EPISODE 1135

[INTRODUCTION]

[00:00:00] JM: The Java ecosystem is maturing. The GraalVM high-performance runtime provides a virtual machine for running applications in a variety of languages. TornadaVM extends the Graal compiler with a new backend for OpenCL. TornadoVM allows the offloading of JVM applications on to heterogeneous hardware.

Juan Fumero works on TornadoVM. He joins the show to talk about the use case for TornadoVM, the design and the engineering that underlies the system. We also talk about the overall Java ecosystem.

We are looking for writers and podcasters for Software Engineering Daily. If you're interested, send an email to erica@softwareengineeringdaily.com.

[INTERVIEW]

[00:00:47] JM: Juan, welcome to the show.

[00:00:50] JF: Hi, Jeff. Pleased to meet you.

[00:00:51] JM: Today we're talking about TornadoVM, which is a system for offloading JVM applications on to various pieces of hardware. Why is this useful? Why do we need to offload JVM applications to heterogeneous hardware?

[00:01:07] JF: Right. So this is a good question. The motivation for running or for having something similar to TornadoVM or having TornadoVM is that many users nowadays need computational power for many applications, like big data application, machine learning, deep learning, this kind of application. And normally those applications could be [inaudible 00:01:28] by using GPU behind on an FPGA. Normally, for deep learning, for example, GPUs goes well as well as Tensorflow course, something like this.

But many of the users are not experts on this type of hardware, because apart from knowing the programming model, you have to know architecture details, complete architecture details. So looking at the most common programming languages now, Java is one of those, right? Java is widely used in academia and industry as well, and it's a good target to [inaudible 00:02:02] applications, compute applications on [inaudible 00:02:05] hardware.

[00:02:08] JM: And TornadoVM is closely related to the Graal JIT compiler. Explain what the GraalVM does.

[00:02:14] JF: Yeah. So GraalVM is a big project, okay? GraalVM is a framework for programming compilers, okay? So it provides an interface to deal with a compiler. It internally has a JIT compiler, just in time compiler. That's a component we use in Tornado. But also has other components like Truffle, which is an API on top of Graal that allows you to run interpreter languages or dynamic language on top of Graal, right? Also, the components, like native images and so on.

So the one we care about in Graal is the JIT compiler, is a compiler framework basically. And one of the things that important for us is that Graal is easily extensible. We extend Graal for generating OpenCL underneath.

[00:03:06] JM: What is OpenCL?

[00:03:07] JF: Yeah, OpenCL, it's a standard by Khronos Group, proposed by Apple at the time. It was 2011. And is a standard for computing with graphics. Basically, you're going to use accelerators, okay? And the standard provides you a set of utilities as the functions that are in the standard in the document, as well as the language within OpenCL kernel to write applications for a Window [inaudible 00:03:35] programming model.

[00:03:37] JM: And why is that important?

[00:03:40] JF: Before OpenCL was CUDA. Actually, OpenCL is kind of the causing of CUDA, if you like. And before CUDA, what people were doing is try to set applications using the graphics API, like OpenGL, or Direct3DX by Microsoft. So that was very, very hard. And instead of doing

that, what these people come along – Actually, one of the main guy was Ian Buck, was one of the creators of CUDA. In his thesis, his PhD thesis, he proposed a Brook language, which is a way of – It's a programming language that allows you to use the computer graphics not only for rendering, but for compute.

[00:04:24] JM: Interesting. Okay. So let's give a little bit of back story for TornadoVM. How did it come to be?

[00:04:29] JF: Yeah. TornadoVM actually started as a project in UK, it's still an academic project, to demonstrate or tried to demonstrate how to accelerate applications using Java. So one of the main goals was to accelerate application for a computer graphics and computer vision, it's called key fusion. Basically, you have a Kinect Microsoft camera, and you have to bind a lot of Java code around – This application contains around 7,000 line of Java code. And various groups that were working together, one group was working on how to set those applications using C, C++, OpenMP, another group with CUDA to do it in Java. And to do so, inside, they built Tornado, and that's what Tornado actually come from.

