02:46:52 distributed data parallel (DDP)

time to bring out the heavy weapons uh you've noticed that so far we've only been using a single GPU for training but actually I am paying for eight gpus here and so uh we should be putting all of them to work and in particular they are going to collaborate and uh you know optimize over tokens at the same time and communicate so that um uh they're all kind of collaborating on the optimization for this we are going to be using the distributed data parallel from pytorch there's also a legacy data parallel which I recommend you not use and that's kind of like you know Legacy distributed data parallel Works in a very simple way we have eight gpus so we're going to uh launch eight processes and each process is going to be assigned to GPU and for each process the training Loop and everything we've worked on so far is going to look

pretty much the same H GPU as far as it's concerned is just working on exactly what we've built so far but now Secret L there's eight of them and they're all going to be processing slightly different parts of the data and we're going to add one more part where once they all calculate their gradients there's one more part where we do a average of those gradients and so that's how they're going to be collaborating on uh the computational workload here so to use all eight of them we're not going to be launching our script anymore with just um pytorch train gbt2 piy we're going to be running it with a special command called torrun in pytorch we'll see that in a bit and torrun uh when it runs our python script we'll actually make sure to run eight eight of them in parallel and it creates these environmental variables where each of these processes can look up which uh basically which one of the processes it is so

for example torron will set rank local Rank and World size environmental variables and so this is a bad way to detect whether uh DDP is running so if we're using torch run if DDP is running then uh we have to make sure that K is available because I don't know that you can run this on CPU anymore or that that makes sense to do um this is some um setup code here the important part is that there's a world size which for us will be eight that's the total number of processes running there's a rank which is um each process will basically run the ex exact same code at the exact same time roughly but all the process the only difference between these processes is that they all have a different dtp rank so the um gpu0 will have DDP rank of zero GPU 1 will have uh rank of one Etc so otherwise they're all running the exact same script it's just that DDP rank will be a slightly different integer and that is the way for us to

coordinate that they don't for example run on the same data we want to we want them to run on different parts of the data and so on now local rank is something that is only used in a multi- node setting we only have a single node with ag gpus and so local rank is the rank of the GPU on a single node so from 0 to seven as an example but for us we're mostly going to be running on a single box so the things we care about are Rank and World size this is eight and this will be whatever it is depending on the GPU uh that uh that this particular instantiation of the script runs on now here we make sure that according to the local rank we are setting the device to be Cuda colon and colon indicates which GPU to use if there are more than one gpus so depending on the local rank of this process it's going to use just the appropriate GPU so there's no collisions on which GPU is being used by which process and finally there's a Boolean

variable that I like to create which is the DDP rank equ equal Z so the master process is arbitrarily process number zero and it does a lot of the printing logging checkpointing Etc and the other processes are thought of mostly as a compute processes that are assisting and so Master process zero will have some additional work to do all the other processes will uh will mostly just be doing forward backwards and if we're not using DDP and none of these variables are set we revert back to single GPU training so that means that we only have rank zero the world size is just one uh and and we are the master process and we try to autodetect the device and this is world as normal so so far all we've done is we've initialized DDP and uh in the case where we're running with torrun which we'll see in a bit there's going to be eight copies running in parallel each one of them will have a different Rank and now we have to make

sure that everything happens uh correctly afterwards so the tricky thing with running multiple processes is you always have to imagine that there's going to be eight processes running in parallel so as you read the code now you have to imagine there's eight you know eight python interpreters running down these lines of code and the only difference between them is that they have a different DDP rank so they all come here they all pick the exact same seed they all make all of these calculations completely unaware of the other copies running roughly speaking right so they all make the exact same calculations and now we have to adjust these calculations to take into account that there's actually like a certain world size and certain ranks so in particular these micro batches and sequence lengths these are all just per GPU right so now there's going to be num processes of them running in parallel so we have to adjust this

right because the grum steps now is going to be total B size divide B * T time U DDP R size because each um process will will do B * T and there's this many of them and so in addition to that we we want to make sure that this fits nicely into total batch size which for us it will because 16 * 124 * 8 8 gpus is 131 uh K and so 524288 this means that our gratum will be four with the current settings right so there's going to be 16 * 124 process on each GPU and then there's a GP pus so we're going to be doing 131,000 tokens in a single forward backward on the 8 gpus so we want to make sure that this fits nicely so that we can derive a nice gradient accumulation steps and uh yeah let's just adjust the comments here times uh DDP World size okay so each GPU calculates this now this is where we start to get run into issues right so we are each process is going to come by a print and they're all going to print so we're going to

have eight copies of these prints so one way to deal with this is exactly this master process variable that we have so if Master process then guard this and that's just so that we just print this a single time because otherwise all the processes would have computed the exact same variables and there's no need to print this eight times um before getting into the data loader and we're going to have to refactor it obviously maybe at this point is uh we should do some prints and uh just take it out for a spin and exit at this point so import sis and S start exit and print IM GPU um DDP rank IM GPU DDP Rank and that um print by so uh so now let's try to run this and just see how this works so let's take it for a spin just so we see what it looks like so normally we use to launch python train gpd2 P like this now we're going to run with torch run and this is what it looks like so torch run Standalone number of processes for example is eight

