**MIT CSAIL FastCode Seminar: Yuanming Hu – 12/07/2020**

2
00:00:07.049 --> 00:00:12.780
Julian Shun: Hey everyone, welcome to the FastCode seminar. So, today our speaker is going to be Yuanming Hu.

3
00:00:14.040 --> 00:00:22.260
Julian Shun: Yuanming is a fourth year PhD student at MIT CSAIL working with professors Fredo Durand and Bill Freeman.

4
00:00:23.070 --> 00:00:32.490
Julian Shun: And his research is focused on compilers and high-performance physical simulation with applications to graphics and artificial intelligence.

5
00:00:33.420 --> 00:00:41.190
Julian Shun: That leads to the design and implementation of the Taichi programming language which he's going to tell us about today.

6
00:00:41.850 --> 00:01:02.160
Julian Shun: Yuanming also received many fellowships for his work, including the Edward Webster fellowship, a SNAP Research Fellowship Adobe research fellowship and a Facebook Research Fellowship. So I'm very excited to hear about Yuanming's work today on Taichi. So I'll turn it over to Yuanming.

7
00:01:02.670 --> 00:01:08.040
Yuanming Hu: Thank you so much Julian for having me here. And for the kind introduction. Let me try to share my screen.

8
00:01:09.900 --> 00:01:10.230
Okay.

9
00:01:12.270 --> 00:01:17.130
Yuanming Hu: So welcome everybody. And today I'm going to talk about the Taichi

10
00:01:17.190 --> 00:01:18.000
Yuanming Hu: Programming language.

11
00:01:18.360 --> 00:01:32.250
Yuanming Hu: There are two main topics of this talk, of course, we'll do a brief intro and after the intro, we're going to talk about spatially sparse computation and then differential programming. I'm Yuanming Hu from MIT CSAIL and I work with Fredo Durand and Bill Freeman here.

12
00:01:33.390 --> 00:01:34.500
Yuanming Hu: So what is Taichi?

13
00:01:35.520 --> 00:01:42.030
Yuanming Hu: Well, Taichi is a high performance programming language designed for spatially sparse and differentiable computation.

14
00:01:42.390 --> 00:01:51.780
Yuanming Hu: And it's actually a project with a very long history. I started this project when I was an undergrad back in the year 2016 and it got a very limited success.

15
00:01:52.500 --> 00:02:04.740
Yuanming Hu: Because at that time, it was still a computer graphics library and it was used by roughly 20 people around the world to publish four papers, and then in January 2019,

16
00:02:05.610 --> 00:02:17.010
Yuanming Hu: we started the conversion from a library to a standalone programming language so that we can get higher performance and more productivity and we published two papers along this thread and

17
00:02:17.520 --> 00:02:23.340
Yuanming Hu: One is at SIGGRaph Asia on spatially sparse computation for stimulation. The other one is for differential programming.

18
00:02:24.810 --> 00:02:38.520
Yuanming Hu: So one of the main application field of Taichi is for high performance physical stimulation. Here I demonstrated simulation with over 100 million particles simulated only single immediate GPU using Taichi and

19
00:02:39.420 --> 00:02:46.350
Yuanming Hu: This one we're using a 4K by 4K by 4K sparse grid which will be a ridiculous amount of memory, if we do everything.

20
00:02:46.590 --> 00:03:01.830
Yuanming Hu: In a dance manner, but we have a specialty sparse programming system that allows us to allocate space in only consuming regions, which I'll talk about later. And normally this solver only takes around 50, sorry, 500 lines of code, and you can get the code on GitHub.

21
00:03:03.480 --> 00:03:14.160
Yuanming Hu: So there are two missions of the Taichi project. The first mission is to explore novel language abstractions and compilation techniques so that we can accelerate and make video computing more productive.

22
00:03:14.790 --> 00:03:23.850
Yuanming Hu: We're actively exploring new research opportunities using Taichi so that we can publish more papers, of course. And the second mission of Taichi

23
00:03:24.450 --> 00:03:39.330
Yuanming Hu: Is to pragmatically simplify the process of high performance computer graphic development or deployment because you know computer graphics applications tend to have a lot of dependencies, and writing a high performance graphic system is not easy and we wish Taichi to

24
00:03:41.160 --> 00:04:00.090
Yuanming Hu: solve this problem. And it's actually already a pretty popular project on GitHub. Now we have nearly 7000 commits and over 10K stars and over 1K pull request from 56 developers. And we also have achieved over 400K downloads from people all around the world.

25
00:04:02.010 --> 00:04:16.410
Yuanming Hu: So, at this time, Taichi is a high performance domain specific language. It is embedding Python and we have JIT Compiler reading super fast so that we can somehow translate our Python code into very high performance.

26
00:04:17.040 --> 00:04:23.550

Yuanming Hu: excipital kernels on CPUs, or GPUs and Taichi is designed for computer graphics applications.

27
00:04:23.880 --> 00:04:37.470
Yuanming Hu: We really, really care about productivity and portability. We want one Taichi program written once in around everywhere, and we want every Python programmer capable of writing Taichi programs, after watching a one hour tutorial video and

28
00:04:38.280 --> 00:04:46.050
Yuanming Hu: The major computational pattern in Taichi is what we call it data oriented Palo megakernels, and this is very different from other systems such as TensorFlow or Pytorch.

29
00:04:46.230 --> 00:04:56.490
Yuanming Hu: Because in TensorFlow or Pytorch your kernels tend to be the customizable kernels in TensorFlow tend to be a very lightweight kernels, with very, very low arithmetic intensity but in Taichi

30
00:04:57.120 --> 00:05:03.090
Yuanming Hu: You can consider all the operations are fused into a single kernel that can execute very, very efficiently with GPUs.

31
00:05:03.540 --> 00:05:11.160
Yuanming Hu: We also try to decouple data structure from computation, which is an idea from the Highline programming language that allows us to somehow leverage

32
00:05:11.550 --> 00:05:22.500
Yuanming Hu: The modern memory architect memory hierarchy and modern computer architecture, so that we can get the best data structure working for a specific problem only specific architecture.

33
00:05:23.430 --> 00:05:40.620
Yuanming Hu: And notably we provide something we call spatially sparse tensors and users can use them as a tensor that is just dense. I'll talk about this in greater detail. And because deep learning is so popular. And we're also supporting two federal programs I will cover this later.

34

00:05:41.760 --> 00:05:51.930
Yuanming Hu: So the goal is to improve computer graphics R and D productivity and let's look at what a Taichi problem look like. Here's a fractal computation program. And it's usually the first

35
00:05:52.740 --> 00:06:04.200
Yuanming Hu: Taichi program for a lot of people. As you can see, we're just computing a very simple per pixel computation. And here, here's a Taichi program and it looks exactly the same

36
00:06:04.770 --> 00:06:14.490
Yuanming Hu: As Python but there are only some very small tricks. So, in the very beginning, you have to import Taichi STI. And then you initialize Taichi you allocate your fields.

37
00:06:15.270 --> 00:06:19.980
Yuanming Hu: Here we are allocating a 2D pixel array and then you can define your computational kernel

38
00:06:20.910 --> 00:06:31.050
Yuanming Hu: Here we have a function that computes the square of a complex number. And we also have a kernel that automatically paralyzes over all the pixels and

39
00:06:31.740 --> 00:06:42.180
Yuanming Hu: In the kernel. We can do whatever you want. You can do while loop you can do if branching, you can do a lot of fancy computation here and in the main program, you can just call your kernel

40
00:06:43.470 --> 00:06:50.220
Yuanming Hu: Python, and this is just normal Python code, you can interact Taichi with whatever Python library, you would like to interact with.

41
00:06:52.260 --> 00:07:07.740
Yuanming Hu: Although the front end of Taichi is in Python we do have a kind of involved middle end and back end implementation in superclass. And we have a we have our own intermedia reputation and now we support I think seven back ends and

42
00:07:09.450 --> 00:07:23.820

Yuanming Hu: Basically we support LLVM and that goes to CPU and CUDA via intermedia PDFs and we also support source resource to C 99 and of course OpenGL and Metal that allows Taichi to actually run on mobile phones.

43
00:07:25.110 --> 00:07:33.750
Yuanming Hu: For more information on this kind of low level front-end compiler on longtime design implementation, I do have a very different talk which I call the life of a Taichi kernel.

44
00:07:34.620 --> 00:07:41.520
Yuanming Hu: So if you're interested in low level or internal implementation of Taichi, you can feel free to check out that link. For this talk. I'm just going to

45
00:07:43.170 --> 00:07:48.960
Yuanming Hu: Talk about some very, very, very high level ideas and what makes Taichi work and what makes it, what makes it more

46
00:07:49.980 --> 00:07:52.170
Yuanming Hu: Productive compared to other systems.

47
00:07:53.610 --> 00:08:02.250
Yuanming Hu: So in today's talk, I'm going to talk about the first class sparsely data structure in Taichi and this is the paper we published in SIGGraphAsia 2019 and

48
00:08:03.450 --> 00:08:17.190
Yuanming Hu: In this part we somehow decouple data structures from computation. And we also provide a syntax for people to compose sparse data structure primitives into very, very complex sparse data structure that have different properties on different problems.

49
00:08:18.450 --> 00:08:31.350
Yuanming Hu: We also provide incrementally domain specific data access optimizations on Taichi IR in the Taichi compiler so that users don't have to worry about the complexity with English data structure because the compiler will just use the majority of the work for you.

50
00:08:32.850 --> 00:08:39.840

Yuanming Hu: We also will also talk about the differential programming part in Taichi. And this is a paper we published in ICLR 2020 and

51
00:08:41.640 --> 00:08:48.180
Yuanming Hu: In this work, we propose a two scale … system that is very tailored for writing defensible physical simulators.

