PROFESSOR: So it's my pleasure to introduce Professor Saman Amarasignhe as our guest lecturer today. So Saman Amarasignhe is a professor in the EECS Department at MIT. And he's also the associate department head. He's an expert in compilers, domain specific languages, and autotuning. In fact, he was the designer of the OpenTuner framework that you've been using for your homework assignments.

So today Saman is going to tell us about some of his recent work on domain specific languages and also on autotuning. So let's give Saman Amarasignhe a round of applause.

[APPLAUSE]

SAMAN AMARASIGNHE: Thank you. OK, so I used to teach this class for many, many years. Unfortunately, now I am administrator, so I don't-- Julian and Charles get to have the fun teaching the class. So hopefully you guys enjoyed your-- now you're starting-- you are done project one, two, and three and going into project four? Yeah, project four is really fun and--

[LAUGHTER]

It will look big and daunting, but at the end, you'll enjoy it, and especially all the amount of time people spend working on it. So I think I'm making you scared here, more than anything. OK, so let's get into the talk today. So I will talk to you about domain specific languages and a little bit about autotuning, how it leads to autotuning.

So why is domain specific languages? So we are all used to general purpose languages that we use all day. And those languages are set up to capture a very large sort of what people might want to do in programming.

However, a lot of times there are specific areas, specific domains, either some area in-- that you want to implement, or the certain patterns you want to implement code that has a lot of interesting properties that in a general purpose language it's very hard to describe. And a lot of times it's basically very hard, especially from compiler point of view, to take advantage.

Because it has to work for everybody.

So domain specific languages basically has this lot of integrated benefits. Because if you know that you are-- what you're building has a certain shape, certain set of properties, if the language captured this, and if you're building on that, it could be much easier to build. It should have a lot of clarity. It's very easy to maintain that kind of thing. It's very easy to test.

And also, the other thing is, it's very easy to understand. Because the domain is very clearly described. If-- you can build a library, but somebody can go and do weird things in library. If it is built into the language, it's set in stone. You can't go and say, oh, yeah, I'm going to change something here. Let me do some weird thing here. It's built into the language. So it stays there. It makes it much easier for programmers [INAUDIBLE].

But from my point of view, the domain specific language I really like are the languages where I know I can take advantage of knowledge of the domain experts to get really good performance. So a lot of times, domain experts say, ah ha, in this domain, I can do-- OK, there's some linear algebra, but I know this kind of algebra that I can do to simplify the expression.

That algebra might only work on that domain. It's very hard to put some complex algebra into C++ or C. But in that domain, I can say, ha, I can call it up. So you can write any expression that I can simplify it.

And also, there are a lot of idioms in each domain. So some domain might say, OK, look, I am going to represent a graph that I'm going to talk about. In the normal C++, you create a bunch of classes. You do these very complicated things. The idiom is hidden in there.

First of all, C++ doesn't know that I had to look for graphs. But even if you had look for graphs, you can try graphs in hundreds of millions of ways. But if it is a first class supporting the language, I don't have to work heroically to extract that. It's there. I can easily see that. So most of my compiler can be doing useful things in there.

And most of the time, the other thing is, if you build a domain specific language right, you can leave the complex, the lower level decision, to the compiler. And if you-- C++, you might be tempted to say, eh, I know some optimization. Let me do something here. Oh, let me do some of the optimizations here.

So I have been working on optimization all my life. And a lot of times, when you write a compile optimization part, you spend half of or more than half of your time undoing the crazy optimization the programmer did, like you guys are learning.

So this-- you think you know better. You go do something. And that might work well then, but believe me, that code survives 20 years later. And 20 years later, that looks like a really stupid thing to do. And then you look at it and say, OK, now I had to undo everything in the compiler, do the right thing in the current architecture.

And because of that, if you capture the right level, I will let the compiler do the work in here. And then as the architectures keep maturing, as the problems keep changing, I don't have to worry. I don't have to undo these parts in here.

So again, I'm coming to the performance engineering class and telling you guys, leave the performance to the compiler. But that's the nice thing, that if the compiler can do the most of your work, much nicer job. So don't doubt the compiler.

So I'm going to talk about three parts in here. One is three different programming languages in here, domain specific languages, GraphIt, Halide, and then OpenTuner, which is not just the language, but the framework in here. And between GraphIt and Halide, you will see some patterns. And then we'll see whether you found the pattern that we are working on in here.

So GraphIt, this is a product that I worked with Julian. So if you have any questions of GraphIt after today, you can definitely go ask Julian. He knows probably more about graphs and GraphIt-- more about graphs than probably anybody on this planet. So he's a good resource to talk about graphs.

So talking about graphs, graphs everywhere. So if you go to something like Google and do some search, Google has represented the entire knowledge on the internet as a big graph. They have done a huge amount of graph processing behind you. That is how-- what guides your search in there.

Or if you go, again, maps, or something like Uber, it will find your directions. The entire road network is a graph. And it's trying to find things like shortest path in this graph to give you the map.

And if you go to a recommendation engine to get a recommendation for a movie, if you get a really cool movie you like, that's because there's a huge graph between everybody who's--

which movie they've watched, and their likings to those movies. And they are looking and comparing you to them and recommending that. That's all can be viewed as graphs in here.

And even if you go to an ATM and try to do a transaction, there's a very fast graph analysis back to say is this a fraudulent transaction or not? So most of the transactions people have done, all the connectivities in the back in there, before the time that actually the money pops out of your ATM machine, it has done a bunch of graph processes to understand, OK, this seems like a good transaction. So I will actually give you the money.

Sometimes you say-- you get this other message that mean the graph processing decided there might be some weird thing going on there. So a lot of these things that some of them, like maps and graphs has-- maps and these transactions have very fine latency thing in there. You have to get this thing done right. You have to get good directions. Especially if you take a wrong turn, you need to get the next set of directions done very fast before you go hit some Boston bad weird traffic. So these things have to work fast.

And other things, like recommendations and Google Search, is huge graph. They build the entire web, then all the recommendations have to do a huge amount of processing. So performance matters a lot in these applications.

So let me dive down a little bit deeper into show what graphs means, what graph processing means. So one of the very well known graph algorithms is called PageRank. Anybody knows [INAUDIBLE] page? How many have heard of PageRank?

OK, what does page stand in page rank?

AUDIENCE: Larry Page.

SAMAN AMARASIGNHE: Larry Page. So the first algorithm Google did-- I don't think this is anywhere near Google at this point-- was this algorithm, PageRank. So it ranked these pages. But it was developed by Larry Page. So it depends on either page-- it's web pages or Larry Page. We don't know. But people think it's Larry Page is PageRank.

