDIENCE: 4 to 7, yeah. So that is a--

PROFESSOR: So that would be the next one, right? 4 to 7.

AUDIENCE: When you get cache line. You are not using the fully cache line. There is no spatial locale. You are using one from the cache line.

PROFESSOR: So this code is using this one and then this one. It's not using the rest. So it's not very efficient code.

AUDIENCE: So the cache line is holding the 0 to 3 and the 4 to 7. [INAUDIBLE] n plus 2 just reading--

[INTERPOSING VOICES]

PROFESSOR: Right. And those are fixed. So if you just a dice up memory in our machine into 64 byte sizes, those are the things that come in together whenever you access anything on that line.

AUDIENCE: And on this particular axis, you never actually get the 4 to the 7 in the--

PROFESSOR: Well we eventually do. Until we get there, yes, that's right. Until we get there.

Now, of course, we're also accessing A and C, but turns out to this analysis it sufficient to show that we're getting n cubed misses just on the matrix B. In order to say hey, we've got a lot of misses here.

So this was the case where n was bigger than the size of a cache. So the situation is a little bit different if n is large but still actually less than m over b.

So in this case, we suppose it n squared is bigger than m. So the matrix doesn't fit in memory. So that's what this part of the equation is, m to the 1/2 less than n is the same as n squared is bigger than memory. So we still don't fit in memory, but in fact it's less than some constant times m over b.

And now let's look at the difference with what happens with the caches as we go through the algorithm. So we essentially do the same thing. Once again, we're going to assume LRU.

And so what happens is we're going to go down a single row there. But now, notice that by the time I get down to the bottom, basically I've accessed fewer than some constant times m over b locations. And so nothing has gotten kicked out yet.

So when I go back to the top for the next access to B, all these things are still in memory. So I don't take a cache fall, a cache miss in those cases. And so we keep going through.

And basically this is much better because we're actually getting to take advantage of the spatial locality. So this algorithm takes advantage of the spatial locality. If n is really big it doesn't, but if n is just kind of big, then it does. And then if n is small enough, of course, it all fits in cache and there's no misses other than those needed to bring it in once. And then the same thing happens once you go through the next one.

So in this case, what's happening is we have n squared over b misses per run through the matrix B, and then we have n times that we go through. Once for every row of A. So the total then is n cubed over b the cache block size.

So depending upon the size, we can analyze with this, that this is better because we get a factor of B improvement. But it's still not particularly good. And it only works, of course, if my side of my matrix fits in the number of lines of cache that I have. Yeah, question.

AUDIENCE: Can you explain in-- I don't understand why you have n cubed over b?

PROFESSOR: OK, so we're going through this matrix n times. And for each one of those, we're running through this thing. So this thing basically, I get to go b times through, because all these things are going to be in memory when I come back to do them again.

And so it's only once every B columns that I take a miss. I take a miss and then I get to the other b minus 1 access is that I get cache hit. And so the total here is then n squared over b. So therefore a total of n cubed over b. So even this is not very good compared to what we can actually do if we exploit the cache well.

So let's go on and take a look. We saw this before. Let's use tiling.

So the idea of tiling is to say, let's break our matrix into blocks of s times s size. And essentially what we do is we treat our big matrix as if we're doing block matrix multiplications of things of size s by s.

So the inner loop here is doing essentially the matrix multiplication. It's actually matrix multiply and add. The inner three loops are just doing ordinary matrix multiplication, but on s-sized matrices.

And the outer loop is jumping over matrix by matrix for each of those doing a matrix multiply as its elemental piece. So this is the tiling solution that you've seen before. We can analyze it in this model to see, is this a good solution. So everybody clear on what the code does? So it's a lot of four loops, right? Yeah.

AUDIENCE: There should be less than n somewhere? There's like an and something.

PROFESSOR: Oh yeah. That must have happened when I coped it. That should be j less than n here. It should just follow this pattern. i less than n, k less than n, that should be j less than n there.

Good catch. I did execute this. That must have happened when I was editing.

