EPISODE 1203

[INTRODUCTION]

[00:00:00] JM: The incredible advances in machine learning research in recent years often take time to propagate out into usage in the field. One reason for this is that such state-of-the-art results for machine learning performance rely on the use of handwritten idiosyncratic optimizations for specific hardware models or operating contexts. When developers are building machine learning powered systems to deploy in the cloud and at the edge, their goal is to ensure that the model delivers the best possible functionality and end user experience. And importantly, the hardware and software stack may require different optimizations to achieve that goal. OctoML provides a SaaS product called the Octomizer to help developers and AI ops teams deploy machine learning models most efficiently on any hardware in any context. The Octomizer deploys its own ml models to analyze your model topology and optimize benchmark and package the model for deployment. The Octomizer generates insights about model performance over different hardware stacks and helps you choose the deployment format that works best for your organization.

Luis Ceze and Jason Knight join the show from OctoML. They joined the show to talk about how OctoML is automating deep learning engineering and why it's so important to consider hardware when building deep learning systems as well as the evolving field of deep learning.

[INTERVIEW]

[00:01:23] JM: Guys, welcome to the show.

[00:01:24] JK: Great to be here.

[00:01:25] LC: Thanks for having us. Great to be here.

[00:01:28] JM: What are the problems with machine learning deployments in 2021?

© 2021 Software Engineering Daily

[00:01:34] JK: Essentially, machine learning is still just too hard. And especially in deployment, a lot of effort has been spent on the training frameworks and data annotation and management frameworks, but the deployment side of things in part because the increase in the number and variety of hardware backends that we've seen over the last few years makes deployment even more difficult.

So in 2021, you're looking at a number of problems. One is, typically, vendors, Silicon vendors, hardware vendors, have software specific solutions for their hardware. They recommend you deploy too. And these often vary in terms of the performance you get and the type of coverage you see when you're trying to deploy workloads to a given piece of hardware and they have different APIs and different installation. And so you have to learn the ins and outs of these as you go forward.

And then whenever there's a problem in terms of performance or coverage for one of these operators in these underlying software stacks, then you have to go in and, if you can even, try and fix this and patch around it writing low-level code for that specific hardware. Typically, performance and portability and go-to-market, the resulting slowdowns and getting your model to actual production are the three kind of pains we see in this space.

[00:02:58] JM: And is there room to build tooling to improve that process? I mean, because there's so much heterogeneity. It seems really hard to build tools for this space.

[00:03:10] LC: Oh, yeah. Great question, Jeff. Basically, what the TVM, the Apache TVM project started with the observation of this large collection of models and growing collection of models that the world cares about and it's starting to be embedded in a bunch of applications more and more. And there's a growing set of hardware targets that these models need to run on. So this cross product is large. And as Jason said, it's a very complex ecosystem and many, many layers to get from the framework down to hardware. We started TVM because we wanted to form a clean, unified abstraction across all hardware to the frameworks so the data

scientists can target and then and automatically optimize and really tune the model to specific hardware targets in a clean and unified way.

So you're right, yeah. There's a lot of heterogeneity and that's exactly the genesis of the TVM project. And in TVM, the way it works in a nutshell, we can go as deep as you think it's interesting here that it has its own internal representation of these models that enables a lot of automatic optimizations and code tuning to specific hardware targets in automated fashion and it uses machine learning to do that, because precisely to deal with the intricacies of tuning models and the low-level libraries that Jason's talking about to specific how targeting target involves quite a bit of heavy engineering that's done by hand when we replace that with a machine learning-guided engine to optimize the code.

[00:04:35] JK: Yeah, going even further back, if you pick up the Dragon compiler book or look into compiler design as it's been taught for decades, then you'll see that the techniques that are used to construct compilers are rooted from those days that typically heuristic-driven set of compiler passes. And to your point, Jeff, the heterogeneity and complexity and just raw performance demands of modern accelerator hardware and general purpose hardware as well leads to more and more complexity. And the innovation of TVM was, “Well, we have new tools now and machine learning is one of those tools. So can we take machine learning and apply that back to the the process of compiling machine learning workloads themselves?” And that led to TVM and led to all of the performance and portability and time market savings that we're seeing now.