52
00:08:49.320 --> 00:08:52.410
Yuanming Hu: So for the first part, let's just talk about sparse data structures.

53
00:08:54.480 --> 00:09:07.650
Yuanming Hu: As you know, SIGGRAPH every year has a lot of physical simulation papers and here's a MPM simulation example where we cut a lot of animals such as armadillo or here's a bunny here and of course you can cut cheese.

54
00:09:07.740 --> 00:09:20.010
Yuanming Hu: You can stir a few million sand particles in a bowl so that you get a fancy SIGGraph pattern and get to get the reviewer happy about your submission.

55
00:09:21.150 --> 00:09:25.290
Yuanming Hu: So here's another kind of simulation work at SIGGRAPH where you have to

56
00:09:26.370 --> 00:09:38.070
Yuanming Hu: simulate finite animals and then do you topology organization here in this work. We are optimizing the internal material distribution in a bird beak structure so that the bird beak can be

57
00:09:38.520 --> 00:09:48.840
Yuanming Hu: more practical for a bird to eat anything. We want the bird beak to be as stiff as possible with respect to x and all those here, you're, you're seeing the optimization process.

58
00:09:51.240 --> 00:10:02.820
Yuanming Hu: As you know, in simulations. People really tend to use very, very high resolution, like, here we are using a 3K by 2K by 2K grid. And if you count all the active boxes

59
00:10:03.330 --> 00:10:11.040
Yuanming Hu: Those are the actual boxes there. We actually use in the simulation. It's over 1 billion boxes and we do have to do a lot of low

60
00:10:11.790 --> 00:10:21.540
Yuanming Hu: Low level performance engineering to actually make this kind of simulation fit in a single workstation with I guess if I remember correctly, that's 512 gigabytes of memory.

61
00:10:22.140 --> 00:10:29.370
Yuanming Hu: And the code is not present, you have to write a VX instructions to do very low level multi grade optimizations to make the simulation converge faster.

62
00:10:30.270 --> 00:10:42.840
Yuanming Hu: So the nasty trade off here is if you want, really, really high resolution, then the simulation is just going to run ridiculously slowly in the bird beak optimization example, it took us one whole week for the simulation to finish.

63
00:10:43.860 --> 00:10:53.190
Yuanming Hu: But if you want performance, then you have to write very low level code, either in superfast or CUDA, and this kind of treat is obviously really bad you can't get

64
00:10:54.060 --> 00:11:02.610
Yuanming Hu: Both performance and productivity simultaneously, because if you go to the high level programming system, you get productivity. But the program runs very slowly. If you

65
00:11:03.120 --> 00:11:13.920
Yuanming Hu: Do low level programming, then usually the productivity isn't going to be great. So the question is, how do we get here for simulation program specifically, is there a way for us to actually

66
00:11:14.640 --> 00:11:29.820
Yuanming Hu: Get both productivity and performance. The answer is yes. And we should definitely leverage domain specific abstractions that are specifically to simulation and then we can somehow design a more tailored system for applications.

67

00:11:32.550 --> 00:11:47.130
Yuanming Hu: So here's a 3 billion particles simulation simulator using the material point method and render using path tracing using both the simulator and renderer are returning Taichi. So there's a very interesting thing here, which is

68
00:11:48.180 --> 00:11:59.850
Yuanming Hu: Although we have to allocate a very large volume, volume for the whole simulation domain, but we actually use a very small part of it, which means that region of interest is a tiny fraction of the whole body volume.

69
00:12:00.180 --> 00:12:14.910
Yuanming Hu: And this is definitely one property that we need to leverage. So let's call this spatial sparsity which means regions of interest only occupy a very small fraction of the whole body volume. And there are a lot of sparsity

70
00:12:15.960 --> 00:12:20.490
Yuanming Hu: There a lot of types of sparsiity and for different types of sparsity people invent different types

71
00:12:20.970 --> 00:12:33.840
Yuanming Hu: Of data structures for sparse matrices. They're already a lot of data structure in a lot of research on this route actually and most notably is the Taco paper the tensile algebra compiler work and we're

72
00:12:34.860 --> 00:12:47.430
Yuanming Hu: Efficient computation on this kind of sparse matrix operation can be effectively optimized. But in this work in the Taichi work we're focusing on spatial sparsity which is kind of different from

73
00:12:48.870 --> 00:13:02.340
Yuanming Hu: General sparsity because our system, sorry, because now the system. Now the active box or distribution in our simulation are usually globally sparse but locally dense. So this is kind of a

74
00:13:02.700 --> 00:13:18.150
Yuanming Hu: special property of physical simulation, because if you have something here, then with very, very high probability you're going to have a different particle surrounding it. So this kind of sparsity makes one makes a lot of optimization makes sense. For example, blocking and tackling.

75
00:13:20.670 --> 00:13:30.060
Yuanming Hu: So in graphics. People have invented a lot of fancy sparse data structures. One notable example is the VDB video structure which is invented by Museth in the year.

76
00:13:30.510 --> 00:13:35.940
Yuanming Hu: 2013 I guess the data structure was actually invented way before this, but they get the paper published in that year.

77
00:13:36.420 --> 00:13:44.190
Yuanming Hu: And essentially, it's a data structure that is very similar to the B tree. And it's a shallow multi level sparse box of grids.

78
00:13:44.880 --> 00:13:54.120
Yuanming Hu: And the other very interesting little structure is the sparse page grid or people call this SP grid and it is a even shallowerer squared system where people

79
00:13:54.600 --> 00:14:03.120
Yuanming Hu: Where developers utilize virtual memory system of modern computer architecture, so that they can use the TLB as a cache as the page table and

80
00:14:03.720 --> 00:14:19.260
Yuanming Hu: They view the memory system the virtual memory system as a hash table and of course a lot of other fancy tricks, such as Morton coding and bitmasks so that they can record both the topology and the data very efficiently and improve memory access spatial locality.

81
00:14:21.150 --> 00:14:29.850
Yuanming Hu: So here's a simulation in Taichi. And usually people will leverage this kind of multi level sparse box or data structure in here.

82
00:14:30.150 --> 00:14:41.820
Yuanming Hu: At the top left, you see some water pouring and on the top right, it's the utilized one by one by one boxes in lower left is four by four by four in lower right 16 by 16 by 16 you can see

83

00:14:42.390 --> 00:14:57.600
Yuanming Hu: The boxes the three boxes pictures that are actually a 3D tree like structure. It's something similar to actually but usually for to reduce memory directions, people use a very much shallower tree, which means a much higher ranking factor.

84
00:14:59.370 --> 00:15:07.860
Yuanming Hu: So the idea of sparse data structure is very straightforward. And it's kind of a magical that you can allocate memory as well. So you use it.

85
00:15:08.520 --> 00:15:20.880
Yuanming Hu: You use the memory in the part that only the part that you actually need. However, in reality, using sparse data structure is actually it's not easy, actually, it can be pretty hard because there are a lot of complex

86
00:15:21.780 --> 00:15:29.370
Yuanming Hu: Complex conditions such as boundary conditions, you have to take care of take care of the boundaries. We have to maintain the data structure topology.

87
00:15:29.700 --> 00:15:38.880
Yuanming Hu: You have to do memory, a memory management because whenever you're writing to a inactive box so you have to allocate that you have to do personalization load balancing.

88
00:15:39.180 --> 00:15:45.450
Yuanming Hu: And I have to say the most daunting thing here is the data structure overhead. So let me talk about data structure overhead.

89
00:15:46.230 --> 00:15:50.250
Yuanming Hu: So ideally, when someone is using data structure, they would actually expect

90
00:15:50.820 --> 00:16:00.600
Yuanming Hu: Something like 90% of time spent on essential computation and then 10% of the time on data structure overhead. However, in reality, it's usually not the case.

91
00:16:01.050 --> 00:16:11.580

Yuanming Hu: It's often the case that you actually spend 90% of your clock cycles on just accessing data structure and then 10% only central computation and sometimes even worse. Why is that

92
00:16:12.210 --> 00:16:24.990
Yuanming Hu: Well, if you do some a little bit of counting, you will find that when you're using a specific structure, you have to use something like hash table look up, which is a 10s of clock cycles and CPU or even over 100 clock cycles and GPUs.

93
00:16:25.350 --> 00:16:30.990
Yuanming Hu: And there are a lot of memory direction because your data structure is dynamic. There are pointer look ups and

94
00:16:31.890 --> 00:16:38.790
Yuanming Hu: That usually gives you something like cache messages or TLB messages And you have to use known allocations. And by doing that, you have to

95
00:16:39.720 --> 00:16:54.480
Yuanming Hu: There has to be some kind of locking or synchronization or some other complex and time consuming operations and of course there are branching on CPU that that means mipredictions on GPU. That's what divergence, neither of them are good for performance.

96
00:16:55.710 --> 00:17:04.410
Yuanming Hu: Of course, it is always possible that you can do low level engineering to reduce the data structure overhead. But once you start doing that you will find that the productivity

97
00:17:05.070 --> 00:17:10.260
Yuanming Hu: Is going to reduce because low level engineering takes time, in the worst thing is that

98
00:17:10.920 --> 00:17:23.970
Yuanming Hu: By start doing low level engineering your data structure starts to couple with the algorithm and it will make it very, very difficult to explore different data structure designs and find the optimal one because the cost of redesigning a data structure. It's just so high.

99

00:17:25.500 --> 00:17:31.770
Yuanming Hu: Here's an example of data structure access versus actually a computation in here.

100
00:17:32.490 --> 00:17:39.390
Yuanming Hu: If you do some benchmark, you'll find that in this two level data structure where we have a hashtag on the top and a bit mass array in the middle.