So here's the analysis of work. So what's going on in the work? So here we have, basically the outer loop is going n over s times, each loop. So there's cube there times the inner loops here which are going each s times. So times s cubed.

Multiply that through, n cubed operations. That's kind of what you'd expect. What about cache misses?

So the whole idea here is that s becomes a tuning parameter. And whether we choose s well or poorly influences how well this algorithm works. So the idea here is we want tune s so that the submatrices just fit into cache.

So in this case, if I want a matrix to fit into cache, I want to be about the size of the square root of the cache size. And this is where we're going to use the tall-cache assumption now. Because I want to say, it fits in cache, therefore I can just assume it all fits in cache. It's not like the size fits but the actual data doesn't, which is what happens with the short cache.

So the tall-cache assumption implies that when I'm executing one of these inner loops, what's happening? When I'm executing one of these linear loops, all of the matrices are going to fit in cache. So all I have is my cold misses, if any, on that submatrix.

And how many cold misses can I have? Well the size of the matrix is s squared and I get to bring in b bytes of the matrix each time. So I get s squared over b misses per submatrix.

So that was a little bit fast, but I just want to make sure-- it's at one level straightforward, and the other level it's a little bit fast. So the point is that the inner three loops I can analyze if I know that s is fitting in cache. The inner three loops I can analyze by saying, look it's s squared data. Once I get the data in cache, if I'm using an optimal replacement, then it's going to stay in there. And so it will cost me s squared over b misses to bring that matrix in for each of the three matrices.

But once it's in there, I can keep going over and over it as the algorithm does. I don't get any cache misses. Because those all fitting in the cache. Question? Everybody with me? OK.

So then I basically have the outer three loops. And here I don't make any assumptions whatsoever. There's n over s iterations for each loop. And there's three loops. So that's n over s cubed. And then the cost of the misses in the inner loop is s squared over b. And that gives me n cubed over bm to the 1/2 if you plug in s being m to the 1/2.

So this is radically better because m is usually big. Especially for a higher level cache, for an L2 or an L3. m is really big.

What was the value we had before for the best case for the other algorithm when it didn't fit in matrix? It was n cubed over b. b is like 64 bytes. m is like the small L1 cache is 32 kilobytes. So you get to square root the 32 kilobytes. What's that?

So that's 32 kilobytes is 2 to the 15th. So it's 2 to the 7.5. 2 to the 7 is 128. So it's somewhere between 128 and 256.

So if we said 128, I've got a 64 and a 128 multiplier there. Much, much better in terms of calculating. In fact, this is such that if we tune this properly and then we say, well what was the cost of the cache misses here, you're not going to see the cost of the cache misses when you do your performance analysis. It's all going to be the work. Because the work is still n cubed.

The work is still n cubed, but now the misses are so infrequent, because we're only getting one every-- on the order of 64 times 128, which is 2 to the 6th times 2 to the 7th is 2 to the 13th is 8K. Every 8,000 or so accesses there's a constant factor in there or whatever, but every 8,000 or so accesses we're getting a cache miss. Uh, too bad.

If it's L1, that cost is four cycles rather than one. Or that cost us 10 cycles if I had to go to L2 rather than one, or whatever. So is to the point is, that's a great multiplier to have. So this is a really good algorithm. And in fact, this is the optimal behavior you can get for matrix multiplication.

Hong and Kung proved back in 1981 that this particular strategy and this bound was the best you could do for matrix multiplication. So that's great. I want you to remember this number because we're going to come back to it. So remember it's b times n to the 1/2, b times square root of m in the denominator.

Now there's one hitch in this story. And that is, what do I have to do for this algorithm to work well? It says right up there on the slide. Tune s. I've got to tune s.

How do I do that? How do I tune s? How would you suggest we tune s?

AUDIENCE: Just run a binary [INAUDIBLE].

PROFESSOR: Yeah, do binary search on s to find out what's the best value for s. Good strategy. What if we guess wrong?

What happens if, say, we tune s, we get some value for it. Let's say the value is 100. So we've turned it. We find 100 is our best value. We run it on our workstation, and somebody else has another job running.