[00:05:25] JM: So can you tell me more what kind of use case, like a high-level use case would be appropriate for TornadoVM?

[00:05:32] JF: Yes. Actually, Tornado can be used like any other applications you use for GPUs. It can be suitable for Tornado as well. That means that a matrix multiplication, deep learning, machine learning, which those applications you use intensively matrix multiplication, but also computer graphics, computer vision. Also cases for fin tech, cases for healthcare systems, natural language processing, artificial intelligence. Those are the cases that are well-suite for Tornado.

But that's one aspect, which is use Tornado to program GPUs. But Tornado can be used also to program FPGAs, which is another technology, another type of hardware. And FPGAs are very good at exploiting pipeline parallelism. Instead of data parallelism, pipeline parallelism. All the type of applications like digital signal processing are very good to exploit in those hardware. And Tornado can target also those.

[00:06:31] JM: So, multi-core CPUs, GPUs, field programmable gate arrays, FPGAs. Could you just tell me about like what kinds of applications need to get deployed to these kinds of devices?

[00:06:44] JF: Yeah. We're talking about some examples, machine learning, deep learning, these kind of applications. As well as data-intensive applications. For example, a physics simulation. We have a use case for embody computation. Basically have a set of stars in a galaxy or planets. Given the position and velocity of each planet, the star you want to simulate in next step. And that's very computational intensive. And you have a huge amount of data. And that's a very good case, or those are very cases for GPUs.

[00:07:17] JM: We talked a little bit CUDA and OpenCL as heterogeneous programming languages. What is a heterogeneous programming language?

[00:07:24] JF: Yeah. Traditionally, we use CPUs, censor processing units, and that has been the way until 2005, '6 kind of. And later on because GPUs came along as a compute unit as well, not only for rendering, but for compute. GPUs are different architecturally than CPUs. So you don't have an homogeneous system. You have a heterogeneous system. Actually, the GPUs contains – It's all memory, instructions and so on. How they execute instructions are different. How they are programmed are totally different, and the same with FPGAs basically.

[00:08:04] JM: And so CUDA and OpenCL, they're not actually languages that people program in. People write higher level things on top of them?

[00:08:12] JF: Right So many engineers actually use CUDA and OpenCL. I would say CUDA more than OpenCL. But you'll require an expert programmer here. Because apart from the programming model, which can be learned obviously, to make good use and to make the most of performance, you have to know architectural details.

For example, I can give you a quick sample. So on GPUs, you can copy data to L1 cache, if you want. That's not possible on CPUs. But as is the responsibility of the programmer to copy data there. But L1 cache is not coherent. So the programmer has other barriers. And you can imagine all these complexity coming along. When you have one thread, two threads, you can feel it out. But those GPUs normally run thousands of threads in parallel.

And this has an extra level of complexity. So you have to understand the model and you have to understand the architecture. Even more, people programming in CUDA, normally, they optimize for one GPU for one vendor. Even if the same vendor, you change that GPU. Most likely you have to tune again your application. Okay. So we want to avoid that with Tornado basically.

And the same applies with OpenCL. So OpenCL as I say is like a cousin of CUDA. It's very similar, the programming model. The language is a bit different. Obviously, function calls are different. But how you program is very similar. And, again, you have to know architecture details like this, or how to deploy threads, things like that.

[00:09:45] JM: Tell me more about what gets built on top of a language like CUDA.

[00:09:50] JF: In what sense?

[00:09:52] JM: Well, so what kinds of applications are written on top of CUDA or OpenCL?

[00:09:57] JF: Right. So we're talking about the same type of applications as Tornado targets as I said in the beginning. Now, it's mostly used for deep learning and artificial intelligence. But you can program pretty much any compute in any data-intensive application as soon as you comply with the restrictions that CUDA has and OpenCL has as well. So we're talking about, as I say, artificial intelligence, machine learning, deep learning, chemistry, physics simulation, fin tech, these kinds of applications.