[00:05:32] JM: Can you go a little bit deeper on what TVM is, the Apache TVM and the motivation for it?

[00:05:41] JK: Sure. So TVM is an open source deep learning compiler and runtime and it's actually a tool kit. It's quite flexible as opposed to just a compiler frontend with very few knobs. It's has the compiler frontend, a nice easy to use command line that can take any machine learning or deep learning model and then compile that to an efficient, executable form of that model that you can run on any given piece of hardware. But due to the flexibility, there's a

number of interesting pieces in that. One is that inside the TVM compiler engine, as we mentioned, we're using machine learning-guided search similar to what you would see in kind of alpha zero or one of these kind of ML-guided search procedures to find the most efficient way to represent that model execution on a given piece of hardware more efficiently than what you would find for what humans are able to do by hand for a generic broader swath of design space, because we can tune for that particular workload, or the machine learning-guided procedure can tune.

The other main use case for TVM in addition to compiling whole models and then getting a deployable binary out the other side and deploying that and getting performance and portability advantages is that if you're a researcher and you're experimenting with new variations of transformers or convolution variants or recurrent models, then TVM gives you a rich set of tools to describe algorithms at a high-level of abstraction and yet get good performance code at the bottom for any hardware that you're interested in, NVIDIA GPUs, AMD GPUs, intel CPUs, AMD CPUs, et cetera. And so both the kind of full model compilation and kind of custom workload, custom algorithm experimentation and compilation are the two primary use cases.

I guess there's the third, which is for Silicon vendors themselves, looking up the stack, they need to get their piece of hardware accessible to a broader user base and they want to enable these higher level abstractions, and TVM enables you as a hardware Silicon vendor with a much lower amount of effort to get a high-performance compiler with all of these kind of frontends that are user-friendly Python and deep learning framework integrations in front of your users more quickly.

[00:08:20] JM: So how does TVM fit into a typical workflow of a machine learning developer?

[00:08:30] JK: Right. So if you're wanting to deploy your model and you just want to go fast and you want to maybe experiment with new platforms or just optimize for the platform you're currently using, then today once you've trained your model using TensorFlow, PyTorch, Keras, whatever, then you have that serialized model sitting on disk. You pass that into TVM through

either the command line interface or the Python API or C++ if you'd like and you optimize that for the hardware that you're interested in. And through that compilation optimization procedure, you then get a binary out the other side, a shared library with whatever API you want on top, C API, Python API, Java API, whatever. And then you integrate that back into your broader application code. The APIs are quite straightforward. And then you ship that alongside the rest of your application. And then I'm happy to get into the other two workloads as well if you're the researcher or the Silicon vendor software provider.

[00:09:39] JK: Oh! This is like a quick summary that I think might bring it all together here, is that from the hardware vendor's point of view, their goal is to make sure that machine learning end users make the most out of their hardware without them having to worry about catering to every single specific user to every single specific framework, which happens to be the case today, right? Simply not scalable even for the big hardware vendors.

Now, from machine learning and user point of view, what they want is not having to worry about what they need to do to their models in developing them to actually make them run well in whatever hardware targets they want that to run, right? So even not having to worry about the harder vendors specific software stack, which often tends to be not the friendliest and not the easiest to use. So abstracting that away is a significant advantage to machine learning end users soon.

[00:10:30] JM: So the TVM compiler is going to analyze my model and re-architect it for a lower level execution? Am I understanding it correctly?

