Summary of “Perseus: Reducing Energy Bloat in Large Model Training”

Brandon Zhang (bdwzhang), Justin Paul (justpaul), Sreya Gogineni (gosreya), Sarah Stec (sastec)

Problem and Motivation

Training large AI models, like the GPT series or Llama-405B, requires an enormous amount of energy. These energy demands inflate operational costs and even present a challenge for scaling AI models: one of the primary difficulties in scaling data centers to run larger AI workloads has been with delivering enough power. Hardware advancements offer general-purpose performance gains, but software can take advantage of application-specific characteristics to reduce energy inefficiency in the training workflow itself.

Not all of the energy used during pipelined training processes directly contributes to throughput- some of this energy expenditure can be removed without slowing down training. This energy gone to waste, termed energy bloat, stems from two sources. Intrinsic energy bloat is caused when certain pipeline stages are run at maximum speed to finish faster, expending more energy, but have their results sit idly while waiting for stages on the critical path, the sequence of pipeline stages that takes the longest to run, to finish. Extrinsic energy bloat is caused in multi-pipeline setups when whole pipelines are run at high speeds to finish quickly but have to wait idly for straggler pipelines to finish due to unexpected issues like thermal throttling, hardware failure, or I/O bottlenecks. By minimizing energy bloat, large models can be trained using less energy without a loss in training speed.

Related Works

Previous works that attempt to improve the energy efficiency of model training include Optsched, which attempts to balance workloads across resources to optimize energy use, and Zeus, which captures the energy-time tradeoff for single GPU training. Both of these methods fall short in scalability for large model training workloads.

Currently, large training workloads rely on pipeline parallelism, introduced by GPipe and Pipedream. While these systems massively increased training throughput, they were not optimized for energy efficiency and faced imbalances in pipeline completion times. Some works simply fill the idle time from early completed pipeline stages to insert other useful work, but doing so can slow down the pipeline or require repartitioning the pipeline and are limited in what work they can execute as the memory is being mostly used by the pipeline. EnvPipe, rather than inserting work, slows down the computation of pipeline stages to match the runtime of the critical path. However EnvPipe makes the often incorrect assumption that the final pipeline stage is the critical path, and it doesn’t account at all for extrinsic energy bloat.

Solution Overview

The paper introduces an optimization framework, Perseus, that attempts to remove both intrinsic and extrinsic energy bloat. The goal is to identify critical paths and stragglers so that parallel stages and pipelines can be set to run slower and finish at the same time. Perseus does so by determining the optimal clock speed for each stage and pipeline to minimize the energy used without increasing the overall training time.

In order to find opportunities to save power, Perseus creates a directed acyclic graph (DAG) that notates *expected* time and energy consumption called the "energy schedule." Each edge in the DAG represents a computation task (forward or backward pass in a layer) and nodes represent dependencies (for a particular node, incoming edges must finish before outgoing edges can begin). The DAG allows Perseus to determine the critical path and then decide which non-critical node computations can be slowed down to reduce power consumption.

Perseus consists of a server and a client. The client is responsible for profiling the time and energy usage of the pipelines and pipeline stages. After the client sends profiling information from the initial stages of training, while the model continues to train, the server creates a DAG and reasons about various energy energy schedules. When an optimal energy schedule has been computed based on the critical paths, it sends the optimal running times to the client to enforce for each node. The client then sets GPU clock frequencies. When the client sets a slower clock frequency for a task, the task will take longer to complete, but will consume less energy and spend less time idle waiting for the critical path. As the client continues profiling and detects stragglers, it will ask the server for an updated energy schedule time that it should employ.

Perseus largely outperformed EnvPipe in energy savings and Google's T5 3B model training showed that Perseus provided 28.5% energy savings with no throughput loss.

Limitations

Perseus relies on effective straggler detection in order to make use of the energy-saving mechanism, a difficult task due to the unpredictability of straggler behavior. Perseus also requires GPUs that support modulating clock speeds.

Using Perseus for model training has some challenges as it is not easily integrated into the Megatron-LM and Deepspeed frameworks, which are preferred for training very large models.

Future Research Directions

A future direction would be to better integrate Perseus with Megatron-LM and Deepspeed frameworks. As mentioned in class q&a, it would be good to look for ways to incorporate Perseus DAG optimization framework into inference. Perseus also requires reasonably accurate profiler measurement data. Future research could be done in both directions: how much more optimal could Perseus be if its profiling has little noise and how well can it perform when profilers may have lots of noise causing inaccurate readings.

