01:28:14 Tensor Cores, timing the code, TF32 precision, 333ms

format okay so first of all what are tensor cores well tensor course tensor core is just an instruction in the a100 architecture right so so what it does is it does basically a little 4x4 Matrix multiply so uh this is just matrix multiplication here of 4x4 matrices and there are multiple configurations as to what Precision any of these matrices are it in what Precision the internal accumulate happens and then what is the output Precision input precisions Etc so there's a few switches but it's basically a 4x4 multiply and then anytime we have any operations that require Magic multiplication uh they get broken up into these into this instruction of little 4x4 multiply and so everything gets broken up into this instruction because it's the fastest way to multiply matrices and it turns out that most of the computational work that we're doing up above uh all of it really is matrix multiplication most of the work computationally happens in the linear layers um linear linear Etc there's a few things sandwiched in between so there's some additions in residuals there's some G nonlinearities there's some layer Norms Etc but if you just time them you'll see that these are nothing like basically the in Transformer is just a bunch of Matrix multiplications really um and especially at this small scale 124 million parameter model actually the biggest matrix multiplication by far is the classifier layer at the top that is a massive Matrix multiply of going from 768 to 50257 and that Matrix multiply dominates anything else that happens in that Network roughly speaking so it's Matrix multiplies that become a lot faster which are hidden inside our linear layers and they're accelerated through tensor course now the best reference I would say for tensor course is basically just go to the um a 100 architecture white paper and then it's pretty detailed and but I think people it's like relatively readable mostly if you half understand what's happening um so figure 9 tensor float 32 so this is the explanation basically for tf32 and what happens here and you see that there's many configuration options here available so the input operands and what precisions are they in the accumulator and um what um basically the um the internal representation within the instruction when you do the accumulate of this matrix multiplication so the intermediate plus equals um of the intermediate little vector multiplies here that all happens in fp32 and then uh this is an aex improvement as I mentioned to the Ops that we get so tf32 specifically we're looking at this row here and the way this works is um normally fp32 has 32 bits tf32 is the exact same bits we have one sign bit we have eight exponent bits except the mantisa bits get cropped in the float and so basically um we end up with just 19 bits instead of 32 bits because the last 133 bits get truncated they get dropped um and all this is internal to the instruction so none of it is visible to anything in our pytorch uh none of our pytorch code will change all of the numbers will look identical it's just that when you call the tensor core um instruction internally in the hardware it will crop out these 13 bits and that allows it to uh calculate this little Matrix multiply significantly faster 8X faster now of course this speed up comes at a cost and the cost is that we are reducing the Precision our accumulate is still an fp32 our output is fp32 our inputs are fp32 but internally things get truncated in the operand to perform the operation faster and so our results are starting to be a bit more approximate but empirically when you actually train with this you basically can't tell the difference so the reason I like tf32 is because if you can tolerate a little bit of a Precision fudge um then this is free like none of your codes sees this it's fully internal to the operation and the operation to you just go 8X faster and it's a bit more approximate and so it's a pretty sweet spot I would say in optimization and uh let's see what that looks like first so I've set up our Cod to just time the uh iterations so import time I changed the hyper parameters so that we have something a bit more that reflects uh kind of workload that we want to run uh because we want to do a fairly large run at the end of this so let's use batch size 16 and let's now use the actual gpt2 um maximum sequence length of 10,24 tokens uh so this is the configuration and then for 50 iterations I'm just doing something very lazy here I'm doing time. time to get the current time and then this is the optimization Loop and now I want to time how long this takes now one issue with working with gpus is that as your CPU um when your CPU runs it's just scheduling work on GPU it's ordering some work right and so it send a request and then it continues running and so we can actually it can happen sometimes that we sort of um speed through this and we queue up a lot of kernels to run on the GPU and then the CPU sort of like gets here and takes time at time but actually the GPU is still running because it takes it time to actually