What happens then? That other job starts sharing part of that cache. So the effective cache size is going to be smaller than what we turned it for. And what's going to happen? What's going to happen in that case? If I've tuned in for a given size and then I actually have to run with something that's effectively a smaller cache, does it matter or doesn't matter?

AUDIENCE: Is it still tall?

PROFESSOR: Still tall.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So if you imagine this fit exactly into cache, and now I only have half that amount. Then the assumption that these three inner loops is running with only s squared over b misses is going to be totally out the window. In fact, it's going to be just like the case of the first algorithm, the naive algorithm that I gave. Because the size of matrix that I'm feeding it, s by s, isn't fitting in the cache.

And so rather than it being s squared over b accesses, it's going to be much bigger. I'm going to end up with essentially s cubed accesses if the cache, in fact, gets enough smaller.

It's also one thing you have to put in there is what I like to call voodoo. Whenever you have a program and you've got some parameters that, oh good we've got these parameters we get to tweak to make it go better. I call those voodoo parameters. Because typically setting them is not straightforward.

Now there are different strategies. One, as you say, is to do binary search by doing it. There are some programs, in fact, which when you start them up you call an initialization routine. And what they will do is automatically check to see what size is my cache and what's the best size should I do something on and then use that when you actually run it later in the program. So it does an automatic adaptation automatically when you start. But the more parameters you get, the more troublesome it becomes.

So let's take a look. For example, suppose we have a two-level cache rather than a one-level cache. Now I need to have something that I tune in for L1 and something that I tune for L2. So it turns out that if I want to optimize s and t, I can't do it in more with binary search because I have two parameters. And binary search won't suffice for figuring out what's the best combination of s and t.

And generally multidimensional searches are much harder than one-dimensional searches for optimizing. Moreover, here's what the code looks like. So now I've got, how many four loops? 1,2,3,4,5,6,7,8,9 nested for loops.

So you can see the voodoo is starting to make this stuff run. You really have to be a magician to tune these things appropriately. I mean, if you can can do it that's great. But if you don't do it, OK.

So now what about three levels of cache? So now we need three tuning parameters. Here s, t and u, we have 12 nested four loops. I didn't have the heart to actually write out the code for the 12 nested for loops. That just seemed overhead.

But our new halo machines, they have three levels of caches.

So let's tune for all the levels of caches. And as we mentioned, in a multi-program environment, you don't actually know what the cache size is, what other programs are running. So it's really easy to mistune these parameters.

AUDIENCE: [INAUDIBLE] don't you have a problem because you're running the program for a particular n, and you don't necessarily know whether your program is going to run faster or slower--

PROFESSOR: Well what you're usually doing, is you're tuning for s not n, right? So you're assuming--

AUDIENCE: No, no a particular n.

PROFESSOR: But the tuning of this is only dependent on s. It doesn't depend on n. So if you run it for a sufficiently large n, I think it's reasonable to assume that the s you get would be a good s for any large n. Because the real question is, what's fitting in cache? Yeah--

AUDIENCE: How long does it take to fill up the cache relative to the context each time?

PROFESSOR: Generally you can do it pretty quickly.

AUDIENCE: Right. So why does it matter if your have multiple users, if you can fill it [INAUDIBLE].

PROFESSOR: No because, he may not be using all of the cache, right? So when you come back, you're going to have it polluted with a certain amount of stuff. I think it's a good question.

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, so anyway, so this is the-- yeah question.

AUDIENCE: So if n is really large, is it possible that the second row of the matrix never loaded?

PROFESSOR: If n is really large--

AUDIENCE: Because n is really large, right? Just the first row of the matrix will fill up all the caches.

PROFESSOR: It's LRU and in B you're going down this way. You're accessing things going down. OK, good. So let's look at a solution to these alternatives.

What I want to in particular take a look at is recursive matrix multiplication. So the idea is you can do divide and conquer on multiplying matrices because if I divide each of these into four pieces, then essentially I have 8 multiply adds of n over 2 by n over 2 matrices. Because I basically do these eight multiplies each going into the correct result.