101
00:17:40.320 --> 00:17:50.790
Yuanming Hu: Allocate accessing one element that can actually take you over 50 clock cycles and only more than CPU is super scaler and AVX doing the actual stencil computation, it's actually

102
00:17:51.420 --> 00:18:01.110
Yuanming Hu: Less than 0.5 clock cycles per element. So the funny, funny thing here is without low level engineering sometimes, actually in most cases,

103
00:18:01.470 --> 00:18:16.770
Yuanming Hu: Dense data structures are often faster for problems with not extreme sparsity, just because compilers can optimize your dense data access much better than optimizing sparse data access and dense data access overhead is much lower than sparse ones.

104
00:18:18.210 --> 00:18:33.600
Yuanming Hu: So, here, here. I actually visualize the process of accessing a dense array in here. If you have a dense data structure accessing one element is super easy. It's just a re-addressing that will not take you more than I guess

105
00:18:34.050 --> 00:18:43.740
Yuanming Hu: Two or three instructions on x86. However, if we have a sparse data structure things will get much harder and here as you can see, starting from the root.

106
00:18:44.250 --> 00:18:52.500
Yuanming Hu: You have to first access the hash table and hash table access is not cheap. It's actually very costly and then you do the re-addressing.

107
00:18:53.130 --> 00:18:59.730

Yuanming Hu: If have a second access, you have, you have to access the hash table once again and then do the array addressing

108
00:19:00.360 --> 00:19:17.280
Yuanming Hu: So, as you may have noticed we have accessed the hash table twice. If at compile time we can somehow deduct this can redundant access, then we can achieve higher performance. So actually reducing redundant access is one of the keys to high performance spatially sparse computation.

109
00:19:18.480 --> 00:19:30.000
Yuanming Hu: So that's why we designed the Taichi programming language. So at a very, very high level, we try to decouple the the structure from computation. And we do have to set up languages, just as many other

110
00:19:30.480 --> 00:19:40.260
Yuanming Hu: Speech Language, especially highlight. So we have a computational kernel, which is the imperative computational language that somehow you can define for example here, a 2D Laplacian stansell here.

111
00:19:40.860 --> 00:19:54.840
Yuanming Hu: And we have a data structure description language that allows developers to design a hierarchy of data structure. By the way, this was still in the past from 10 and now we have completely migrated to Python to make the language, even easier to use.

112
00:19:55.920 --> 00:19:57.780
Yuanming Hu: And then we have a system and

113
00:19:58.800 --> 00:20:04.740
Yuanming Hu: Data structure access optimization system actually doing these this time domain specific optimization is super important.

114
00:20:05.100 --> 00:20:14.820
Yuanming Hu: And our runtime system handles Palo Alto polarization and of course memory management and by combining all this together we achieve kind of

115
00:20:15.360 --> 00:20:22.950

Yuanming Hu: Good performance boost. Like, if you look at for entry and benchmark the sample against manually optimized code.

116
00:20:23.670 --> 00:20:30.150
Yuanming Hu: All of the benchmark baselines are manually optimized and that's why they're ridiculous like they use shared memory.

117
00:20:30.780 --> 00:20:37.890
Yuanming Hu: For for the MPM example and for the MTP CG people do a lot of operator fusion and

118
00:20:38.700 --> 00:20:50.040
Yuanming Hu: Do a lot of free conditioning to make it run faster and of course do some blogging at Harding to make memory access better. But even if we're comparing against those manually written

119
00:20:50.940 --> 00:21:01.410
Yuanming Hu: baselines are Taichi system can run over four x faster and they'll force because the compiler does all most of the optimization work and the programmer can code.

120
00:21:02.580 --> 00:21:07.140
Yuanming Hu: All the spots systems data structures as if they are dense our code is much, much shorter

121
00:21:08.670 --> 00:21:11.220
Yuanming Hu: Defining computation intention is super easy. You can just

122
00:21:12.750 --> 00:21:19.110
Yuanming Hu: Write computation as if every data structure is Dennis in here. Where do you find a different stance or so.

123
00:21:21.210 --> 00:21:34.890
Yuanming Hu: There are quite a few interesting things here. So the first thing is that also hear you envy can be specialty sparse. We are programming on them as if they are dentists and we're also supporting Paolo for loops that allows us to somehow leverage the

124
00:21:36.030 --> 00:21:47.550
Yuanming Hu: Streaming multi processing on GPU in which is pretty similar to kuda or SPC, it will look over only active elements in the spouse to structures. This is super important because this allows us to

125
00:21:48.000 --> 00:21:57.300
Yuanming Hu: Skip the reading that we don't really care about. So, and we also support complex control flow, such as F and while actually here's a retreats are written in Taichi

126
00:21:57.810 --> 00:22:16.260
Yuanming Hu: Is a body match with retreats are running on GPU and because it runs very fast. Users can just take the parameters on the fly. But the point here is really tracers have a lot of looping brown chain and this kind of a very complex control flow and teacher, you can indeed handle that.

127
00:22:18.420 --> 00:22:28.380
Yuanming Hu: Describing the instructors in Taichi is not hard. So we only have 16 instructor primitives, you have something we call structure notes and then they can combine very easily and describe the the structures.

128
00:22:28.680 --> 00:22:35.370
Yuanming Hu: In just two lines of code, you know, those kind of deconstruct libraries used to take people, thousands of lines, lines of code to right but now

129
00:22:35.670 --> 00:22:44.940
Yuanming Hu: With this kind of a data structure DESCRIPTIVE LANGUAGE we can very easily. This design this kind of data structures and replicate their performance and

130
00:22:46.800 --> 00:22:58.740
Yuanming Hu: The more interesting thing here is that we can actually take features of different data structures we like recombining them and invent our own data structure that is more tailored for different problems. Here we are taking

131
00:23:00.000 --> 00:23:11.820
Yuanming Hu: The tree structure from the DB and the modern Cody and bit mass feature from SP grid, and then we can get something we call SPV DB, which is a founding director of that somehow leverage is that the good words of

132
00:23:13.710 --> 00:23:15.120
Yuanming Hu: Tools to data structures.

133
00:23:16.170 --> 00:23:25.260
Yuanming Hu: Here's a simulation only bounded sparse greatest structure in were smashing it snowball on the ground and as you can see there's a very undesirable boundaries. The simulation domain.

134
00:23:25.770 --> 00:23:32.880
Yuanming Hu: However, without changing much code. We have very easily switch the data structure from the bounded those rupture to the outbound data structure.

135
00:23:33.240 --> 00:23:41.100
Yuanming Hu: And stimulation. This is incredibly useful because sometimes people just don't want to see your particles hitting the simulation domain which is not natural.

136
00:23:42.720 --> 00:23:48.240
Yuanming Hu: So the key to achieving high performance in to enable programmers to write code.

137
00:23:49.260 --> 00:23:53.700
Yuanming Hu: Freely is our access simplification in the techy compiler and

138
00:23:54.630 --> 00:24:04.230
Yuanming Hu: Here's a here again, here's a workflow of how attached. You can always compiled from compiled from Python to back ends and we have a lot of domain specific document.

139
00:24:04.590 --> 00:24:11.970
Yuanming Hu: Optimization within the workflow, so that we can somehow leverage the data structure information and do a lot of access optimization

140
00:24:13.470 --> 00:24:17.550
Yuanming Hu: So the idea of our access simplification is actually very, very simple. So

141
00:24:18.600 --> 00:24:24.990

Yuanming Hu: We actually only do two things. So, first thing is to instead of representing all data access as a

142
00:24:25.530 --> 00:24:37.320
Yuanming Hu: Entry and access we break down the whole pieces of data structure accesses into smaller pieces and finer granularity. Then we get something we call micro access instructions and then we can just use

143
00:24:38.010 --> 00:24:45.750
Yuanming Hu: A magazine that is very, very similar to traditional common sub expression elimination to simplify the instructions that we get a extremely

144
00:24:46.530 --> 00:24:52.830
Yuanming Hu: More efficient or simplified micro access set of instructions. Here's a visualization of

145
00:24:53.490 --> 00:25:02.010
Yuanming Hu: What are accessing the big it actually does. So here we are having on the left we are having a bunch of a optimize the accesses and we have

146
00:25:02.430 --> 00:25:11.580
Yuanming Hu: A blue axis, a red axis and the orange axis and by breaking down those accesses into smaller pieces, we can just eliminate the redundant accesses and then

147
00:25:12.810 --> 00:25:21.240
Yuanming Hu: On the right we have the optimized accesses and some of the access can even be inferred at compile time being a maybe a few

148
00:25:21.900 --> 00:25:30.810
Yuanming Hu: Buys away from a previous actors, then you don't need to even they traverse the legal structure. You only have to compute one pointer and offset it to get the other pointer.

149
00:25:31.380 --> 00:25:38.670
Yuanming Hu: We also include other optimization, such as duty lightning shared memory will see on GPUs. We also avoid unnecessary data.

150

00:25:39.120 --> 00:25:55.050
Yuanming Hu: Activation which is for sparse boxes that are inactive when you are reading it. We are also supporting better vector eyes load on CPUs because own even with ABS or effects to the gathering of instruction is much, much more extensive than a vector eyes load.

151
00:25:56.580 --> 00:26:02.790
Yuanming Hu: So compared to the hand optimize baselines, we're getting pretty promising results so

152
00:26:03.990 --> 00:26:10.800
Yuanming Hu: What is the reason for hyper higher performance at a very high level, there are three reasons. The first one is we can do

153
00:26:11.940 --> 00:26:20.310
Yuanming Hu: Interesting index analysis in second one is instruction granularity. And the final one is the data excellent antics. Let's look at the example in the following slide.

154
00:26:20.790 --> 00:26:31.560
Yuanming Hu: Here we're comparing a we're looking at a financial analyst. And so, and here we are trying to load a backer from a spouse way so that we can

155
00:26:32.040 --> 00:26:45.300
Yuanming Hu: Do some kind of FM current stance or Colonel computation. Here we're loading boxes 1234 and as you can see initially they are, by the way, this is still pseudo code, the actual IRS started more complex, but

156
00:26:45.750 --> 00:26:58.110
Yuanming Hu: Here I'm just using a easier to to read version here. So as you can see here the initial IR has for load instructions and then we are making a vector. Out of the four load. The load results.