[00:10:26] JM: So why would I do this in the Java environment? I frequently hear about Tensorflow and PyTorch as being the primary programming models for doing deep learning. What would I want to do in the Java environment?

[00:10:42] JF: Yeah. So because of this complexity of programming those hardware, GPUs, and FPGAs, and you can think about another type of hardware coming in the future. Most likely, you will want to have more type of specialized hardware in the future, we are tending to specialized, all these type of pipelines in hardware.

And having an application and then trying to rewrite application with another programming model, another primary model and try to make it work, it takes time. So instead of doing that, we want to abstract all of these and make a layer up, right? We go up. And right just what Java does. Write once, run everywhere. So we want to have the same philosophy, basically. The idea is that just write your application in Java. In Java world, you should not think too much about the actual hardware is going to run. This is a runtime underneath. This is going to take care of how to compile that for the actual hardware, for the perfect hardware, for actual GPU and FPGA and how to manage data, how to deploy threads, how to deal with L1 cache and so on and so forth.

[00:11:54] JM: So I wanted to use TornadoVM today, what would I have to do?

[00:11:59] JF: Right. So right now, what you need is the driver, the OpenCL driver. So Tornado generates OpenCL for you. So if you want, I can talk a bit about Tornado just to get what you need.

[00:12:14] JM: Sure. Please do.

[00:12:16] JF: Tornado, the input for Tornado is a byte code, the Java byte code. And the output is the OpenCL code. And then we need an OpenCL driver underneath. So what you need is an OpenCL driver. If you're running – If you have an Nvidia GPU, you can have like the driver for Nvidia, plus the CUDA runtime. If you have OpenCL device [inaudible 00:12:39] Nvidia, which is the same tool that you need. Or you have an Nvidia device. You will need a driver plus the AMD SDK for OpenCL. For Intel, you would need something similar as the Intel runtime for CPU, or the Intel driver for the integrated GPU. Apart from that, you only need Java. That's all.

[00:12:58] JM: Okay. Let's talk about the stack in a little more detail. So TornadoVM at the top level, I'm defining tasks. I'm doing annotations. I'm running the tasks on task schedules. Give me a description of what's happening beneath the API.

[00:13:16] JF: Yes. So let's talk about the tasks actually. The programmer writes tasks, and you can group many tasks together in what we call a task schedule. Each task is basically a Java method. You want to accelerate. What's happening underneath is that the user writes those tasks, those task schedules. We compile with javac. There is no modification in the language.

It's just a library. And then when we'll run the application, we get the task schedules, the group of tasks. And then we start in the TornadoVM side. We'll start building the OpenCL programs. To do so, the first thing that we do is to build a data flow graph. Basically, we analyze how data is flowing across tasks. If you have a few upgrades coming in, a few upgrades coming out, how many arrays can be reused in the middle? We build a graph to figure out these things.

Once we have this graph built, what we do is to optimize it. So we say, "Ah! This variable or this buffer can be reused on copy twice," because GPUs and CPUs have different memories of spaces. Basically, that means that the user should allocate memory at the beginning on the device, then perform at it a transfer, and then run the kernel. Tornado does that for you automatically. And the goal is not to copy more arrays or more variables than you need. So that's the goal of the data flow analyzer.

After the data flow is optimized, what we do is generate internal byte codes. That's more engineering side. Basically, those byte codes, for us, as a runtime, as a compile engineers, to make it modular, let's say and easy to write. Basically, each byte code is an operation between a CPU and a GPU. For example, we have one code that copy the data, copy in from CPU to the device. For example, from the CPU to the GPU. We have another byte code to say launch a kernel, which is launch a task, which is related to launch a method. And the first time we launch a method, we compile from Java byte code to OpenCL. Once we have the compilation done, we manage memory. We allocate the right buffers, the right pace, and then we'll run the application for you. That's the overall process.

[00:15:42] JM: The TornadoVM byte code, is that the same as Java byte code?

[00:15:46] JF: No. No. No. TornadoVM byte codes are very – That's totally different. TornadoVM byte code is just a subset of the operations that we want to execute. As I say, for example, copy in one buffer from CPU to GPU, add a barrier to wait for the results. Launch one kernel. Those are Tornado byte codes. Actually, we have a set of maybe 10, 11 byte codes.