So you have a graph in here. So this graph algorithm, what it does it is [INAUDIBLE] to some iterations, either it's max_iter on to some convergence in here. So what it first do is it will go around, look at all its neighbors, and calculate, basically rank a new rank out of all my neighbors. So that means what is-- how good are my neighbors? What's their rank? And

what's their contribution to me?

So it means being known to a good person and having a connection to a very well known-- in this case, a super web page-- means I am highly ranked. So I am more influential, because I'm closer to something in there.

So what it does is, basically, it will go-- each node calculates some value, and propagate to all the neighbors, and aggregating. So entire graph participating in that. And then, what happens is each node will go about calculating its new rank in there. From looking at old rank, it get modified a little bit towards a new rank. And then they swap old ranks and new ranks.

So this is the two computations that you iterate over that. And you have to do for the entire graph in here. So, of course, you can run this. This will run very, very slowly if you run this.

So if you want to get performance, you write this piece of code. So this piece of code, basically, is huge. And it runs 23 times faster than what's in the previous graph in here on a 12 core machine. It basically had multi-threaded so we'd get parallel performance. It is load balanced because, as you know, graphs are very unbalanced. So you get load balance. If you have non-uniform memory access machines, things like multiple socket machines, it will take advantage of that. It advantage of caches-- a lot of things happening in this piece of code.

But, of course, you know this is hard to write this piece of code. But also, worse, you might not know what to do, what the right optimizing-- you might only iterate. You might try many things. And this is very hard. Every time you change something, if you say, ah, I want to do something a little bit different, that I had to write a very complicated piece of code, get it all right, get everything working before I test in here. So this is why we can use a DSL for this one.

So let me go a little bit, talk about graph algorithms and say this seems like a new set of [INAUDIBLE]. So what do people do with graphs? So when they say graph algorithms, I'm going to go a little bit deep down to show you what type of things represent these graphs.

There's one class of graph algorithms that are called topology-driven algorithms. That means the entire graph participates in the computation.

For example, Google Search-- before you do Google Search, it will do the entire basic collection of all the web links. It will build this huge graph and do huge amount of processing to basically able to do the search in here. Recommendation engine-- so every, probably few weeks, or whatever it is, it will collect everybody's recommendations, have this huge data, and

you're going to process that and do this recommendation engine. So this is applied for the entire graphs, and sometimes billions or trillions of nodes have to go into this computation.

Another set of algorithms is called data-driven algorithms. So what that means is you start with certain nodes. And then you keep going to its neighbors and its neighbors processing data in here as we do that.

And the kind of algorithms that fit in this category are things like if you have a map, if I had to find the shortest path, that means I have, probably two paths. I don't have to get in from direction from here to Boston. I don't have to go through New York nodes in New York. I just have to go through my neighbors connected to me. So I am basically operating on a certain area with some connections and processing that.

So these are data-driven algorithms. So I might have a huge graph. But my computation might only work on a small region or a small part of the graph in these algorithms.

So when you traversing through a graph doing that, there are multiple ways of doing graph traversals. And this is why optimization is hard. Because there are many different ways of doing things. And each has different set of outcomes you can get.

So I see a lot of graph algorithms. I need to get something from my neighbors. One way to get something my neighbors is I can calculate what the neighbor-- all my neighbors might want and give it to all the neighbors. Or I can go change all the neighbors to update my value.

So why do you think-- OK, you have done some programming in here. What do you think about this? Is this a good way? So if I want to update everybody, I will calculate what I will do, and I'll go change everybody, all my neighbors.

AUDIENCE: This is not as parallel as it could be.

SAMAN AMARASIGNHE: Not as parallel as it could be. I think you are getting to a point. But why is not that parallel?

AUDIENCE: Well, if you're doing the same thing with the neighbors, you might as well just tell your neighbors to do the work for you.

SAMAN AMARASIGNHE: Yeah, but-- that's a very good point. So if you are doing a data-driven, if I'm doing, that's not good. But if everybody is doing that to their neighbor, so then I have parallelism. So everybody

might say, you are updated your neighbor. You are updating your-- everybody is updating their neighbors. So now there's another problem showing up. What's the problem if everybody tried to update their neighbors? Back there.

AUDIENCE: There's a determinancy race.

SAMAN AMARASIGNHE: There's a race in there. Because everybody's going right in there. So if you want to get this actually right, you have a bunch of issues here. You want to basically do atomic updates. Because you need to lock that thing. So it has get atomically updated in here.

And this is nice. But I don't have to traverse anything. Because everybody I need to update, I actually go and update it. That's nice way to do that, especially if it is not a global thing. So if I'm propagating, I will update my neighbors. And I can propagate that down.

Another way to do that is pull schedule. That means if I-- everybody look at-- ask their neighbors, OK, do you have-- what you have to-- give it to me. And I collect everything from my neighbors. And I update myself. So is there a race condition now?

How many people say there is a race? How many people think there's no race?

So what happens is I'm reading from all the neighbors. Everybody is reading from the neighbors. But I am only updating myself. So because of that, I'm the only one who's writing me. So I don't have a race. So it is really nice you don't have a race.

But I might-- if I'm doing a data-driven transformation, I might not know that I need to get updated. Because the update comes from that person. And that means I might be asking you, do you have anything to send? And you might say no. So in that sense, I might basically doing a lot of extra computation than necessary. Because I might not know that I have data I need to get. But I had to ask you whether I should do this. But I don't have any, basically, need to do any synchronization.

Another interesting thing is I can take this graph, and I can basically partition the graph. And once I partition the graph, I can basically say, OK, this core get this graph. This core get this graph. Or this processor node get this graph. What's the advantage of partitioning a graph? Why do I want to partition a graph-- large graph into small pieces?

Of course, you had to do a good partition. You can't do arbitrary partition. So what happens if I do a good partitioning? I don't tell the word, because then the answer comes out in there. OK,

let me see if anybody else, you have answered-- anybody else want to answer? Come on. You have to-- [INAUDIBLE].

What happened if I take apart, and find two different groups, and separate them, and give this one to one and this one to another? What do I get?

AUDIENCE: You get some parallelism.

SAMAN AMARASIGNHE: I get parallelism, also. But other thing, if I have a lot of connected things going to-- these connected things going to that person, what else can I get? Locality-- have you heard? Did you do locality in the class?