Summary of Class Discussion

Q: How often is the straggler detection happening?
A: The profiling for detecting stragglers is happening while the training is running. One thing to note is the client/server will not slow down any tasks to an iteration time greater than T* even if the task at T* is expected to finish sooner than the straggler because energy usage would increase past iteration time T*.

Q: Why, when you keep increasing the clock frequency, does the energy go back up?
A: Energy (Watt-hours) = power*time. As you reduce the clock speed, the power may keep decreasing, but at some point the power savings cannot make up for the time increase, and the total energy use starts to increase.

Q: How do you know a straggler is going to be slow ahead of time since it may be slow due to external factors you can't predict?
A: The client, anytime it detects a straggler, sends a request to the server for its precomputed optimal energy schedule given computation task X as a straggler. But straggler detection by the client is indeed difficult- it could be straggling transiently, for example- there are whole papers dedicated to straggler detection and people continue to work on improving methods.

Q: Why would Perseus, a framework that is more focused on software optimization, have such a different impact across different hardware?
A: Though the optimizations could be similar, the hardware architectures are very different from one another: there are different energy/time frontiers for different hardwares. Hyperscalers will definitely be looking for opportunities to make chips even more energy/time efficient.

Q: How can the Perseus architecture be applied to inference?
A: Computations can be run slower because tasks are bottlenecked by memory movement. Perseus requires a DAG in order to determine the critical (and non-critical paths that can achieve energy savings). One natural combination is to take the DAG created in Parrot where there are dependencies between requests. Another is applying it within each model instance in inference, but there is probably less energy to be saved because it is a simple forward pass.

Q: Is Perseus being used anywhere you know of? Do you have plans to continue to work on Perseus?
A: There is a startup based in Europe using it! There are a lot of industry interest, but a complaint is that it's not very easily integrated with Megatron and Deepspeed. Works well with the HuggingFace transformer library, but for very large models, people would prefer to use Megatron. There's also a lot of work required to maintain Perseus (fixes and enhancements), and it's hard for outside contributors to help because distributed training requires a lot of resources to test.

Summary of “DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency”

Brandon Zhang (bdwzhang), Justin Paul (justpaul), Sreya Gogineni (gosreya), Sarah Stec (sastec)

Problem and Motivation

LLMs are widely adopted in a variety of applications, causing LLM inference clusters to receive large quantities of queries. To keep up with consumer demands, these LLMs have Service Level Objectives (SLOs) that outline performance expectations for each query (i.e. speed, reliability, etc.), forcing GPUs to run with maximum performance. This leads to excessive energy consumption on power-hungry GPUs and significant carbon emissions.

Ideally, this excessive energy consumption can be minimized by dynamically reconfiguring GPU resources to different workloads. While this approach is intuitive, there are three key challenges with this dynamic reconfiguration. First, LLM workloads are heterogeneous, leading to differing energy-performance profiles at the request granularity. For instance, greater request lengths, high request loads, high-compute bound models, and differing SLOs can all contribute to a higher energy-performance profile. Second, LLM workloads are highly dynamic in model diversity, request length, and number of requests, causing energy-optimal configurations to quickly become sub-optimal. Finally, reconfigurations may incur significant overheads.

Related Works

Relevant works have explored energy-efficiency for CPU workloads, but researchers have also begun exploring the energy properties of GPU workloads. More specifically, recent works have focused on managing energy consumption for deep neural network (DNN) inference and training via the following techniques: frequency scaling, autoscaling, and resource partitioning and mapping These prior works are limited in that they only optimize efficiency based on one facet of energy consumption.

With respect to efficient LLM inference serving, recent works suggest a variety of approaches, including memory and key-value cache management, node and cluster-level scheduling, and the use of heterogeneous resources and platforms. These works primarily focus on improving LLM inference from the latency or throughput perspectives, which may not necessarily improve energy efficiency.

Solution Overview

The paper proposes DynamoLLM, an LLM inference framework which dynamically reconfigures clusters based on load characteristics to minimize energy while meeting SLOs. Upon instantiating a DynamoLLM instance, the system will profile the expected energy consumption on the specified model, varying request length, number of requests, GPU

frequency, and model parallelism. This profiling also informs future decisions as DynamoLLM aims to reconfigure clusters to minimize energy consumption by adjusting the nodes per pool, GPU frequency, and tensor parallelism.