[00:10:44] JK: Yeah. Without something like TVM, what happens is your model is described in a high-level framework like TensorFlow or PyTorch and whatever their serialized model representation. It's just a higher level representation of what those computations and sequencing and inputs and outputs are. And typical execution frameworks will then iterate over that model and invoke pre-packaged libraries of optimized versions of convolution or batch normalization or matrix multiplication and invoke those in sequence. And the way that TVM does this instead is by actually analyzing and evaluating your entire model and then generating

that low-level code directly for your model and then compiling that into a shared library on disk and then invoking those specialized kernels instead of these generic kernels that are pre-packaged by Silicon vendors. So that's the slight difference between how TVM operates, and it actually compiles your model to specific kernels for your version of the model versus just generic versions that are that are pre-packaged by your hardware vendor.

[00:12:09] JM: Interesting. So do you have to write all the logic to have that optimization code for each different kind of target Silicon? Or I guess I don't know the interface between machine learning model execution and hardware well enough to ask the right questions.

[00:12:34] JK: Yeah. Yeah. No. It's a great question. This sounds too good to be true, right? How can you get a high-level description of a mathematical computation down to efficient code when this has been really the goal of compiler design for decades now, in the HPC realm, in the compiler realm? And the way this works is that, essentially, in TVM's construction, someone has gone in and for a general class of hardware like CPUs or all GPUs described what are the high-level ways of kind of breaking down or transforming loop nests for dense linear algebra or just even numeric code bases and enumerated kind of not all possible but most of the common techniques for transforming or manipulating these what we call loop nests to get them from a high-level naive form of description that would be effectively the same computation, but terrible performance, and transform those instead to high-level or high-performance low-level code.

But the key difference between traditional compilers and TVM is that instead of humans heuristically describing when to apply these different types of operations and in what sequence and only in these cases do this one but only followed by this one. TVM only requires someone to describe the set of operations and then TVM will discover the appropriate set and ordering of those for this particular workload automatically as opposed to kind of a set of heuristics being developed over decades by compiler engineers trying to handle all cases.

[00:14:24] JM: Got it. As a machine learning team, are they consuming less resources? Give me a bit of a description for the macro impact of this.

[00:14:39] JK: Yeah. So the primary three benefits of TVM, applying a compiler like TVM to your machine learning workflow that you see higher performance, because we can specialize, or TVM can specialize your workload for your workload and for your hardware. We typically see more than 2x performance gains on average than the best baseline, other baseline available on that hardware target. Portability is another one. So even if you don't change your hardware platform today, perhaps you've already optimized for cost and latency for what you need today. But this enables you to potentially experiment with other platforms down the line. Experiment with edge computation as well to offload costs or for privacy reasons and execute this in addition or maybe in partial scenarios or on the edge in addition to your cloud workload or migrate the entire workload. And then the third is time to market, right? So, “Oh, now there's a new variant of transformer that you want to try out,” and because TVM is a compiler is so flexible, then you're much less likely to hit the kind of gotchas that are typically seen in terms of, “Oops! The vendor-specific provided inference engine doesn't support that model yet.” So you have to wait for them to support the 3D version of this model that you're interested in or for more color channels than what they expected people to assume. Yeah, it's much quicker.

And we’ve talked to many users who from after the models trained, getting a model in production is on the order of weeks and often months and even full quarter or two quarters on average. So it's depending on the scale and complexity of the deployment. That's without TVM by the way.

[00:16:34] LC: Yeah. So I just went to – Outside the perspective that a lot of the benefits apart from automation from months to hours really stems from really, really awesome performance, right? So we offer anywhere from 2x, sometimes 30x better performance. And that has a bunch of ways of reaping those benefits, right? One is if you make something 10x faster, you're likely enabling an application that wasn't possible before. It's especially important for edge, for example. If you're doing computer vision in the edge, if you don’t hit a certain frame rates, then it's not ready for deployment. But also you can see that performance gain as a way of saving on costs if you're deploying it at scale in the cloud, right? If you make something 20x faster, use 20x fewer cloud resources, or you can also, as Jason said, since you can use TVM to

easily see how your model runs in a highly optimized way in a variety of different hardware can help you procure what is the best instance in the cloud that gives you the highest throughput per dollar. But everything stems from the fact that TVM is really good at really specializing a model specific harder target and give a lot of performance as a result. It might be worth spending a minute on the Octomizer, Jason. Just to say abstracting TVM away even more, right?