So that means-- the partition means my-- the thing I'm working, I am only working on a small amount. And that might, if I'm lucky, fit in my cache. And that would be very nice then everybody's has to go to every node in here.

So if I partition this properly, I will get good locality in here. It's actually written there, whoops. So my answer was in there, so improved locality. But, of course, now I might have a little bit extra overhead. Because I know I might have to replicate some nodes, stuff like that. Because it's in both sides.

So another interesting in properties of graphs is when you look at data structures until now, things like arrays, the size matters. These represent what array fits in the cache, and stuff like that. Graphs, there are some other properties of the graphs in here.

So if you go to social networks-- social network is a graph-- what's the interesting property in social networks you have observed?

AUDIENCE: Connectedness.

SAMAN AMARASIGNHE: Connectedness-- there are people like me that probably have 20 friends in there and has a very little number of connections. And then there are celebrities who have millions of connections in here.

So the interesting thing is, if you look at a social network graph, you have this relationship called power law relationship. That means-- there's exponential code. There are some people here, like very well-known celebrities that might have millions and millions of users in here-- connections in neighbors, or likes, or whatever it is in that node. And there are people like me

sitting here that has very little people connected to the rest of the world.

So this is normally-- people have observed these big social network type graphs-- you have this kind of exponential relationship in here. So the web has exponential relationship. A social network has this kind of relationship in there. So those things you have to do very interesting things when you process these graphs. Because there are certain connections that matter, certain nodes that matter a lot, or has a big impact than other nodes.

Then there are other graphs that have a bounded-degree distribution. If you have a road network, the maximum connection, probably you might have an intersection that has six roads coming to together in there. You don't have a million roads connecting into one place anywhere in there. So that doesn't happen. So this is a lot more flatter, a lot more bounded-degree distribution graphs in here.

They have lots of excellent locality in here because, of course, all the roads in Cambridge might be connected. But roads in Cambridge can be separated from roads in New York City. So there they are separated. They are locality-- nice locality in these kind of graphs. So even the-- if you say the graph be the same size, the shape of the graph matters in computation, a lot of times.

So what happens is now when you want to operate on these graphs, you have to look at three interesting properties. One property is, OK, how much parallelism my algorithm, what I'm trying to do to this graph is going to get?

It's like a Goldilocks type thing. You don't want too much parallelism. If you say, I have algorithm that huge amount of parallelism, if I can't take advantage, it's not useful. So you need to get a parallelism good enough that I can actually use it.

Then I really like to have locality. Because if I have a locality, my caches will work. Everything will be nearby. I can get-- runs things fast. If I, every time, I have to get something from main memory, it can be very, very slow. So I want to get locality.

But the interesting thing about graphs is to get localities and get some of these, you might have to do some extra work. So if you saw that graph got divided into two different graphs, I had to add extra nodes in here. I might write some extra data structures, so do some extra computation. So I might have to do some extra work in here.

So in certain things, I might not be that work efficient. So I might get really good parallelism

and locality, but I am doing too much work. So, for example, if I want to-- assume I want to find one node's neighbor, very way to get good parallelism, everybody finds their neighbor. OK, but that's not efficient. I mean, most of the computation's not useful. So there, you can do things that you are doing extra work than necessary. Then that can get much faster other things. But you have to be careful on doing that. So you have this balance in there.

So certain algorithms will fit in different places in here in this tradeoff space. So push algorithm will fit in here. So, for example, if you go to something like a pull algorithm, what you might find is you are doing less work efficient. Because you might do a little bit more work. But it might be better in locality and parallelism, because you don't have to do locks in here.

And then you do something like partitioning. It gets really good locality in partitioning. But you are doing extra work. And also, because in your partition, you might limit your parallelism in here. So you might less parallelism, but you get really good locality.

So all this is basically large tradeoff space in here. And then when you keep adding more and more things you can do, it fits into this big tradeoff space.

So how do you decide what to go in the tradeoff space is a very important thing-- decision. So it depends on the graphs. If you have power law graphs, you might want to do something. If you have a more limited distributed graph, you want to do something else.

And the power law graphs, sometimes you might do something different for the high connected edges versus others. You might not even differentiate between that.

It depends on the algorithm. So if you are doing-- visiting all the nodes, whereas as a data-driven algorithm, you might do something different. It also depends on the hardware you're running. So, for example, if you are doing a Google search, basically indexing, you're running an algorithm that has to operate on the entire graph in here. And the graph is a power law graph in that. And you're running on a cluster. So the right thing might be something like a pull schedule with some partitioning and something like a vertex parallel, or some kind of a parallelism scheme in here might give you the best performance.

But in the other side of the Google, if you're trying to do a map, and if you're trying to give you directions, you have a very different type of a graph. You are doing a data-driven algorithm in that graph. And you might be running on a single machine. Because you need to give direction fast for each individual time. You might have a very different type of algorithm you want to run

this graph, the push algorithm in vertex parallel, perhaps, some combination in there.

And, of course, if you get a bad algorithm or bad set of-- way of doing it, you can be very bad. You can get hundreds of thousands times slower than the best you can achieve. So it matters to find the right thing, right way of doing things.

So this is where GraphIt came in. GraphIt is a domain specific language, basically, we developed. And one thing GraphIt did was we said, OK, look, the algorithm is mostly constant. But how you process the-- how you go about it is very different.

So we want to separate these things. So the first thing we did was come up with the algorithm, which is what do you want to compute? It's very high level. It don't tell you how we are computing that-- saying this is my algorithm. I aim to process these nodes. And this is the computation I want to do in there.

And you separate it with an optimizational schedule how to compute. So we'd say, OK, to do this algorithm, you had to do a push schedule, do this type of parallelism-- each separately. And the nice thing is that is now, if the graph changed or if the matching changed, I can give you a different schedule in here.

So let me show you some examples. First, look at the algorithm in here. So we show three different types of things you want to do. So you want to do the entire graph in here, have the data-driven. Or I might want to just operate on the vertices in here.

So this one we have-- the language provides a very simple way of doing that. Language has this function saying, if there are edges, all the edges of the graph, apply-- you can give a function. The function takes the, basically, nodes and the edges, basically-- it to basically carry out this computation, a very simple way of doing that. So this is the representation.

So the nice thing, the simplicity of programming now. If I write it in C, it will look like a big blob of ugly code. In the domain specific language, all you have to write is this-- make life very simple.

Or if you're a data-driven language, I have to say, OK, I start with this set of vertices to compute in here. And here are the vertices I am going to in here, the vertex sent here. And then I do some filtering. Because I might not go visit everybody. There are some filtering of what you can do.

