Calculon: a Methodology and Tool for High-Level Codesign of Systems and Large Language Models

Mikhail Isaev
michael.v.isaev@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Nic McDonald
nimcdonald@nvidia.com
NVIDIA
Salt Lake City, Utah, USA

Larry Dennison
ldennison@nvidia.com
NVIDIA
Westford, Massachusetts, USA

Richard Vuduc
richie@cc.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

This paper presents a parameterized analytical performance model of transformer-based Large Language Models (LLMs) for guiding high-level algorithm-architecture codesign studies. This model derives from an extensive survey of performance optimizations that have been proposed for the training and inference of LLMs; the model's parameters capture application characteristics, the hardware system, and the space of implementation strategies. With such a model, we can systematically explore a joint space of hardware and software configurations to identify optimal system designs under given constraints, like the total amount of system memory. We implemented this model and methodology in a Python-based open-source tool called Calculon. Using it, we identified novel system designs that look significantly different from current inference and training systems, showing quantitatively the estimated potential to achieve higher efficiency, lower cost, and better scalability.

ACM Reference Format:
Mikhail Isaev, Nic McDonald, Larry Dennison, and Richard Vuduc. 2023. Calculon: a Methodology and Tool for High-Level Codesign of Systems and Large Language Models. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607102

1 INTRODUCTION

We consider the task of conducting high-level analyses for algorithm-architecture codesign of distributed clusters and transformer-based Large Language Models (LLMs) [3, 4, 36, 39, 40]. By "high level," we mean focusing on developing and using fast and coarse-grained analytical or semi-empirical models of the software and hardware, a stage of design that precedes detailed simulation or implementation on actual hardware. The goal is to estimate the best-case relative improvements that might come from significant changes to the system or software, as well as combinations of system and software configurations that might be unusual or otherwise costly and difficult to implement. Since high level models are expected to be much cheaper than detailed simulation, they should facilitate rapid exploration of a potentially large parameter space during the early phases of codesign.

In this work, our starting point is Megatron, a large family of open-source LLM instances developed by NVIDIA [44]. The cost of training such models is high: a version of Megatron having one trillion (1T) parameters was recently trained over 84 days on 450 billion tokens using 3,072 NVIDIA A100 Graphics Processing Unit (GPU) and executing more than 1,000 zettaFLOP (1×10^{21} floating-point operations) [29, 30]. This cost roughly equals seven hundred years on a single GPU and over six million dollars (US) assuming a single GPU at $1 per hour cloud-GPU rates. Incurring such costs is commonplace; the PaLM-540B model recently trained by Google used 2,572 zettaFLOP with similar numbers of Tensor Processing Units (TPUs) and more than 8 million TPU hours [6]. Such high costs strongly motivate any combination of algorithmic, software, or hardware redesign that can reduce them.

Consequently, there are proposed enhancements in algorithms and software [29, 37, 38] and options for hardware acceleration [18, 31]. However, selecting a good configuration in practice has relied primarily on heuristic reasoning [29]. While these proposals have yielded impressive results, it has also been observed that distributed training runs at a FLOP/s rate well below 50% of peak despite the prevalence of matrix multiply (GEMM) operations [34].

The challenge is that the codesign landscape is quite large, making it hard to reason about the impact of major changes to arbitrary combinations of hardware and software. For example, consider that limited GPU memory capacity requires dividing a large model among processors. Doing so can be achieved via model parallelism, which combines two strategies known as tensor parallelism and pipeline parallelism [29]. However, when using NVIDIA A100 GPUs, the size of the NVLink domain is 8, which can limit tensor parallelism performance [32]. To compensate, one can increase the degree of pipeline parallelism—but that may in turn produce other inefficiencies such as reduced utilization due to pipeline bubbles and needing to recompute intermediate features in light of memory constraints [13, 29]. Alternatively, one might improve the computer network to support larger tensor parallelism domains; companies and researchers have indeed considered doing so [17, 32]. However, if memory capacity is the root issue, then a more cost-effective

This work is licensed under a Creative Commons Attribution International 4.0 License.
SC '23, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607102

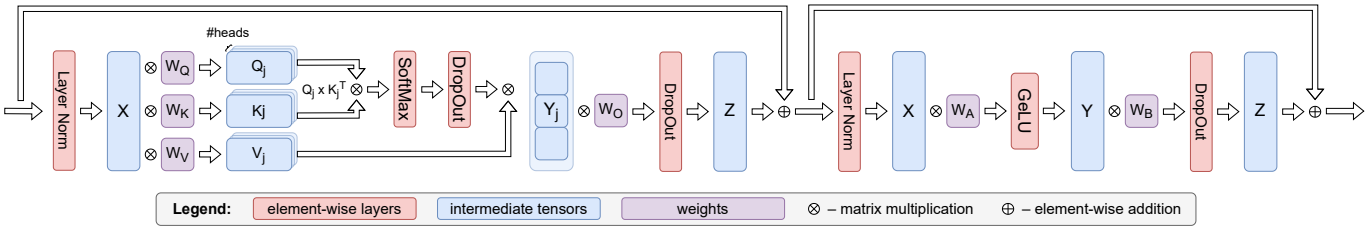

Figure 1: The transformer block structure of a typical LLM, such as GPT-3 or Megatron.

strategy is to increase capacity. Overall, this example shows that codesign should carefully consider and delicately balance memory capacity, memory bandwidth, processing throughput (i.e., FLOP/s), network bandwidth, and network scalability, all of which interact with choices made in software. Therefore, to reason about these trade-offs we seek a principled analysis framework that can be extended or adapted in a hardware and software landscape undergoing rapid and continual evolution.

We propose one such model-driven approach for high-level codesign of LLM training and inference systems. We first identify a parameterized space of possible configurations that span major system features and common algorithmic and software implementation strategies (Section 2). We then develop a unified analytical model to estimate the end-to-end performance of LLM training as a function of the configuration parameters. This model allows us to pose, mathematically, a constrained optimization problem: find the configuration that yields the best performance given fixed system constraints such as memory capacity and system or network size.

We encode this performance model and model-optimizer in a tool called Calculon. The parameter space includes the structure and number of weights in the LLM model, the implementation strategy, and a schematic description of the hardware system (Table 1). Since Calculon's model is analytical, it can calculate and return a complete breakdown of projected training or inference time quickly, typically in much less than a 1 ms per configuration. It thus becomes possible to search an entire configuration space having many millions of combinations in only a few minutes on a standard desktop computer.

This paper includes several analyses we have conducted using Calculon. The modeling formulae themselves are complex to write out in full; therefore, to save space, we focus on the analyses as "proof-of-concept," with detailed formulas appearing in Calculon's open-source repository, where our formulas and assumptions appear in full, allowing replication of our results or modification of our assumptions.[1] The analyses showcase Calculon's potential to facilitate high-level codesign, revealing several system and optimization insights that may contradict conventional wisdom:

(1) None of the existing software-parallelism strategies is uniformly the "best." However, there is an optimal split-parallelism strategy that balances system resources well, with the exact optimum depending on system parameters, as we show.
(2) The speed of LLM training can be a sensitive function of system size, with performance variability (ratio of highest to lowest performer) exceeding 6×. These "efficiency cliffs" come from difficulties mapping LLM structures to a given number of processors when sizes do not "divide evenly."
(3) Adding a second high-capacity tier of memory for tensor offloading reduces performance variability across various LLM configurations and sizes, enabling efficient training of larger models. Moreover, the bandwidth requirement for efficient offloading is within current technological capabilities.

While these findings are estimates, they suggest quantitatively what performance improvements are *possible*, a critical first step for LLM codesign. Our methodology is systematic and rigorous, enabling future exploration via more detailed experiments and simulation.

2 ANALYTICAL MODEL

Calculon is a Python-based analytical performance model for LLM training and inference on large-scale distributed systems. The core analytical model performs a single calculation of time and resource usage. This model is given 3 specifications: the LLM, the system the LLM is running on, and the execution strategy describing how the LLM is run on the system.

2.1 LLM Configuration

We adopt the framework of Megatron [44] for describing the structure of transformer-based [49] LLMs. Megatron can be used to reimplement models such as GPT2 [36] or GPT3 [3], Chinchilla [11], LLaMa [48], and many other popular LLMs.

