CHARLES LEISERSON: So welcome to 6.172. My name is Charles Leiserson, and I am one of the two lecturers this term. The other is Professor Julian Shun. We're both in EECS and in CSAIL on the 7th floor of the Gates Building. If you don't know it, you are in Performance Engineering of Software Systems, so if you found yourself in the wrong place, now's the time to exit.

I want to start today by talking a little bit about why we do performance engineering, and then I'll do a little bit of administration, and then sort of dive into sort of a case study that will give you a good sense of some of the things that we're going to do during the term. I put the administration in the middle, because it's like, if from me telling you about the course you don't want to do the course, then it's like, why should you listen to the administration, right? It's like--

So let's just dive right in, OK? So the first thing to always understand whenever you're doing something is a perspective on what matters in what you're doing. So the whole term, we're going to do software performance engineering. And so this is kind of interesting, because it turns out that performance is usually not at the top of what people are interested in when they're building software. OK? What are some of the things that are more important than performance? Yeah?

AUDIENCE: Deadlines.

CHARLES LEISERSON: Deadlines. Good.

AUDIENCE: Cost.

CHARLES LEISERSON: Cost.

AUDIENCE: Correctness.

CHARLES Correctness.

LEISERSON:

AUDIENCE: Extensibility.

CHARLES LEISERSON: Extensibility. Yeah, maybe we could go on and on. And I think that you folks could probably make a pretty long list. I made a short list of all the kinds of things that are more important than performance. So then, if programmers are so willing to sacrifice performance for these properties, why do we study performance? OK? So this is kind of a bit of a paradox and a bit of a puzzle. Why do you study something that clearly isn't at the top of the list of what most people care about when they're developing software?

I think the answer to that is that performance is the currency of computing. OK? You use performance to buy these other properties. So you'll say something like, gee, I want to make it easy to program, and so therefore I'm willing to sacrifice some performance to make something easy to program. I'm willing to sacrifice some performance to make sure that my system is secure. OK?

And all those things come out of your performance budget. And clearly if performance degrades too far, your stuff becomes unusable. OK? When I talk with people, with programmers, and I say-- you know, people are fond of saying, ah, performance. Oh, you do performance? Performance doesn't matter. I never think about it.

Then I talk with people who use computers, and I ask, what's your main complaint about the computing systems you use? Answer? Too slow. [CHUCKLES] OK? So it's interesting, whether you're the producer or whatever. But the real answer is that performance is like currency. It's something you spend.

If you look-- you know, would I rather have $100 or a gallon of water? Well, water is indispensable to life. There are circumstances, certainly, where I would prefer to have the water. OK? Than $100. OK? But in our modern society, I can buy water for much less than $100. OK? So even though water is essential to life and far more important than money, money is a currency, and so I prefer to have the money because I can just buy the things I need.

And that's the same kind of analogy of performance. It has no intrinsic value, but it contributes to things. You can use it to buy things that you care about like usability or testability or what have you. OK? Now, in the early days of computing, software performance engineering was

common because machine resources were limited. If you look at these machines from 1964 to 1977-- I mean, look at how many bytes they have on them, right? In '64, there is a computer with 524 kilobytes. OK?

That was a big machine back then. That's kilobytes. That's not megabytes, that's not gigabytes. That's kilobytes. OK? And many programs would strain the machine resources, OK? The clock rate for that machine was 33 kilohertz. What's a typical clock rate today?

AUDIENCE: 4 gigahertz.

CHARLES LEISERSON: About what?

AUDIENCE: 4 gigahertz.

CHARLES LEISERSON: 4 gigahertz, 3 gigahertz, 2 gigahertz, somewhere up there. Yeah, somewhere in that range, OK? And here they were operating with kilohertz. So many programs would not fit without intense performance engineering. And one of the things, also-- there's a lot of sayings that came out of that era. Donald Knuth, who's a Turing Award winner, absolutely fabulous computer scientist in all respects, wrote, "Premature optimization is the root of all evil."

And I invite you, by the way, to look that quote up because it's actually taken out of context. OK? So trying to optimize stuff too early he was worried about. OK? Bill Wulf, who designed the BLISS language and worked on the PDP-11 and such, said, "More computing sins are committed in the name of efficiency without necessarily achieving it than for any other single reason, including blind stupidity." OK? And Michael Jackson said, "The first rule of program optimization-- don't do it. Second rule of program optimization, for experts only-- don't do it yet." [CHUCKLES]

OK? So everybody warning away, because when you start trying to make things fast, your code becomes unreadable. OK? Making code that is readable and fast-- now that's where the art is, and hopefully we'll learn a little bit about doing that. OK? And indeed, there was no real point in working too hard on performance engineering for many years.