And then once you figure out exactly the things you are computing, here's a function to go and apply to that. So I can give you some very nice way of basically subsetting my graph with certain properties, select those things, and now go compute there.

And if you're only doing vertices, say, OK, for each vertices, again, I can filter, saying this subset or something go to that computation. So language-wise, it's very simple. This is what all you had to do.

Now if you look at PageRank, PageRank has two interesting update functions. What is-- one is updating, going-- looking at edges. So what it says is new rank, I get the destination edge. And it gets updated using all the source edges in here. This is the update function, very simple update function.

And then once you do that for each, basically, vertex, I go do internal update. I give these two functions and put them together into driver. And the driver says run this function, run this function, and I'm done.

OK, so I can write this code at higher level, much simpler, much nicer, much more elegant way. It's much easier to understand. It's easier than even the simple C++ code to understand what's going on if you write it in this way.

So this is the first advantage of a domain specific language. I can do this.

Then the next thing you can do is now I can come up with the schedule. So schedules should be easy to use. And it should be powerful enough I should be able to get the best speed possible. Because I can tell you all the crazy things I can do to the code.

So here's my program here for PageRank. And so what I can do is, for this algorithm, I can provide this schedule in here. And this schedule basically says, OK, look at this guy, s1. I marked it in there. For s1, I want to do SparsePush type computation. This is how I want to process this one.

And then, by looking at that, I can generate a pseudo code that looks like this that basically first goes through a source node, because I'm doing push from source to destination. And then I'm going through all the destination nodes of that source. And I'm going to actually go and update them. So I can do this very simple updating here.

But this might not get you that performance. I say, ah ha, I want to do this in parallelism. I want

to run this parallel. And then when I do that, it will automatically generate, say ah ha, now, I will make this two parallel. And now I can't do simple updates. I have to atomic add. So here's my atomic add operation-- so the graph in here.

Then you might think, and say, mm, do I want to do the push? Can I do a pull? So if I do a pull chain, it will basically switch the-- in here. Now I am going from destination to source. I changed order in there. And now I don't have to do that atomic update. Because I am pulling everything to my node and updating here.

And then, of course, if you want to do some kind of partitioning, I can also say partitioning, it's-- now we created a sub-graph in here. And for the sub-graph, I am doing this partitioning. So I can keep changing all these things. Look, I didn't touch this. My algorithm still stays same. I'm changing my scheduling. I can play with this schedule.

Nice thing about that is now if you keep playing with the schedule, here's the kind of performance I get. The first guy was sequential, pretty bad performance. The next guy, I just parallelized in here. I got some performance in here. But it had all the synchronization. So I changed the order of execution. And I got an even better performance. And now I partitioned, got [INAUDIBLE] performance.

So this is the order of doing that. But, of course, you can play with many, many different combinations. And what GraphIt has is huge number of different combinations you can play with.

So there are a lot of different optimizations. You can do direction optimizations, push, pull, doing a sparse, dense, different parallelization, cache, NUMA optimization, and also data layout, things like structures of arrays, array of structure layout, additional data structures that simplify computation. All these things I can specify in here.

And then you can play with it. It's not clear which one wins. It depends on the algorithm, depending on the graph shape, graph size, depending on the machine you run. So most of the time, if you are a performance engineer, you'll be trying different things, and looking at the performance, and say, this doesn't get good cache behavior. OK, let me try different things.

So you want to iterate. And these iterations, you want to do fast. And this will do that.

So let me tell you a little bit of results. This is a-- I have to explain. This a little bit of a complicated graph. So what we looked at was shown against bunch of different benchmarks, a

bunch of different frameworks that do graphs.

So what they says is here is a program, PageRank, ran all on a graph like general graph in here. One means it ran the fastest. This ran about 8% slower. This ran 50% slower. This ran 3x slower, and 8x lower for that graph.

The interesting thing is as you add more different graphs, the performance changes. So, in fact, even though we ran fastest for this road graph, which are a very different type of graph, this framework ran the-- provided-- got the fastest result. Because the graph is different. So it might be doing something that's better.

The interesting thing is since we had-- because most of other frameworks will have a couple of built in things they try. They don't do, give you all this ability to try all this optimizing. They say, ah ha, I know this. This is really good. I will do that. It works for certain things, not for everybody.

And so if you look at the entire different breadth-first search, connected components, shortest path algorithms, what you find is some frameworks are good sometimes. They might be really bad in other times, to either some algorithms, some type of data, they can be really bad.

So this algorithm was really kind of good at this data set, but really bad in this data, and really kind of not good in this algorithm. We are most of the time good all the time. The reason is we don't make a few decisions. In GraphIt, what it will do is it will give you this ability to try different things.

And depending on the graph, depending on the algorithm, some optimizations might work better than the other. This is exactly what you guys have been doing in the class. You are trying different optimizations by hand. The difference is every time you thought about optimizing, you had to go change the entire program to make that work. Here you just change the scheduling language one way, recompile, run, measure, and you can do this fast.

Any questions so far before I switch gears?

AUDIENCE: [INAUDIBLE]

SAMAN AMARASIGNHE: OK. So I'm going to switch to another domain specific language. You will find a lot of simulated, lot of parallelism in here. This was intentional. I could have talked on many different domain specific languages. But I took another one that you-- almost has kind of a mirror

similarities of what's going on. And you will see a pattern in here, hopefully. And after this, I will ask you what the patterns are.

This language is Halide. It was originally developed for image processing. And its focus is-- the graphs focus on this past graph data structures. Halide's focused on-- because images are dense, regular structures. You do regular computation on the images. And you process this thing. And you have a very complex pipeline. Like, for example, camera pipeline do many very complex algorithms to the image before you get from the bits coming out of your CCD to the beautiful picture you see in Facebook.

And the primary goal of Halide was you want to match and exceed the hand-optimized performance, basically. This was the property we want to do. And we want to reduce the rote amount of programming that normally a performance engineer has to do to achieve this thing. And we want to also increase the portability, the ability to take that program from different machines to different.

So let me give you an example. Here is a three by three blur example. So what this does is this [INAUDIBLE] two loops go in the x direction and do a blur in x direction, get the-- average the three values next to each other. And then it will go-- the result of that, do it in y direction, and average that.

OK, very simple filter that you might want to do for image, you can run this. This is valid C code. But if you want to get performance, you want to generate this guy. This thing, on the other hand, ran about 11 times faster than this one. This has done tile. It has fused multiple loops. It has vectorized. It has multi-threaded. It had to do some redundant computation I'll get to a little bit later.