For training, Megatron uses synchronous mini-batch stochastic gradient descent and the Adam optimizer for weight updates [2, 19]. It may employ transformer-based encoder blocks, decoder blocks, or both. Each transformer block has the same structure and consists of a multi-head attention block followed by a multilayer perceptron (MLP) block, with normalization and residual connection between them, as shown in Fig. 1 (adapted from Shoeybi et al. [44]). Several hyperparameters define the blocks: the hidden size (`hidden`) is the size of the embeddings and MLP layers, the number of attention heads (`attn`) of certain size, the sequence size (`seq`) of the input text, training batch size (`batch`), micro-batch size (`m`), and the number of transformer blocks (`blocks`).

2.2 Hardware Configuration

Calculon models a processor-based distributed system in which computation is assigned to either "matrix" execution or "vector" execution. The performance of each type of computation may be parameterized by input size in a specification file. This feature is useful when, for instance, smaller general matrix multiply (GEMM) operations run at a lower fraction of peak than larger ones [33].

[1] https://github.com/calculon-ai/calculon

Table 1: Optimizations related to LLM training grouped in families related to a system component or particular parallelism they target. Families and optimizations within families are sorted by year. Arrow upward/downward direction represents an increase/decrease in the metric. Color represents an increase (green) or decrease (red) in the performance. The presented relative significance of each optimization's effect is based on experimental evaluation of the search space. Range represents space of Calculon's input parameters.

Optimization	Year	Related system	Comp time	Comp util	Mem time	Mem cap	Mem BW	Net time	Net BW	Range
Data parallelism (DP) [55]	1989	network	–	↑	–	↑↑↑	–	↑	↑	1 .. `batch`
DP overlap [23]	2017	network	↑	↓	–	–	–	↓↓↓	–	true/false
Optimizer sharding [22]	2019	network	↓	–	–	↓↓	–	–	–	true/false
Recompute [5, 10]	2000	compute	↑↑	–	–	↓↓↓	–	–	–	full/attn/none
Fused layers [26]	2018	compute	–	↑↑	↓↓	↓↓	↓	–	–	true/false
Microbatch training [13]	2019	compute	–	↑↑	–	↑↑↑	–	–	–	1 .. `batch`/DP
Pipeline parallelism (PP) [7, 13]	2012	network	↑	↓↓	–	↓↓	–	↑	↑	1 .. `blocks`
PP 1F1B schedule [7, 28]	2012	network	–	–	–	↓↓	–	–	–	true/false
PP interleaving [29]	2021	network	↓	↑↑	–	↑	–	↑	↑↑	1 .. `blocks`/PP
PP RS + AG [20]	2022	network	–	–	–	–	–	↓	↓↓	true/false
Tensor parallelism (TP) [7, 21, 43]	2012	network	↓↓	↓	–	↓↓	↓↓	↑↑↑	↑↑↑	1 .. `attn`
TP RS + AG instead AR [29]	2021	network	–	–	↑	↑	–	↓	↓	true/false
Sequence parallelism (SP) [20]	2022	network	↓	–	↓	↓↓	↓	↑	↑	true/false
TP redo for SP [20]	2022	network	–	–	–	↓	–	↑	↑	true/false
TP overlap [52]	2022	network	↑	↓	–	–	–	↓↓	–	none/pipe/ring
Weight offload [42]	2021	memory	–	–	↑	↓↓↓	↑	–	–	true/false
Activation offload [42]	2021	memory	–	–	↑	↓↓↓	↑	–	–	true/false
Optimizer offload [42]	2021	memory	–	–	↑	↓	↑	–	–	true/false

The memory system of the processor is modeled as a two-level hierarchy wherein the first level is used for direct computation and the second level is used for stashing bulk data until a later time when it is needed (i.e., offloading). Both memory systems have programmable capacities, bandwidths, and size-based efficiencies.

Each computational operation (e.g., GEMM) is fed to a processing model that determines how long it will take. This model considers both the time spent in raw compute (i.e., FLOPs) and the time spent in raw memory accesses [33].

Each processor is able to connect to an arbitrary number of networks. Each network is programmed with a size, bandwidth, latency, and efficiency. A network also has a specification of how it handles each specific operation, which is also the mechanism that models the performance benefits of in-network collectives [32]. Each network also has a value of how much processing power is taken from the processor while the network is operating at full bandwidth. This value is explicitly used when modeling the performance degradation of overlapping communication with computation.

2.3 Execution Configuration

Many performance optimization techniques and implementation strategies have been proposed for transformer-based model training. We surveyed these methods and present them in Table 1. The large compute requirements of LLMs dictate distributed training using many processors or accelerators and using various modes of parallelism. Megatron employs three parallelization strategies: data parallelism (DP), pipeline parallelism (PP), and tensor parallelism (TP). These strategies can lead to a complex execution schedule (Fig. 2). In this schedule, execution of transformer blocks is intertwined with communication related to TP, PP, and DP.

We implement TP as presented in Megatron [44]. While other TP partitioning schemes are possible, the selected one tries to minimize the number of communication instructions per single transformer block, which in the case of small TP partition sizes also reduces the amount of communication traffic. We implement PP using an interleaved schedule presented in [29], as shown in Fig. 2. DP is implemented with an optional overlap. In this case, DP communication for each layer is scheduled as soon as the last microbatch is propagated through that layer, as shown in Fig. 2(b).

Calculon implements all of optimizations from the Table 1. Each optimization is parameterized with its acceptable input-range in the column *"range"*. The space of techniques that Calculon captures grows combinatorially, a major challenge for the implementation. While individual techniques can be described with formulae, they must be combined carefully to ensure their interactions are captured and feasibility constraints are accounted for, as Calculon does.

We based the Calculon performance model on many transformer-specific optimizations described in the open literature. The starting point is Megatron [44], which combines many optimization families including DP [55], activation recompute [5, 10], and TP [44]. The Megatron team later added many PP-related features[13, 28, 29], including micro-batching, 1F1B and interleaved scheduling, and TP-related optimizations [20, 29], such as splitting all-reduce communication into reduce-scatter and all-gather to optimize PP traffic and adding sequence parallelism. Calculon also implements most optimizations from DeepSpeed [41], including optimizer sharding [22, 37], and tensor offloading [38, 42]. Several optimizations

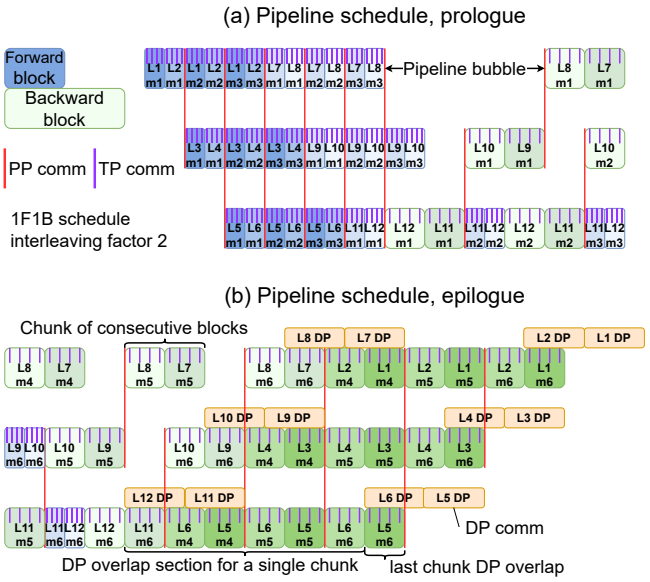

Figure 2: A pipeline schedule of LLM training with an interleaved schedule [29]. Every transformer block (Fig. 1) is identified by its block number starting with L and microbatch id starting with m. Several consecutive blocks in each processor are grouped into chunks. The backward pass for every block-microbatch combination is scheduled right after the data for it becomes available. A pipeline schedule consists of a prologue phase shown in (a) and epilogue phase shown in (b), with the in-between steady phase omitted for brevity. The epilogue demonstrates data parallelism (DP) communication overlapped with backward pass of other transformer blocks.

developed for inference are included as well [1, 35]. Finally, we also included activation fusion for some layers [26] and overlapping DP and TP communication with compute [23, 52].

We do not consider many techniques that make LLM training available on smaller systems with costs of significant slowdowns [38, 46], nor those that target arbitrary networks with worse performance for regular transformer structures [12, 15, 16, 51]. We also do not consider implementation strategies that may non-trivially affect model fidelity, including compression and asynchronous training techniques.