If you look at technology scaling and you look at how many transistors are on various processor designs, up until about 2004, we had Moore's law in full throttle, OK? With chip densities doubling every two years. And really quite amazing. And along with that, as they

shrunk the dimensions of chips-- because by miniaturization-- the clock speed would go up correspondingly, as well.

And so, if you found something was too slow, wait a couple of years. OK? Wait a couple of years. It'll be faster. OK? And so, you know, if you were going to do something with software and make your software ugly, there wasn't a real good payoff compared to just simply waiting around. And in that era, there was something called Dennard scaling, which, as things shrunk, allowed the clock speeds to get larger, basically by reducing power. You could reduce power and still keep everything fast, and we'll talk about that in a minute.

So if you look at what happened to from 1977 to 2004-- here are Apple computers with similar price tags, and you can see the clock rate really just skyrocketed. 1 megahertz, 400 megahertz, 1.8 gigahertz, OK? And the data paths went from 8 bits to 30 to 64. The memory, correspondingly, grows. Cost? Approximately the same.

And that's the legacy from Moore's law and the tremendous advances in semiconductor technology. And so, until 2004, Moore's law and the scaling of clock frequency, so-called Dennard scaling, was essentially a printing press for the currency of performance. OK? You didn't have to do anything. You just made the hardware go faster. Very easy.

And all that came to an end-- well, some of it came to an end-- in 2004 when clock speeds plateaued. OK? So if you look at this, around 2005, you can see all the speeds-- we hit, you know, 2 to 4 gigahertz, and we have not been able to make chips go faster than that in any practical way since then. But the densities have kept going great. Now, the reason that the clock speed flattened was because of power density.

And this is a slide from Intel from that era, looking at the growth of power density. And what they were projecting was that the junction temperatures of the transistors on the chip, if they just keep scaling the way they had been scaling, would start to approach, first of all, the temperature of a nuclear reactor, then the temperature of a rocket nozzle, and then the sun's surface. OK? So that we're not going to build little technology that cools that very well.

And even if you could solve it for a little bit, the writing was on the wall. We cannot scale clock frequencies any more. The reason for that is that, originally, clock frequency was scaled assuming that most of the power was dynamic power, which was going when you switched the circuit. And what happened as we kept reducing that and reducing that is, something that used to be in the noise, namely the leakage currents, OK, started to become significant to the point

where-- now, today-- the dynamic power is far less of a concern than the static power from just the circuit sitting there leaking, and when you miniaturize, you can't stop that effect from happening.

So what did the vendors do in 2004 and 2005 and since is, they said, oh, gosh, we've got all these transistors to use, but we can't use the transistors to make stuff run faster. So what they did is, they introduced parallelism in the form of multicore processors. They put more than one processing core in a chip. And to scale performance, they would, you know, have multiple cores, and each generation of Moore's law now was potentially doubling the number of cores.

And so if you look at what happened for processor cores, you see that around 2004, 2005, we started to get multiple processing cores per chip, to the extent that today, it's basically impossible to find a single-core chip for a laptop or a workstation or whatever. Everything is multicore. You can't buy just one. You have to buy a parallel processor.

And so the impact of that was that performance was no longer free. You couldn't just speed up the hardware. Now if you wanted to use that potential, you had to do parallel programming, and that's not something that anybody in the industry really had done. So today, there are a lot of other things that happened in that intervening time. We got vector units as common parts of our machines; we got GPUs; we got steeper cache hierarchies; we have configurable logic on some machines; and so forth.

And now it's up to the software to adapt to it. And so, although we don't want to have to deal with performance, today you have to deal with performance. And in your lifetimes, you will have to deal with performance in software if you're going to have effective software. OK? You can see what happened, also-- this is a study that we did looking at software bugs in a variety of open-source projects where they're mentioning the word "performance." And you can see that in 2004, the numbers start going up. You know, some of them-- it's not as convincing for some things as others, but generally there's a trend of, after 2004, people started worrying more about performance.

If you look at software developer jobs, as of around early, mid-2000s-- the 2000 "oh oh's," I guess, OK-- you see once again the mention of "performance" in jobs is going up. And anecdotally, I can tell you, I had one student who came to me after the spring, after he'd taken 6.172, and he said, you know, I went and I applied for five jobs. And every job asked me, at every job interview, they asked me a question I couldn't have answered if I hadn't taken 6.172,

and I got five offers. OK?

And when I compared those offers, they tended to be 20% to 30% larger than people who are just web monkeys. OK? So anyway.

[LAUGHTER]

That's not to say that you should necessarily take this class, OK? But I just want to point out that what we're going to learn is going to be interesting from a practical point of view, i.e., your futures. OK? As well as theoretical points of view and technical points of view. OK?

So modern processors are really complicated, and the big question is, how do we write software to use that modern hardware efficiently? OK? I want to give you an example of performance engineering of a very well-studied problem, namely matrix multiplication. Who has never seen this problem? [LAUGHS] Yeah. OK, so we got some jokers in the class, I can see. OK.