And it basically gives a near roof-line optimum performance. That means it's using the machine resources to this max. Because this has a bunch of floating point operations. So it's basically floating point unit is running at the max performance. So there's nothing much else you could do to this one. But you write this thing. And this is not that easy.

So this project started some time ago with one of my-- the person who did it-- going to Adobe. He went to Adobe. And they had this thing called a local laplacian filter in Camera Raw, and Lightroom, and Photoshop projects in here.

The reference implementation was about 300 lines of code. But the implementation that they

used was about 1,500 lines of code. It took three months of one of their best engineers to get to that performance. But it made sense. Because that engineer was able to get 10x faster by trial and error for this piece of code. It's a non-trivial piece of coding here to go do that.

So the student, Jonathan, who's now a professor at Berkeley, he basically, in one day, in 60 lines of Halide, he was able to beat 2x of Adobe code in some sense. And then Adobe, in those days, didn't generate any code for GPUs. Because they decided GPUs are changed too fast. And they can't keep up updating for GPUs in every generation. Then they-- because of that, they were not-- the Adobe applications were not using GPUs. So if you ran Photoshop, it's not going to use a GPU. Even if you mention it has a GPU.

So Jonathan still had some time left in the day. So he said, OK, let me try to write on GPUs. So he just-- basically the same code, he changed GPUs and got a 9x faster than the fastest Adobe had ever had for this piece of code. So how did he do it?

Again, the key principle here is decoupling algorithm from schedule. So algorithm, again, is what is computed. And the algorithm defined the pipeline of very simple pure functions operating in there. And execution order, parallelism, all those things is left for the schedule.

The pipeline of Halide just looks like this for the blur filter. It says, OK, get the image in x dimension. And do it a blur in the y dimension. That's all. And the image size is-- because it's operating on the entire image, you don't have loops in here. That's all you have to say there.

Then you have to come up with a schedule. Again, the same way when and where it's computed, to be simple, that you need to be able to tell that. And it has be powerful. You need to be able to get the hand-optimized performance or better by doing this.

Something looks a little bit familiar. Because it's all these things, a lot of work you do performance kind of fit into this genre. You need to do a trade off between locality, parallelism, and redundant work. That's what you look for in here.

So let's look at the three things you need to do. First, you need to get parallelism. Parallelism is you need to keep the multi-cores and vector units happy and probably the GPU busy. But if you have too much parallelism, it's not going to help you. I mean, nobody is going to take advantage [INAUDIBLE] parallelism. So let's look at a piece of code in here.

So assume I am going to say, I'm going to run all these things parallel and all these things parallel afterwards. If you have three cores-- great, I got a lot more parallelism. I got six times

parallelism. Hurrah, nobody's going to use that. It's not that useful to get six times parallelism in here.

On the other hand, if you run like this, one at a time, you have parallelism of one. That's not that good. Because you're going to not use the machine. So what you really want is something basically-- OK, wait till it's done-- that actually do parallelisms of three might be the best way of running that machine to get best performance. You don't want too much. You don't want too little. You want to get the exact right thing.

The next interesting thing you need to get is locality. Normally, when you do image processing, what you do is you change everything in the image in one filter. Then the next filter has to go in and change everything in the image. So what happens if one filter ran through the entire image and the next come and start running through the entire image? What happens, basically? Is that good? I give the entire image, say you, do my first color correction. And I will do some kind of aberration correction afterwards. So what happens if you do something like that? Entire image, process one filter, then the next filter takes the image and process the entire [INAUDIBLE] or whatever multi-megapixel image-- oh, you-- you're on a [INAUDIBLE]. OK, back there.

AUDIENCE: You end up kicking [INAUDIBLE].

SAMAN AMARASIGNHE: [INAUDIBLE] cache. Because if the image is large, it doesn't fit in the cache. It's not that great to do this. You won't get things in the cache in here.

So assume I go like this, processing the entire first row before you go to the second row. So what happens now here is we need to start touch this one-- I need to read these two values. And those two are-- the last time I read them was way before I started.

So I [INAUDIBLE] them. I went through all the image. And I come back to that. And this distance-- by the time I reach here, I might-- these two might be out of the cache. And when I go back there, oops, it's not in the cache. I have a problem in that.

So the other way, a right way to do that might be trying it this way. If I run it like this, basically, what happens is as you run-- when I touch this, I won't have-- I want to get these three to run this thing. Last time I read this one was just before, in the previous iteration.

So to get to that-- I just touched it. So the next guy uses it, the next guy, and after, after. I go

to my window. I've never touched that again. I have a really good locality in here. So I want to operate it that way. I won't get good locality in here.

So redundant work is a very interesting thing. Sometimes, if you want to get both locality and parallelism, you might have to do some extra work, a little bit of extra work. So assume in this one I had to process these elements parallel if I want to run these three. Because these three needs all these four elements in there. These three need these four.

If I want to run these two parallel in two different cores, it might be better if both calculates these two values. Because I don't have to synchronize and stuff. I can say, the left guy, calculate four values. And then I can do the three. The right guy, calculate the four values. And then I can do the three. I can do that parallelly.

But now the middle two guys, these two get calculated twice. Because both needs it. And so what that means is-- oops, you can keep that.

So sometimes, to do everything, I might have to do some redundant work. So the way to look at that is I can put this into this scheduling framework. I can map my computation bandwidth. That means coarse interleaving with low locality. That means I finish everything before I go back in here between two things. If I run two things, I finish this one before I go to the next one.

Fine interleaving means I process one element one, duh duh duh duh, go back and back and forth in here. That's my two options here.

Other side is storage granularity. What that means is-- storage granularity very low means I calculate something, I don't remember. Next time I want it, I recalculate it again.

Very high storage granularity means once I calculate it, I will remember it forever. Anytime you need that value, I have it back for you. So that means I have to get it to you from anywhere I calculated.

Storage granularity low means my process, I calculate, I use, I throw it out. If anybody else want it, they'll recalculate again.

So now you can have many different computations in different places of this space in here. So if you want to compute something here, this is the scheduling language. That means I run this one, and I run this one. I have no redundant computation, very coarse grained interleaving.

That means I run the entire thing, and then the next entire thing.

You can go very fine [INAUDIBLE] in here. I'll calculate this one. And I'll calculate these three again, these three again. So everything is calculated multiple times. When you need it, I recalculate every time I need something. I don't store anything in here.