To minimize energy wasted by idling, DynamoLLM uses historic data to forecast the peak request load, then its cluster manager will allocate the minimal number of GPUs needed per node pool. Each pool handles a different input/output request length and is optimized for its request type. Incoming requests are processed by a BERT model to predict output lengths and are routed to the corresponding node pool. This ensures requests can meet their SLOs, while minimizing the number of GPUs reduces idling and thus saves energy. After every epoch (i.e. 30 minutes), the cluster manager will dynamically reconfigure node pools again to iteratively improve energy efficiency.

DynamoLLM also recognizes that a lower GPU frequency corresponds to lower power and energy consumption. To leverage this observation, DynamoLLM's instance managers find the optimal GPU frequency profile to process requests while still meeting SLOs. For instance, with a lower workload, nodes can afford to operate at a lower frequency, which will operate slower, but will return a response within the SLOs while using less energy.

Reducing tensor parallelism in DynamoLLM can also reduce energy consumption by reducing high-bandwidth communication between GPUs. Like instance managers controlling GPU frequency, pool managers limit tensor parallelism (number of GPUs per instance) to minimize energy based on profiled data. To compensate for this decrease, pool managers increase model parallelism (number of instances) in their pools to ensure all GPUs remain utilized.

While adjusting these knobs can reduce energy consumption, the associated reconfigurations can yield significant overhead. DynamoLLM accounts for this by only reconfiguring when it will save energy, and by readjusting knobs in intervals proportional to the cost of reconfiguration. For instance, reallocating nodes to pools incurs significant reconfiguration costs due to time intensive tasks like creating a VM and loading model weights, so this is done infrequently. DynamoLLM uses various specific optimizations to further reduce overhead. To maintain performance stability, all reconfigurations are staggered which prevents sudden spikes in latency.

Due to its consistent dynamic reconfiguration across multiple knobs, DynamoLLM was found to use 53% less energy and reduce 61% of customer costs compared to non-energy-aware systems.

Limitations

DynamoLLM does not attempt to minimize energy in pipeline parallelism, leaving it for future work. It also does not justify its categorization of request lengths into short, medium, and long (as opposed to more or fewer categories) or its percentile thresholds for these categories. Additionally, by sometimes decreasing its performance to reduce energy, DynamoLLM leaves itself more vulnerable to system overload (which can lead to drastic measures like squashing requests) compared to solutions which do not attempt to minimize energy consumption.

Future Research Directions

Future research could attempt to categorize requests by characteristics other than just input/output lengths to allow further optimizations for each type of request, for example the paper considers Conversation requests and Coding requests. The predictions of load characteristics could also be improved to reduce mispredictions and the issues that come along with them. The techniques in DynamoLLM could also be adapted to non-GPU sources of power consumption, such as CPUs. More types of SLOs could also be considered by future research, such as SLOs concerned with the latency of the full output instead of individual tokens. Any of the issues outlined in limitations could also be addressed by future work.

Summary of Class Discussion

Q: How is the estimation of output length done in DynamoLLM?
A: DynamoLLM uses a BERT-based prediction model, which is the current state-of-the-art method in predicting LLM request outputs.

Q: How can Dynamo know if SLOs will be satisfied by looking at a single request? Wouldn't pending requests change the pool's ability to satisfy an SLO?
A: DynamoLLM uses historical data to determine the peak load for each pool, and will provision resources with the assumption that each pool will run at that peak load. This allows each pool to handle each individual request while meeting the SLO, as the pool will always be running at or below that peak load. DynamoLLM will also reevaluate each pool's peak load after every epoch.

Q: After prediction, is it possible to "go back" and switch configuration after determining the prediction was incorrect?
A: Nope! Mispredictions are inevitable. If a request length is over-predicted, we simply use too much energy, but we still meet our SLOs. If we under-predict the output length, its SLOs may not be met, which impacts user experience. In this case, DynamoLLM triggers an "emergency event," where it may try to re-order requests or amp up the GPU frequency to process requests faster.

Q: Prediction models cannot always predict output length correctly which could cause requests to be sent to the wrong pools. Could this hurt DynamoLLM's efficiency?
A: Yes, and the authors address this in evaluations done on Azure. They compare TTFT and and energy with varying mispredictions from 0-50% in 10% increments and found that increasing mispredictions increase energy and TTFT, but both are still better than compared to SinglePool for up to 40% mispredictions.