So multiply A11 B11 and add it into C11. Multiply A12 B21 add it into C11 and so forth. So I can basically do divide and conquer. And then each of those I recursively divide and conquer.

So what's the intuition by why this might a good scheme to use?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, we're not going to do parallel yet. Just why is this going to use the cache well?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah eventually I get down to a size where the matrix that I'm working on fits into cache, and then all the rest of the operations I do are all going to be cache hits. It is taking something and it it's doing what the tiling is doing but doing it blindly.

So let's take a look. Here's the recursive code. So here I have the base case if n is 1, I basically have a one by one matrix and I just simply update c, with a times b. And otherwise what I do, is I'm going to do this by computing offsets. So generally when you're dealing with matrices, especially if you want fast code, I usually don't rely on two-dimensional addressing, but rather do the addressing myself and rely on the compiler to do common subexpression elimination.

So, for example, here what I'm going to do is compute the offsets. So here's how I do it. So first of all, in practice what you do, is you don't go down to n equals 1. You have some cutoff. Maybe n is 8 or something. And at that point you go into a specialized routine that does a really good 8 by 8 multiply.

And the reason for that is you don't want to have the function call overheads. This function call is expensive to do two floating point operations here. So you'd like to have a function call and then do 100 floating point operations or something. So that you get a better balance. Do people understand that?

So normally to write recursive codes you want a course in the recursion. Make it so you're not going all go the into way down to n equals 1. But rather are stopping short and then doing something that doesn't involve a lot of overhead in the base case of your recursion. But here I'll explain it as if we went all the way down to n equals 1.

So then what we do is, if this is a submatrix, which is basically what I'm showing here. We have an n by n submatrix. And it's being pulled out on a matrix of size row size, of width row size.

So what I can do is, if I want to know where the elements of the beginning of matrices are, well the first one is exactly the same place that the input matrix is. The second one is basically I have to add n over 2 to the location in the array. The third one here, 21, I have to basically add n over 2 rows to get the starting point of that matrix.

And for the last one I have to add n over 2 and n over 2 rows and n over 2 plus 1 rows to get to that point. So I compute those and now I can recursively multiply with sizes of n over 2 and perform the program recursively. Yeah--

AUDIENCE: So you said it rightly. You're blindly dividing the matrix up til you get something to fit the cache. So essentially--

PROFESSOR: Well and you're continuing. The algorithm is completely blind all the way down to n equals 1.

AUDIENCE: This could never be better if the other one-- your computer's version is well-tuned. Because the applications are the same, but this one you have all the overhead from the [INAUDIBLE].

PROFESSOR: Could be.

AUDIENCE: At the end, you still need to make a multiplication and then go back and look at all of the--

PROFESSOR: Could be. So let's discuss that later at the end when we talk about the differences between the algorithms. Let's at this point, just try to understand what's going on in the algorithm. Question--

AUDIENCE: [INAUDIBLE]

PROFESSOR: n over 2 times row size plus-- plus n over 2. It should be row size plus 1. You're right. Good, bug. Should be n over 2 times row size plus 1.

So let's analyze the work assuming the code actually did work. So the work we can write a recurrence for. So here we have the work to solve an n by n matrix problem.

Well if n is 1, then it's just order one work-- constant amount of work. But if n is bigger than 1, then I'm solving eight problems of size n over 2, plus doing a constant amount of work to divide all those up. So everybody understand where I get this recurrence?

Now normally, as you know, when you do algorithmic work, we usually omit this first line because we assume a base case of constant if it's one. I'm actually going to keep it. And the reason is because when we do caching the basic cases are important. So everybody understand where this recurrence came from?

So I can use the master theorem or something like that to solve this. In which case the answer for this is what? Those of you who have the master theorem in your hip pocket.

What's the solution of this recurrence? People remember? Who's has heard of the master theorem? I thought that was kind of a prerequisite or something of this class, right? So you might want to brush up on the master theorem for the quiz next week.