So it's good. I have a lot of locality. But I'm doing a lot of recomputation.

And then here, you have something like a sliding window. Basically, you are not recalculating anything. But you are sliding in there. You have a little bit less parallelism. And then you could capture this entire spectrum in between in here. And you can get different levels of fusion of these tiles. And you can calculate-- so I don't recalculate everything. I recalculate a few things in here. These two get recalculated.

And then you can-- I'll go through this fast [INAUDIBLE]. You can use all these operations. So here is the interesting thing. So here is I am showing you different schedules at different points in here. So I'm going to run this game. This is on time. So what it says is this is doing-- you're going through the first input, [INAUDIBLE] the middle one, [INAUDIBLE] the output in here.

So this has all locality, lot of redundant work, good patterns of locality. All patterns have not good locality. In here is some kind of intermediate thing. So what it shows is these are no good. A good balance between locality, parallelism, and some redundant work seem to do really well. This guy finished the fastest.

So what you do is you write different schedules for these things. And you keep running. And we figured out what schedule works. So this is kind of trial and error part you have to do in here.

So if you look at what's going on in here, what you see here is-- there's some example-- is bilateral filter computation here. What it says is the original is about 122 lines of C++ code. And you found something with a good parallelism in here. But we could write it in 32 lines of Halide in here. And we were able to get about 6x faster than CPU.

But the best algorithm was somebody hand wrote for the paper on GPUs. And what it did it was it gave up some parallelism for much better locality. And if you give up some parallelism, much better locality, because we can optimize in that, we got faster than their handwritten algorithm. So we can change something.

Here's, again, another algorithm that is doing segmenting in here. And it was written in MATLAB. And MATLAB is a lot less lines of code, of course. But in Halide, it's a little bit more line, because you're not just calling library functions. And Halide was 70 times faster.

And if you run into GPU verses MATLAB, it's about 100-- 1,000 times faster. It's not because you're running bad MATLAB loops. In fact, what MATLAB did was it called very well hand-optimized libraries.

But the problem with calling libraries, there's no locality. I called a really fast library for first routine. It runs really fast. And then you call the next routine that has to [INAUDIBLE] the entire image again. And now my image is completely off the cache.

So what happens is between these very fast libraries, you're bringing the image from cache. And when you have something like a library, you can't fuse library functions together.

In Halide, we can confuse them together and say, oh, I take this line of the image, and I will do everything on that before I move to the next thing. So I can do much faster.

My feeling is each function probably, in MATLAB was faster, because they have a handwritten really fast thing. But the copying of data from-- the moving from cache, the cache effects, was really slowing it down.

So here's the thing that we showed before. This is a very complicated algorithm. It's what we call a pyramidal algorithm. So what it does is you take a [INAUDIBLE] in here. And you divide it into a bunch of blocks in here in each level of pyramid. And you do some computation, do some look up, and do some up sampling in here. You do some addition computation and compute that. And then you create more and more smaller and smaller images in here. You do-- you basically [INAUDIBLE] image pyramid in here.

And so to do this right, it's not that simple. What that means, in each of these level, there are different balances you want to be. If you have a lot of data, parallelism is not that important at that point. Because you have parallelism anyways. You probably had to focus a lot more on locality.

But when you get to the smaller amount, I think parallelism matters. So you have to come up with very interesting balances between those. So many, many things to tune at every level. There's not three things. There's hundreds of different levels.

So the nice thing about Halide is you can play with all these things. You can play with all these different concepts and figure out which actually gives the fastest performance in that.

So a little bit of, I would say, bragging rights in here for Halide, Halide left MIT about, I think, six years ago. And right now, it's everywhere in Google. So it's on Android phones. It started to Google Glass. It doesn't exist anymore, but in that-- and in fact, any-- all the images, all the videos uploaded to YouTube right now, they do front end processing. And that processing pipeline is written in Halide.

And they switched to Halide because Halide code was about, I think 4-5% faster than the previous version. And 4-5% faster for Google was multi-million dollars saved for them, because there's so many videos getting downloaded from that.

So recently, there's a Photoshop announcement that's saying they have an IOS version of Photoshop from Adobe. They just announced it. I don't think it's even out yet. And the entire Photoshop filters are written in this new version using Halide.

Qualcomm released this processor called Snapdragon image processor. So they built that processor to do image processing in there. And the programming language to program that processor is basically Halide. So you write the code in Halide. So that is the kind of-- the assembly level that makes it available for this in here. And also, Intel is using that. So there's lot of use of this system at this point, which is really fun to see academic project getting to a point it's very heavily used.

And part of that is because it's very useful. Because people realize they need to optimize these code, because cameras and stuff, performance matter. And it needs to look-- having some poor engineer spending months in the corner, just trying out those things, you can try the same things, and lot more by do it faster.

OK so let me ask you a question. So now between Halide and GraphIt, what did you find? A bunch of similarities-- I want to figure out are there any interesting similarities you guys found between these two projects?

AUDIENCE: They both allow you to try optimizations really fast.

SAMAN AMARASIGNHE: So part of that is also a lot of times compilers are kind of black box. We know everything, just feed us, we'll give you the really fast code. And the problem is, they're never the fastest. So if you really care about performance, you get 90%. Then you get really frustrated-- now what do

I do? But this was, OK, I'm not going to-- you are better at what to do. But I'll make you life simpler. So we still want the performance engineer. It's not the person who just don't understand performance feed, you get fast code. We need a performance-- but we want to make performance engineers life easier.

So both of them, said, OK, we need performance engineer. We can't automate it. We don't know how to automate all these things. There's too much complexity. But we will let you, performance engineer, explain what to do. But we'll make your life very simple. What else?

AUDIENCE: Something that was cool was both of these languages can do algorithmic level optimizations [INAUDIBLE], which is pretty different from what compilers like GCC are explained [INAUDIBLE].

SAMAN AMARASIGNHE: Yeah, because-- I wouldn't say alg-- you can do a lot of domain specific optimization. So algorithmic optimization is one level higher. You can say, ah ha, I have a better algorithm.

So, OK, I don't want to do a quick search here. I can do insertion sort. Because quick sorting and insertion sort might be faster for a certain class in there. So that is level change we don't do.

Or, worse yet, I can say-- this happens in a lot of things in machine learning-- yeah, if I just drop a number here, I'm OK. I don't have to get the compute exactly right. Oh yeah, if I-- I don't have to calculate everything. If I calculate for 10 people, it's good enough.