Many techniques implemented in Calculon are de facto standards for LLMs, such as optimizer sharding [22] and microbatch training [13]. Other techniques are scattered among many code bases, such as interleaved schedule for PP only implemented in Megatron [29]. A strength of Calculon is estimating performance for combinations of techniques that have not yet been attempted.

2.4 Performance Calculation

LLMs are implemented in a series of replicated transformer blocks (Fig. 1). Taking advantage of the regular structure, Calculon analyzes the minimum number of unique block structures to make its performance prediction, reusing the results when appropriate. In contrast, many non-LLM specific performance tools needlessly replicate the performance prediction of all blocks independently. This modeling-time optimization allows Calculon to complete a full analysis in under a millisecond.

Calculon distinguishes edge blocks with point-to-point communication for PP and separates the effects of each technique between layers in the transformer block and across the execution stages, such as forward and backward pass, optimizer step, communication phases, etc. Doing so permits distinguishing the effects of interacting model components and a faithful expression of how techniques interact. For example, Calculon forbids DP communication overlap during the optimizer step if optimizer sharding is enabled, and it throttles offloading when high bandwidth memory (HBM) memory is in active use while allowing it during compute-only or network communication phases.

Given a description of the LLM, the system, and the execution strategy, Calculon performs a complete performance estimation using its analytical model and outputs the statistics which contain relevant information about total performance (e.g., batch time, sample rate, etc.) as well as a breakdown of where the time was spent and how much of the available resources were used. The time breakdown includes forward pass, backward pass, recomputation (if used), optimizer, and more. For network time, it reports the amount of time each type of parallelism was communicating on the network and reports how much time was exposed blocking computation. When offloading is used, Calculon reports the amount of total time performing offloading over the second level memory system and also the amount of time exposed (if any) in this process. For memory systems, Calculon reports how much memory was used and which types of data used them (e.g., weights, optimizer state, activations, etc.). Finally, Calculon reports efficiency of the LLM execution.

Fig. 3 shows an example output of running GPT3 175B on 4,096 NVIDIA A100 GPUs each with 80 GiB of memory connected over NVLink clusters of 8 and InfiniBand HDR networks between them. The execution specifies TP=8, PP=64, and DP=8. The total batch time was 16.7 seconds and 20% of that was spent in recomputing the activations during the backward pass. Of the available 80 GiB of HBM memory, 17.4 GiB was used and 29% of the utilized memory was used for optimizer state.

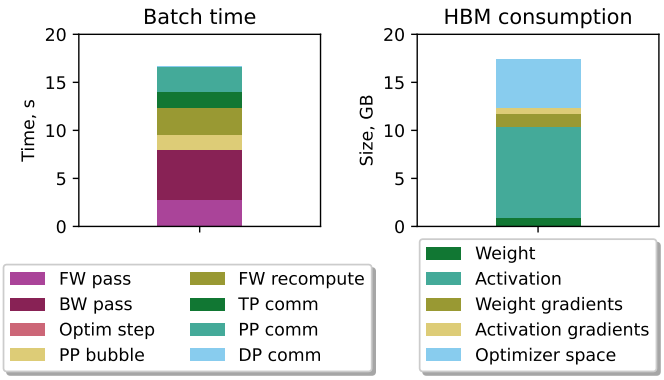

Figure 3: Time and memory consumption for running GPT3 175B across 4,096 GPUs using TP=8, PP=64, DP=8.

2.5 Validation

Table 2: Calculon's validation comparing performance prediction to performance measured on the A100-based Selene supercomputer for full activation recomputation and sequence parallelism plus selective activation recomputation. Performance is batch time measured in seconds.

		22B	175B	530B	1T
Full	**Selene**	1.42	18.13	49.05	94.42
	Calculon	1.40	18.03	49.89	90.08
	Delta	1.72%	0.56%	-1.72%	4.60%
Seq+Sel	**Selene**	1.10	13.75	37.83	71.49
	Calculon	1.14	13.64	34.47	66.04
	Delta	-3.33%	0.81%	8.87%	7.62%

We validated Calculon's modeling accuracy against measured performance of various LLMs on NVIDIA's A100-based Selene supercomputer [20]. We compare the results for modeling Megatron applications sized 22B, 175B, 530B, and 1T, while performing full activation recomputation as well as when applying sequence parallelism with attention-only activation recomputation. The comparison appears in Table 2. For these 8 runs on Selene, Calculon's prediction averages 3.65% error, and the maximum error is 8.87%.

3 RELATED WORK

For codesign, one expects event-driven hardware modeling to be more accurate than analytical modeling but also orders of magnitude slower. NVArchSim [50], a state-of-the-art GPU simulator, can take a day to model 1 second of a single GPU's execution time. Network simulators such as SuperSim takes on the order of a day to model 1 second of network communication between 128 endpoints [14, 27]. SST, a state-of-the-art parallel network simulator, shows a similar speed with single-core modeling [53] and, therefore, requires significant time to simulate the training of Megatron on thousands of accelerators. On the application modeling side, compiler-based models like ParaGraph can extract both application models and optimizations from the compiled representation of the applications [14]. However, it might not be possible to deduce all of the performance optimizations and implementation strategies developed for Megatron, and introducing them explicitly is a very time-consuming process. Furthermore, compiler-based approaches generally work well only for existing hardware.

On the execution side, several projects consider automated model parallelization and partitioning. FlexFlow splits a single iteration along samples, operators, attributes, and parameters dimensions but does not consider pipelining [25]. DAPPLE, PipeDream, and Tarnawski et al. focus on pipeline parallelism scheduling [8, 28, 47]. These efforts do not consider TP or parallelization strategies separate from a broader space of execution optimization techniques.

GShard [24] and GSPMD [54] provide a way to implement parallel transformer or automatically parallelize transformer via XLA compiler [9] after code annotation. XLA compiler supports many other optimizations as well. GSPMD focuses on finding the optimal configuration for existing systems. Alpa [56] builds on top of these projects adding an automated search for optimal combined TP and PP model split. Unlike Calculon, they do not consider a unified space of hardware and software configurations and do not target future system design, both being key features of Calculon. Also, Calculon, being an analytical performance model with fast prediction time, can explore vast configuration spaces. Other solutions, typically working as part of compiler infrastructure or runtime, either cover a smaller space or use heuristics to limit the search, focusing on running training with good performance on the available system.

Finally, NaaS proposes a joint codesign of optimized neural network (NN) and accelerators to achieve the best performance per unit power [57]. Calculon focuses on optimizing large-scale system design and system-level optimizations rather than NN layers.

4 PERFORMANCE TRADE-OFFS

We used Calculon to explore the landscape of system configurations defined by our analytical modeling (Section 2). The two studies that follow study implementation trade-offs given a fixed system (Section 4.1) and how LLM structure, software, and hardware factors interact when exploring the full space of configurations (Section 4.2).

4.1 Parallelization Analysis

We consider the training of Megatron-1T LLM [29] on a baseline system configured with 4,096 NVIDIA A100 GPUs with a global batch equal to 4,096. We try all combinations of TP, PP, and DP to determine the resource allocation with the best performance.

We denote a splitting by (t, p, d) where $t \times p \times d = 4{,}096$ GPUs and each of t, p, and d factors measure the degree of TP, PP, and DP exploitation, respectively. Smaller values of t, p, or d imply "less" of that type of parallelism, with 1 being the minimum value of each. In this case study, we assume a software implementation that employs, as a memory-saving strategy, both optimizer sharding and 1F1B scheduling (Table 1). We then try all possible settings of (t, p, d) where we set the NVLink domain size to $t \leq 32$; we specifically set the NVLink size to the number of GPUs in the TP domain to shed light on the effects (implicit costs) of TP. The impact of changing (t, p, d) on execution time and memory consumption are summarized in Fig. 4, and from it, we can make several observations.

First, over-emphasizing any one mode of parallelism leads to a stark performance drop, as every parallelization strategy has a dominating cost component. The top row of Fig. 4 shows how increasing any of t, p, or d to their extreme values (e.g., when $d = 32$, letting $t = 1$ and $p = 128$ or $t = 32$ and $p = 4$), execution time is relatively high. For TP, the culprit is the visible communication costs ("TP comm") that increase as t increases while compute utilization drops due to thinning operands in the local matrix multiplications. For PP, the cost of pipeline bubbles—in particular, idle time between forward and backward passes— increases as p increases. For DP, communication costs ("DP comm") increase as d increases.