So, you know, it takes n cubed operations, because you're basically computing n squared dot products. OK? So essentially, if you add up the total number of operations, it's about 2n cubed because there is essentially a multiply and an add for every pair of terms that need to be accumulated. OK? So it's basically 2n cubed. We're going to look at it assuming for simplicity that our n is an exact power of 2, OK?

Now, the machine that we're going to look at is going to be one of the ones that you'll have access to in AWS. OK? It's a compute-optimized machine, which has a Haswell microachitecture running at 2.9 gigahertz. There are 2 processor chips for each of these machines and 9 processing cores per chip, so a total of 18 cores. So that's the amount of parallel processing.

It does two-way hyperthreading, which we're actually going to not deal a lot with. Hyperthreading gives you a little bit more performance, but it also makes it really hard to measure things, so generally we will turn off hyperthreading. But the performance that you get tends to be correlated with what you get when you hyperthread. For floating-point unit there, it is capable of doing 8 double-precision precision operations. That's 64-bit floating-point operations, including a fused-multiply-add per core, per cycle. OK?

So that's a vector unit. So basically, each of these 18 cores can do 8 double-precision operations, including a fused-multiply-add, which is actually 2 operations. OK? The way that

they count these things, OK? It has a cache-line size of 64 bytes. The icache is 32 kilobytes, which is 8-way set associative. We'll talk about some of these things. If you don't know all the terms, it's OK. We're going to cover most of these terms later on.

It's got a dcache of the same size. It's got an L2-cache of 256 kilobytes, and it's got an L3-cache or what's sometimes called an LLC, Last-Level Cache, of 25 megabytes. And then it's got 60 gigabytes of DRAM. So this is a honking big machine. OK? This is like-- you can get things to sing on this, OK?

If you look at the peak performance, it's the clock speed times 2 processor chips times 9 processing cores per chip, each capable of, if you can use both the multiply and the add, 16 floating-point operations, and that goes out to just short of teraflop, OK? 836 gigaflops. So that's a lot of power, OK? That's a lot of power. These are fun machines, actually, OK?

Especially when we get into things like the game-playing AI and stuff that we do for the fourth project. You'll see. They're really fun. You can have a lot of compute, OK? Now here's the basic code. This is the full code for Python for doing matrix multiplication. Now, generally, in Python, you wouldn't use this code because you just call a library subroutine that does matrix multiplication.

But sometimes you have a problem. I'm going to illustrate with matrix multiplication, but sometimes you have a problem for which you have to write the code. And I want to give you an idea of what kind of performance you get out of Python, OK? In addition, somebody has to write-- if there is a library routine, somebody had to write it, and that person was a performance engineer, because they wrote it to be as fast as possible. And so this will give you an idea of what you can do to make code run fast, OK?

So when you run this code-- so you can see that the start time-- you know, before the triply-nested loop-- right here, before the triply-nested loop, we take a time measurement, and then we take another time measurement at the end, and then we print the difference. And then that's just this classic triply-nested loop for matrix multiplication. And so, when you run this, how long is this run for, you think? Any guesses?

Let's see. How about-- let's do this. It runs for 6 microseconds. Who thinks 6 microseconds? How about 6 milliseconds? How about-- 6 milliseconds. How about 6 seconds? How about 6 minutes? OK. How about 6 hours?

[LAUGHTER]

How about 6 days?

[LAUGHTER]

OK. Of course, it's important to know what size it is. It's 4,096 by 4,096, as it shows in the code, OK? And those of you who didn't vote-- wake up. Let's get active. This is active learning. Put yourself out there, OK? It doesn't matter whether you're right or wrong. There'll be a bunch of people who got the right answer, but they have no idea why.

[LAUGHTER]

OK? So it turns out, it takes about 21,000 seconds, which is about 6 hours. OK? Amazing. Is this fast?

AUDIENCE: (SARCASTICALLY) Yeah.

[LAUGHTER]

CHARLES LEISERSON: Yeah, right. Duh, right? Is this fast? No. You know, how do we tell whether this is fast or not? OK? You know, what should we expect from our machine? So let's do a back-of-the-envelope calculation of--

[LAUGHTER]

--of how many operations there are and how fast we ought to be able to do it. We just went through and said what are all the parameters of the machine. So there are 2n cubed operations that need to be performed. We're not doing Strassen's algorithm or anything like that. We're just doing a straight triply-nested loop.

So that's 2 to the 37 floating point operations, OK? The running time is 21,000 seconds, so that says that we're getting about 6.25 megaflops out of our machine when we run that code, OK? Just by dividing it out, how many floating-point operations per second do we get? We take the number of operations divided by the time, OK?