157
00:26:59.250 --> 00:27:12.450
Yuanming Hu: So after lower axis we are breaking down each access into two pieces were first getting the block for boxer one and then for walks along we get the box. So from the block. So you see

158
00:27:12.930 --> 00:27:25.740

Yuanming Hu: Instead of just representing one data access as a MTN access. We're breaking it down into two levels. And first of all, is to get the four by four blocks. And then we get a walk so out of the four by four block.

159
00:27:27.630 --> 00:27:31.800
Yuanming Hu: Then we do some index analysis here as you can see, we're just

160
00:27:33.150 --> 00:27:41.190
Yuanming Hu: Plugging in all the indices and then we'll get we can get hopefully get more information. So, and we stayed a structured information we

161
00:27:43.500 --> 00:27:43.800
Yuanming Hu: Sorry.

162
00:27:46.350 --> 00:28:02.880
Yuanming Hu: And we still structure information we can do some kind of influence, because here we know the block size is 16 and then we can just a computer block ID to be something like I over 16 or I pass one over 16 and by using integer division property, which is a

163
00:28:04.020 --> 00:28:11.010
Yuanming Hu: Very funny thing here. We can just replace i plus one was i plus zero, because we know it's going to be

164
00:28:11.760 --> 00:28:19.110
Yuanming Hu: We know i is going to be a multiple of four and we're dividing the result of the ad by 16 and adding

165
00:28:19.530 --> 00:28:31.770
Yuanming Hu: 123 and divided by 16, where is the model for doesn't change with result. So we can safely do this and then we can do. Come on, somebody, somebody especially elimination. So now, get rid of

166
00:28:32.880 --> 00:28:38.520
Yuanming Hu: Three or four instructions and then for the box office. We can do the same thing. Now, resuming and we see

167

00:28:39.630 --> 00:28:43.650
Yuanming Hu: We're just a computing three or four boxes.

168
00:28:44.700 --> 00:28:49.260
Yuanming Hu: Out of a block based pointer and now we can

169
00:28:50.280 --> 00:29:01.800
Yuanming Hu: Do with influence. We know that all the boxes. They're just neighbors of each other. So there are continuous piracy memory and then we can just issue a single vector I slowed to drastically improve the load performance.

170
00:29:03.390 --> 00:29:09.810
Yuanming Hu: So that's for the index analysis. I think one of the more important thing is the IRA Glenn Lowry spectrum. And as you can see

171
00:29:10.170 --> 00:29:14.910
Yuanming Hu: A lot of system just represent accesses in MTN manner they do something like x ha

172
00:29:15.210 --> 00:29:22.920
Yuanming Hu: And there are actually a lot of choice to represent this kind of data structure access and you can do layer level wise access as we just did just now.

173
00:29:23.160 --> 00:29:31.050
Yuanming Hu: You can do the techy arrow thing which is slightly finer granularity computer to layer wise access. You can also repeat that everything is El de mar or even machine code.

174
00:29:31.890 --> 00:29:38.220
Yuanming Hu: So the thing is there's a trade off here and by breaking down your code into smaller pieces.

175
00:29:38.820 --> 00:29:49.290
Yuanming Hu: Your code is hardly allies, but it actually exposes slightly more about optimization opportunities by using a very, very coarse high level intermediate representation

176

00:29:49.590 --> 00:29:57.750
Yuanming Hu: You're going to hide a lot of optimization opportunities and but that's just maybe easier to analyze. So we really want to

177
00:29:58.200 --> 00:30:08.640
Yuanming Hu: Achieve a trade off between getting more optimization opportunities and we want our system still easy to analyze. So that's why we picked it out to be in this sweet spot.

178
00:30:09.450 --> 00:30:18.750
Yuanming Hu: So our IR design has first capital structure Ayers, which will cost structure, no trees, and if you are interested in this, you can check my other talk, which is the life of

179
00:30:19.080 --> 00:30:29.670
Yuanming Hu: Her know that PDF file and there I dive into this structural treaty line. But here, the important thing is that the data structure our entire sky is the first

180
00:30:30.090 --> 00:30:38.340
Yuanming Hu: first class citizen, so that the compiler knows what it's doing. And the compiler can do to my specific understanding and optimization of the data structure I are

181
00:30:39.690 --> 00:30:51.930
Yuanming Hu: We also have a set of hierarchical SSH competition I are so we make a hierarchical instead of CFD plus basic ballsy IBM style I are just because we want to keep in the loop information and in the FEMA

182
00:30:52.440 --> 00:31:00.930
Yuanming Hu: Load example we just saw the loop information here which tells us that i is a multiple for it's actually critically important for a lot of data access optimizations.

183
00:31:02.040 --> 00:31:16.200
Yuanming Hu: We, we do a progressive lowering from the ST all the way down to back end executable code and for most some backhands we just in time compiler via VPN. The others fail sources or its completion.

184
00:31:17.460 --> 00:31:29.880

Yuanming Hu: So here's a pain. Colonel and the hour looks like this. Pretty, pretty involved. But you see, it's a pretty similar to, it's actually lower level than CUDA but it actually has a similar structure.

185
00:31:31.440 --> 00:31:40.020
Yuanming Hu: Again hierarchical SSA stereotyped Paolo differential. I'm going to talk about the fungibility later and first kind of data structure accesses

186
00:31:42.210 --> 00:31:55.200
Yuanming Hu: And our data structure accesses has a kind of trivial assumptions or it's semantics. For example, there's no point or aliasing and you know point or ideas and can really stop a lot of compiling compiling optimization from happening. So

187
00:31:56.130 --> 00:32:03.570
Yuanming Hu: X, Y and big title, you will never overlap, unless a is the same field sp. So this kind of a

188
00:32:04.080 --> 00:32:10.590
Yuanming Hu: Pointer aliasing analysis is much easier in touch the computer to in a more general purpose language where you have pointers pointing everywhere.

189
00:32:11.190 --> 00:32:20.640
Yuanming Hu: And all memory accesses are down through this kind of index sorry field plus index semantics, you always use a small squared plus I set up indices

190
00:32:21.000 --> 00:32:30.540
Yuanming Hu: To do memory access. There's no other memory access can modify the state. So this furthers give some further information for the compiler to do the optimization

191
00:32:31.980 --> 00:32:38.130
Yuanming Hu: As the only way data structures can get modified is to write access, which means read access doesn't modify anything

192
00:32:39.600 --> 00:32:39.990
Yuanming Hu: And

193

00:32:41.220 --> 00:32:45.360
Yuanming Hu: No memory allocation during read no exception if read is out of range.

194
00:32:46.560 --> 00:32:58.620
Yuanming Hu: However, we do have a debug mode that actually detects this kind of other autobahn axis. So the overall design for this data structure access semantics is to enable compile optimization and make programmers life easier.

195
00:33:00.030 --> 00:33:08.340
Yuanming Hu: So to wrap it up. We just now describe a system that allows us to achieve both productivity and performance. So by

196
00:33:09.240 --> 00:33:12.300
Yuanming Hu: Developing a data structure abstraction, we allow the programmers.

197
00:33:12.900 --> 00:33:22.410
Yuanming Hu: To use a spouse to structures as a player dense so that for the majority of physical simulationtasks. They don't have to worry about sparsity or changing valuable agreed anymore that allows

198
00:33:22.800 --> 00:33:33.360
Yuanming Hu: That makes your life much easier and to compensate for performance we propose a set of abstraction specific compiler optimizations that works very well for text based application domain.

199
00:33:34.830 --> 00:33:43.290
Yuanming Hu: And by the company. I was in the front data structures were finally able to achieve slightly a little bit more performance and a little bit more productive, just because

200
00:33:45.240 --> 00:33:57.780
Yuanming Hu: The users can then explore data structures rapidly and then now they're guided by benchmark performance and changing only the structure to the other one is super, super easy. Just a few lines of code change actually

201
00:34:00.630 --> 00:34:09.570
Yuanming Hu: So that's for the first part. Do you have any questions at this point, maybe I can take one or two in case I just go too fast and people lose me completely

202
00:34:12.510 --> 00:34:13.350
Yuanming Hu: If not, yeah.

203
00:34:13.440 --> 00:34:23.940
Julian Shun: I have a question. So, so I'm wondering how, like, how does the user find these optimizations. Is it all done automatically like using auto tune or something.

204
00:34:25.170 --> 00:34:25.800
Yuanming Hu: Um,

205
00:34:26.970 --> 00:34:39.300
Yuanming Hu: So I think there are two things to follow for the data access optimization is done by the compiler and for looking for the optimal data structure, usually at this point, we still doesn't have a

206
00:34:40.380 --> 00:34:44.070
Yuanming Hu: Auto tuner at this point. So usually people just a

207
00:34:45.270 --> 00:34:59.130
Yuanming Hu: design space, it's not so huge. It's just usually something like to level data structure resist three level data structure and for the leaf level and for all the intermediate level, you haven't determining, you have to determine. I brought in factor and

208
00:35:00.300 --> 00:35:05.760
Yuanming Hu: At this point, we're still doing all the tuning manually, it's not auto tune

209
00:35:07.200 --> 00:35:15.000
Yuanming Hu: And we're, we're in the planning to somehow develop a auto tuner so that people can be even more simplified. That's a great question.

210
00:35:15.570 --> 00:35:16.860
Julian Shun: Great, thank you. So

211
00:35:17.700 --> 00:35:18.720

Saman Amarasinghe: I have a small question.