So basically it's a n over b, so it's n to the log base 2 of 8. So that's n cubed, n to the log base 2 of 8 is n to the n cubed. And that's bigger than the order one here, so the answer is order n cubed. Which is a relief, right? Because if weren't order n cubed we would be doing a lot more work than one of the looping algorithms.

However, let's actually go through and understand where that n cubed comes from. And to do that I'm going to use the technique of a recursive tree, which I think all of you have seen. But let me go through it slowly here to make sure, because we're going to do it again when we do cache misses and it's going to be more complicated.

So here's the idea. I write down the left hand side the recurrence, w of n. And now what I do is I substitute, and I draw it out as a tree. I have eight problems of size n over 2. So what I do is I replace that with the thing that's on the right hand side, I've dropped the theta here, but basically put just a constant one here. Because I'll take into account the thetas at the end.

So I have a one here, and then I have, loops that should be a w. Should be w n over 2. That's a bug there.

And then I replace each of those. OK, wn over 2, sorry that should be wn over 4. Ah, more bugs. I'll fix them up on after lecture.

So this should be w of n over 4. And we go all the way down to the bottom to where I hit the base case of theta 1. So I built out this big tree that represents, if you think about it, that's exactly what the algorithm is going to do. It's going to walk this tree doing the work. And what I've just simply put up here is to work it does at every level.

So the first thing we want to do is figure out what's the height of this tree. Can somebody tell me what the height of the tree is? It is a log n. What's the base? Log base 2 of n, base because at every level if I hadn't made a mistake here, I'm actually having the argument. So I'm having the argument at each level. So the height is log base 2 of n. So LG is notation for log base 2.

So if I have log base 2 of n, I can count how many leaves there are to this tree. So how many leaves are there? Well I'm branching a factor of eight at every level.

And if I'm going log base 2 levels, the number of leaves is 8 to the log base 2. So 8 to the log base 2 of n. And then with a little bit of algebraic magic that turns out that's the same as n to the log base 2 of 8. And that is equal to n cubed. So I end up with n cubed leaves.

Now let's add up all the work that's in here. So what I do is I add across the rows. So the top level I've got work of one. The next level I work of eight. The next I have work of 64. Do people see the pattern? The work is growing how? Geometrically.

And at this level I know that if I add up all the leaves I've got work of n cubed. Because I've got n cubed leaves, each of them taking a constant. And so this is geometrically increasing, which means that it's all born in the leaves.

So the total work is order n cubed. And that's nice. It's the same work is the looping versions. Because we don't want to increase that. Questions? Because now we're going to do cache misses and it's going to get hairy, not too hairy, but hairier.

So here we're going to cache misses. So the first thing is coming up with a recurrence. And this is probably the hardest part, except for the other hard part which is solving the recurrence.

So here what we're doing is, we have the same thing is that I'm solving eight problems of size n over 2 and to do the work in here. I'm taking basically order one cache misses. However I do, those things work out. Plus the cache misses I have in there.

But then at some point, when I'm claiming is that I'm going to bottom out the recursion early. Not when I get to n equals 1, but in fact when n squared is less than some constant times the cache size. For some sufficiently small concept. And what I claim, at that point, is that the number of cache misses I'm going to take at that point, I can just, without doing any more recursive stuff, I can just say it's n squared over b.

So where does that come from? So this basically comes from the tall-cache assumption. So the idea is that when n squared is less than a constant times the size of your cache, constant times the size of m, then that means that this fits into-- the n by n matrices fit within m. I've got three of them. I've got C, A and B. So that's where I need a constant here. So they're all going to fit in memory.

And so if I look at it, all I have to do is count up the cold misses for bringing in those submatrices at the time that n hits this threshold here of some constant times m. And to bring in those matrices is only going to cost me n squared over b cache misses.

And once I've done that, all of the rest of the recursion that's going on down below is all operating out of cache. It's not taking any misses if I have an optimal replacement algorithm. it's not taking any more misses as I get further down. Questions about this part of the recurrence here? So people with me?