The peak, as you recall, was about 836 gigaflops, OK? And we're getting 6.25 megaflops, OK? So we're getting about 0.00075% of peak, OK? This is not fast. OK? This is not fast. So let's do something really simple. Let's code it in Java rather than Python, OK? So we take just that

loop. The code is almost the same, OK?

Just the triply-nested loop. We run it in Java, OK? And the running time now, it turns out, is about just under 3,000 seconds, which is about 46 minutes. The same code. Python, Java, OK? We got almost a 9 times speedup just simply coding it in a different language, OK? Well, let's try C. That's the language we're going to be using here.

What happens when you code it in C? It's exactly the same thing, OK? We're going to use the Clang/LLVM 5.0 compiler. I believe we're using 6.0 this term, is that right? Yeah. OK, I should have rerun these numbers for 6.0, but I didn't. So now, it's basically 1,100 seconds, which is about 19 minutes, so we got, then, about-- it's twice as fast as Java and about 18 times faster than Python, OK?

So here's where we stand so far. OK? We have the running time of these various things, OK? And the relative speedup is how much faster it is than the previous row, and the absolute speedup is how it is compared to the first row, and now we're managing to get, now, 0.014% of peak. So we're still slow, but before we go and try to optimize it further-- like, why is Python so slow and C so fast? Does anybody know?

AUDIENCE: Python is interpreted, so it has to-- so there's a C program that basically parses Python pycode instructions, which takes up most of the time.

CHARLES LEISERSON: OK. That's kind of on the right track. Anybody else have any-- articulate a little bit why Python is so slow? Yeah?

AUDIENCE: When you write, like, multiplying and add, those aren't the only instructions Python's doing. It's doing lots of code for, like, going through Python objects and integers and blah-blah-blah.

CHARLES LEISERSON: Yeah, yeah. OK, good. So the big reason why Python is slow and C is so fast is that Python is interpreted and C is compiled directly to machine code. And Java is somewhere in the middle because Java is compiled to bytecode, which is then interpreted and then just-in-time compiled into machine codes. So let me talk a little bit about these things.

So interpreters, such as in Python, are versatile but slow. It's one of these things where they said, we're going to take some of our performance and use it to make a more flexible, easier-to-program environment. OK? The interpreter basically reads, interprets, and performs each program statement and then updates the machine state. So it's not just-- it's actually going through and, each time, reading your code, figuring out what it does, and then implementing it.

So there's like all this overhead compared to just doing its operations. So interpreters can easily support high-level programming features, and they can do things like dynamic code alteration and so forth at the cost of performance. So that, you know, typically the cycle for an interpreter is, you read the next statement, you interpret the statement. You then perform the statement, and then you update the state of the machine, and then you fetch the next instruction. OK?

And you're going through that each time, and that's done in software. OK? When you have things compiled to machine code, it goes through a similar thing, but it's highly optimized just for the things that machines are done. OK? And so when you compile, you're able to take advantage of the hardware and interpreter of machine instructions, and that's much, much lower overhead than the big software overhead you get with Python.

Now, JIT is somewhere in the middle, what's used in Java. JIT compilers can recover some of the performance, and in fact it did a pretty good job in this case. The idea is, when the code is first executed, it's interpreted, and then the runtime system keeps track of how often the various pieces of code are executed. And whenever it finds that there's some piece of code that it's executing frequently, it then calls the compiler to compile that piece of code, and then subsequent to that, it runs the compiled code.

So it tries to get the big advantage of performance by only compiling the things that are necessary-- you know, for which it's actually going to pay off to invoke the compiler to do. OK? So anyway, so that's the big difference with those kinds of things. One of the reasons we don't use Python in this class is because the performance model is hard to figure out.

OK? C is much closer to the metal, much closer to the silicon, OK? And so it's much easier to figure out what's going on in that context. OK? But we will have a guest lecture that we're going to talk about performance in managed languages like Python, so it's not that we're going to ignore the topic. But we will learn how to do performance engineering in a place where it's easier to do it. OK?

Now, one of the things that a good compiler will do is-- you know, once you get to-- let's say we have the C version, which is where we're going to move from this point because that's the fastest we got so far-- is, it turns out that you can change the order of loops in this program without affecting the correctness. OK? So here we went-- you know, for i, for j, for k, do the

update. OK?

We could otherwise do-- we could do, for i, for k, for j, do the update, and it computes exactly the same thing. Or we could do, for k, for j, for i, do the updates. OK? So we can change the order without affecting the correctness, OK?

And so do you think the order of loops matters for performance? Duh. You know, this is like this leading question. Yeah? Question?

AUDIENCE: Maybe for cache localities.

CHARLES LEISERSON: Yeah. OK. And you're exactly right. Cache locality is what it is. So when we do that, we get-- the loop order affects the running time by a factor of 18. Whoa. Just by switching the order. OK? What's going on there? OK? What's going on?