212
00:35:20.190 --> 00:35:32.010
Saman Amarasinghe: Start with Python. How do you deal with the tire type system. How do you deal with dynamic types. When you go to you. I have you resolve types it or yeah

213
00:35:33.090 --> 00:35:45.300
Yuanming Hu: That's a good question. Actually, we were not entirely sure if everything's doable to switch a system in Python. But it turns out to be very doable. So Python is a dynamic has dynamic types.

214
00:35:46.320 --> 00:35:57.270
Yuanming Hu: However, it's perfectly doable. That when translating Python to Taichi, we look at the St. And then we somehow do type inference and make everything static so

215
00:35:58.230 --> 00:36:11.670
Yuanming Hu: The techy kernel language, which is a very similar language of Python and it's actually possible by the Python posture posture. It's strongly typed. So this allows our compiler to actually do the thing. Otherwise, a compilation will be super hard.

216
00:36:12.090 --> 00:36:16.050
Saman Amarasinghe: It's very interesting, because we are doing this seek language, which is also

217
00:36:17.100 --> 00:36:23.400
Saman Amarasinghe: Very similar. So it might be interesting to see and compare some notes and see where we did this, we can learn from each other.

218
00:36:24.240 --> 00:36:31.980
Yuanming Hu: Yeah. That will be great. Yeah, actually I do. I do have some some slides and maybe I should show them after the talk.

219
00:36:32.400 --> 00:36:40.950
Yuanming Hu: And that's all know how how he us how tight on the ice is translated to Taichi and I'm super interested in talking about this.

220
00:36:41.040 --> 00:36:46.080
Saman Amarasinghe: I didn't talk to me because I want to get the see people also to listen to this thing here.

221
00:36:46.740 --> 00:36:50.970
Yuanming Hu: Okay, okay. Yeah, we can maybe get you another talk the other day.

222
00:36:52.320 --> 00:37:06.540
Yuanming Hu: Okay, yeah, thanks for mentioning that I think I always enjoy working with different people to learn, learn new ideas. So okay, so let me continue talking about differential programming for integration.

223
00:37:07.800 --> 00:37:08.220
Yuanming Hu: So,

224
00:37:09.420 --> 00:37:14.580
Yuanming Hu: My master's thesis was actually on defense for physical simulation of simulations and here's a

225
00:37:14.970 --> 00:37:27.480
Yuanming Hu: One work with it, which we call chain Queen and we named this simulator, king, queen, because first of all you can evaluate the gradients union chain rule. And it's called king queen, because the chain rule is just going to be ridiculously

226
00:37:28.920 --> 00:37:42.540
Yuanming Hu: Boring and airport and I, it took me quite a long time to get it right. So it's a queen for for for the applications of channels. So one funny interesting one. Very nice application of this kind of different

227
00:37:49.020 --> 00:37:51.300
Julian Shun: Oh, sick. Sick. We lost

228
00:37:53.280 --> 00:37:53.610
Yuanming Hu: Is

229
00:37:54.390 --> 00:37:55.350

Julian Shun: After they can tell you

230
00:37:55.650 --> 00:38:00.840
Yuanming Hu: Because TensorFlow is designed for teachers and the computational power is not really suitable for

231
00:38:01.290 --> 00:38:02.430
Julian Shun: Later, because it's good.

232
00:38:03.630 --> 00:38:04.050
Yuanming Hu: So,

233
00:38:05.220 --> 00:38:11.670
Yuanming Hu: That's why we develop this kind of auto dev system. We didn't actually as soon as you can see is a relatively

234
00:38:11.700 --> 00:38:12.330
Tiny part

235
00:38:14.460 --> 00:38:27.990
Yuanming Hu: Actually works pretty well. So the project we call default, it does this kind of defendable programming protection and this allows us to empty and optimize neural network controllers using gradient descent.

236
00:38:29.490 --> 00:38:36.960
Yuanming Hu: Here we are. Here we are showing three different differential simulators and their control using neural networks and

237
00:38:37.410 --> 00:38:47.100
Yuanming Hu: In the past, people have to use reinforcement learning, which is not so efficient way of doing the optimization, but now with Steve Taichi, we can directly evaluate the gradients using alternative and get higher performance.

238
00:38:48.480 --> 00:38:53.970
Yuanming Hu: So a lot of people have questions between differential program and deep learning. So what are they, they're actually very, very similar.

239
00:38:54.510 --> 00:38:59.220
Yuanming Hu: The high level idea is to evaluate the loss function gradient which with respect to a certain

240
00:38:59.760 --> 00:39:10.680
Yuanming Hu: Set of free parameter, so that you can learn or optimize using gradient isn't actually deep learning is a special case of differential programming, but differently, it gets so popular today so people know about deep learning, but

241
00:39:11.730 --> 00:39:16.410
Yuanming Hu: For defensive programming. There are relatively few people that has a better understanding

242
00:39:17.400 --> 00:39:26.670
Yuanming Hu: So there although they're actually similar, but there are some differences when people are talking about deep learning. They usually focus on a set of

243
00:39:27.300 --> 00:39:35.280
Yuanming Hu: relatively limited set of operations that are provided by TensorFlow or other deep learning systems, one example being the conclusion colonel and

244
00:39:36.630 --> 00:39:44.310
Yuanming Hu: You know kuda has immediate has this kind of optimized Cody and operated that is highly optimized for those frequently used deploy any operations.

245
00:39:44.850 --> 00:40:01.170
Yuanming Hu: And when talking about differential programming, people usually expect the program to be slightly more diverse and more exotic. For example, you might want to do stanzas or gathering scattering or fine grained Bronte and loops and those are actually pretty important in simulation, but

246
00:40:02.730 --> 00:40:05.610
Yuanming Hu: In testicle or pie chart is hard to implement something like this.

247
00:40:06.630 --> 00:40:13.290
Yuanming Hu: And granularity is a different is another issue because in deep learning, you have large data blobs and you have a

248
00:40:14.250 --> 00:40:27.960
Yuanming Hu: You usually have ways height channels and they're a batch size which is the 40 tensor and usually they are pretty large. But for simulation your data granularity is fairly significant smaller and you want a finer granularity programming language.

249
00:40:29.010 --> 00:40:29.460
Yuanming Hu: So,

250
00:40:31.470 --> 00:40:43.350
Yuanming Hu: Although we're talking about the differences there. The key technique for determining and our application in different programming is still the same, which is reverse motive and

251
00:40:44.310 --> 00:40:56.460
Yuanming Hu: The techy programming language has this this kind of defied the extension which is a trivial extension to Taichi and that allows automatic differentiation for physical simulation and by

252
00:40:57.030 --> 00:41:07.200
Yuanming Hu: designing this by proposing this kind of new language we achieve for the first time, instead of key language design decisions such as differential imperative.

253
00:41:07.530 --> 00:41:14.340
Yuanming Hu: Parallel and mega kernels and Medicare nose is one of the most important thing because achieving Magneto in TensorFlow more pipeline is actually

254
00:41:21.120 --> 00:41:22.950
Julian Shun: If we lost him again.

255
00:41:24.390 --> 00:41:26.310
Julian Shun: Just wait a couple seconds.

256
00:41:26.400 --> 00:41:36.600
Yuanming Hu: Okay, so that we can do more floss provide fish from memory. And we also want to reduce the ku, Tokyo launches, which might take you roughly like 10 microseconds per kilo which

257
00:41:36.630 --> 00:41:40.200
Yuanming Hu: Can be quite a waste. If recommendation is actually lighter than that.

258
00:41:42.150 --> 00:41:46.440
Yuanming Hu: And our language allows programmers to easily beautiful physical module.

259
00:41:48.210 --> 00:41:59.190
Yuanming Hu: Program is now entry and differentiable. And if you build a system like this intensive or pork pie apple pie torch. If you're a CM airy timestamp takes you roughly 1000 I ops in TensorFlow.

260
00:41:59.460 --> 00:42:19.980
Yuanming Hu: Then you can easily end up with a TensorFlow graph with over a million nodes which can take TensorFlow something like 15 minutes to compare and start back to the year 2019

261
00:42:24.720 --> 00:42:27.240
Yuanming Hu: So we definitely want to do a manual

262
00:42:28.680 --> 00:42:29.820
Yuanming Hu: Gradient evaluation.

263
00:42:31.290 --> 00:42:38.670
Yuanming Hu: Because that is just as so painful and airport, and this is why exam was spent. Actually, one who afternoon to figure out the gradients here and

264
00:42:39.570 --> 00:42:50.940
Yuanming Hu: Even if you figure out the gradients a lot and sometimes even if the gradient has some bugs in it as long as, as long as the gradient difference between the true gradient

265
00:42:52.440 --> 00:42:52.920
Yuanming Hu: Sorry.

266

00:42:54.060 --> 00:42:55.080
Yuanming Hu: Am I still lost

267
00:42:55.560 --> 00:42:56.760
Julian Shun: Are we good here. You know, it's just

268
00:42:56.760 --> 00:42:59.310
Julian Shun: Audio gets cut off once in a while.