work through the work that was scheduled to run and so you're just building up a queue for the GPU and so actually if you need to you want to wait toat data synchronize and this will wait for the GPU to finish all the work that was scheduled to run up above here and then we can actually take the time so basically we're waiting for the GPU to stop this iteration take time and then we're going to just print it so so here I'm going to run the training Loop and here on the right I'm watching Nvidia SMI so we start off at zero um we're not using the GPU and then by default P will use gpu0 so we see that it gets filled up and we're using 35 GB out of 80 gabt available and then here on the left we see that because we've cranked up the batch size now it's only 20 batches to do a single Epoch on our tiny Shakespeare and we see that we're seeing roughly a th000 milliseconds per iteration here right so the first iteration sometimes is slower and that's because pytorch might be doing a lot of initializations here on the very first iteration and so it's probably initializing all these uh tensors and buffers to hold all the gradients and I'm not 100% sure all the work that happens here but uh this could be a slower iteration when you're timing your logic you always want to be careful with that but basically we're seeing a th000 milliseconds per iteration um and so this will run for roughly 50 seconds as we have it right now so that's our Baseline in flo 32 one more thing I wanted to mention is that if this doesn't fit into your GPU and you're getting out of memory errors then start decreasing your batch size until things fit so instead of 16 try eight or four or whatever you need to fit um the batch into your GPU and if you have a bigger GPU you can actually potentially get away with 32 and so on uh by default you want to basically max out has Max Max out the batch size that fits on your GPU and you want to keep it nice numbers so use numbers that have lots of powers of two in them so 16 is a good number 8 24 32 48 These are nice numbers but don't use something like 17 uh because that will run very inefficiently on a GPU uh and we're going to see that a bit later as well so for now let's just stick with 16124 and uh the one thing that I added also here and I ran it again is I'm calculating a tokens per second throughput during training because we might end up changing the backat size around over time but tokens per second is the objective measure that we actually really care about how many tokens of data are we training on and what is the throughput of tokens that we're getting in our optimization so right now we're processing and training on 163,000 tokens per second roughly and that's a bit more objective metric okay so let's now enable tf32 now luckily pytorch makes this fairly easy for us and uh to enable tf32 you just need to do a single line and is this and when we go to the py documentation here for this function basically this tells pych what kind of kernels to run and by default I believe it is highest highest Precision for mat M and that means that everything happens in float 32 just like it did before but if we set it to high as we do right now Matrix multiplications will not use tensor flow 32 when it's available my GPU is a100 so it's an ampere series and therefore tf32 is available if you have an older GPU this might not be available for you but for my GPU it's available and so what I expect P to do is that every single place where we see an nn. linear inside there there's a matrix multiplication and I expect that matrix multiplication now to be um running on tensor course utilizing the TF 32% so this is the single line of change that is I believe necessary and let's rerun this now we saw that um in terms of the throughput that is promised to us we're supposed to be getting 8X roughly so let's see what happens and that 8X came from here right um 8X and it also came from looking at it um here 156 T flops instead of of 19.5 okay so what actually happened uh so we're seeing that our throughput roughly 3x not aex so we are going we're from 1,000 milliseconds we're going down to 300 milliseconds and our throughput is now about 50,000 tokens per second so we have a roughly 3x instead of 8X so what happened and basically What's Happening Here is again a lot of these workloads are memory bound and so even though the tf32 offers in principle a lot faster throughput all of these numbers everywhere are still float 32s and it's float 32 numbers that are being shipped all over the place through the memory system and is just costing us way too much time to shuttle around all this data and so even though we've made the multiply itself much faster uh we are memory bound and we're not actually seeing the full benefit uh that would come from uh this napkin math here uh that said we are getting one a 3X faster throughput and this is free um single line of code in P torch all your variables are still float 32 everywhere it just runs faster and it's slightly more approximate but we're not going to notice it basically uh so that's tf32 okay so let's now continue so we've