Second, each parallelism strategy affects memory usage differently, as shown in the bottom row of Fig. 4. While TP cuts both weight and activation memory costs, PP reduces only weights. Due to how interleaved pipeline scheduling is implemented, we need to keep an even larger activation space as we would have to with no PP. DP cannot reduce activation or weight storage.

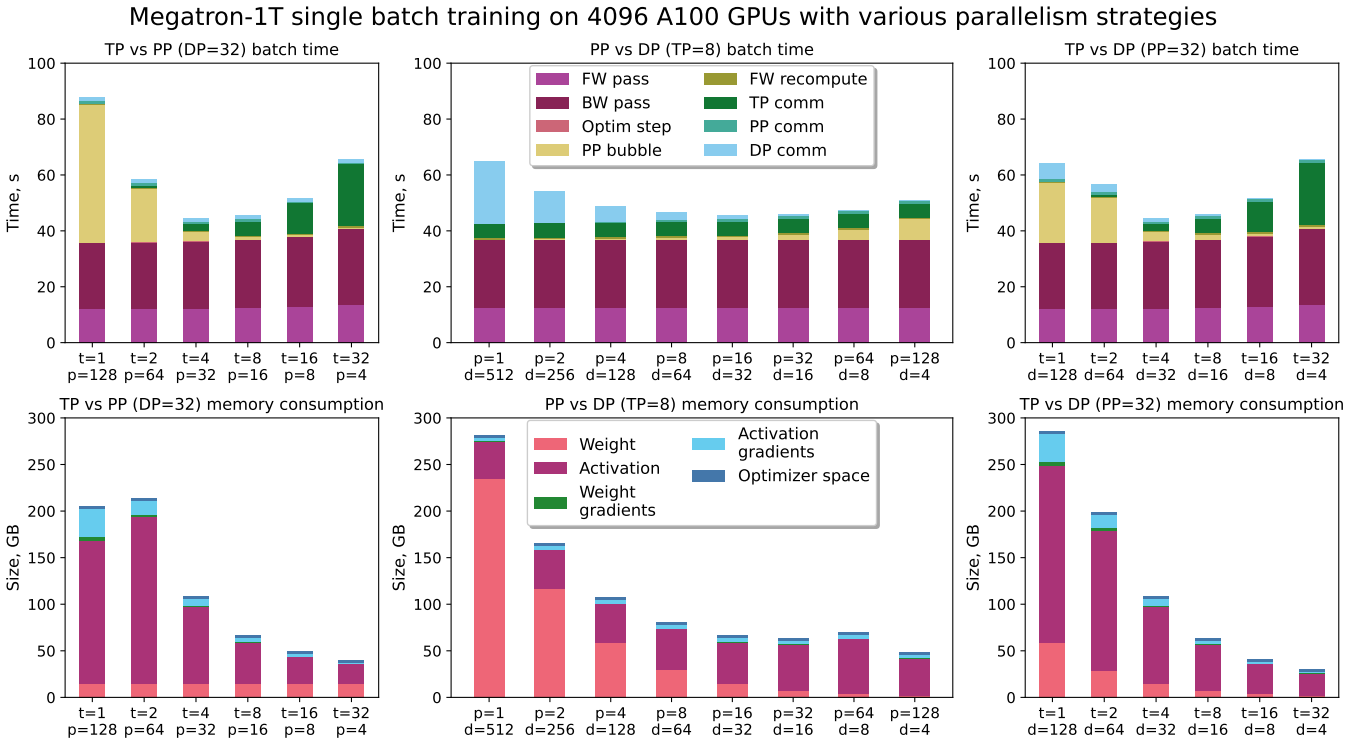

Figure 4: Parallelization strategies analysis, t indicates TP factor, p - PP, and d - DP. We can see that high TP and DP suffer from communication overheads, while high PP suffers from scheduling bubble. On the memory side, TP helps reduce weight and activation pressure, while PP helps only with weight space reduction. High DP helps balance optimizer space by sharding it, achieving the same efficiency as PP and TP.

Third, the dependence of overall performance on (t, p, d) appears to behave like a convex function, yielding an *optimal* parallelization-split in execution time (where valleys occur in execution time) with some minimum constraint on memory usage (i.e., since overall memory usage decreases nearly monotonically as t and p increase).

In short, a good parallelism split reduces visible communication time, improves compute utilization by reducing PP bubbles and increasing local matrix multiply sizes, and implies a minimum total required memory. Calculon illuminates these quantitatively.

4.2 Optimizations Analysis

While an optimal parallelization strategy exists per Section 4.1, the best configuration depends on the structure of the LLM, the available system resources, and what other software implementations are selected. We illustrate these variations in Fig. 5.

First, Fig. 5(a) shows the variation in execution time and memory usage for training a baseline implementation of Megatron-1T [29] assuming a per-GPU HBM memory capacity of 80 GiB. Dashes indicate infeasible configurations due to memory capacity limits. Among the feasible configurations, the minimum execution time is attained for $(t, p) = (8, 32)$ with 79 GiB, just under capacity.

Next, suppose we enable additional software or hardware techniques. For instance, Fig. 5(b) considers the addition of sequence parallelism (Table 1) and its associated performance-enhancing techniques [20]. The optimal configuration shifts slightly to $(t, p) = (16, 64)$ while also lowering the memory requirement to 72 GiB. Enabling all compatible techniques from Table 1, as shown in Fig. 5(c), yields a variety of configurations that could be chosen to minimize either time ($(t, p) = (16, 4)$) or memory capacity (40 GiB at $(t, p) = (8, 32)$), as desired. Additional optimizations, including optimizer sharding and TP and DP communication overlapped with computation, significantly increase the number of possible mappings and move the optimal parallelization point towards higher TP and DP with lower PP. And doubling memory capacity yields Fig. 5(d), shifts such points even more in that direction.

5 OPTIMAL STRATEGY SEARCH

Beyond the examples in Section 4, we observed numerous instances of complex interactions in configuration parameters.

5.1 Optimal Execution

There were many instances where selecting a memory-saving technique frees enough memory to allow a different software technique needing memory to use the "newly available" capacity to decrease running time. Discovering these effects motivates the design of Calculon to allow exploring of the full combinatorial space.

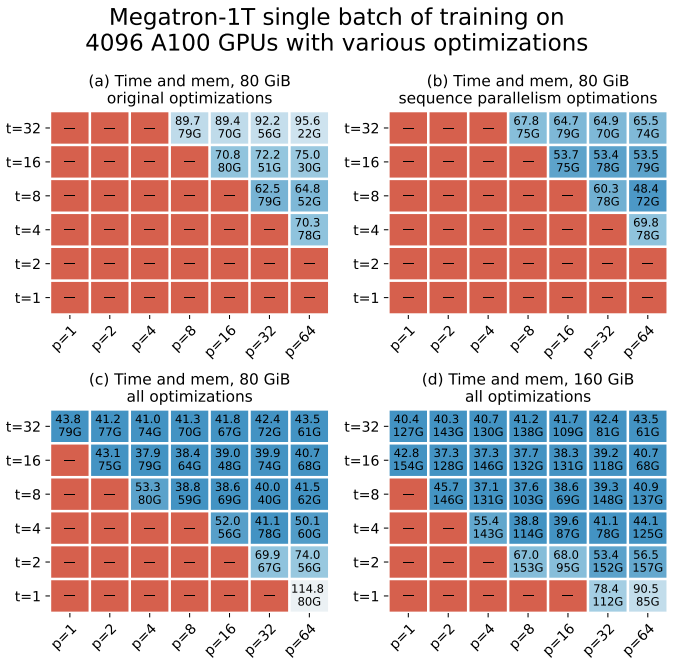

Figure 5: Batch time training with various optimizations and constraints (dashes indicate infeasible configurations due to a lack of memory). The top number shows the best batch time, and the bottom number shows the required memory to run, t and p measure the degree of TP and PP . (a) time for a system with 32 A100 with 80 GiB HBM in a single NVLink domain and original optimizations set [29]; (b) same with partial recompute and sequence parallelism [20]; (c) same with all optimizations from Table 1; (d) same with memory capacity increased to 160 GiB per GPU.