269
00:43:00.570 --> 00:43:03.390
Yuanming Hu: Cool. Um, okay, so

270
00:43:04.620 --> 00:43:17.790
Yuanming Hu: I always say that my internet connection is not stable. For some reason, I will just continue and okay so we definitely want to get rid of this kind of manual gradient distribution pattern workflow, which is super tedious and airport

271
00:43:19.980 --> 00:43:27.780
Yuanming Hu: So reverse mode alternative is not hard. Actually, we are using the source code transform approach and essentially we're just doing

272
00:43:28.140 --> 00:43:40.560
Yuanming Hu: For every corner. We're just doing a reverse accumulation of the green and contribution from edge, as I say, our instruction to its opposite. And so it's it's actually a relatively easy thing to do in Taichi

273
00:43:42.450 --> 00:43:51.300
Yuanming Hu: So here's one key difference between Taichi and our system. So in Taichi or alternative is to scale. And we have a higher level of scale were

274
00:43:51.960 --> 00:44:04.440
Yuanming Hu: Outside of mega Kronos we use a very lightweight type to record a colonel launches and within America know we use the source code transform this actually allows us to preserve the the arithmetic intensity

275
00:44:05.100 --> 00:44:10.740

Yuanming Hu: In the for mega Kronos in that in the backward America. America los we really want to achieve the same

276
00:44:11.910 --> 00:44:20.850
Yuanming Hu: Same performance. I mean the same magnitude of performance as the for Colonel so using source code transform within the mega Kronos is super easy. We want to

277
00:44:21.540 --> 00:44:32.220
Yuanming Hu: Do all the we want to catch the Indymedia resulting registers in comparison to storing all data to GPU memory, which is kind of a lot of ways for memory bandwidth

278
00:44:33.510 --> 00:44:39.510
Yuanming Hu: There are a lot of related work we're which are doing great things are on domain, but here for physical stimulation

279
00:44:41.340 --> 00:44:56.340
Yuanming Hu: We have to say that he is one of the unique system that allows people to actually ride high performance differential physical simulators. So that's a lot for the compiler parent. Let's talk about some applications. So those applications are pretty pretty

280
00:44:57.780 --> 00:45:01.440
Yuanming Hu: Interesting and probably yummy, I guess. Here we have a

281
00:45:02.490 --> 00:45:12.840
Yuanming Hu: Jedi like robot and you have four pieces of muscles and two per in Slack. So the muscle can either expand or contract, you can see in the color code.

282
00:45:13.440 --> 00:45:22.080
Yuanming Hu: Blue means contract and red means expand and our goal is to use brute force gradient descent to make the robot move to the right.

283
00:45:22.350 --> 00:45:30.450
Yuanming Hu: As you can see, just after at gradient descent iterations. The robot can learn how to jump, which is pretty incredible. And we're pretty happy to have this result.

284
00:45:31.710 --> 00:45:44.370
Yuanming Hu: Here's a 3D version. And this is the initial guess where we are. The robot does nothing and just after 40 gradient descent iterations, it learns to somehow move its, its legs to crawl forward.

285
00:45:45.390 --> 00:45:55.980
Yuanming Hu: And we can also combine liquid with different objects because our differential simulation differential programming language is relatively easy to use. We can easily add

286
00:45:56.430 --> 00:46:08.400
Yuanming Hu: More physical phenomenon in here we are adding a little bit of liquid who makes a robust life harder. So after 400 intuitions is children's to move to the right, despise there's water.

287
00:46:09.750 --> 00:46:20.850
Yuanming Hu: Here's our here's our mastering system which are pretty cute. So it seems as robust as to how to learn to contract or expand the box or the muscles so that we can walk to the right.

288
00:46:21.930 --> 00:46:32.700
Yuanming Hu: By the way, all the example from def Taichi can be reproduced from our public repositories script, essentially, if you have Python, you can install it via tab and then you can just reproduce the results.

289
00:46:33.960 --> 00:46:34.770
Yuanming Hu: Here's a

290
00:46:36.000 --> 00:46:48.870
Yuanming Hu: More funny application we're teaching the computer to play builders and the goal is to adjust the position have lost the of the white builder, so that the blue ball can hit the black destination using gradient descent.

291
00:46:50.730 --> 00:46:59.310
Yuanming Hu: One more thing rigid body simulation still a bunch of muscles and some ready bodies, our goal is to teach the rigid body no boss to move to the right.

292

00:47:01.500 --> 00:47:11.310
Yuanming Hu: So this is indefensible incompressible fluid simulation, we are optimizing for a initial velocity fuel so that after a bunch of time steps we get in touch the pattern.

293
00:47:11.880 --> 00:47:30.390
Yuanming Hu: Is kind of a great advertisement for the project. And here's a defensible waterway of simulation and you can solve for A initial Highfield so that you get a taxi pattern. Let's see, forgetting that okay, you guys see that okay the techy pattern. After some gradient descent iterations.

294
00:47:31.680 --> 00:47:34.020
Yuanming Hu: So here's my favorite one. Here we have a

295
00:47:34.620 --> 00:47:46.680
Yuanming Hu: N D federal program. We have a water simulator. We have a water render and we have a deep neural network attached to the end of the program. So we have three pieces every piece is defensible. And this is what we can do so.

296
00:47:47.460 --> 00:47:57.720
Yuanming Hu: We're, we're using a fox square image and VG tells us, okay, this is a Fox Zero. I know it's a square. I don't know. It's a fox. We're also video is doing a better job than me.

297
00:47:58.170 --> 00:48:05.610
Yuanming Hu: For this case and we can actually apply a water report to the center of limited of the image that you get so that you get some refraction here.

298
00:48:05.970 --> 00:48:16.290
Yuanming Hu: And of course you can also solve for A initial report distribution and after simulation and after refraction and after VG classification

299
00:48:16.650 --> 00:48:32.160
Yuanming Hu: It tells you that it's a goldfish were using a gradient descent methods to generating a something we call adversarial waterway so that we can generate a slight perturbation of the image that confuses the virtual confuses the BG to think this image is a goldfish.

300
00:48:34.740 --> 00:48:43.830

Yuanming Hu: So here's one interesting thing building dope ass differential physical simulator is not easy and sometimes differentiating physical stimulators using different

301
00:48:44.160 --> 00:48:49.140
Yuanming Hu: Using reverse mode alternative does not always here useful gradients of the

302
00:48:49.710 --> 00:48:56.850
Yuanming Hu: Physical system being stimulated. That's quite a long sentence, but let me explain what's happening here. So the gradients can easily go wrong if we

303
00:48:57.240 --> 00:49:04.440
Yuanming Hu: Just use brute force reverse motive. Consider this example we are, you have a little party ball hits a frictionless ground.

304
00:49:04.830 --> 00:49:16.740
Yuanming Hu: No gravity no friction fully elastic. So if you look at the initial high and final higher you have this relationship which is initial High Pass final height is a constant. If you're adjusting the initial height.

305
00:49:18.030 --> 00:49:24.390
Yuanming Hu: Which means if you raise the initial height of the bar if we if we're simulating for a constant amount of time.

306
00:49:24.720 --> 00:49:33.810
Yuanming Hu: Then we have the relationship derivative of the final height with respect to with the initial higher is negative one, which is a by definition right by by the physical loss.

307
00:49:34.230 --> 00:49:46.440
Yuanming Hu: But the difference was stimulated me tell you that the gradient this one instead of the negative value. So what is happening here. Let's think let's demonstrate what is happening here using a relatively larger time step.

308
00:49:47.490 --> 00:49:52.590
Yuanming Hu: You know, in simulation, way back down time integration into a few times steps in here as you can see

309
00:49:53.520 --> 00:50:01.800
Yuanming Hu: When we were raising the initial height, the final height is actually still racing except for a few discontinuities which is really bad and

310
00:50:02.160 --> 00:50:11.490
Yuanming Hu: If you pop the curves out if you plot the final height initial height plot your see that you get this kind of sawtooth pattern where buyers steadily increasing the initial height.

311
00:50:12.000 --> 00:50:16.620
Yuanming Hu: The final hide still increases, which is not desirable, you still get the correct tendency

312
00:50:17.430 --> 00:50:25.020
Yuanming Hu: But you get the gradients are just completely the wrong. So the question is, is there any way for us to get the orange curve here well

313
00:50:25.950 --> 00:50:36.000
Yuanming Hu: It's actually already something in computer graphics. So does something like this which which we call precise time of impact. So instead of treating the collision events in discrete timestamps

314
00:50:36.780 --> 00:50:48.750
Yuanming Hu: If we do some continuous time approximation of the collision then differentiating the physical simulation system can give you indeed give you the correct radiant without handling.

315
00:50:49.230 --> 00:50:58.200
Yuanming Hu: Just by using reversible alternative so fixing the wrong gradients here is really, really important. And by fixing the gradients. The robot can do much better.

316
00:50:58.530 --> 00:51:12.300
Yuanming Hu: without fixing ingredients, you see the last curves are like the red ones but but after fixing ingredients you get green curves which is significantly better optimization. So here's a visualization optimize without time of impact fix

317
00:51:13.920 --> 00:51:30.000

Yuanming Hu: Can barely make any progress, but if I optimize with Tommy some of impact it optimizes very, very well. But this is even more interesting if we test the optimized controller on then environments with time of impact everything still works.

318
00:51:31.080 --> 00:51:41.790
Yuanming Hu: Without type of impact, everything works. So the thing is when only for simulation is needed, without him of impact the simulator is good enough. But if you need to train that you should definitely

319
00:51:42.240 --> 00:51:54.060
Yuanming Hu: Use a higher precision for simulation skim the takeaway here is differentiating physical simulators does not always use us for gradients for the physical simulated physical system being simulated

320
00:51:54.750 --> 00:52:05.250
Yuanming Hu: In a simulation good enough for will forward might not be good enough for back propagation. So check out our paper for more details on building simulators with more numerical stability.

321
00:52:06.270 --> 00:52:07.680
Yuanming Hu: And we definitely have morning from

322
00:52:15.750 --> 00:52:19.290
Yuanming Hu: Forum. We have a techy con conference where people talk about

323
00:52:20.490 --> 00:52:23.130
Yuanming Hu: Taichi backhands and implementation details.

324
00:52:24.450 --> 00:52:35.640
Yuanming Hu: So that's mostly it and let me finish my talk by one quadrant Confucius infrastructure is key to success. And we believe that a good set of infrastructure for computer graphics

325
00:52:36.030 --> 00:52:47.070
Yuanming Hu: Has been missing for too long time and we really wished he to be something that comes up that for graphics developers and thank you and I'm happy to take a few questions.