[00:16:12] JM: Okay. And how does the TornadoVM byte code get translated into JVM byte code?

[00:16:18] JF: Right. So the whole runtime is implementing in Java. So for the JavaVM, the Tornado runtime is another Java application. The only thing that the Java application, Tornado in this case, has the ability to also compile methods. That's the only thing.

But the byte codes that Tornado generates, they're never going to be executed on the device. They only run on the host, which means on the CPU. So if you're running with open JDK, for example with C1 and C2 which you run multiple times, the actual runtime, Tornado runtime, is going to get compiled with C2. If you run with Graal, the actual runtime is going to get compiled with Graal.

[00:17:02] JM: Okay. Well, let's use your canonical example, which is matrix multiplication. So I program a matrix multiplication using the TornadoVM. What happens when I execute that matrix multiplication?

[00:17:15] JF: Yeah. So going back to all the steps I provide before, which can be a bit confusing. So the user writes a matrix multiplication and then creates a task schedule and then there is a method in the API, in the Tornado API that says execute. So once you call execute, we build this data flow graph I talk about. Matrix multiplication has two inputs, matrix A and matrix B. I want to open matrix C.

So the data flow optimizer will say, "Ah! You have two inputs. One output. That's all." Then we generate byte codes for that, Tornado byte codes and we say, "We have to variables to copy." So let's copy first one. Copy in for the first. Copy in for the second. Then we launch the kernel, which is the matrix multiplication. And then we copy back the result. Copy out C variable. And that's what TornadoVM executes.

So the first time we execute the byte code launch, we call Graal to compile from Java byte code to OpenCL. So we take the methods that represent matrix multiplication. All the bytes codes [inaudible 00:18:27] multiplication. We call Graal, the Graal compiler, and then we optimize that code to generate the OpenCL for you.

[00:18:35] JM: Can you give a more complex example?

[00:18:38] JF: Yeah. One complex example could be MapReduce, that you have multiple tasks. So we have one task to do a map operator. Map means you have a bunch of – An array as an input, and you apply a function for each element of the input datasets. And your output is exactly the same size, but the function applies for each element.

And then you have a reduction, which means that the previous operation that has an array, that reviews all elements into a single one. For example, a sum, okay? In this case we build that data flow. And let's say you have two variables, A and B, then you have some intermediate value, which is let's call it C. And then you have an output. But the output's color value. So let's call it D.

So the data flow will build these data flow graph. We say copy A and B, because that's your input data. And then D will say, "Let's keep it on the GPU. You don't need these variables on the host again, on the CPU again. So you can leave it there. So you don't have to transfer back the result."

And TornadoVM will say, "Transfer data. Copy in A or B." Then you have to allocate a space for C obviously with the intermediate value. We allocate the data. We don't copy anything. We keep it there. We launch the kernel, the first one. And then, immediately, we launch the second kernel. Taking the results from the previous operation. And then we only transfer back a single element at the end. That's data flow a bit more complicated.

And then from that – I mean, there is no limit of the task, the amount of task you want. You can program. You can have 100 tasks if you want to. The actual limit will be the code we can fit on the device basically. Because, underneath, we generate OpenCL.

[00:20:33] JM: Okay. Can we take a step back here for people who have gotten lost at this point? What are we actually talking about? What is TornadoVM? What value does it add?

[00:20:42] JF: Yeah. So TornadoVM is an easy to accelerate your applications without worrying about how or details. And the user, at the end of day, you don't have to worry about these byte codes. Nothing like that. This is just how it works internally. So the value that we add is that the input application is something easy to understand. You write a serial code. So if you program

code on OpenCL, you have to write parallel code. You have to think in parallel. Meanwhile in Tornado, you write serial code. Each task is a serial code. Then Tornado will parallelize it for you basically.