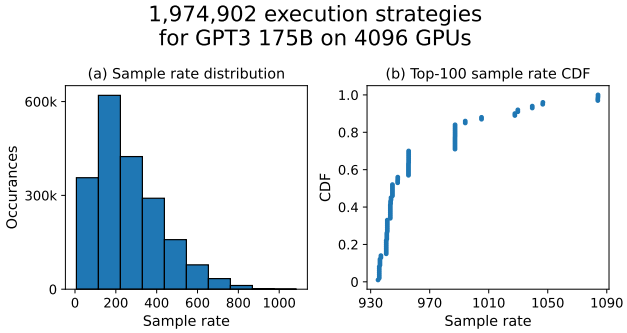

Figure 6: All possible execution strategies for GPT3 175B model training on 4,096 GPUs. (a) Histogram of the sample rate. (b) CDF plot of top-100 configurations.

To get a coarse sense of this space, we considered all execution configurations for training the GPT-3 175B-parameter model on a 4,096-GPU system. There were 10,957,376 calculations possible. Out of all possible configurations, only 1,974,902 were feasible (~18%) as the rest would not have sufficient resources to run. The histogram in Fig. 6(a) depicts the distribution of estimated performance of the feasible runs measured as sample-processing rate, that is, the number of data samples processed per second of LLM training. While the histogram has 10 bins, the 2 rightmost bins that correspond to configurations within 20 % of the best configuration are indistinguishable with the naked eye. We observed only 30 configurations, or less than 0.002% of the full space, achieved performance within 10 % of the best configuration. Figure 6(b) considers just the 100 best performers as an empirical cumulative distribution function. Only about ten attain performance within 5 % of the best. Thus, good configurations may be like needles in a haystack.

Anecdotally, several of these best-configurations did not match our expectations or commonly reported heuristics. For example, some of the best partitioning strategies included setting TP to 4 and PP to 2, moving both under the capacity of an 8-GPU NVLink domain, whereas conventional wisdom assumes TP should be used to saturate NVLink with PP needing only a slower network.

For these reasons we implemented an optimal execution search engine in Calculon. Unlike the procedure described in Section 2, which yields a result based on a single execution configuration, this search engine exhaustively tries all possible execution configurations for a specific system, system size, and LLM and returns the best performer and its statistics. Because Calculon calculates performance so quickly, a standard multi-core desktop computer is able to search the entire configuration space in minutes.

5.2 Optimal System Size

The most important "variable" affecting performance is the structure of the LLM model itself, which can impose many constraints on what partitioning strategies will be most effective, memory capacity requirements, and the ideal system size (number of processors). For example, for best system utilization, PP should split transformer blocks evenly, while TP should split attention heads and the neurons in MLP block evenly, possibly cutting them so the local matrix multiplication size is divisible by a large power of 2, typically 128.

One way this interaction between LLM shape and system parameters manifests is in the phenomenon of "efficiency cliffs," which are sudden drops in performance among system configurations that are close in size. We can observe these cliffs in Fig. 7. It considers three different LLMs: GPT3 175B [3], Turing-NLG 530B [45], and Megatron-1T [29] models. Given a system with a certain number of GPUs (x-axis, considering only multiples of 8 GPUs), we search the full configuration space for the best performer that utilizes exactly that many GPUs and plot its performance as a point (y-axis, higher is better). Each data point is an exhaustive search performed as described in Section 5.1. While the overall trend (envelope) steadily increases with system size, observe that the performance variation in "top performers" also grows dramatically. The Turing-NLG has many LLM size-parameters that are not powers of two, making it particularly tricky to map. This phenomenon occurs for all models and implies that even though one might buy a system with some number of GPUs, mapping a specific model might utilize a smaller

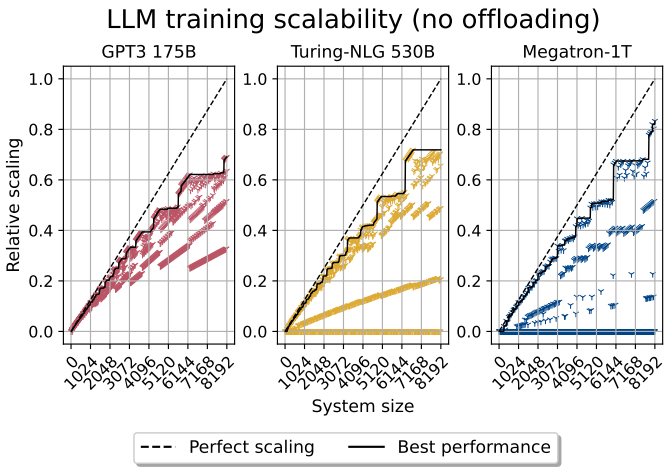

Figure 7: Scaling LLM efficiency training on up to 8,192 GPUs. "Efficiency cliffs"—sudden drops in performance—become worse as the system size increases due to the difficulty of finding a good mapping of the LLM to the system.

number due to these cliffs. Moreover, in two of the LLMs fewer configurations can run at all, as represented by the increasing number of points with zero relative performance.

In short, Calculon can be used to find the optimal execution strategy for a particular system by exhaustively searching the configuration space for every possible system partition size. For the 3 LLMs presented in Fig. 7 this required 467,553,284 calculations. Indeed, the case of a "single-use" system in the context of systems for LLM training is not out of the question, and this finding that arbitrary increases in system size might not deliver the best performance or efficiency, counters some prior claims [56]. Right-sizing the system in light of such phenomena could mean the difference between deciding to use or acquire a relatively smaller system knowing that, due to efficiency cliffs, there might not be a usable configuration without increasing the system scale by a lot.

6 OFFLOADING ANALYSIS

An interesting type of analysis enabled by Calculon is tensor offloading. Recall that some software techniques aim to reduce memory requirements while others trade increased memory requirements for better performance. In hardware, there are at least two options to alleviate capacity issues. One is to directly increase the size of the processor's memory system, typically HBM for top performing processors and accelerators. Unfortunately, scaling HBM capacity is limited by strict and expensive technological constraints. An alternative is to employ an external high-capacity memory, such as Central Processing Unit (CPU) memory or compute express link (CXL)-attached memory. Software strategies can exploit this approach via zero-offload [42] and zero-infinity [38] for tensor offloading to CPU memory. These methods alleviate memory capacity issues but may incur other costs, such as increased computation (e.g., under activation checkpointing) or increased communication costs due partly to limited bandwidth to external memory.

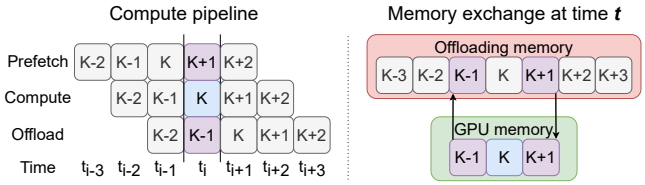

Figure 8: Tensor offloading scheme modeled in Calculon

We can use Calculon to analyze what the minimum bandwidth requirements for the offloaded memory system should be to make the offloading effective. In this analysis, we assume that offloading overlaps only with compute and network operations and not with memory operations to HBM memory. This assumption avoids potential compute throttling that depends on communication with HBM memory. Also, we assume that offloading could be performed with CPU or a direct memory access (DMA)-like engine similar to Tensor Memory Accelerator (TMA) in the NVIDIA H100 [31], and thereby requiring no computational resources on the training accelerator. We analyze the required offloading capacity and bandwidth, assuming we only store a single transformer block currently required for computation in HBM, and allocate space worth of a single block's tensors that are being prefetched and offloaded, as shown in Fig. 8. We fully overlap computation for a current block with the offloading of the results of the previous block computations and prefetching operands for the next one.

The bandwidth required for such seamless tensor offloading is

$$\texttt{Bandwidth}_{\text{offload}} \geq \frac{\texttt{Size}_{\text{tensor}}}{T_{\text{compute}}}. \quad (1)$$

Depending on the size of the offloading tensors, the highest bandwidth may be required when prefetching weights during the forward pass; offloading activations during the forward pass; or prefetching activations, weights, and optimizer during the backward pass.

While weights and optimizer space size depend only on the model parameters, compute time and activation size depend on model parameters and micro-batch size. Micro-batch size heavily depends on the amount of available memory and, as such, on hardware and other software implementation choices, as we want to increase micro-batch size as much as possible as long there is enough memory. The best choice of offloading memory bandwidth and capacity depends on the combination of LLM and system.