for us because we have eight gpus uh and then change of2 Pi so this is what the command would look like and torch run again we'll run eight of these so let's just see what happens so first it gets a little busy so there's a lot going on here so first of all there's some warnings from distributed and I don't actually know that these mean anything I think this is just like the code is setting up and the processes are coming online and we're seeing some preliminary failure to collect while the processes come up I'm not 100% sure about that but we start to then get into actual prints so all the processes went down and then the first print actually comes from process 5 uh just by chance and then it printed so process 5 basically got here first it said I'm process on GPU 5 buy and then this these prints come from the master process so process 5 just finished first for whatever reason it just depends on how the operating system

scheduled the processes to run uh then gpu0 ended then GPU 3 and two and then uh probably process 5 or something like that has uh exited and and DDP really doesn't like that because we didn't properly dispose of uh the multi-gpus um setting and so process group has not been destroyed before we destruct uh so it really doesn't like that and in an actual application we would want to call destroy process group uh so that we clean up DDP properly and so it doesn't like that too much and then the rest of the gpus finish and that's it so basically we can't guarantee when these processes are running it's totally but they are running in parallel we don't want them to be printing um and next up let's erase this next up we want to make sure that when we create data loader light we need to now make it aware of this multi-process um setting because we don't want all the processes to be loading the exact same data we want

every process to get its own chunk of data so that they're all working on different parts of the data set of course so let's adjust that so one particular particularly simple and a naive way to do this is we have to make sure that we pass in the rank and the size to the data loader and then when we come up here we see that we now take Rank and processes and we save them now the current position will not be zero uh because what we want is we want to stride out all the processes so one way to do this is we basically take S.B times salt. T and then multiply it by the process rank so proc process rank 0 will start at zero but process rank one now starts at B * T process rank two is starts at 2 * B * D Etc so that is the initialization now we still they still do this identically but now when we advance we don't Advance by B * T we advance by B * T times number of processes right so basically um the total number of tokens that

we're um consuming is B * T * number processes and they all go off to a different Rank and the position has to advance by the entire chunk and then here B * T time uh s. num processes + one would be to exceed number of tokens then we're going to Loop and when we Loop we want to of course Loop in the exact same way so we sort of like reset back uh so this is the simplest change that I can uh find for kind of a very simple distributed data Lo light and um you can notice that if process rank is zero and non processes is one then uh the whole thing will be identical to what we had before but now we can have actually multiple processes uh running and this should work fine um so that's the data loader okay so next up once they've all initialized the data loader they come here and they all create a GPT model uh so we create eight GPT models on eight processes but because the seeds are fixed

here they all create the same identical model they all move it to the device of their Rank and they all compile the model and because the models are identical there are eight identical compilations happening in parallel but that's okay now none of this uh changes because that is on a per step basis and we're currently working kind of within step because we need to um just uh all the all the changes we're making are kind of like a within step changes now the important thing here is when we construct the M model we actually have a bit of work to to do here get loits is deprecated so uh create model we need to actually wrap the model into the distributed data parallel container so um this is how we wrap the model into the DDP container and these are the docs for DDP and they're quite extensive and there's a lot of caveats and a lot of things to be careful with because everything complexifies times 10 when multiple

processes are involved but roughly speaking this device IDs I believe has to be passed in now unfortunately the docs for what device IDs is is is extremely unclear uh so when you actually like come here this comment for what device IDs is is roughly nonsensical um but I'm pretty sure it's supposed to be the DDP local rank so not the DDP rank the local rank uh so this is what you pass in here this wraps the model and in particular what DDP does for you is in a forward pass it actually behaves identically so um my understanding of it is nothing should be changed in the forward pass but in the backward pass as you are doing the backward pass um in the simpl setting once the backp passes over on each independent GPU each independent GPU has the gradient for all the parameters and what DDP does for you is once the backward pass is over it will call what's called all reduce and it basically does an

average across all the uh ranks of their gradients and and then it will deposit that average on every single rank so every sing Single rank will end up with the average on it and so basically that's the communication it just synchronizes and averages the gradients and that's what DDP offers you now DDP actually is a little bit more um it is a little bit more involved than that because as you are doing the backward pass through the layers of the Transformer it actually can dispatch Communications for the gradient while the backward pass is still happening so there's overlap of the uh communication of the gradient and the synchronization of them and uh the backward pass and uh this is just more efficient and um uh to do it that way so that's what DDP does for you um forward is unchanged and backward is mostly unchanged and we're tacking on this average as we'll see in a bit okay so now

let's go to the uh optimization nothing here changes let's go to the optimization here the inner loop and think through the synchronization of uh these gradients in the DP so basically by default what happens as I mentioned is when you do l. backward here it will do the backward pass and then it will synchronize the gradients um the problem here is because of the gradient accumulation steps Loop here we don't actually want to do the synchronization after every single La step backward because we are just depositing gradients and we're doing that serially and we just want them adding up and we don't want to synchronize every single time that would be extremely wasteful so basically we want to add them up and then on the the very last uh it's only on the very last step when micro when micro step becomes gratak steps minus one only at that last step do we want to actually do the alberu uh to average up the