So when I get down to something of size n squared, where the submatrix is size n squared, the point is that I'll bring in the entire submatrix. But all the stuff that I have to do in there is never going to get kicked out, because it's small enough that it all fits. And an optimal algorithm for replacement is going to make sure that stuff stays in there, because there's plenty of room in the cache at that point. There's room for three matrices in the cache and a couple of other variables that I might need and that's basically it. Any questions about that?

So let's then solve this recurrence. So we're going to go about it very much the same way. We make draw a recursion tree. So those of you are rusty in drawing recursion trees, I can promise you there will be a recursion tree on the quiz next Thursday.

I think I can promise that. Can I promise that? Yeah, OK I can promise that.

The way I like to do it, by the way, is not to try to just brought out all at once. In my own notes when I do this I always draw it step by step. I copy over and just do a step by step.

You might think that that's extensive. Gee, why do I have to draw every one along the way? Well the answer is, it's a geometric process. All the ones going up to the last one are a small amount of the work to draw out the last one. And they help you get it correct the first time. So let me encourage you to draw out the tree iteration by iteration. Here I'm going to just do replacement.

So what we do is we replace with the right hand side to do the recursion. And replace that. And once again I made the bug, that should be in over 8. Sorry, n over 4 here. n over 4. And then we keep going down until I get to the base case, which is this case here.

Now comes the first hard part. How tall is this tree? Yeah--

AUDIENCE: [INAUDIBLE] square root of n over b. You want n squared to be cm, not [INAUDIBLE].

PROFESSOR: So here's the thing, let's discuss, first of all, why this is what it is. So at the point where n squared is less than cm, that says that it's going to cost us n squared over b. But n squared is just less than cm, so therefore, this is effectively m over b.

Good question. So everybody see that? So when I get down to the bottom, it's basically costing me something that's about the number of lines I have in my cache, number of misses to fill things up.

The tricky thing is, what's the height? Because this is crucial to getting this kind of calculation right. So what is the height of this tree? So I'm having every time.

So one way to think about it is, it's going to be log bas 2 of n, just as before, minus the height of the tree that is hidden here that I didn't have to actually go into because there are no cache misses in it. So that's going to occur when n is approximately m, cm, sorry when n is approximately square root of cm.

So I end up with log of n minus 1/2 log of cm. That's the height here. Because the height at this point of the tree that's missing because they're no cache, I don't have to account for any cache misses in there, is log of cm to the one half, based on this. Does that follow for everybody? People comfortable? Yeah? OK, good.

So now what do we do? We count up how many leaves there are. So the number of leaves is 8, because I have a branching factor of 8, 2 whatever the height is. Log n minus 1/2 log of cm.

And then if I do my matrix magic, well that part is n cubed, the minus becomes a divide, and now 8 to the 1/2 log of cm is the square root of n cubed, which is m to the 3/2. Is that good?

The rest of it is very similar to what we did before. At every level I have a certain number of things that I'm adding up. And on the bottom level, I take the cost here, m over b, and I multiply it by a number of leaves. When I do that I get, what?

I get n cubed over b times m to the 1/2. This is geometric. So the answer is going, in this case, just going to be the sum of a constant factor times the large thing.

And why does this look familiar? That was the optimal result we got from tiling. But where's the tuning parameters? No tuning parameters. No tuning parameters.

So that means that this analysis that I did for one level of caching, it applies even if you have three levels of caching. At every level you're getting near optimal cache behavior. So it's got the same cache misses as with tiling.

These are called cache-oblivious algorithms. Because the algorithm itself has no tuning parameters related to cache. Unlike the tiling algorithm. That's a cache-aware algorithm.

The cache-oblivious algorithm has no tuning parameters. And if it's an efficient one. So, by the way, our first algorithm was cache-oblivious as well. The naive one. It's just not efficient.

So in this case we have an efficient one. It's got no voodoo turning of parameters, no explicit knowledge of caches, and it passively autotunes itself. As it goes down when it fits things into cache it fits them and uses things locally. And then it goes down and it fits into the next level of cache and uses things locally and so forth.

