01:48:15 torch.compile, Python overhead, kernel fusion, 130ms

now okay so as we can see we are currently at about 300 milliseconds uh per iteration and we're now going to reach for some really heavy weapons in the pie torch Arsenal and in particular we're going to introduce torch. compile so torch. compile is really quite incredible infrastructure from the pytorch team and it's basically a compiler for neural networks like it's almost like GCC for CN C++ code this is just this GCC of neural nuts so came out a while ago and extremely simple to use um the way to use torch compile is to do this it's a single line of code to compile your model and return it now this line of code will cost you compilation time but as you might guess it's going to make the code a lot faster so let's actually run that because this will take some time to run but currently remember we're at 300 milliseconds and we'll see what happens now while this is running I'd like to explain a little bit of what torch. compile does under the hood uh so feel free to read this page of P torch but basically there's no real good reason for you to not use torch compile in your pie torch I kind of feel like you should be using almost by default if you're not uh unless you're debugging and you want your code to run really fast and there's one line here in torch compile that I found that actually kind of like gets to why this is faster speed up mainly comes from reducing python overhead and GPU read wrs so let me unpack that a little bit um okay here we are okay so we went from 300 milliseconds we're now running at 129 milliseconds so this is uh 300 129 about 2.3x Improvement from a single line of code in py torch uh so quite incredible so what is happening what's happening under the hood well when you pass the model to torch compile what we have here in this NN module this is really just the algorithmic description of what we'd like to happen in our Network and torch compile will analyze the entire thing and it will look at what operations You' like to use and with the benefit of knowing exactly what's going to happen it doesn't have to run in What's called the e mode it doesn't have to just kind of like go layer by layer like the python interpreter normally would start at the forward and the python interpreter will go okay let's do this operation and then let's do that operation and it kind of materializes all the operations as it goes through uh so these um calculations are dispatched and run in this order and the python interpreter and this code doesn't know what kind of operations are going to happen later but torch compile sees your entire code at the same time and it's able to know what operations you intend to run and it will kind of optimize that process the first thing it will do is will it will take out the python interpreter from the forward pass entirely and it will kind of compile this entire neural net as a single object with no python interpreter involved so it knows exactly what's going to run and we'll just run that and it's all going to be running in efficient code uh the second thing that happens is uh this read write that they mentioned very briefly so a good example of that I think is the G nonlinearity that we've been looking at so here we use the n and G now this here is me uh basically just breaking up the inang Galu uh which you remember has this formula so this here is the equivalent implementation to what's happening inside g algorithmic l it's identical Now by default if uh we just we using this instead of ending. G here what would happen without torch compile well the python interpreter would make its way here and then it would be okay well there's an input well let me first let me raise this input to the third power and it's going to dispatch a kernel that takes your input and raises it to the third power and that kernel will run and when this kernel runs what ends up happening is this input is stored in the memory of the GPU so here's a helpful example of the layout of what's happening right you have your CPU this is in every single computer there's a few cores in there and you have your uh Ram uh your memory and the CPU can talk to the memory and this is all well known but now we've added the GPU and the GPU is a slightly different architecture of course they can communicate and it's different in that it's got a lot more course than a CPU all of those cores are individually a lot simpler too but it also has memory right this high bandwidth memory I'm sorry if I'm botching it hbm I don't even know what that stands for I'm just realizing that but uh this is the memory and it's very equivalent to uh RAM basically in the computer and what's happening is that input is living in the memory and when you do input cubed this has to travel to the GPU to the course and to all the caches and registers on the actual chip of this GPU and it has to calculate the all the elements to the third and then it saves the result back to the memory and it's this uh travel time that actually causes a lot of issues so here remember this memory bandwidth we can communicate about 2 terabytes per second which is a lot but also we have to Traverse this link and it's very slow so here on the GPU we're on chip and everything is super fast within the chip but going to the memory is extremely expensive takes extremely long amount of time and so we load the input do the calculations and load back the output and this round trip takes a lot of time and now right after we do that we multiply by this constant so what happens then is we dispatch another kernel and then the result travels back all the elements get multiplied by a constant and then the results travel back to the memory and then we take the result and we add back input and so this entire thing again travels to the GPU adds the inputs and gets written back so we're making all these round