So the user only needs to think what do I want. Not how do I want to be performed. Tornado will manage how do you want basically to get maximum performance. And that's the main value we add. Another thing is Tornado can dynamically change the device. So because now we have plethora of multiple devices, we have one CPU, one GPU or multiple GPU, perhaps an FPGA. So, now, where to run the code? Now it becomes a mess, right? Because I want to maximize performance. But Tornado that has a mode that can figure out which one is the best to achieve good performance, better performance.

[00:22:07] JM: And how much better performance are we talking about?

[00:22:10] JF: This depends on the device you run. But typically, we're talking about 1, 2, or 3 orders of magnitude faster than JavaVM compiled with C2 GraalVM. For example, DFT application is a typical application for digital signal processing. It's a very simple program. It's just [inaudible 00:22:30] loops and a bunch of calculations of square roots and things like that.

And you can get easily 1000, 2000X compared to Java C2 compiler, or GraalVM. Most of the performance because of the power of the GPU, right? Of the device in this case. Because we use Graal and we compile with Graal, we can optimize things further that you don't see statically, like roll the loop, and then evaluate constants and so on. So we can get also a bit faster in that sense. But normally, the speed up comes from the device.

So just to sum up about your question. So on GPUs, we get around two to three orders of magnitude on FPGAs. Essentially, it's harder to get performance from OpenCL. But in some cases, we're talking about one to two orders of magnitude in the best cases scenario.

[00:23:23] JM: You mentioned that this is a good fit for pretty much any data-intensive applications. So let's say I've already got a data-intensive application written. Is there a way to port it to the TornadoVM?

[00:23:34] JF: Yeah. We're actually working with some of the partners we have in the project to do this. They have already existing applications and they're going to port it to TornadoVM. As soon as you comply with TornadoVM restrictions, we don't support the whole Java, okay? We happen to support a subset. It's possible to run it with TornadoVM. Yes. But it really depends on the input applications.

[00:23:58] JM: So the output of TornadoVM, is it a Java program or is it a C program? What is the lowest level execution situation?

[00:24:09] JF: The output of TornadoVM is an OpenCL C program source code, which means that we will need another compiler to compile to the final binary. But because we execute OpenCL with OpenCL programming model, that's part of the actual driver in OpenCL. So we pass the kernel we generate to the driver, and then we'll do the final compilation.

So that's for the branch or for the code we have now in public. But now we're working on another – It's an extension, actually. So instead of going to OpenCL, we go to PTX, okay? It's the machine code, assembly code kind of, it's ASTRA code for Nvidia devices, okay? Parallel execution throughout this code. And in this case, we also need the final driver, the final JIT compiler in the driver. But it's lower level. So PTX is like close to assembly code instead of C code. Yeah.

[00:25:06] JM: Tell me more about the task scheduling layer of TornadoVM.

[00:25:10] JF: Yeah. The task schedules are light API we provide to identify the parallelism. So one of the things we do is to exploit parallelism. We don't detect parallelization. That's a hard problem. So we need a way to identify which methods the developer wants to accelerate. We do that via the task schedules. And the task schedule, in my opinion, is a very simple API. You say compose a task and you pass a pointer or a reference to the method you want to execute. Okay? You just define the class. and the methods, and you get that reference. And then you pass the input-output parameters for that method.

And then you can concatenate. You can compose as many task as you want. Task. task. task. task. in one task schedule. Then the benefit of having multiple tasks in one task schedule is that

all the task are going to be compiled in a single compilation unit You can have multiple task schedules, and each task schedule execute one task. In that case, we are going to generate one code, one big OpenCL compare task schedule. Or you can combine all of them in one basically. You have flexibility for doing that. Yeah.

[00:26:22] JM: So TornadoVM can be used to execute languages other than Java. Can you explain why that's useful?

[00:26:30] JF: Yeah. Actually, I started these as a part of my PhD back in 2013. We actually – It's not in Tornado, but potentially could be. We execute our code directly. The benefit is that some of these languages have a potential of not really accelerate functions. R, for example, the basic type is a vector. And there are many, many operations with vector types. And that's a good fit, very good fit for accelerating with GPUs. So instead of calling a native library, you just call your functions as you normally do. And then underneath will be a compiler that can run on a very fast hardware, like a GPU. So we do the same in Tornado through the Truffle framework. Yeah.