In order to determine optimal offloading memory parameters, we set up Calculon with an offloading memory of infinite capacity and infinite bandwidth, and reported utilized capacity and bandwidth. The results for a 1 trillion-parameter LLM are presented in Fig. 9(a) for performance and HBM consumption, and in Fig. 9(b) for offloading memory bandwidth and capacity utilization. Utilized bandwidth almost reaches 600 GB/s, while required memory capacity is as high as 4 TiB. However, due to the greedy nature of the search, Calculon reports required bandwidth for the absolute best performing configuration. As seen in Fig. 6, there might exist other configurations that have close enough performance but demanding fewer resources. With the desire to choose an offloading memory that is realistic and practical, inspecting Fig. 9(a,b) shows that there exist many configurations with good performance where

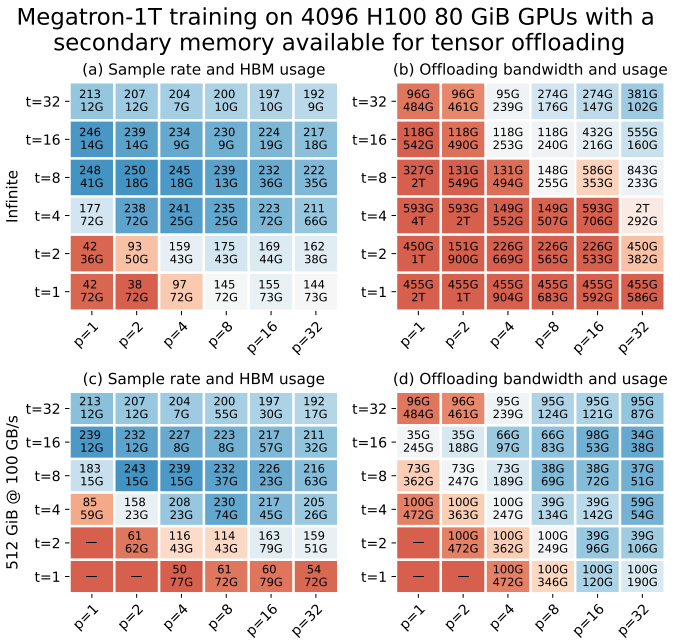

Figure 9: Tensor offloading study. t and p measure the degree of TP and PP. (a) shows sample rate (top number) and HBM consumption (bottom number) with ideal offload memory with infinite size and bandwidth. (b) shows offloading bandwidth (top number) and capacity (bottom number) consumption. (c) and (d) show the same for offloading memory with 512 GiB capacity and 100 GB/s bandwidth. Note that with resource abundance Calculon finds strategies that consume significantly more resources. With reasonable resource constraints, Calculon finds execution strategies with similar performance utilizing much less resources.

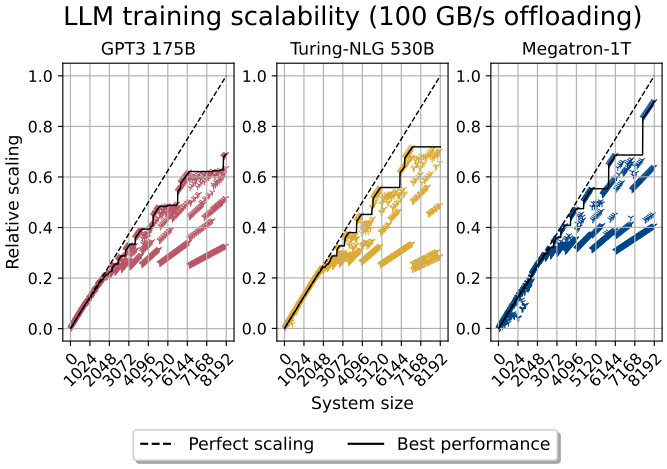

Figure 10: Scaling LLM efficiency training on up to 8,192 GPUs with offloading. Offloading helps keep higher efficiency for LLMs with higher parameter counts.

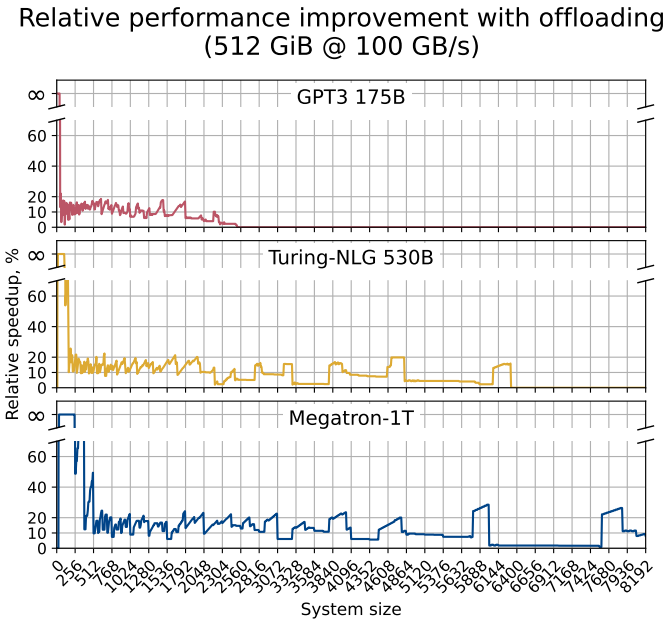

Figure 11: Relative speedup of LLM training on up to 8,192 GPUs due to available offloading. While providing moderate performance improvement for large system sizes, offloading can be instrumental for fine-tuning LLMs on small systems.

the offloading memory requirements could fit in 512 GiB capacity at 100 GB/s.

Fig. 9(c,d) shows the same analysis where we set the offloading memory to 512 GiB capacity at 100 GB/s. Comparing Fig. 9(a) and (c) shows the performance slowdown using offloading memory with restricted configuration is well within 5% for many cases. At the same time, utilized capacity and bandwidth is much lower, as seen comparing Fig. 9(b) and (d). For the best performing split-parallelism configuration with $(t, p) = (8, 2)$ the performance drop is less than 3% while utilizing 56% of bandwidth and 45% capacity of the system with an infinite offload memory.

The existence of an offloading memory significantly affects HBM utilization and the choice of optimizations for the best performing configuration. With offloading memory, the majority of configurations, including the most performant ones, do not utilize more than 20 GB of fast HBM. The abundance of slower memory makes using higher DP possible while making PP less appealing. While both TP and DP communication overlap are available, Calculon suggests configurations with higher DP and a value of TP no more than 16. TP up to 16 can achieve best performance with a single dimensional distribution, as described in [29], since distributing GEMM across more dimensions works better only with larger TP partition sizes [35]. The reason for the relatively higher DP is that it is distributed across processors in the slower network, which requires fewer processors to fully saturate network bandwidth. In the case of NVIDIA GPUs and NVLink, we consider allocating up to 15 % of cores for running NCCL kernels, whereas driving the slower network only requires 2 % of cores. Using 15 % of cores slows

Table 3: Price and performance results for various systems and LLMs under a fixed budget of $125M.

				GPT-3 175B			Turing-NLG 530B			Megatron 1T		
HBM3	DDR5	Price	Max GPUs	GPUs	Perf	Perf/$M	GPUs	Perf	Perf/$M	GPUs	Perf	Perf/$M
20G	0	$22.2k	5616	5472	1117	917	5600	349	280	5120	93	82
40G	0	$25k	5000	4992	1154	924	4984	447	359	4864	219	180
80G	0	$30k	4160	3744	990	881	4080	390	318	4096	224	182
120G	0	$40k	3120	3120	861	690	3120	282	226	3072	172	140
20G	256G	$24.8k	5048	4680	1209	1044	5040	518	415	4896	283	234
40G	256G	$27.5k	4544	4536	1172	939	4320	386	325	4544	261	209
80G	256G	$32.5k	3840	3744	990	813	3840	403	323	3840	228	182
120G	256G	$42.5k	2936	2928	786	632	2880	306	250	2720	160	139
20G	512G	$32.2k	3872	3744	990	820	3864	405	325	3872	230	184
40G	512G	$35k	3568	3568	943	755	3360	355	302	3504	209	170
80G	512G	$40k	3120	3120	837	671	2880	306	266	3072	184	150
120G	512G	$50k	2496	2328	642	552	2496	266	213	2456	151	123
20G	1T	$42.2k	2952	2944	790	635	2880	306	251	2944	177	142
40G	1T	$45k	2776	2672	724	602	2760	293	236	2720	164	134
80G	1T	$50k	2496	2328	642	552	2496	266	213	2456	151	123
120G	1T	$60k	2080	2080	579	464	2016	226	187	2080	130	104

down concurrent GEMM and exposes some amount of the communication time. (If the process of GPU design compartmentalization proceeds further, with blocks similar to TMA being able to move data across the network, this finding is likely to change.)