[00:17:51] JK: Yeah, that's a great point. So we've been talking primarily about TVM and its benefits and we've seen TVM be so successful at delivering these three benefits of performance portability and time to market. But we noticed that the one trend, it was that TVM is still a tool kit. And even though the community's done a lot of work and making it easy to use, it's still a relatively advanced piece of software. It's even hard to describe on a podcast, for example. And so we realized there was an opportunity to go and take TVM and its advantages and benefits to an even broader class of developer and data scientist. And so that is what led us to the creation of OctoML and specifically our first product that we call the Octomizer, and happy to go into this more, but the Octomizer is designed to take TVM at its core and make this even easier to use and more applicable in a wider variety of scenarios because we have all the hardware hooked up internally and all the configurations set up for the user and we have all the packaging and build processes hooked up internally. So you literally just get from your model to optimized binary, whether that’s a shared library in a tarball or a Python wheel, binary packaged, ready to go, or even a full Docker image or the GRPC wrapper, whatever format is easiest for you, and you don't need to worry about what CUDA version is being built with, or is that CUDA version compatible with the parameters I passed into TVM and all of these types of things?

[00:19:29] JM: So let's continue to walk through this I think just to refresh people and make sure that we're giving a clear explanation for what OctoML does. So the steps are, first, there's a model that's uploaded. You've got your model in TensorFlow or PyTorch, whatever else. It gets optimized and benchmarked and packaged for whatever hardware platforms you need. And then the performance of the model gets compared across different cloud instance types to

evaluate the requirements of the model. Can you talk a little bit more about that process of evaluating the requirements needed for a model and assessing how it should be deployed?

[00:20:24] JK: Right. Yeah. It varies a lot, and I've been continually impressed at the variety we've seen from user and customer feedback. And some people need, for example, just raw performance and they don't care what hardware it is, what batch size is required. Even accuracy is – They're quite willing to trade off accuracy for raw throughput. All the way to the other side where it's highly accuracy-sensitive. And then there's high latency concerns with regards to models that are in the tight loop of a user request reply type scenario. And so that was one of the foundational principles of what we've built at OctoML is that we need to kind of be flexible enough and get out of the way of what users have the needs for their applications and give them the flexibility they need while automating all the rest of the details away.

And we've designed the experience to be as straightforward and seamless as possible where you upload your model, specify the batch size that you're interested in, and then we optimize for that batch size and model input shape and give you the best possible performance for that resulting batch size and hardware configuration. And so what that then enables the user to do is then determine and do their own sweeps of, “Well, let's try a few different batch sizes. Maybe I'm trying to optimize for throughput and I'm going to try large batch size regimes and maybe try a few options. Compare the resulting performance on various hardware types and then use the data that the Octomizer provides and the performance gains to really eek the most out of the systems that are in the regime of operating that they're looking at.

On the other hand, on the latency-sensitive regime maybe you want to kind of lower the batch size as much as possible while still staying – Or increase it as much as possible still staying under a latency target, but you're going to start from a small batch size and kind of work your way up to increase throughput or lower your costs as long as you're meeting that SLA. And so we've designed Octomizer to be agnostic and flexible to support all these different use cases. And then as we roll out quantization as well, then being able to look at – So that's one point I should make, is that right now all the optimizations that we've been talking about in terms of performance are accuracy preserving. So there's no degradation of accuracy. We’re taking the

exact same computation and just executing it more efficiently, whereas coming soon in our roadmap is actually doing the quantization and accuracy – Lossy optimizations on top of that for users that are willing to take that tradeoff. And then giving those decisions back in the hands of the end user to enable them to compare and take consideration of, “Okay, now I can look at the full Pareto curve of accuracy and performance and let me choose based on my knowledge the application where I'm able to make that tradeoff. And so really putting the power back in the user's hands while automating the details of exactly what depth of search are we doing for any given kernel and what layer fusion depth we're performing in terms of the optimizations in the backend? So those are all kind of hidden from the user.