[00:27:19] JM: The Truffle framework. What is that?

[00:27:21] JF: Yeah. So Truffle is a compile framework on top of GraalVM basically, as a Graal compiler. So basically it's a framework that allows you to write AST interpreters, abstract syntax tree interpreters. And Graal knows the structure, the structure of a Truffle ASTs and it say, "Well, to efficiently compile Truffle ASTs into machine code." Basically, if you want to use Graal compiler from another language and efficiently, if you want to implement your own language, you might use Truffle AST framework. They have been implementing many languages. I think R, Python, Groovy, JavaScript, NodeJS, Scala I believe, LLVM. I think they have a port for LLVM as well.

[00:28:08] JM: Can you tell me some of the industrial use cases for TornadoVM?

[00:28:13] JF: Yeah. So TornadoVM is part a European project. It's called E2Data, and there are many partners there. Some of them are from industry. So we have a very nice use case from healthcare. It's a company in London called EXUS. What they do is to – Well, let me give

you a little background over the problem, because it's a nice problem. So in UK, we have the NHS, basically the healthcare system. And there's a problem when a patient goes to a hospital and apparently there's a probability that the same patient is going to readmitted later on for some whatever reason, for conditions, health conditions, age, whatever.

So what this company is doing is try to predict if a patient is going to be readmitted. Yes or no? In a short period of time in the same hospital. Because the NHS system can reschedule resources accordingly. Imagine now with the coronavirus. Well, that's another situation. But upfront, you can make decisions. Let's put more resources, more doctors in that hospital, because that hospital are going to receive more patients shortly.

So what they did is to apply machine learning. I think underneath, they're doing some select logistic regression. It's an application fully written in Java. And they're accelerating their training phase of this application. So, so far, what they have got is they have accelerated the sequential application between Java, the training phase from 2500 seconds to 180 seconds. So that gives you an speed up of 14X, 15X by running on GPUs automatically.

[00:29:58] JM: That's pretty impressive. And can you tell me more about the tool chain for TornadoVM?

[00:30:04] JF: Yeah. The tool chain, you can use just your favorite editor if you like, IntelliJ [inaudible 00:30:12], Eclipse. And you can use exactly the same tools as Java has, same debuggers, etc. Because at the end a Tornado application is just a Java application. It's also integrative with Graal compilers. So if you know Graal and you want to go deep, you can even see their compile graph. You can interact with a compile graph and so on. So it's fully integrative in that environment.

So the user writes an application using these editors or terminal and compiles with normal javac. And then to run it, you just run Tornado. But actually, Tornado becomes – You need to type a lot of parameters to run Tornado. It's just a Java program with a lot of flags to enable Tornado. But this is still a Java program. So it's a normal workflow as another Java program. The only thing we tell, use this OpenCL library. This is located in this path. Enable Tornado with these options, and that's all.

[00:31:06] JM: Are you the lead programmer of TornadoVM?

[00:31:09] JF: Yes. I'm currently the lead developer of the project. Yes.

[00:31:13] JM: What has been the process for developing and maintaining it? Can you tell me some about the engineering process?

[00:31:18] JF: Yeah. So I was hired because I mentioned a bit earlier, I did my PhD in similar topic. I also took Graal to accelerate Java programs on GPUs only. And then I took a step further to accelerate our programs automatically. So no need for external libraries. And then at the end, when I was finishing my PhD, I heard about this project and then I contact with my current bosses now Christos Kotselidis. And they told me that they're opening a position because they have a European project to work in something very similar. A big broader, because Tornado is being integrating into one of the frameworks called Apache Flink. But the idea was something similar. That excited me, because it's an opportunity to apply what I learned from my PhD into Tornado. Since then, what we do is – What I did in the beginning was try to get the tool up to speed. More test cases, unit testing, more integrations with Graal, more compiler phases, etc. More maturity. Better recommendation. That has been the process.