So we're going to talk about this in more depth, so I'm just going to fly through this, because this is just sort of showing you the kinds of considerations that you do. So in hardware, each processor reads and writes main memory in contiguous blocks called cache lines. OK? Previously accessed cache lines are stored in a small memory called cache that sits near the processor.

When the processor accesses something, if it's in the cache, you get a hit. That's very cheap, OK, and fast. If you miss, you have to go out to either a deeper level cache or all the way out to main memory. That is much, much slower, and we'll talk about that kind of thing. So what happens for this matrix problem is, the matrices are laid out in memory in row-major order. That means you take-- you know, you have a two-dimensional matrix. It's laid out in the linear order of the addresses of memory by essentially taking row 1, and then, after row 1, stick row 2, and after that, stick row 3, and so forth, and unfolding it.

There's another order that things could have been laid out-- in fact, they are in Fortran-- which is called column-major order. OK? So it turns out C and Fortran operate in different orders. OK? And it turns out it affects performance, which way it does it. So let's just take a look at the access pattern for order i, j, k. OK? So what we're doing is, once we figure out what i and what j is, we're going to go through and cycle through k. And as we cycle through k, OK, C i, j stays the same for everything.

We get for that excellent spatial locality because we're just accessing the same location. Every single time, it's going to be in cache. It's always going to be there. It's going to be fast to

access C. For A, what happens is, we go through in a linear order, and we get good spatial locality. But for B, it's going through columns, and those points are distributed far away in memory, so the processor is going to be bringing in 64 bytes to operate on a particular datum. OK?

And then it's ignoring 7 of the 8 floating-point words on that cache line and going to the next one. So it's wasting an awful lot, OK? So this one has good spatial locality in that it's all adjacent and you would use the cache lines effectively. This one, you're going 4,096 elements apart. It's got poor spatial locality, OK? And that's for this one.

So then if we look at the different other ones-- this one, the order i, k, j-- it turns out you get good spatial locality for both C and B and excellent for A. OK? And if you look at even another one, you don't get nearly as good as the other ones, so there's a whole range of things. OK? This one, you're doing optimally badly in both, OK? And so you can just measure the different ones, and it turns out that you can use a tool to figure this out.

And the tool that we'll be using is called Cachegrind. And it's one of the Valgrind suites of caches. And what it'll do is, it will tell you what the miss rates are for the various pieces of code. OK? And you'll learn how to use that tool and figure out, oh, look at that. We have a high miss rate for some and not for others, so that may be why my code is running slowly. OK?

So when you pick the best one of those, OK, we then got a relative speedup of about 6 and 1/2. So what other simple changes can we try? There's actually a collection of things that we could do that don't even have us touching the code. What else could we do, for people who've played with compilers and such? Hint, hint. Yeah?

AUDIENCE: You could change the compiler flags.

CHARLES LEISERSON: Yeah, change the compiler flags, OK? So Clang, which is the compiler that we'll be using, provides a collection of optimization switches, and you can specify a switch to the compiler to ask it to optimize. So you do minus O, and then a number. And 0, if you look at the documentation, it says, "Do not optimize." 1 says, "Optimize." 2 says, "Optimize even more." 3 says, "Optimize yet more." OK?

In this case, it turns out that even though it optimized more in O3, it turns out O2 was a better setting. OK? This is one of these cases. It doesn't happen all the time. Usually, O3 does better than O2, but in this case O2 actually optimized better than O3, because the optimizations are

to some extent heuristic. OK? And there are also other kinds of optimization. You can have it do profile-guided optimization, where you look at what the performance was and feed that back into the code, and then the compiler can be smarter about how it optimizes.

And there are a variety of other things. So with this simple technology, choosing a good optimization flag-- in this case, O2-- we got for free, basically, a factor of 3.25, OK? Without having to do much work at all, OK? And now we're actually starting to approach 1% of peak performance. We've got point 0.3% of peak performance, OK?

So what's causing the low performance? Why aren't we getting most of the performance out of this machine? Why do you think? Yeah?

AUDIENCE: We're not using all the cores.

CHARLES LEISERSON: Yeah, we're not using all the cores. So far we're using just one core, and how many cores do we have?

AUDIENCE: 18.

CHARLES LEISERSON: 18, right? 18 cores. Ah! 18 cores just sitting there, 17 sitting idle, while we are trying to optimize one. OK. So multicore. So we have 9 cores per chip, and there are 2 of these chips in our test machine. So we're running on just one of them, so let's use them all. To do that, we're going to use the Cilk infrastructure, and in particular, we can use what's called a parallel loop, which in Cilk, you'd call cilk_for, and so you just relay that outer loop-- for example, in this case, you say cilk_for, it says, do all those iterations in parallel.