gradients so to do that we come here and um the official sanctioned way by the way is to do this no sync context manager so pytorch says this is a context manager to disable gradient synchronization across DDP processes So within this context gradient will be accumulated and basically when you do no sync there will be no communication so they are telling us to do with DDP no sync uh do the gradient accumulation accumulate grats and then they are asking us to do DDP again with another input and that backward and I just really don't love this I I just really don't like it uh the fact that you have to copy paste your code here and use a context manager and this is just super ugly so when I went to this source code here you can see that when you enter you simply toggle this variable this require backward grat sync and this is uh being toggled around and changed and this is the variable that basically uh if you

step through it is being toggled to determine if the gradient is going to be synchronized so I actually just kind of like to use that directly uh so instead what I like to do is the following right here before the L back backward if we are using the DDP then um then basically we only want to synchronize we only want this variable to be true when it is the final iteration in all the other iterations inside the micr steps we want to be false so I just toggle it like this so required backward graph sync should only turn on when the micro step is the last step and so I'm toggling this variable directly and I hope that that impacts last St backwards and this is a naughty thing to do because you know they could probably change the DDP and this variable will go away but for now I believe this this works and it allows me to avoid the use of context managers and code duplication I'm just toggling the variable and then Lop backward will not synchronize

most of the steps and it will synchronize the very last step and so once this is over uh and we come out every single um rank will suddenly magically have the average of all the gradients that were stored on all the ranks so now we have to think through whether that is what we want and also um if this suffices and whether how it works with the loss and what is loss AUM so let's think through through that now and the problem I'm getting at is that we've averaged the gradients which is great but the loss AUM has not been impacted yet and the and this is outside of the DDP container so that is not being averaged um and so here when when we are printing Los AUM well presumably we're only going to be printing on the master process uh rank zero and it's just going to be printing the losses that it saw on its process but instead we want it to print the loss over all the processes and the average of that loss because we did

average of gradients so we want the average of loss as well so simply here after this uh this is the code that I've used in the past um and instead of LF we want Lum so if DDP again then this is a p torch distributed I import it where do I import it uh oh gosh so this file is starting to get out of control huh so if uh so import torch. distributed as dist so dist. ALU and we're doing the average on Lum and so this lakum tensor exists on all the ranks when we call all use of average it creates the average of those numbers and it deposits that average on all the ranks so all the ranks after this um call will now contain L AUM uh averaged up and so when we print here on the master process the L AUM is identical in all the other ranks as well so here if Master process oops we want to print like this okay and finally we have to be careful because we're not processing even more tokens so times DDP World size that's

number of tokens that we've processed up above and everything else should be fine uh the only other thing to be careful with is as I mentioned you want to destroy the process group so that we are nice to nickel and it's not going to uh to uh to DDP and it's not going to complain to us uh when we exit here so that should be it let's try to take it for a spin okay so I launched the script and it should be uh printing here imminently we're now training with 8 gpus at the same time so the gradient accumulation steps is not 32 it is now divide 8 and it's just four uh so um otherwise this is what the optimization now looks like and wow we're going really fast so we're processing 1.5 million tokens uh per second now so these are some serious numbers and the tiny shakespare data set is so tiny that we're just doing like so many Epoch over it most likely but this is roughly what looks like um one thing that I had to fix by the way is that this was model. configure

optimizers which Now doesn't work because model now is a DDP model so instead this has to become raw model. configure optimizers where raw model is something I create here so right after I wrap the model into DDP uh I have to create the raw model which in the case of DDP is a model. module is where it stores the raw and then module of gpt2 as we have it which contains the uh configure optimizers function that we want to call so that's one thing that I have to fix otherwise this seems to run now one thing you'll notice is that when you actually compare this run and the numbers in it to the just running a single GPU you'll notice that this is single GPU run with 32 gratum the numbers won't exactly match up and uh that's kind of a boring reason for why that happens uh the reason for that is that in the data loader we're basically just iterating through batches and slightly different way because now we're looking for an entire

page of data and if that page uh for all the gpus if that chunk exceeds the number of tokens we just Loop and so actually the single GPU and the H GPU process will end up um resetting in a slightly different Manner and so our batches are slightly different and so we get slightly different numbers but one way to convince yourself that this is okay it just make the total batch size much smaller and the b and a t and then um so I think I used uh 4 * 124 * 8 so I used 32768 as a total patch size and then um so I made sure that the single GPU will do eight creting accumulation steps and then the multi-gpu and then you're reducing the boundary effects of the data loader and you'll see that the numbers match up so long story short we're now going really really fast the optimization is mostly consistent with gpt2 and three hyper parameters and uh we have outgrown our tiny Shakespeare file and we want to

upgrade it so let's move to next to that next

so let's now