trips from the memory to actually where the comput happens because all the tensor cores and alus and everything like that is all stored on the chip in the GPU so we're doing a ton of round trips and pytorch uh without using torch compile doesn't know to optimize this because it doesn't know what kind of operations you're running later you're just telling it raise the power to the third then do this then do that and it will just do that in that sequence but torch compile sees your entire code it will come here and it will realize wait all of these are elementwise operations and actually what I'm going to do is I'm going to do a single trip of input to the GPU then for every single element I'm going to do all of these operations while that memory is on the GPU or chunks of it rather and then I'm going to write back a single time so we're not going to have these round trips and that's one example of what's called kernel fusion and is a major way in which everything is sped up so basically if you have your benefit of onet and you know exactly what you're going to compute you can optimize your round trips to the memory and you're not going to pay the the memory bandwidth cost and that's fundamentally what makes some of these operations a lot faster and what they mean by read writes here so let me erase this because we are not using it and yeah we should be using torch compile and our code is now significantly faster and we're doing about 125,000 tokens per second but we still have a long way to go before we move on I wanted to supplement the discussion a little bit with a few more figures uh because this is a complic topic but it's worth understanding on a high level uh what's happening here and I could probably spend an entire video of like two hours on this but just the preview of that basically so this chip here that is uh the GPU this chip is where all the calculations happen mostly but this chip also does have some memory in it but most of the memory by far is here in the high bandwidth memory hbm and is connected they're connected um but these are two separate chips basically now here this is a zoom in of kind of this cartoon diagram of a GPU and what we're seeing here is number one you see this hbm I I realize it's probably very small for you but on the sides here it says hbm and so that that's the links to the hbm now the hbm is again off chip on the chip there are a large number of these streaming multiprocessors uh every one of these is an

SM there's 120 of them in total and this is where the a lot of the calculations happen and this is a zoom in of a single individual as it has these four quadrants and see for example tensor core this is where a lot of the Matrix multiply stuff happens but there's all these other units to do all different kinds of calculations for fp64 fp32 and for integers and so on now so we have all this uh logic here to do the calculations but in addition to that on the chip there is memory sprinkled throughout the chip so L2 cache is some amount of memory that lives on the chip and then on the SMS themselves there's L1 cache I realized it's probably very small for you but this blue bar is L1 and there's also registers um and so there is memory stored here but the way this memory is stored is very different from the way memory is stored in hbm uh this is a very different implementation uh using um just in terms of like what the Silicon looks like it's a very different implementation um so here you would using transistors and capacitors and here it's a very different implementation uh with SRAM and what that looks like but long story short is um there is um memory inside the chip but it's not a lot of memory that's the critical point so this is some C this is a example diagram of a slightly different GPU just like here where it shows that for example typical numbers for CPU Dam memory which is this thing here you might have one tab of this right but it would be extremely expensive to access especially for a GPU you have to go through the CPU here now next we have the hbm so we have tens of gigabytes of hbm memory on a typical GPU here but it's as I mentioned very expensive to access and then on the chip itself everything is extremely fast within the chip but we only have couple 10 megabytes of memory collectively throughout the Chip And so there's just not enough space because the memory is very expensive on the chip and so there's not a lot of it but it is lightning fast to access in relative terms and so basically whenever we have these kernels um the more accurate picture of what's Happening Here is that we take these inputs which live by default on the global memory and now we need to perform some calculation so we start streaming the data from the um Global memory to the uh chip we perform the calculations on the chip and then stream it back and store it back to the global memory right and so if we are if we don't have torch compile we are streaming the data through the chip doing the calculations and saving to the memory and we're doing those round trips many many times but uh if it's torch compiled then we start streaming the memory as before but then while we're on the chip we're we're we have a chunk of the uh data that we're trying to process so that chunk now lives on the chip while it's on the chip it's extremely fast to operate on so if we have kernel Fusion we can do all the operations right there in an element-wise fashion and those are very cheap and then we do a single round trip back to the global memory so operator Fusion basically allows you to keep your chunk of data on the Chip And do lots of calculations on it before you write it back and that gives huge savings and that's why torch compile ends up being a lot faster or that's one of the major reasons uh so again just a very brief intro to the memory hierarchy and roughly what torch compile does for you