The compiler and runtime system are free to schedule them and so forth. OK? And we could also do it for the inner loop, OK? And it turns out you can't also do it for the middle loop, if you think about it. OK? So I'll let you do that is a little bit of a homework problem-- why can't I just do a cilk_for of the inner loop? OK? So the question is, which parallel version works best?

So we can parallel the i loop, we can parallel the j loop, and we can do i and j together. You can't do k just with a parallel loop and expect to get the right thing. OK? And that's this one. So if you look-- wow! What a spread of running times, right? OK? If I parallelize the just the i loop, it's 3.18 seconds, and if I parallelize the j loop, it actually slows down, I think, right? And then, if I do both i and j, it's still bad. I just want to do the outer loop there.

This has to do, it turns out, with scheduling overhead, and we'll learn about scheduling

overhead and how you predict that and such. So the rule of thumb here is, parallelize outer loops rather than inner loops, OK? And so when we do parallel loops, we get an almost 18x speedup on 18 cores, OK? So let me assure you, not all code is that easy to parallelize. OK? But this one happens to be.

So now we're up to, what, about just over 5% of peak. OK? So where are we losing time here? OK, why are we getting just 5%? Yeah?

AUDIENCE: So another area of the parallelism that [INAUDIBLE]. So we could, for example, vectorize the multiplication.

CHARLES LEISERSON: Yep. Good. So that's one, and there's one other that we're not using very effectively. OK. That's one, and those are the two optimizations we're going to do to get a really good code here. So what's the other one? Yeah?

AUDIENCE: The multiply and add operation.

CHARLES LEISERSON: That's actually related to the same question, OK? But there's another completely different source of opportunity here. Yeah?

AUDIENCE: We could also do a lot better on our handling of cache misses.

CHARLES LEISERSON: Yeah. OK, we can actually manage the cache misses better. OK? So let's go back to hardware caches, and let's restructure the computation to reuse data in the cache as much as possible. Because cache misses are slow, and hits are fast. And try to make the most of the cache by reusing the data that's already there. OK? So let's just take a look.

Suppose that we're going to just compute one row of C, OK? So we go through one row of C. That's going to take us-- since it's a 4,096-long vector there, that's going to basically be 4,096 writes that we're going to do. OK? And we're going to get some spatial locality there, which is good, but we're basically doing-- the processor's doing 4,096 writes. Now, to compute that row, I need to access 4,096 reads from A, OK? And I need all of B, OK?

Because I go each column of B as I'm going through to fully compute C. Do people see that? OK. So I need-- to just compute one row of C, I need to access one row of A and all of B. OK? Because the first element of C needs the whole first column of B. The second element of C needs the whole second column of B. Once again, don't worry if you don't fully understand this, because right now I'm just ripping through this at high speed. We're going to go into this

in much more depth in the class, and there'll be plenty of time to master this stuff.

But the main thing to understand is, you're going through all of B, then I want to compute another row of C. I'm going to do the same thing. I'm going to go through one row of A and all of B again, so that when I'm done we do about 16 million, 17 million memory accesses total. OK? That's a lot of memory access.

So what if, instead of doing that, I do things in blocks, OK? So what if I want to compute a 64 by 64 block of C rather than a row of C? So let's take a look at what happens there. So remember, by the way, this number-- 16, 17 million, OK? Because we're going to compare with it. OK?

So what about to compute a block? So if I look at a block, that is going to take-- 64 by 64 also takes 4,096 writes to C. Same number, OK? But now I have to do about 200,000 reads from A because I need to access all those rows. OK? And then for B, I need to access 64 columns of B, OK? And that's another 262,000 reads from B, OK? Which ends up being half a million memory accesses total. OK?

So I end up doing way fewer accesses, OK, if those blocks will fit in my cache. OK? So I do much less to compute the same size footprint if I compute a block rather than computing a row. OK? Much more efficient. OK? And that's a scheme called tiling, and so if you do tiled matrix multiplication, what you do is you bust your matrices into, let's say, 64 by 64 submatrices, and then you do two levels of matrix multiply.

You do an outer level of multiplying of the blocks using the same algorithm, and then when you hit the inner, to do a 64 by 64 matrix multiply, I then do another three-nested loops. So you end up with 6 nested loops. OK? And so you're basically busting it like this. And there's a tuning parameter, of course, which is, you know, how big do I make my tile size?

You know, if it's s by s, what should I do at the leaves there? Should it be 64? Should it be 128? What number should I use there? How do we find the right value of s, this tuning parameter? OK? Ideas of how we might find it?

AUDIENCE: You could figure out how much there is in the cache.

CHARLES LEISERSON: You could do that. You might get a number, but who knows what else is going on in the cache while you're doing this.

AUDIENCE: Just test a bunch of them.

CHARLES LEISERSON: Yeah, test a bunch of them. Experiment! OK? Try them! See which one gives you good numbers. And when you do that, it turns out that 32 gives you the best performance, OK, for this particular problem. OK? So you can block it, and then you can get faster, and when you do that, that gave us a speedup of about 1.7.