So that kind of changes, you can't do. Because that's very contextual. Like, for example, a lot of time, if you are doing things like machine learning, there's no right answer. You need to have a good answer.

So sometimes good means you can not do certain things. And you need to find what things you shouldn't-- you cannot do, that you get a huge benefit, but you don't lose that much. That level you can't do that. That's the next level of [INAUDIBLE] is saying, OK, how do you do that? How do you-- when somebody say, OK, look, I can train it for 10 iterations versus 100-- ah, 10 is good enough.

I can't-- if your code is written to train for 100 iterations, I can't tell you, oh yeah, 10 is good enough. That is a decision that has to be a lot higher level than what I can make. So that's a-- that-- there's an interesting level that can still exist on top of that, which we can't automate that

easily. But we might be able to make, still, a language, like a schedule language, give you that option. That's a cool option to give, say try some of these things that actually change the algorithm. But within the algorithm, that means I'll still give you the same answer. I will try different things.

Any other things? Any other things you guys thought that was interesting? How about from somewhere here? What are the interesting things you found? Back there.

AUDIENCE: They both involve a lot of trial and error.

SAMAN AMARASIGNHE: Yes, both involve a lot of trial and error. I mean, this is the modern computer systems. Everything is extremely complicated. There's no right way of doing things when you look at this pretty large piece of code. And there might be a lot of-- there are caches, parallelism, locality, a lot of things that can go right. And so you might have to try out many things.

So if you know the answer, if you come up with the, I know exactly, every time, I have the right answer, that's amazing. But even the best performance person might not be able to look at a piece of code and say, ah ha, I know your solution. You do a lot of trial and error. This kind of supports that. And you probably have figured that one out for most of your projects. It's not like you went and say, ah, I know what to do. You probably did many different trials.

And I know from this, lot of things, you actually either have no impact or slow down the code. And you say, oh, that didn't work. And all if-- [INAUDIBLE] you leave in your codes shows all the crazy things you've tried, and nothing happened. So that's an interesting thing. What else? Anything else, a little bit differently that you see on this one?

AUDIENCE: I was just wondering, are there other similar domains that don't have something like this [INAUDIBLE]?

SAMAN AMARASIGNHE: So interesting question-- are there any other domains that don't have something like this? People are working on similar things to machine learning these days. That seems to be their domain, and tensor flow, and all those people are trying to do-- to build systems like similar-- like frameworks that you can get that.

I mean, that's a very-- I think-- the way I have operated is I go talk to people. And sometimes you find this poor graduate student, or postdoc who want to do some research but spending all of their time basically optimizing their piece of code, because they can't get their performance. And then that might be a good domain.

You find these people in physics. You find these people in biology. And I am actually talking to-- because, for example, in biology, a lot of this gene sequencing stuff is-- there are very similar things you have to do. But they seem to be spending all this time writing the code, and then-- mired in code complexity. OK, can you do something in that?

I mean, the key thing is this is a good way to-- a nice thing about MIT is there are good-- a lot of very smart people in many different domains trying to push the state of the art. And who's spending all this time cursing in front of a computer program to get to a point they want to do, because-- not because they don't know the algorithm, because the amount of data they have to deal with-- astronomy, I mean they get these multiple telescopes, that deluge of data. And most of the time, they know what they have to do. They just have to-- can't write the program to do it fast enough. So there might be domains like that, if you look at that.

And there might be domains from application and domains from patterns. Like sparse matrices or graphs are patterns, which-- or not only on a single application. I mean, it works in multiple places.

There might be other patterns. Say, this is-- if you want to do research, this might be interesting piece of doing research. And I have spent my life finding different domains and a bunch of people that spend their lifetime just hand hacking things and telling them, OK, let me see if we can do some nice abstraction.

Anything that you guys found that's interesting? So to both of them, what are-- what's the space that they operated on to optimize programs?

AUDIENCE: [INAUDIBLE].

SAMAN AMARASIGNHE: Done for me, no. What I'm saying is, what are the things that you're trying to optimize? There's a nice space of three different things-- parallelism, locality, and redundant work.

My feeling is, as you go as a performance engineer, that's going to be your life. If I add additional things, there might be algorithmic things that you completely get rid of well. But most of the time, we are-- all of you will be [INAUDIBLE] performance will be working on some kind of multi-core vector GPU type units. You have to get parallelism. So getting parallelism is important.

But then, if you don't have locality, it doesn't matter. Because most of the time you're waiting to

get data from all the way from memory. So you had to get good locality.

And then more-- a lot of times you can do that really well if you do some extra computation. But if you do too much extra things, that's going to, oh well, that's not going to help you.

So it's all about playing the distribution. You've got a final project. That's exactly what you're going to do. You might say, ah, if I can do some extra work, OK, I can do this faster. But oops, no, this extra pre-compute pass, or whatever, it's not-- I can't amortize the cost. So there's these three things that you're trading over that. So that's one interesting thing.

Another thing is we made it available for the programmers to do this scheduling language. But can you make it-- can you think of a way to make it a little bit easier for programmers than doing a scheduling language? What can I do? What's the nice thing about scheduling languages?

It's very simple. It has a very simple pattern.

AUDIENCE: [INAUDIBLE]

SAMAN AMARASIGNHE: Yeah, I mean, that's-- the number of options-- it's not like you can write any program. There are certain things you can do in the schedule. So if you know that space, you can sort of use doing that smartly. What else can we do with it?

AUDIENCE: Test them all.

SAMAN AMARASIGNHE: Test them all. That's one approach there.

AUDIENCE: Use autotuning, trying to find [INAUDIBLE].

SAMAN AMARASIGNHE: We can do autotuning. So that switched into the autotuning part of this talk. So performance engineering basically, most of the time, is finding right these crazy things. Like you start looking at, I think, probably as Charles talked, what this voodoo parameters, like, OK, what's the right block size? And it has a big impact, finding that.

A newer memory allocation project, you had to find the right strategy, right memory allocation. You searched through a bunch of these things. You and GCC compiler, there are, I think, about 400 different flags for GCC. And you can actually get factor of four performance by having a [INAUDIBLE] 200 flags into GCC. It's crazy. And that 200 flags is not the same for

every program. And, of course, some programs will crash in some places. Most of the time, it'll slow down or speed up.

So you can just give all the flags of GCC and autotune that. And you can get factor of two to four performance in there. It's just crazy. And then, because it do weird things in there, and then 01, 02, 03 will only do certain amount. So 03 doesn't-- it's not always right. You can try all these things. So you can tune that. And scheduling Halide, scheduling GraphIt, all these things can be autotuned.