[00:24:02] JM: And how do you see deployments for machine learning models changing in the near future? Like obviously OctoML can only work on one part of the deployment frustrations at a time, but how do you see things changing in the near future?

[00:24:22] JK: Yeah, it's a great question. There's a couple of interesting trends we're seeing. One is the shift to edge that I mentioned. While cloud deployment still is by far the largest and most common deployment avenue, there's more and more interest in deploying to the edge either in addition to or kind of migrating to the edge directly, and so that I've talked a little bit about the complexities there. Another trend is the kind of increase in adjacent services and needs. So performance and coverage are just two of these. But is their data drift, right? Is the data that I'm seeing and doing inference time after deployment the same or close enough statistically to what I was training on? And am I expecting to get good accuracy as a result of that?

Explainability is another one. There's a great paper about the technical debt of machine learning that goes into the myriad of extra concerns around deployment. And our thought here on this is that we fit into the deployment process in terms of this portability and performance angle. And because we're so flexible, we play nicely with all these other components. So we've already talked with other companies in the area of model monitoring, data drift detection, explainability, and already determined that our solution fits really nicely with these solutions in a complementary way, because if there's an explainability model that you've licensed from

some vendor or a data drift kind of add-on module, then those can either be compiled alongside your model as just other types of computations that spit out kind of various metrics that are ingested by those adjacent platforms or solutions. Or post-deployment, you just plug up – Because the model that we give you, you deploy that in whatever application and realm you want, then you're free to hook up whatever solutions, either homegrown, or third-party provided solutions for all of these surrounding capabilities and needs as machine learning deployment becomes less of a dark art and more of a DevOps type culture or ML ops type culture and all of these best practices become the norm instead of kind of the exception. And so that's how we're positioning. And, of course, we have ideas on how these can be done even better and kind of closer integrated to the solution, but today we're really focused on the portability and performance side of things because there's already a lot to chew.

[00:27:06] LC: Yeah. And just to add one more, a couple more trends there actually. One is it's pretty clear now that applications will have, and some of them already do, a collection of models that they use. There's a model to do computer vision and there might be decision trees. It might be a language model I'll integrate into single applications. So working with ensemble of models well is something that I think is going to grow importance. Now the second one is language model is getting extremely popular. It has implications on a lot of the practicalities of getting these models to production. One, these models are huge. Think of it as hundreds of billions of parameters in some cases. And second, they're much trickier to extract with the performance of then your typical computer vision model. So that's going to bring interesting challenges as they become more and more part of a commonplace application.

[00:28:00] JM: I'd like to go a little bit deeper into the implementation, the technology of the compiler that you've built with TVM. So I guess, I mean, compilation is just very complicated and difficult. Can you just give me an overview for how a machine learning model compiler works?

[00:28:24] JK: Yeah, certainly. So as with most compilers, it's a multi-level process. So starting at the top of the stack, you have the highest level description of the algorithms or deep learning model, and this is on the order of – We call it the graph layer. Specifically for TVM, it's called

the relay level or intermediate representation, and this is essentially a high-level description of the course level computations, and this is on the order of do this convolution with these parameter sizes. Then followed by this batch normalization, then followed by this element-wise, et cetera, right? It's just a high-level description of a series of mathematical computations on kind of Tensor-level objects.

And so there's a number of optimizations that actually occur here like constant propagation, dead code elimination, some types of fusion, et cetera. And so these are done by kind of a series of pattern matching and other operations. And then underneath that, you have a lower level representation. We call this the Tensor IR or TIR level, and this takes those high-level descriptions of the computations comp2d, like literally the string literal comp2d and now breaks that down into the fundamental element-wise scalar operations in a loop nest fashion. So over these dimensions, do a for loop. And at each element, do this single operation and accumulate into this temporary and then write out the end results into this scalar value and as part of this tensor.