OK? So we're now up to-- what? We're almost 10% of peak, OK? And the other thing is that if you use Cachegrind or a similar tool, you can figure out how many cache references there are and so forth, and you can see that, in fact, it's dropped quite considerably when you do the tiling versus just the straight parallel loops. OK? So once again, you can use tools to help you figure this out and to understand the cause of what's going on.

Well, it turns out that our chips don't have just one cache. They've got three levels of caches. OK? There's L1-cache, OK? And there's data and instructions, so we're thinking about data here, for the data for the matrix. And it's got an L2-cache, which is also private to the processor, and then a shared L3-cache, and then you go out to the DRAM-- you also can go to your neighboring processors and such. OK?

And they're of different sizes. You can see they grow in size-- 32 kilobytes, 256 kilobytes, to 25 megabytes, to main memory, which is 60 gigabytes. So what you can do is, if you want to do two-level tiling, OK, you can have two tuning parameters, s and t. And now you get to do-- you can't do binary search to find it, unfortunately, because it's multi-dimensional.

You kind of have to do it exhaustively. And when you do that, you end up with--

[LAUGHTER]

--with 9 nested loops, OK? But of course, we don't really want to do that. We have three levels of caching, OK? Can anybody figure out the inductive number? For three levels of caching, how many levels of tiling do we have to do? This is a gimme, right?

AUDIENCE: 12.

CHARLES LEISERSON: 12! Good, 12, OK? Yeah, you then do 12. And man, you know, when I say the code gets ugly when you start making things go fast. OK? Right? This is like, ughhh!

[LAUGHTER]

OK? OK, but it turns out there's a trick. You can tile for every power of 2 simultaneously by just solving the problem recursively. So the idea is that you do divide and conquer. You divide each of the matrices into 4 submatrices, OK? And then, if you look at the calculations you need to do, you have to solve 8 subproblems of half the size, and then do an addition. OK? And so you have 8 multiplications of size n over 2 by n over 2 and 1 addition of n by n matrices, and that gives you your answer.

But then, of course, what you going to do is solve each of those recursively, OK? And that's going to give you, essentially, the same type of performance. Here's the code. I don't expect that you understand this, but we've written this using in parallel, because it turns out you can do 4 of them in parallel. And the Cilk spawn here says, go and do this subroutine, which is basically a subproblem, and then, while you're doing that, you're allowed to go and execute the next statement-- which you'll do another spawn and another spawn and finally this.

And then this statement says, ah, but don't start the next phase until you finish the first phase. OK? And we'll learn about this stuff. OK, when we do that, we get a running time of about 93 seconds, which is about 50 times slower than the last version. We're using cache much better, but it turns out, you know, nothing is free, nothing is easy, typically, in performance engineering. You have to be clever.

What happened here? Why did this get worse, even though-- it turns out, if you actually look at the caching numbers, you're getting great hits on cache. I mean, very few cache misses, lots of hits on cache, but we're still slower. Why do you suppose that is? Let me get someone-- yeah?

AUDIENCE: Overhead to start functions and [INAUDIBLE].

CHARLES LEISERSON: Yeah, the overhead to the start of the function, and in particular the place that it matters is at the leaves of the computation. OK? So what we do is, we have a very small base case. We're doing this overhead all the way down to n equals 1. So there's a function call overhead even when you're multiplying 1 by 1. So hey, let's pick a threshold, and below that threshold, let's just use a standard good algorithm for that threshold. And if we're above that, we'll do divide and conquer, OK?

So what we do is we call-- if we're less than the threshold, we call a base case, and the base

case looks very much like just ordinary matrix multiply. OK? And so, when you do that, you can once again look to see what's the best value for the base case, and it turns out in this case, I guess, it's 64. OK? We get down to 1.95 seconds.

I didn't do the base case of 1, because I tried that, and that was the one that gave us terrible performance.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Sorry. 32-- oh, yeah. 32 is even better. 1.3. Good, yeah, so we picked 32. I think I even-- oh, I didn't highlight it. OK. I should have highlighted that on the slide. So then, when we do that, we now are getting 12% of peak. OK?

And if you count up how many cache misses we have, you can see that-- here's the data cache for L1, and with parallel divide and conquer it's the lowest, but also now so is the last-level caching. OK? And then the total number of references is small, as well. So divide and conquer turns out to be a big win here. OK?

Now the other thing that we mentioned, which was we're not using the vector hardware. All of these things have vectors that we can operate on, OK? They have vector hardware that process data in what's called SIMD fashion, which means Single-Instruction stream, Multiple-Data. That means you give one instruction, and it does operations on a vector, OK? And as we mentioned, we have 8 floating-point units per core, of which we can also do a fused-multiply-add, OK?

