02:06:54 nice/ugly numbers. vocab size 50257 50304, 93ms

now getting to one of my favorite optimizations and it is simultaneously the dumbest and the most brilliant optimization and it's always a little bit surprising to me um anyway so basically I mentioned a few minutes ago that there are some numbers that are nice and some numbers that are ugly so 64 is a beautiful nice number 128 is even nicer 256 is beautiful what makes these numbers beautiful is that there are many powers of two inside them you can divide by two many times and uh examples of ugly numbers are like 13 and 17 and something like that prime numbers numbers that are not even and so on and so pretty much you always want to use nice numbers in all of your code that deals with neural networks or Cuda because everything in Cuda Works in sort of like powers of two and lots of kernels are written in terms of

powers of Two And there are lots of blocks of sizes 16 and uh 64 and so on so everything is written in those terms and you always have special case handling for all kinds of uh logic that U when your inputs are not made of nice numbers so let's see what that looks like basically scan your code and look for ugly numbers is roughly theistic so three times is kind of ugly um I'm not 100% sure maybe this can be improved but this is uh this is ugly and not ideal um four times is nice so that's uh that's nice 1024 is very nice that's a power of two 12 is a little bit suspicious um not too many powers of two 768 is great 50, 257 is a really really ugly number um it's first of all it's odd so uh and there's no not too many powers of two in there so this is a very ugly number and it's highly suspicious and then when we scroll down all these numbers are nice and then here we have mostly nice numbers except for 25 so in this

configuration of gpt2 XL a number of heads is 25 uh that's a really ugly number that's an odd number and um actually this did cause a lot of headaches for us recently when we're trying to optimize some kernels uh to run this fast um and required a bunch of special case handling so basically these numbers are we have some ugly numbers and some of them are easier to fix than others and in particular the voap size being 50257 that's a very ugly number very suspicious and we want to fix it now when you when you fix these things uh one of the easy ways to do that is you basically um increase the number until it's the nearest power of two that you like so here's a much nicer number it's 50304 and why is that because 50304 can be divided by 8 or by 16 or by 32 64 it can even be divided by 128 I think yeah so it's a very nice number um so what we're going to do here is the GPT config and you see that we initialized B

cap size to 50257 Let's override just that um element to be 50304 okay so everything else stays the same we're just increasing our vocabulary size so we're adding it's almost like we're adding fake tokens uh so that book up size has powers of two inside it now actually what I'm doing here by the way is I'm increasing the amount of computation that our network will be doing if you just count the the flops on like do the math of how many flops we're doing we're going to be doing more flops and we still have to think through whether this doesn't break anything but if I just run this uh let's see what we get uh currently this ran in maybe 96.5 milliseconds per step I'm just kind of like eyeballing it and let's see what kind of a result we're going to get uh while this is compiling let's think through whether our code actually works okay when we increase the vocap size like this let's look at where vocap size is actually used so we swing up

to the inet and we see that it's used inside the embedding table of course so all the way at the bottom of the Transformer and it's used at the classifier layer all the way at the top of the Transformer so in two places and let's take a look and we're running at 93 so 93 milliseconds instead of 96.5 so we are seeing a roughly yeah 4% Improvement here uh by doing more calculations and the reason for this is we fixed we've made an ugly number into a nice number let's I'm going to come into the explanation for that a little bit again but for now let's just convince ourselves that we're not breaking anything when we do this so first of all we've made the the wte the embedding table for the tokens we've made it larger it's almost like we introduced more tokens at the bottom and these tokens are never used because the gbt tokenizer only has tokens up to $50,000 256 and so we'll never index into the rows that we've added so we're wasting

a little bit of space here by creating memory that's never going to be accessed never going to be used Etc now that's not fully correct because this wte weight ends up being shared and ends up being used in the classifier here at the end so what is that doing to the classifier right here well what what that's doing is we're predicting additional Dimensions at the classifier now and we're predicting probabilities for tokens that will of course never be present in the training set um and so therefore the network has to learn that these probabilities uh have to be driven to zero and so the logits that the network produces have to drive those dimensions of the output to negative Infinity but it that's no different from all the other tokens that are already in our data set um or rather that are not in our data set so Shakespeare only probably uses let's say a th000 tokens out of 50,000 to 57 tokens so most of the tokens are already being driven

to zero probability by the optimization we' just introduced a few more tokens now that in a similar manner will never be used and have to be driven to zero in probability um so functionally though nothing breaks we're using a bit more extra um memory but otherwise this is a harmless operation as far as I can tell but and we're adding calculation but it's running faster and it's running faster because as I mentioned in Cuda so many kernels use uh block tiles and these block towels are usually nice numbers uh so powers of two so calculations are done in like chunks of 64 or chunks of 32 and when your um when your desired calculation doesn't neatly fit into those block tiles um there are all kinds of boundary kernels that can kick in to like do the last part so basically in a lot of kernels they will chunk at up your input and they will do the nice part first and then they have a whole second second phase where they come back to any

that like uh remains uh and then they process the remaining part and the kernels for that could be very inefficient and so you're basically um spinning up all this extra compute and is extremely inefficient so you might as well pad your inputs and um make it fit nicely and usually that empiric lens up actually running faster um so this is another example of a 4% Improvement that we've added and this is something that also torch compile did not find for us you would hope that torch compile at some point could figure an optimization like this out uh but for now uh this is it and I also have to point out that we're using pytorch nightly so that's why we're only seeing 4% if you're using pytorch 2.3.1 or earlier you would actually see something like 30% Improvement just from this change from changing it to from 50,000 to 57 to 50304 so again one of my favorite examples also of having to understand the under the hood and how it

all works and to know what kinds of things to Tinker with to push the performance of your code okay so at this point we have