326
00:52:48.090 --> 00:52:49.110
Julian Shun: Right, so excellent

327
00:52:52.740 --> 00:52:57.600
Julian Shun: So anyone have any questions feel free to speak up or and also type in the chat.

328
00:53:04.440 --> 00:53:18.060
Julian Shun: So I have another question. I'm wondering if they're ready like optimizations that are currently not supported by the High Level interface you have and if there are plans to extend a interface for other optimizations.

329
00:53:23.610 --> 00:53:24.690
Julian Shun: Oh, we

330
00:53:25.740 --> 00:53:26.730
Julian Shun: We lost him again.

331
00:53:28.890 --> 00:53:30.270
Julian Shun: Wait a couple seconds.

332
00:53:30.270 --> 00:53:33.540
Yuanming Hu: Sorry, yeah. I find that my internet might not be so good.

333
00:53:34.260 --> 00:53:36.750
Julian Shun: Oh yeah, no problem. I can, we can hear you.

334
00:53:41.100 --> 00:53:41.910
Julian Shun: Oh hello.

335
00:53:42.240 --> 00:53:42.810
Yuanming Hu: Hello, today.

336
00:53:43.380 --> 00:53:44.550
Yuanming Hu: Oh yeah, maybe

337
00:53:45.210 --> 00:53:45.630
Julian Shun: A video

338
00:53:46.140 --> 00:53:47.700
Yuanming Hu: Yeah, I can hear you so that

339
00:53:48.480 --> 00:53:51.600
Yuanming Hu: I use smaller network. Anyways, I guess.

340
00:53:52.500 --> 00:53:53.310
Julian Shun: Yeah yeah

341
00:53:54.330 --> 00:53:55.170
Julian Shun: I can hear you now.

342
00:53:58.110 --> 00:53:59.010
Julian Shun: Can you hear me.

343
00:54:07.140 --> 00:54:09.510
Julian Shun: Type it type or direct message.

344
00:54:15.270 --> 00:54:18.240
Yuanming Hu: Yep, I can, I can hear you. But can you hear me.

345
00:54:18.990 --> 00:54:20.220
Julian Shun: Yeah, I can hear you.

346
00:54:20.820 --> 00:54:23.100
Julian Shun: Working. Yeah, it works.

347
00:54:23.250 --> 00:54:23.850
Yuanming Hu: Okay, okay.

348

00:54:25.620 --> 00:54:28.110
Yuanming Hu: Yeah, okay. Oh, I see. Everyone will be

349
00:54:29.970 --> 00:54:33.690
Julian Shun: Yeah, so I i was what I don't know if he answered my question. Ready, but

350
00:54:34.170 --> 00:54:34.920
Julian Shun: I was

351
00:54:35.340 --> 00:54:44.310
Julian Shun: Okay, yeah. So I was wondering if there any like additional optimizations that are currently not supported and that you might want to extend the

352
00:54:45.390 --> 00:54:46.380
Julian Shun: Interface for

353
00:54:47.430 --> 00:54:56.310
Yuanming Hu: Yeah, I think that's a good question. So although we strive to do as much as many optimization, as we could. There are still some

354
00:54:57.450 --> 00:54:59.100
Yuanming Hu: optimizations that we cannot do.

355
00:55:00.240 --> 00:55:04.740
Yuanming Hu: Either due to the design decision that he or due to we don't just don't have enough time.

356
00:55:05.040 --> 00:55:11.550
Yuanming Hu: So we're still working on two interesting things, which is one thing being whole program optimization of Taichi programs.

357
00:55:11.850 --> 00:55:17.010
Yuanming Hu: And this is what we're really interested in because you know for a specialty sparse computation you need to

358
00:55:17.370 --> 00:55:24.270

Yuanming Hu: Generate a list of active elements. And sometimes this kind of auxiliary this generation can take your 90% of timing some extreme cases.

359
00:55:24.690 --> 00:55:35.130
Yuanming Hu: And by up in order to optimize that out, we need to really do some whole program analysis of the whole program so that we can get more information to optimize beyond the current that's

360
00:55:42.600 --> 00:55:51.570
Yuanming Hu: Where people are going lower position and low precision is still one interesting thing for for us. The other thing is that

361
00:55:53.130 --> 00:55:55.590
Yuanming Hu: In general, or we think

362
00:55:56.970 --> 00:56:05.040
Yuanming Hu: Although we can do some reasonable optimization. And I think you mentioned the order to anything. And sometimes the developer still have to do

363
00:56:05.460 --> 00:56:14.490
Yuanming Hu: A lot of manual training to get a optimal data structure and that is very, very promising. Next step, we're going to try all the training of data structures.

364
00:56:15.270 --> 00:56:21.300
Yuanming Hu: And actually we can't even can't do something like a data dependent data structure, all the training like

365
00:56:22.080 --> 00:56:34.980
Yuanming Hu: Changing the sparkles on the fly. Because sometimes it's really, really hard to know what kind of data, you're going to get. But I'm not sure if that's going to bring us super huge performance boost, but that sounds like a crazy an interesting idea.

366
00:56:36.600 --> 00:56:37.770
Julian Shun: Yeah. Great. Thank you.

367
00:56:38.160 --> 00:56:40.470
Yuanming Hu: Okay. Two more questions actually.

368
00:56:40.500 --> 00:56:40.890
Yeah.

369
00:56:42.060 --> 00:56:46.440
Yuanming Hu: When there are thousands of times that. How do you deal with managing and explore ingredients. Yeah, that's a

370
00:56:47.430 --> 00:56:58.170
Yuanming Hu: That's, that's definitely a very good question and you don't know how much we have suffered from the gradients to some stability issue where we're dealing with the differential physical simulators, just like

371
00:56:59.610 --> 00:57:03.750
Yuanming Hu: NLP people were doing a recursive neural networks, when they have maybe

372
00:57:05.670 --> 00:57:15.900
Yuanming Hu: A few hundred nodes in a neural network is a green and do tend to vanish or explode. But what we find is that individual simulators screen and vanishing is usually not a big issue.

373
00:57:16.500 --> 00:57:27.540
Yuanming Hu: Sometimes as long as your system is not overly damned greedy and managing is not a huge issue by physical loss, but we do have a lot of gradient exploding issue so

374
00:57:27.900 --> 00:57:43.590
Yuanming Hu: Our victim of that is super easy. You can just do some creating clamping or do some some penalty to prevent your similar physical parameter or newsletter ways to be too large. And that turns out to be a very, very simple and effective approach to stabilize the gradients.

375
00:57:45.330 --> 00:57:54.180
Yuanming Hu: Hope that answers the question. And there's one more question on Eastern compiler. We're of cash sizes and hierarchy. If so, can a user

376

00:57:54.540 --> 00:58:07.470
Yuanming Hu: Easily specify a new architectures. So that's a that's also a great question. So our solution is to is not to specify the the cash sciences. Our solution is to directly let the user propose.

377
00:58:09.000 --> 00:58:21.390
Yuanming Hu: Data structure data structure parameters such as, How many layers do you want what's up ranking factor. What's the size of the leaf level blocks. So one thing we find that usually for modern computer architecture.

378
00:58:23.040 --> 00:58:38.970
Yuanming Hu: No matter is arm or Intel or four CPUs is almost always something like 32 kilobytes for instruction cash and 30 kilobytes for data cache. And then to 56 kilobytes for our new character and two megabytes per quarter for L three. So what we find here if that

379
00:58:40.560 --> 00:58:50.700
Yuanming Hu: Instead of telling the competitor, the cache size which is a not a huge thing a huge change across different architecture, it's more important to specify the memory layout.

380
00:58:51.120 --> 00:58:59.790
Yuanming Hu: Or data structure specific agents such as a real structural structural ray or even a ways away and which can lead to structure we use do we use

381
00:59:00.480 --> 00:59:08.400
Yuanming Hu: Pointers pointer race. Do you use a bit masks, which are which have different behavior for the memory alligator and for caching hit rates.

382
00:59:08.760 --> 00:59:22.980
Yuanming Hu: So we're actually doing a more end to end approach we directly specify the data structure specification and let benchmark be our guide intelligent compiler and even I think even if we know let's say the the LM caches.

383
00:59:24.180 --> 00:59:36.360
Yuanming Hu: 32 kilobytes is still the only thing we know is that for the for a leaf level note of your data structure. You don't go beyond 30 kilobytes. Otherwise, we're going to getting too big cash.

384
00:59:37.290 --> 00:59:51.060
Yuanming Hu: Working set, but we don't know what should exactly be the size for Katie, a Kb second KB. We have no idea. So we just want to let the users to try and then we can just use benchmark to figure out which one is the best

385
00:59:52.110 --> 00:59:54.570
Richard Barnes: And if I could extend on that question for a moment.

386
00:59:55.590 --> 01:00:01.800
Richard Barnes: It seems so the use of catches and CPU. So like if you missed the catch the program keeps running right

387
01:00:02.550 --> 01:00:13.230
Richard Barnes: But you also spoke about using shared memory within GPUs where those that same thing doesn't apply, right, if you go beyond the shared memory, you're kind of screwed. Could you talk about how you handle that issue.

388
01:00:13.800 --> 01:00:21.120
Yuanming Hu: Exactly. So currently, the certain memory thing is we have a something we call a block local storage.

389
01:00:21.480 --> 01:00:34.230
Yuanming Hu: Abstraction for Sam Raimi which is very similar to a thread local storage for CPU, where you just or everything I want cash. So for a block local storage. We're still relying on the user to specify

390
01:00:35.280 --> 01:00:47.010
Yuanming Hu: A block them. Let's see, we have a three by three by three stand. So, and let's say our leaf block size it's four by five, four, and in order to apply that stand. So usually people have to allocate something like a six by six by six.