And it's at this level that we take that representation of the graph or the computation and then start applying these transformations to either manipulate or rewrite these loop nests into low-level constructs. And depending on the platform that you're interested in, that either goes into Vulcan compute shader or an OpenCL code fragment or x86 LLVM IR or even directly to high-level intrinsics like Vector Intrinsics or or Tensor Intrinsics depending on the hardware platform and capabilities.

And then once you've done that for the entire region of compute that you're looking at, and that's another conversation about what that granularity is, then you compile that using the underlying low-level compiler for that hardware using the OpenCL or CUDA or Vulcan underlying assembler to actually get the resulting executable at the other end machine code. And then that is loaded on the hardware and then actually performance measured benchmark to get the resulting runtime execution time for that fragment of code. And then that data point then is used to train – Well, I guess I should stop there, because that's the compilation flow. And then now you have the kind of flow back of data of how well did that code perform? So

then you can actually choose another way of lowering that code and experiment and using the machine learning guided approach to trying out different flavors of breaking that code down to a lower level.

[00:31:57] JK: Yeah. If you don't mind me jump in. Just one important point here is that, Jeff, whenever you're compiling even just a single operator and even more so you talk about the whole model, there's literally hundreds of millions if not billions of different ways in which you can actually translate an operator into code because you can do a layout in-memory in different ways. You can actually order the loops in different ways, as Jason said, and then you can tile the data structures in different ways. And this gets – Even lead to even larger space when you're talking about tensors with higher rank. We talked about know data structures with many dimensions, right? So the question here is if you had billions of variants of a single piece of code, all of them are valid variants from a correctness point of view. The big question here is how do you pick the fastest one? There's no time to test them all like. Because even if each one of them takes just a second to try, you're talking about billions of seconds of compute. So picking the fastest one is a key question, and you want to do that in a reasonable amount of time. That's where machine learning comes into play, whereas Jason's getting at, is just doing management and building a model of what tends to work, what tends not to work and improving performance and using that to build models to help you navigate this very large space.

[00:33:09] JM: Right. And that sounds really hard. Like building a deep learning or a machine learning-based compiler that's going to be able to iteratively improve the model compilation process. So how do you benchmark the success of that model? So you compile it and then you presumably have to compile it and compile it and compile it again in different kinds of ways in order to have some benchmark for how it actually performs, right? I mean do have to compile it and then execute it and then compile and then execute it again and just go again and again and again?

[00:33:55] JK: That's right. But every time we do it, we create a variant compile and execute and measure its performance. We add the data point in our training sets to build a predictive

model of how well future transformations are likely to do. Such that you can use that as a filter. So if you have pretty high confidence that a transformation is unlikely to yield better performance, you don't even run it. You save that. That's what leads to speed up in the process of exploring this very large space, and this can be significant, right? You can cut down the time to do the whole variant compilation and measurement that takes seconds. You can cut down that to microseconds or maybe hundreds of nanoseconds just by using this machine learning-based filtering that learns from prior executions.

[00:34:40] JM: So I guess the user of the TVM needs to have a training set that they can apply for this application, for this compilation step.

[00:34:51] JK: Not necessarily. Actually, TVM out of the box supports learning this cost model de novo online essentially. And so the initial examples of – Or initial compilations and search will be essentially at random or maybe some slight heuristics and then you'll bootstrap that cost model and search process directly without any training data whatsoever.

[00:35:20] JM: Interesting.

[00:35:20] JK: It obviously you know helps if you do have that training data, because you can get to more efficient kernels more quickly, but it's not necessary.

[00:35:32] JK: That's right. Yeah. The better the data and models that you have to make this prediction of what optimization likely to do better, the faster you get to really good performing really high-performance compiled model. And that's one of the value adds that the Octomizer can offer, is having this dataset just ready to go for the harder targets that customers care about. So that's one of the value adds that we offer. You have this access to this data ready to go if you use TVM via the Octomizer as opposed to ramping up your own use of TVM.