And then because Tornado is an academic project, right now, we have to – It's a balance, right? We have to find a time to find new ideas for papers, academic papers and so on. And another half of the time to integrate those papers into TornadoVM to mature the tools, to mature the workflow, to mature the projects that the students do inside TornadoVM. So it's a bit – Yeah. Basically, it's exciting to have to see both sizes actually.

[00:32:57] JM: Can you tell me more about the GraalVM ecosystem?

[00:33:01] JF: Yeah. You mean, how do we use it with Tornado? Or in general?

[00:33:06] JM: What are some other use cases for GraalVM? We did a show about it a while ago, and I think we also did a show on Quarkus. I'm just curious about 2020 state of Java applications.

[00:33:18] JF: Right. So, GraalVM, the usage right now – I think the best use is to run high-level programming languages on top of Graal. Like the Truffle languages basically, R, Ruby, Scala, especially Scala. Because all these technology, you get a lot of speed up for the applications. Also [inaudible 00:33:38], as you mentioned, the Quarkus project for native images is something that is well-received in the community.

And I don't really know the projects, honestly, directly using Graal apart from us as well. There's a small group, I think, in Boston for a small talk. Another implementation, another Truffle language as well. Yeah.

[00:34:04] JM: So when I execute a task scheduler, the steps are optimizations are applied. Then there's JIT compilation, and then there's execution. And then the Java heap gets updated with the results overtime as the parallel OpenCL threads execute. Am I understanding that workflow correctly? Basically, the execution actually takes place across these multiple threads and OpenCL. And then the results of that are being rendered in a Java heap?

[00:34:37] JF: Yes, correct. Yes. Yes. So the user writes a single thread application. And then we launch the kernel, we pass from single thread into multiple thread application. Actually, we launch depending on the input size, but likely thousands of threads for the same applications. But those threads run on the device. Okay.

Then we sync with the variable again, the results again, from the GPU, the device heap into the Java heap. Again, that's correct. And then from that point, again, it's single thread.

[00:35:14] JM: As you've open sourced TornadoVM and got an input from other people and industry, what kinds of changes have you made to it?

[00:35:22] JF: Yeah. A love that question. So, since we opened, one of the reasons to open is to get feedback. And we have been receiving quite a bit of feedback. I remember last year I gave a talk at JVMLS, and received a feedback, like, "Why don't we generate a PTX? Well, the main reason for OpenCL was because with the same – Because on standard, we can target multiple devices. But we could do that. And then we did it. So it's in the progress, as I said,

before we are developing the PTX backend. That was one of the things that we hear from the community.

Another things we're heard from the community is to improve the programming model. How to improve? To get more code inside Tornado? So we have with us in the European project Gary Frost. Gary Frost is the creator of Aparapi. Basically, it's a similar tool to Tornado to compile Java applications down to OpenCL and then you run.

And he's giving us quite a bit of feedback about how to get more code for TornadoVM. And we are integrating those comments right now. Actually, every week, we are updating Tornado. We're now releasing an official number, Tornado 7.8 or something. But every week, we push changes. Most of the changes came from the feedback right now.

[00:36:51] JM: You mentioned in an article that Microsoft is using TornadoVM? Can you explain that use case?

[00:36:57] JF: No. What I meant is the Microsoft Kinect. It's a use case that we have. But I don't think Microsoft is using Tornado. I'm not aware of that at least.

[00:37:07] JM: Oh I see. Okay. So you have somebody using a Kinect, which is kind of a device that follows you, tracks you, and as you move in space, in real-time. And you can have simultaneously localization and mapping, which is just – It's like scanning your movement, your body movement and accelerating the frameworks per second that it's watching you. That's the use case that TornadoVM has been applied to. Why does that make sense? Explain why that's a good use case of TornadoVM.

[00:37:40] JF: Yeah, because in this case, it's an application that's specifically designed for computer vision. And internally, we have a lot of matrix operations and stuff. So that's why matrix operations and deep learning, we use matrix multiplication and stuff. A very good fit for GPUs in this case. Because GPUs were designed for rendering, and you have a bunch of X, Y, Z color, X, Y, Z pixel in this position, and you apply texture and you apply geometry to those pixels basically. And those operations are quite heavily vector operations and matrix operations.