It handles multi-level caches automatically. And it's good in multi-programmed environments. Because if you end up taking away some of the cache it doesn't matter. It still will end up using whatever cache is available nearly as well as any other program could use that cache. So these are very good in multi-programmed environments.

The best cache-oblivious matrix multiplication, in fact doesn't do an eight way split as I described here. That was easier to analyze and so forth. The best one that I know work on arbitrary rectangular matrix. And what they do, is they do binary splitting.

So you would take your matrix, i times j, So if you take a matrix, let's say it's something like this. So here we have i, k, k, j. And you're going to get something of shape. i times j, right?

What it does, is it takes whatever is the largest dimension. In this case k is the largest dimension. And it partitions either one or both of the matrices along k. In this case, it doesn't do that. And then it recursively solves the two sub-rectangular problems. And that ends up being a very, very efficient fast code if you code that up tightly.

So it does binary splitting rather than-- and it's general. And if you analyze this, it's got the same behavior as the eight way division. It's just more efficient.

So questions? We had a question about now comparing with the tiled algorithm. Do you want to reprise your question?

AUDIENCE: What I was saying was, I guess this answers my question. If you were to tune the previous algorithm properly, and you're assuming it's not in a multi-program environment, the recursive one, it will never be the one that is locked. [INAUDIBLE]

PROFESSOR: So at some level that's true, and at some level it's not true. So it is true in that if it's cache-oblivious you can't take advantage of all the corner cases that you would might be able to take advantage of in a tiling algorithm. So from that point of view, that's true.

On the other hand, these algorithms work even as you go into paging and disks and so forth. And the interesting thing about a disk, if you start having a big problem that doesn't fit in memory and, in fact, is out of core as they call it, and is paging to disk, is that the disk sizes of sectors that can be brought efficiently off of a disk, vary.

And the reason is because in a disk, if you read a track around the outside you can get two or three times as much data off the disk as a track that you read near the inside. So the head moves in and out of the disk like this. It's typically on a pivot and pivots in and out.

If it's reading towards the inside, you get blocks that are small versus blocks that are large. This is effectively a cache line size that gets brought in. And so the thing is that there are actually programs in which, when you run them on disk, there is no fixed size tuning parameter that beats the cache-oblivious one.

So the cache-oblivious one will beat every fixed-size tuning parameters you put in. Because you don't have any control over where your file got laid out on disk and how much it's bringing in and how much it isn't varies.

On the other hand, for in-core thing, you're exactly right. That, in principle, you could tune it up more if you make it more cache aware. But then, of course, you suffer from portability loss and from, if you're in a multi-programmed environment and so forth.

So the answer is, that there are situations where you're doing some kind of embedded or dedicated type of application, you can take advantage of a lot of things that you want. There are other times where you're doing a multi-programmed environment, or where you want to be able to move something from one platform to another without having to re-engineer all of the tuning and testing. In which case it's better to use the cache oblivious.

So as I mentioned, my view of these things is that performance is like a currency. It's a universal medium of exchange. So one place you might want to pay a little bit of performance is to make it so it's very portable as for the cache-oblivious stuff. So you get nearly good performance, but now I don't have that headache to worry about. And then sometimes, in fact, it actually does as well or better than the one. For matrix multiplication, the best algorithms are the cache oblivious ones that I'm aware of.

AUDIENCE: [INAUDIBLE] currency and all the different currencies. Single currency.

PROFESSOR: You want a currency for-- so in fact the performance for this is people who have engineered it to take advantage of exactly the cache size, we can do just as well with the cache oblivious one. And particularly, if you think about it, when you've got three levels hierarchy, you've got 12 loops. And now you're going to tune that. It's hard to get it all right.

So next time we're going to see a bunch of other examples of cache-oblivious algorithms that are optimal in terms of their use of cache. Of course, by the way, those people who are familiar with Strassen's algorithm, that's a cache-oblivious algorithm. Takes advantage the same kind of thing. And in fact you can analyze it and come up with good bounds on performance for Strassen's algorithm just the same. Just as we've done here.