[00:36:04] LC: Also, one thing to note, Jeff, here is that I skipped one step at the high-level IR, which is while we are really excited and see great improvements from doing compilation in TVM, we do realize that there are always going to be areas and situations where the engineers

at places like Intel and NVIDIA have done their homework and written high-performance code that we can leverage out the gate. And so in cases where we can leverage that and we can package those binaries alongside the model, then we can also compare the best code generation of TVM versus the execution of a pre-packaged kernel library from a hardware vendor. And then we can actually choose the best performing one and for each kernel in your model. So you think of a deep learning model as a sequence of computations. Some of those might be faster with TVM code gen. Some might be faster with the prepackaged kernel library. And we can actually do the search over that as well and pick the best performing one. And so get even better performance than just doing code generation for all of them or relying on a kernel library.

[00:37:21] JM: So just to revisit what you said about the machine learning-based approach to iterating on the compiled model. So let's say I get my model, I use TVM to compile it and then I put it into production, and then over time I'm using my model for classification or whatever. Is TVM able to detect over time that there are improvements to be made to the model that is in production?

[00:38:02] JK: That's a great point. So TVM itself is just software that if a user were to download the freshest set of TVM or check out the latest master and then compile it and then try their model again, they might notice an improvement in performance, they might not. And so TVM itself won't do that for you. But uh that's one of the benefits of the Octomizer, is that as we improve TVM and all of the pieces around TVM as part of the Octomizer, then we can continually test. And when there are performance benefits, offer the user, “Hey, your model you uploaded a few weeks ago, we now have a better performing version that's 20% faster. If that's of interest to you, download here.” So that's definitely something we are doing as part of the Octomizer.

[00:38:58] JM: So what is the business model that you guys are pursuing? Obviously this is some really cool technology. I think one of the challenges you see for machine learning tools companies sometimes is what is the best path to monetization. Do you guys have anything in mind at this point?

[00:39:20] LC: Yeah. So it's all around the Octomizer SaaS platform. Essentially being this fully automated machine learning acceleration platform offered as a service with turnkey use and being able to cater to a broad set of users and not just sophisticated machine learning engineers. That's our bets. We already have Octomizer. By the way, we are accepting applications for early access and we have a healthy list of customers to cater to there. So our monetization strategy now is to charge for access to the SaaS platform. And we do have a few uh potential product bets that we have in the works based on customer pool that will likely get there as the platform evolves.

[00:40:08] JK: Also, it's worth noting. In addition to the Octomizer, we're driving another flywheel of improvement both to the TVM ecosystem broadly and back into the Octomizer as well through working with hardware silicon vendors directly in terms of improving TVM's capabilities and performance for their hardware. And so these are companies like AMD and Qualcomm, and we're really excited to work with these partners and more as we continue to improve the TVM ecosystem for them, for their customers and back that helps also in the Octomizer platform directly.

[00:40:51] JM: What are the biggest technical challenges that your team is faced with right now?

[00:40:58] JK: It might be cliche to say this, but I actually think the engineering challenges are the least of our worries just because we have such an amazing team and the ability for the team like to automate difficult things that were once thought impossible continually impresses me. I think, really, just keeping on top of the pace of development of the machine learning ecosystem is probably the main one and that just manifests in so many different ways of the latest quantization scheme, the latest pruning scheme, the latest model types and operators. But those are almost advantages to us because the way we say, we can more quickly roll out support for these quantization and pruning and operators than anyone else out there across more hardware targets at the same time. So it almost benefits us the more quickly the machine learning field evolves.

[00:41:58] JK: Yeah. And I just wanted to add maybe a slightly different way of saying that, is that it's diversity, right? So diversity is growing. More models, more different types of models, fast moving feud. Also, a set of harder targets also growing. But luckily we have two things that allows us to – That gives us confidence that we can be future proof. One is automation. Definitely, automation first in everything we do even what we do internally. Automating the process of onboarding new hardware. Automating the process of dealing with new models and so on.