The improvements to system scalability with 512 GB of offloading memory at 100 GB/s for three LLMs appear in Fig. 10. The impact on smaller models, such as GPT3-175B, is modest. But for larger models, like Megatron-1T, the improvement is much more significant. Moreover, consider models like Turing-NLG 530B, which suffer from efficiency cliffs due to the difficulty of mapping to arbitrary system sizes. These cliffs are mitigiated with offload capacity, thereby reducing the plateaus of suboptimal system sizes. Offload memory capacity provides more options to partition an LLM and may be regarded as a cost-effective method for "future-proofing" larger system acquisitions in support of models that might require more non-power-of-two parallelism configurations.

Calculon estimates typical performance gains for LLM training due to offloading to lie between 10 % and 20 % for Turing-NLG 530B and Megatron-1T LLMs per Fig. 11. Though these values seem modest, the decision to use offloading or not should come after analyzing total cost of ownership (TCO), as even small efficiency gains can accumulate during long system use time.

Where offloading shines is providing better LLM training efficiency at smaller scales. It allows the training of Megatron-1T with high efficiency on less than 256 GPUs, which is not possible without offloading (indicated by "infinite speedup" in the figure). While small systems have a low aggregate performance to train foundational models, LLM fine-tuning, which is believed to be key in successful LLM adaptation, requires several orders of magnitude less compute. However, we still need to have enough aggregate memory capacity to fit LLM. Typical system design would force LLM users to allocate more GPUs, increase model parallelism (e.g., TP and/or PP), and suffer some efficiency loss due to strong-scaling issues. Offloading enables LLM training at smaller GPU counts, permitting higher DP and providing better performance and efficiency.

7 OPTIMAL SYSTEM SEARCH

The key benefit of Calculon is its ability to search across a wide range of software configurations quickly. When determining what hardware to deploy for a data center, designers are often faced with the challenge of choosing an optimal system under a specific budget. In this section, we use Calculon to showcase its ability to quickly choose from a selection of systems in order to optimize performance per price.

For this study, we analyze the training of three different LLMs: GPT3 175B [3], Turing-NLG 530B [45], and Megatron-1T [29] models on different systems. We base our system design around NVIDIA's H100 [31] GPU interconnected in clusters of 8 with NVLink and between clusters with NDR InfiniBand. We use theoretical system components and pricing as follows. An H100 without any HBM3 memory is $20k, which includes all the required infrastructure. HBM3 memory costs $2,250, $5,000, $10,000, and $20,000 for 20 GiB, 40 GiB, 80 GiB, and 120 GiB, respectively. All HBM versions run at 3TB/s. To add a secondary DDR5 memory to an H100 costs $2.5k, $10k, and $20k for 256 GiB, 512 GiB, and 1 TiB, respectively. All secondary memory options run at 100 GB/s per direction.

For a price-aware performance analysis we cap the maximum system cost to $125M. In this case, due to cost difference for each design, the number of H100s deployed is different based on the total system cost constraint. We present all system options across the full permutation of HBM3 and DDR5 options resulting in 16 system designs. Table 3 shows these options. The "Price" column shows the price of each H100 with its options and the "Max GPUs" column shows the maximum GPUs that can be afforded with $125M.

For each system and LLM analyzed, we sweep across the system size space exhaustively finding the absolute best execution strategy. We report the used number of GPUs in the "GPUs" column for

each LLM, as well as the performance (i.e. sample rate) and the performance divided by system cost. Neither the least nor most expensive system design competes very well. The system design with the highest performance (highlighted) is the top performer for all three LLMs. With only 256 GiB of DDR5 memory it was able to offload enough data to keep the active memory usage under 20 GiB in HBM3. This cost saving trade-off allows it to have the second highest system size and maintain a high compute efficiency.

8 CONCLUSION

In conclusion, the value of Calculon lies in its ability to analyze a large codesign space of hardware and software configurations, thereby making it possible to discover new and sometimes surprising configurations that might outperform the best-known state-of-the-art.

To help illustrate this point, we summarize some of the best strategies discovered by Calculon in Table 4, with the resulting performance improvements over the state-of-the-art Fig. 12. There is a 30 % performance improvement compared to previous State-of-the-Art from optimization strategies alone, and potential 30 % more performance per cost improvement from a better system design choice based on our cost model.

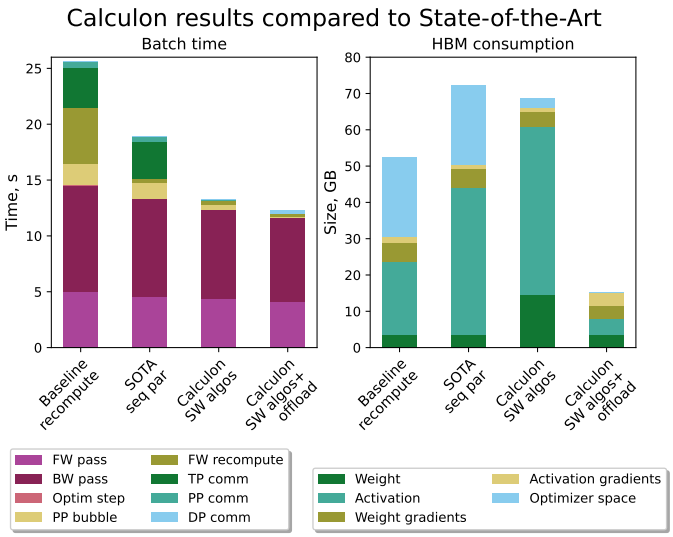

Figure 12: Performance comparison of optimal execution strategies found by Calculon with available State-of-the-Art optimized Megatron-1T implementations.

The specific combination of optimizations discovered by Calculon, to the best of our knowledge, is not implemented in any framework. The combination of activation fusion and optimizer sharding, significantly reduces memory requirements. But then, counter to conventional wisdom, Calculon decided to exploit this reduction by reducing PP and increasing DP (which benefits from more memory). Normally, doing so would incur great communication cost. But by combining a large microbatch size with high PP interleaving and DP communication overlap, all the cost could be hidden behind increased per-microbatch compute time.

Table 4: Comparison of parallelization strategies.

	(t, p, d)	m	PP int	added optimizations	MFU
recompute	(8, 64, 8)	1	2	Full recompute p2p RS+AG	36.67%
seq par	(8, 64, 8)	1	2	Attn recompute RS+AG RS redo for SP	49.61%
Calculon SW optim	(8, 16, 32)	2	8	TP and DP overlap optimizer sharding fused activation –RS redo for SP	70.96%
Calculon SW optim + offload	(8, 1, 512)	6	1	offload memory weight + activation + optimizer offload	76.71%

A second counterintuitive finding is that a high HBM capacity is not necessary for efficient LLM training. In fact, we would prefer to have a large amount of slower and cheaper memory and rearrange execution to better fit the application's implicit data reuse patterns.

Overall, our findings support a hypothesis that a complete search over the joint space of hardware and software configurations can identify optimal configurations that may be hard or impossible to discover using manual heuristics. Indeed, any heuristic-driven process risks biasing exploration of the search space toward certain known types of configurations, thereby missing others. We hope the quick analysis facilitated by Calculon makes broad explorations of unconventional designs more feasible.

(*Note:* Per Section 1, the full model description has been omitted due to space constraints, but all details are available at the open-source Calculon repository.)

REFERENCES

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 46, 15 pages.

[2] Léon Bottou. 1991. Stochastic Gradient Learning in Neural Networks. In *Proceedings of Neuro-Nîmes 91*. EC2, Nimes, France. http://leon.bottou.org/papers/bottou-91c

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain,

William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. https://doi.org/10.48550/ARXIV.1604.06174

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]

[7] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1223–1231.

[8] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 431–445. https://doi.org/10.1145/3437801.3441593

[9] Google, LLC. 2022. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla

[10] Andreas Griewank and Andrea Walther. 2000. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Trans. Math. Softw.* 26, 1 (mar 2000), 19–45. https://doi.org/10.1145/347837.347846

[11] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. arXiv:2203.15556 [cs.CL]

[12] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. https://doi.org/10.1145/3373376.3378530

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. Curran Associates Inc., Red Hook, NY, USA.