So before autotuning, when you have a large search space, what do we normally do? The thing that when we think we are smart, what we do is we build models. We have model for a cache. And say, can we understand the cache? We have all these nice models in here. And using the model, I can predict, ah ha, this is the right block size.

So what's the problem when you try to do a model?

AUDIENCE: Sometime it doesn't work [INAUDIBLE].

SAMAN AMARASIGNHE: Exactly, because most of the time, when you try to do a model, you have to abstract. And most of the time, you-- what you abstract out might be the most important part of the darn thing that we didn't consider. So we built a model for cache. But oops, I could-- didn't figure out pages. Well, the pages made a big difference. So there might be things in real life that matters that didn't fit into your model, or you didn't know it needed to be fit into a model. If you try to put everything, that's too complicated of a model.

So you abstract something out. You can say, I have optimal result for this model. But that optimal result might be way off than the simple result you can get. Because the thing that you didn't put into the model are the important ones. So model doesn't work.

The next thing is what you do, heuristic-based thing. This is where these old people come and say, I know how to do this. In order to do this, you need to do this thing, this thing, this thing. You can come up with some kind of the old grandmother's solution type thing. There are certain things that will always work. And you hardcode them. So you can say if the matrix dimension is more than 1,000, always go to block, or some kind of rules like that.

These rules work most of the time. But obviously, there are certain cases that rules doesn't work. Worse, that rules might be set for a certain machine, certain architecture, all those

things. I'll give you the story.

So GCC has this fast table sort routine. So fast table sort routine says sort using a parallel quick sort. And when the number exceeds 16, switch to in session sort. It's hardcoded in GCC. It's like, wow, some amazing person figured out 16, this amazing number, has to switch from parallel quick sort to insertion sort. So we are trying to figure out, what's the profoundness of this number?

The profoundness of this number is somewhere around 1995 when this code was released. In those machines, that was the right number. That 16 was a really good number to switch from parallelism to doing that, because the cache size, stuff like that. But that 16 survived from 1995 [INAUDIBLE] even two dates there.

Today that number should be like 500. But it's in there, because somebody thought 16 is the right, it's hardcoded in there. It didn't change.

So there are a lot of things in compilers code like that, that they draw, that some programmer said, I know what works here. This fits in there. You put it in there. But there's no rhyme or reason. Because at that time, they had a reason. But it doesn't scale.

So a lot of these heuristics get out of focus very fast. And there's no theory behind this to say, now, how do you update? You had to ask people, why did you put that? And then it's, oh yeah, because my machine has a 32 kilobytes of cache. It's like, oh, OK, that's a different machine what we have today.

So that's the problem in here. And then other thing is you can do exhaustive search. You can say, OK, I'll try every possible thing. The problem here is sometimes my search base is 10 to the power 10. You don't have enough seconds in your lifetime to do that search. So it would be too complicated. And that's where the autotuner comes in.

So-- oh, OK, actually I have a little bit more slides here. So model based solution is you come up with this comprehensive model, like a cache model, or something like that. And you do that. And you can exactly show what's right for the optimal solution in here. But the problem is hard to build models, cannot model everything, and most of the time modeling is the most important thing.

Heuristic-based things are the rule of the thumb kind of solution that you come up and say, it's hardcoded in there. And it's very simple and easy. It works most of the time, if you get it right.

But the problem is too simplistic. It doesn't scale. It doesn't stand the test of time, most of the time in here.

An exhaustive search is great. But the problem is just way too much possibility of searching in here, too big of search base. You can't do that. So this is where you want to prune the search base. And the pruning, the best way to do that is basically use autotuning.

So what autotuning you can do is you can define the space of acceptable values nicely, choose a value at random-- that's what the system will do, try it out there-- and evaluate the performance of that value end to end. Because end to end matters. Because if you try to predict, most of the time, it might not work. And if satisfies the performance that you need, you're done. Otherwise, choose a new value and iterate over there, go to three in there. So this is the kind of thing.

And what you have to do is you need to have a system to figure out how to do this fast, basically what space to basically do that, when to basically think you're done, how to go through the iterating loops through that.

So this is the kind of-- with cartoonish way, what happens is you give a value candidate, you compile the program, you run it with bunch of data, you are running through the bunch, otherwise you are all fitting. You can't run it with one. And you get the results. And you get something like average. And if you go through this loop in here.

And what OpenTuner has done is come up with the ensemble of techniques. So the idea there is when you're searching through a space, you might be at the bottom of a hill of the space. So what that means is there are certain value, if you keep improving in value, that you are getting good better, and better, and better. And at that time, something like a hill climber-- hill climber, or somebody like an [INAUDIBLE] hill climber can actually give you the best performance. You're going very fast in there.

But meet you at the top of the hill for that pyramid, oops, there's no place to go. So that if you tried to hill climb, it's not going to be helpful. So at that time, what do you want to do is do something like a random search in here.

So what this system do, OpenTuner, you do it, it will basically test this request in there. And if something is doing very well, it will give it more time. If not, what it will do is it will say, OK, look, this is-- this technique is not working. Let's try something else. It'll basically allocate the time in

here. So it do this search much faster than otherwise you can do.

So I want to finish this by showing what you need for autotuning for GraphIt. So we have algorithm, and you have a schedule. It's a pain to write this schedule.

In fact, there's a good [? interesting ?] thing in-- when you do Halide, we decided, OK, it should be similar to write the algorithm for Halide and the schedule. Google very fast realized many people won't use Halide. And they-- at about two years, they had about hundreds of programmers who can write the algorithm.

But they only had five people who could write the really good schedule. To write a really good schedule, you need to understand a little bit of the algorithm. You need to understand a little bit of the architecture, a little bit of everything. And that's much harder for people to learn. So getting the schedule right is not that easy.

And same thing in here, because you need to understand a lot unless you do kind of random-- you've got certain arbitrary, but to do it right, you need to know a little bit more.

So what we can do is we can basically give some idea about the graphs and some idea about the algorithm in there. We can autotune these things in there and then generate the schedule.

And so what we found was to generate this schedule, if you do exhaustive search, it runs for days. But if you're using autotuner, OpenTuner, you can find a really good schedule for-- in less than two hours. And, in fact, a few cases we found schedules that run better than what we thought was the best possible schedule. Because it was able to-- because it was able to search much better than our intuition would say in here. And when-- and even if our intuition know it, it has more time to try many different combinations and trying something in-- come something better in here.

So that's all I have today.