And then second, the fact that we are very, very committed and we build – Our growth is also related to the growth of the open source community. So the oversized community is a huge amplifying factor of our efforts that's mutually beneficial with other players in the space as well. So let's say that the way we avert the diversity risk is really by automation and working closely with the ever-growing community of developers. And to give you an idea of size, I think the number as of yesterday, we have 471 contributors of Apache TVM, which is remarkable given how – I mean, how young the project is in a number of ways.

[00:43:07] JM: I meant to ask this question earlier, but what language do you write a compiler in these days?

[00:43:16] JK: Yeah, it's a great question. The core of TVM is written C++ and has been from the beginning, but it's written in such a way that it's easy for other languages to plug-in, drive and interoperate with the C++ machinery through a packed function FFI interface. So what that means is that Python, for example, has very deep integration. You can drive the compiler down to individual function calls if you wish, and we often do this for prototyping in order to get quick experiments for compiler passes and um various regions of the compiler and then moving those into C++ once those are stable and tested out.

We also have more and more Rust contributions coming to the project, and we're pretty excited about the interplay between the Rust ecosystem and things like embedded programming and just webstacks and and data stacks and the ability for that to kind of open

up high-performance code to a broader class of developers and integrating into the kind of higher performance runtime scenarios and embedded and lightweight environments.

Wasm and WebGPU is an example this where you can write native or Wasm accelerated applications in Rust today and then you can then compile TVM models to Wasm or WebGPU code or both and then integrate those to your Rust application for deployment to the browser. So that's just one example of how that language integration. But we also have Java JNI bindings and others as well.

[00:44:58] JK: Yeah, I love that question because we have 14 compilers early on in my career as well. And back then was an Assembly and C. And today, it's not which languages. How many languages do we use? We have scripting and we have declarative languages and – Yeah, anyways, it's just kind of interesting how complex this whole ecosystem got.

[00:45:18] JM: Just to wrap up, OctoML is based on an open source project that came out of academia. How does the academic perspective on machine learning compare to the industry perspective? Is there any difference?

[00:45:38] JK: It depends on which area you're talking about within each of those. There's definitely overlap more so probably in this field than many others. It's definitely much more of a – Well, at least people by and large in industry treat machine learning more as a tool of, "Okay, I want to do an application or feature X, Y and Z and my customers are about these aspects." And machine learning is a component that can help me achieve those goals. And so I'm going to start as a base of what I can find the literature for similar applications, similar model types, and then adapt those to my needs and just find the right solution that works, whereas a lot of the academic work in this space, the kind of machine learning systems community, which is the intersection of kind of machine learning algorithm design and actually executing these and large training systems and deployment systems, etc. It's much more, "Okay, how can we better position the ecosystem for building these kinds of things for the future that will enable the end users typically in the industry to deploy them faster? Train them faster? Train larger models, etc. So it's a little bit more focused on kind of setting the foundation in academia, at

least in the machine learning systems field specifically. I guess you could you could make the same analogy with the algorithm design and model design in areas of academia as well versus how can I use machine learning for some customer-driven or business-driven purpose? But they're pulling the same direction. It's just kind of a slightly different window of focus in terms of how much am I willing to invest? In what time frame am I looking for that kind of return on investment? Whether that's longer in the academic sign or shorter in the – And that's a gross overgeneralization. The windows are more overlapping probably than like I mentioned than anywhere else.

[00:47:46] JM: Okay. Well, Jason, thanks for coming back on the show. It's been a while obviously. It's been fun to watch your uh continued entrepreneurial trajectory.

[00:47:56] JK: Thanks, Jeff. Yeah, it's always a pleasure. Happy to be back. Hopefully next time we'll be even further along that trajectory.

[00:48:06] JM: Great. Cool. Well, thanks, Jason.

[END]