And the GPUs normally do these operations in floating point. And they not only run one thread, or 10 threads, they run thousands of threads on this. So for these reasons, GPUs are a good fit of these type of applications for rendering computer graphic, computational photography, these kinds of stuff.

[00:38:38] JM: Where do you expect to take the project next?

[00:38:41] JF: Well, we're moving by October, November this year once we release the PTX, and that will be our next target. We already released a prototype for running ARM processors. We are moving also towards cloud. So we can run out-of-the-box Tornado on Amazon Cloud, on GPUs and FPGAs, Xilink FPGAs in this case. And we're moving hopefully by mid next year. We'll have complete results. Some of this is I mentioned briefly before. We're integrating Tornado into Apache Flink.

So Apache Flink is a MapReduce framework for batch processing and extreme processing. Mainly focused on extreme processing. So you imagine that you have these framework running, this application running for hundreds of thousands of nodes in a cluster. So each application is compute-intensive whether you have in the final node. So if the node of a server that is going to run your computation contains GPUs and FPGAs, the idea is to switch and flow the computation over there.

So we're moving towards having an application [inaudible 00:39:50]. There is no modification in the language. There is no modification in the API. Nothing. You just run plain Flink, Vanilla Flink, and automatically get accelerated on those nodes by plugging in Tornado. So we're working this project right now. It will be the next step for us.

[00:40:08] JM: Tell me a little bit about your background. How did you end up working on this?

[00:40:12] JF: Yeah. So briefly mentioned, I did my master's at the University of La Laguna in Spain. I did it in a project. It was a coincidence actually. I was working my first job in a lab in super computer, basically, sys administration. But my boss at the time, I was studying the master. Meanwhile, I was working. And I wanted to do the final project on operating systems, but my boss at that job told me I need people, I need persons, I need developers to do this

compiler. I said, “Well, I’m looking for a master project. It could be good.” But, actually, in my mind I was, “Okay. I’m not sure.” My mind is in operating system, not in the compiler. But I like it.

And the project was to support open ACC directives. We implement a compiler for open ACC basically. And I like it. And at that time I started playing with GPU, with CUDA, with OpenCL, and the more I play, the more I like it. So then I moved to PSD in this topic, and I’m doing this with Graal and Java, and that’s what is connected now with what I’m doing in TornadoVM.

[00:41:23] JM: Any broader predictions for how the Java ecosystem is going to develop in the coming years?

[00:41:30] JF: Yeah. Most likely, I’m not the right person to ask this. Most likely, someone in Oracle, Red Hat. But I’m going to give you my vision where I would like to be instead. So, the philosophy of right ones run everywhere. We are not there yet, because we have many devices we cannot run automatically. And that’s why we push this idea of having heterogeneity inside a VM, inside a Java VM.

What I’ve seen now is that for every new hard application, we tend to – I mean, we as a community, researchers, industry, tend to implement now a specific hardware for one particular device. For example, there’s a company in California, Silicon Valley, implementing this huge chip. I mean, huge, literally, because it’s a whole entire area, to apply for machine learning basically. And Microsoft in combination for that company is implementing its own chip for machine learning, on natural language processing. We see similar things for Tensorflow. And Google implemented Tensorcourse mainly around machine learning, but could be any other application in mind, okay? Machine learning, because it’s a hot topic. What I mean by this is that it seems like for every new hard problem, we tend to build new hardware. With new hardware comes new programming models and new tools. And we have a lot of different tools disconnect with each other, a lot of different programming models. And we should do sometimes the other way around. So let’s converge it. Let’s try to make a unified model to run everywhere. And that’s our approach now.

[00:43:12] JM: Well, thank you so much for coming on the show and talk about TornadoVM. It’s been a real pleasure.

[00:43:16] JF: Thank you for having me. It's a pleasure.

[END]