So each vector register holds multiple words. I believe in the machine we're using this term, it's 4 words. I think so. OK? But it's important when you use these-- you can't just use them willy-nilly. You have to operate on the data as one chunk of vector data. You can't, you know, have this lane of the vector unit doing one thing and a different lane doing something else. They all have to be doing essentially the same thing, the only difference being the indexing of memory. OK?

So when you do that-- so already we've actually been taking advantage of it. But you can produce a vectorization report by asking for that, and the system will tell you what kinds of things are being vectorized, which things are being vectorized, which aren't. And we'll talk about how you vectorize things that the compiler doesn't want to vectorize. OK? And in particular, most machines don't support the newest sets of vector instructions, so the compiler

uses vector instructions conservatively by default.

So if you're compiling for a particular machine, you can say, use that particular machine. And here's some of the vectorization flags. You can say, use the AVX instructions if you have AVX. You can use AVX2. You can use the fused-multiply-add vector instructions. You can give a string that tells you the architecture that you're running on, on that special thing. And you can say, well, use whatever machine I'm currently compiling on, OK, and it'll figure out which architecture is that. OK?

Now, floating-point numbers, as we'll talk about, turn out to have some undesirable properties, like they're not associative, so if you do A times B times C, how you parenthesize that can give you two different numbers. And so if you give a specification of a code, typically, the compiler will not change the order of associativity because it says, I want to get exactly the same result.

But you can give it a flag called fast math, minus ffast-math, which will allow it to do that kind of reordering. OK? If it's not important to you, then it will be the same as the default ordering, OK? And when you use that-- and particularly using architecture native and fast math, we actually get about double the performance out of vectorization, just having the compiler vectorize it. OK? Yeah, question.

AUDIENCE: Are the data types in our matrix, are they 32-bit?

CHARLES LEISERSON: They're 64-bit. Yep. These days, 64-bit is pretty standard. They call that double precision, but it's pretty standard. Unless you're doing AI applications, in which case you may want to do lower-precision arithmetic.

AUDIENCE: So float and double are both the same?

CHARLES LEISERSON: No, float is 32, OK? So generally, people who are doing serious linear algebra calculations use 64 bits. But actually sometimes they can use less, and then you can get more performance if you discover you can use fewer bits in your representation. We'll talk about that, too. OK?

So last thing that we're going to do is, you can actually use the instructions, the vector instructions, yourself rather than rely on the compiler to do it. And there's a whole manual of intrinsic instructions that you can call from C that allow you to do, you know, the specific vector instructions that you might want to do it. So the compiler doesn't have to figure that out.

And you can also use some more insights to do things like-- you can do preprocessing, and

you can transpose the matrices, which turns out to help, and do data alignment. And there's a lot of other things using clever algorithms for the base case, OK? And you do more performance engineering. You think about you're doing, you code, and then you run, run, run to test, and that's one nice reason to have the cloud, because you can do tests in parallel.

So it takes you less time to do your tests in terms of your, you know, sitting around time when you're doing something. You say, oh, I want to do 10 tests. Let's spin up 10 machines and do all the tests at the same time. When you do that-- and the main one we're getting out of this is the AVX intrinsics-- we get up to 0.41 of peak, so 41% of peak, and get about 50,000 speedup. OK?

And it turns out that's where we quit. OK? And the reason is because we beat Intel's professionally engineered Math Kernel Library at that point.

[LAUGHTER]

OK? You know, a good question is, why aren't we getting all of peak? And you know, I invite you to try to figure that out, OK? It turns out, though, Intel MKL is better than what we did because we assumed it was a power of 2. Intel doesn't assume that it's a power of 2, and they're more robust, although we win on the 496 by 496 matrices, they win on other sizes of matrices, so it's not all things.

But the end of the story is, what have we done? We have just done a factor of 50,000, OK? If you looked at the gas economy, OK, of a jumbo jet and getting the kind of performance that we just got in terms of miles per gallon, you would be able to run a jumbo jet on a little Vespa scooter or whatever type of scooter that is, OK? That's how much we've been able to do it.

You generally-- let me just caution you-- won't see the magnitude of a performance improvement that we obtained for matrix multiplication. OK? That turns out to be one where-- it's a really good example because it's so dramatic. But we will see some substantial numbers, and in particular in 6.172 you'll learn how to print this currency of performance all by yourself so that you don't have to take somebody else's library. You can, you know, say, oh, no, I'm an engineer of that.

Let me mention one other thing. In this course, we're going to focus on multicore computing. We are not, in particular, going to be doing GPUs or file systems or network performance. In the real world, those are hugely important, OK? What we found, however, is that it's better to

learn a particular domain, and in particular, this particular domain-- people who master multicore performance engineering, in fact, go on to do these other things and are really good at it, OK? Because you've learned the sort of the core, the basis, the foundation--