391
01:00:49.470 --> 01:00:53.100
Yuanming Hu: Piece of local cache of the whole grid and

392
01:00:54.390 --> 01:01:04.620

Yuanming Hu: I think you just point out it pointed out a great point. If six by six by six goes to cache memory, there's gonna be a lot of bad things happening. Something like a

393
01:01:05.970 --> 01:01:13.890
Yuanming Hu: Now saturating all the GPU cores, or even not the PDFs might not even going to compile because there's just not efficient not sufficient.

394
01:01:14.730 --> 01:01:24.660
Yuanming Hu: Shared memory in a single SM so if that happens. Our solution is to ask the ask the user to use a smaller block size and the other option is to, if you have a

395
01:01:25.620 --> 01:01:39.060
Yuanming Hu: Four x by eight by eight leaf block every time we only gather four by four by four portion of it. And then so that we don't go beyond the shared memory limits, but I think it's a great point that

396
01:01:40.020 --> 01:01:54.630
Yuanming Hu: GPU memory or flow is much worse than CPU memory overflow and i think i think this largely requires trial and error, but fortunately in Taichi. The compatibility handles the trial trial and error thing.

397
01:01:55.200 --> 01:02:04.530
Yuanming Hu: So it's not so mechanical, I mean it's mechanical, but the most of the mechanical things come by the compiler so that users can still quickly try a different design strategy.

398
01:02:05.370 --> 01:02:07.890
Yuanming Hu: Thank you. Thanks. Thanks for the great question.

399
01:02:11.100 --> 01:02:15.030
Julian Shun: Great, thanks. So, anyone else have any questions.

400
01:02:16.590 --> 01:02:22.800
Yuanming Hu: Um, I think there's one extra thing which is I spoke more about better with then memory, whether that mean

401

01:02:24.570 --> 01:02:35.010
Richard Barnes: I don't exactly sorry that that was just the contextualization of the previous question that I felt like you were talking a lot about a few, but some such and say, I wasn't seeing the the memory connection.

402
01:02:37.020 --> 01:02:38.730
Richard Barnes: And I feel like I'm answered now.

403
01:02:42.210 --> 01:02:43.470
Julian Shun: Yeah. Great. Thanks.

404
01:02:44.940 --> 01:02:45.660
Yuanming Hu: Lost again.

405
01:02:46.350 --> 01:02:49.710
Julian Shun: Oh yeah, I think you're back knowing

406
01:02:51.810 --> 01:02:53.310
I think it's frozen again.

407
01:02:57.960 --> 01:02:59.520
Julian Shun: Let's give it a couple seconds.

408
01:03:00.030 --> 01:03:01.050
Yuanming Hu: No. Okay.

409
01:03:01.320 --> 01:03:03.180
Yuanming Hu: Hello. Yeah, I'm back. Yeah.

410
01:03:04.230 --> 01:03:04.500
Yuanming Hu: Sorry.

411
01:03:05.010 --> 01:03:05.790
Julian Shun: Very good.

412

01:03:07.950 --> 01:03:14.670
Yuanming Hu: Reason today. Yeah, it's pretty, pretty good. But I think nowadays, whatever. It goes right so follow the code. So

413
01:03:20.640 --> 01:03:21.390
Julian Shun: Okay, cool.

414
01:03:22.530 --> 01:03:25.290
Julian Shun: Yeah, any, any other questions.

415
01:03:25.470 --> 01:03:32.490
Peter Lu: I just had a quick question about could you like compare you know Taichi or def it with Jax.

416
01:03:33.150 --> 01:03:44.460
Yuanming Hu: We did so we did computes diff Taichi was just because he starts before computation. So we actually borrow a example from the jacks website, which is a

417
01:03:45.600 --> 01:03:58.380
Yuanming Hu: fluid simulation example and I think we're something like 2.4 X faster than Jax. And the other thing, why you shouldn't think that are for that example on Jack's it takes Jack's like

418
01:03:59.040 --> 01:04:16.410
Yuanming Hu: Six minutes to compile, just because it's just not designed for a graph was this many nodes in for techy the competition time it was he is listening to me two seconds, and also the performance in Taichi is faster, just because he already fuses everything the mega Colonel but projects.

419
01:04:17.760 --> 01:04:23.670
Yuanming Hu: There's no guarantee that the compiler is going to fuse everything together. I think just I think for simulation.

420
01:04:26.250 --> 01:04:35.550
Yuanming Hu: Actually think for more general simulation catch your have a larger benefit because that fully simulation is still a something like an array based

421

01:04:42.840 --> 01:04:57.630
Yuanming Hu: There's a race operation, but a lot of simulation parents cannot be representatives in desperate operation, but we were too lazy to actually write a more complex system using jack. So we just took that example from the Jackson website and then do the comparison.

422
01:05:02.640 --> 01:05:03.630
Julian Shun: Great, thanks.

423
01:05:06.690 --> 01:05:08.340
Julian Shun: Okay, we lost you again.

424
01:05:11.310 --> 01:05:12.210
Julian Shun: Oh, are you back.

425
01:05:12.840 --> 01:05:13.170
Yuanming Hu: Oh, no.

426
01:05:14.340 --> 01:05:15.960
Julian Shun: I seen that. Yeah, I can hear you know

427
01:05:19.500 --> 01:05:21.690
Julian Shun: Yeah, so there's another question in the chat.

428
01:05:22.080 --> 01:05:27.000
Yuanming Hu: That's techy you work for more general geometric computation, such as mesh area and spatial

429
01:05:33.660 --> 01:05:35.790
Yuanming Hu: Accommodation for specialty spas.

430
01:05:44.640 --> 01:05:45.330
Yuanming Hu: Us.

431
01:05:47.130 --> 01:05:47.610
Yuanming Hu: So,

432
01:05:50.520 --> 01:05:54.150
Yuanming Hu: Because the tetrahedron metrics are intrinsically kind of irregular

433
01:05:56.190 --> 01:06:00.330
Yuanming Hu: How did you special optimization for tetrahedral matters, but I have to say.

434
01:06:01.020 --> 01:06:18.930
Yuanming Hu: If you use if you're just represent like the index arrays or the connectivity using 2D or one, the dancer race, you can still do that entire chain but the good domain specific optimization for specialty sparse thing for optimizing for specialty spar thing. It's not going to

435
01:06:19.980 --> 01:06:27.330
Yuanming Hu: Bring you too much. In that case, if you're interested. But you still get some benefit because you can receive your code using Python, which is still a class nowadays.

436
01:06:27.750 --> 01:06:37.890
Yuanming Hu: A lot of people don't care about specialist. First thing computation. They care about writing GPU cold reading kuda code in Python with your do care about the productivity here.

437
01:06:42.480 --> 01:06:42.810
Yuanming Hu: Thank you.

438
01:06:50.400 --> 01:06:52.410
Julian Shun: Any okay

439
01:06:53.040 --> 01:07:01.680
Yuanming Hu: Tomorrow, Austin texture memories on GPU. Oh, that's a, that's a great question. So currently, one thing is that

440
01:07:03.150 --> 01:07:18.570

Yuanming Hu: So for constant memory. Let's first talk about constant memory. So at this point we don't yet. Use the constant memory, but we do you make use of the constant Ellen cash on on NVIDIA GPUs, because just because

441
01:07:20.280 --> 01:07:36.180
Yuanming Hu: It's relatively easy to detect which pointer is read only entirely. We don't have point aliasing. And then we somehow issue the recall underscore, underscore out eg load instruction which just ignorance, your cash coherence, the policies and gives you higher performance loads.

442
01:07:44.070 --> 01:07:47.190
Yuanming Hu: Especially you can do biding your training.

443
01:07:56.310 --> 01:07:59.730
Yuanming Hu: One major difficulty for do that is

444
01:08:02.190 --> 01:08:06.090
Yuanming Hu: The techy competitor. It may not be. I mean, the

445
01:08:07.440 --> 01:08:16.590
Yuanming Hu: System is designed to be more relatively general purpose than fetching fetching memory. But I do believe for a lot of application where we have a constant.

446
01:08:17.310 --> 01:08:31.020
Yuanming Hu: Texture piece of texture and you want to do the interpolation and loading 70 to stay you do care about performance. They're doing this kind of a texture memory usage is super, super helpful. But the sad thing is that don't CPU will have you seen any bad but maybe it's not too bad.

447
01:08:35.010 --> 01:08:35.430
Richard Barnes: Thank you.

448
01:08:36.000 --> 01:08:36.300
Thanks.

449
01:08:43.170 --> 01:08:43.620
Julian Shun: Great.

450
01:08:45.390 --> 01:08:47.670
Julian Shun: doesn't see what there any issue.

451
01:08:48.450 --> 01:08:48.810
Here.

452
01:08:49.890 --> 01:08:53.880
Julian Shun: Yeah, well, let's thank you again for the talk.

453
01:08:54.810 --> 01:09:02.010
Yuanming Hu: Thank you so much for thank you for making this happen. The end. Also, thanks for some and chairs for inviting me here.

454
01:09:03.540 --> 01:09:07.080
Julian Shun: Yep. Yeah. Yes. Very, very interesting talk.

455
01:09:10.170 --> 01:09:10.980
Julian Shun: Great. Well,

456
01:09:12.930 --> 01:09:14.400
Julian Shun: Because so i mean

457
01:09:14.730 --> 01:09:17.790
Julian Shun: I guess since there's there's no more questions.

458
01:09:20.310 --> 01:09:20.640
Yuanming Hu: Okay.

459
01:09:21.660 --> 01:09:22.200
Yuanming Hu: All I do

460
01:09:22.650 --> 01:09:23.970
Yuanming Hu: Is going bad once again.

461
01:09:25.470 --> 01:09:25.650
Yuanming Hu: You

462
01:09:25.860 --> 01:09:26.970
Yuanming Hu: Can up here.

463
01:09:27.420 --> 01:09:28.470
Julian Shun: Yeah. Yes. Great.