[14] Mikhail Isaev, Nic McDonald, Jeffrey Young, and Richard Vuduc. 2022. ParaGraph: An application-simulator interface and toolkit for hardware-software co-design. In *51th International Conference on Parallel Processing* (Bordeaux, France) *(ICPP 2022)*. Association for Computing Machinery, New York, NY, USA, Article 61, 10 pages. https://doi.org/10.1145/3545008.3545069

[15] Arpan Jain, Ammar Ahmad Awan, Asmaa M. Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G. Anthony, Hari Subramoni, Dhableswar K. Panda, Raghu Machiraju, and Anil Parwani. 2020. GEMS: GPU-enabled memory-Aware Model-Parallelism system for Distributed DNN Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 45, 15 pages.

[16] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020*. 497–511.

[17] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (jun 2020), 67–78. https://doi.org/10.1145/3360307

[18] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[20] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. https://doi.org/10.48550/ARXIV.2205.05198

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[22] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake A. Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, Yuanzhong Xu, and Zongwei Zhou. 2019. Scale MLPerf-0.6 models on Google TPU-v3 Pods. *CoRR* abs/1909.09756 (2019). arXiv:1909.09756 http://arxiv.org/abs/1909.09756

[23] Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. 2017. Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 183–192. https://doi.org/10.1109/HiPC.2017.00030

[24] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. {GS}hard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations*. https://openreview.net/forum?id=qrwe7XHTmYb

[25] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 553–564. https://doi.org/10.1109/HPCA.2017.29

[26] Nenad Markuš. 2018. Fusing batch normalization and convolution in runtime. https://nenadmarkus.com/p/fusing-batchnorm-and-conv/

[27] Nic McDonald, Adriana Flores, Al Davis, Mikhail Isaev, John Kim, and Doug Gibson. 2018. SuperSim: Extensible Flit-Level Simulation of Large-Scale Interconnection Networks. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 87–98. https://doi.org/10.1109/ISPASS.2018.00017

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[29] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[30] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core

[31] NVIDIA. 2022. NVIDIA H100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[32] NVIDIA. 2022. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/

[33] NVIDIA. 2023. NVIDIA Deep Learning Performance. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

[34] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon Emissions

and Large Neural Network Training. https://doi.org/10.48550/ARXIV.2104.10350

[35] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. arXiv:2211.05102 [cs.LG]

[36] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[38] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. https://doi.org/10.1145/3458817.3476205

[39] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical Text-Conditional Image Generation with CLIP Latents. https://doi.org/10.48550/ARXIV.2204.06125

[40] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 8821–8831. https://proceedings.mlr.press/v139/ramesh21a.html

[41] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[43] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. arXiv:1811.02084 [cs.LG]

[44] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. https://doi.org/10.48550/ARXIV.1909.08053

[45] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. https://doi.org/10.48550/ARXIV.2201.11990

[46] Xiaoyang Sun, Wei Wang, Shenghao Qiu, Renyu Yang, Songfang Huang, Jie Xu, and Zheng Wang. 2022. StrongHold: Fast and Affordable Billion-Scale Deep Learning Model Training. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 71, 17 pages.

[47] Jakub Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient Algorithms for Device Placement of DNN Graph Operators. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1296, 13 pages.

[48] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[50] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 868–880. https://doi.org/10.1109/HPCA51647.2021.00077

[51] Guanhua Wang, Kehan Wang, Kenan Jiang, XIANGJUN LI, and Ion Stoica. 2021. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 696–710. https://proceedings.mlsys.org/paper/2021/file/c81e728d9d4c2f636f067f89cc14862c-Paper.pdf

[52] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. 2022. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 93–106. https://doi.org/10.1145/3567955.3567959

[53] Jeremiah J. Wilke, Joseph P. Kenny, Samuel Knight, and Sebastien Rumley. 2018. Compiler-Assisted Source-to-Source Skeletonization of Application Models for System Simulation. In *High Performance Computing*, Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis (Eds.). Springer International Publishing, Cham, 123–143.

[54] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. https://doi.org/10.48550/ARXIV.2105.04663

[55] Xiru Zhang, Michael McKenna, Jill Mesirov, and David Waltz. 1989. An Efficient Implementation of the Back-propagation Algorithm on the Connection Machine CM-2. In *Advances in Neural Information Processing Systems*, D. Touretzky (Ed.), Vol. 2. Morgan-Kaufmann. https://proceedings.neurips.cc/paper_files/paper/1989/file/e3796ae838835da0b6f6ea37bcf8bcb7-Paper.pdf

[56] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[57] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. 2022. Towards the Co-design of Neural Networks and Accelerators. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 141–152. https://proceedings.mlsys.org/paper/2022/file/31fefc0e570cb3860f2a6d4b38c6490d-Paper.pdf

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.8223205
https://doi.org/10.5281/zenodo.8267785
https://doi.org/10.6084/m9.figshare.23996292.v1

ARTIFACT IDENTIFICATION

The paper presents Calculon, a parameterized analytical performance model of transformer-based Large Language Models (LLMs) training implemented in Python. Calculon itself is the main contribution of the paper. Calculon generated all of the studies and their results presented in the paper. We made Calculon open-sourced and available at https://github.com/paragraph-sim/calculon. We are releasing the scripts and data needed for the result reproducibility as part of a special sc23 reproducibility repository. https://github.com/calculon-ai/sc23 The model derives from the extensive survey of performance optimizations proposed for LLMs training. The core analytical model performs a single calculation of time and resource usage for a single application, system, execution strategy set. This model is given three specifications: the LLM, the system the LLM is running on, and the execution strategy describing how the LLM is run on the system, including all the optimizations applied to execution. On top of this model, we implemented a complete space search algorithm that concurrently iterates on millions of specification combinations and reports the ones that perform the best. Each calculation takes about 1 ms of a CPU thread time, allowing it to perform extensive studies over billions of configurations utilizing several thousand CPU core hours. The tool can facilitate quick exploration of future LLM training systems and a better understanding of the performance trade-offs that exist.

REPRODUCIBILITY OF EXPERIMENTS

This document describes the structure of the project and the necessary steps required to reproduce experiments from the paper. All the plots and tables in the paper are generated using Calculon and Python scripts that we share as part of our github repo. To reproduce results in the paper, simply cloning the code from github and running reproducibility scripts locally is enough. Calculon depends only on standard Python libraries and requires Python 3.9 or newer. To regenerate data for larger experiments, only minimal changes to the script are required to run it on slurm or using any similar task manager. Details on how to run Calculon manually are provided in the README.md file in the repository. All the necessary data and scripts are located in the calculon-sc23 repository. The folder is organized with subfolders for every chapter of the paper that contains the results of the experiments. Each chapter folder has further subfolders for every plot and table. These subfolders contain all the data generation scripts, data itself if it cannot be generated on a laptop promptly, and plotting scripts. The result of running the reproducibility scripts should be identical to the results presented in the corresponding chapter of the paper in the corresponding plot or table. There is also a run_all script that produces results for all of the experiments. We will also provide a colab notebook that can regenerate all the plots for the paper. While a single Calculon calculation takes about 1 millisecond, most of the experiments utilize complete search space to find the best configuration with respect to the study search space. Some of the studies involve running billions of calculations. That results in O(1000) CPU core hours. Figures 2, 3, 4, 7, and 10, and Table 2, can be reproduced on a single CPU system in less than an hour by simply running the scripts provided in the corresponding subfolder. Figures 5, 6, 8, 9, and Table 3 take longer to generate data and require a distributed multi-CPU system to collect the data in a reasonable time. As such, we provide the data generated by Calculon in the form of json files located in the corresponding subfolder. Instead of a single reproducibility script, we provide a script that generates the data and the script that produces a plot from it.

ARTIFACT DEPENDENCIES REQUIREMENTS

Calculon is developed in Python 3 and requires Python 3.9 or later and only depends on pypi available packages. As such, it is able to run in any environment that runs Python 3.9+. We successfully used CentOS, Ubuntu, MacOS, and Google colab during our experiments. No pre-existing datasets are needed to run Calculon.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

General instructions on how to install and run Calculon can found here: https://github.com/calculon-ai/calculon

For the SC ‘23 paper, a script is provided at https://github.com/calculon-ai/calculon-sc23 that will run Calculon and plot all results. It runs either locally on a single machine or on a distributed cluster that has a global network filesystem. The script performs all Calculon calculations by issuing single machine jobs (no MPI, SHMEM, etc. needed). In total O(10,000) jobs will be launched, many of which are only a few minutes long. The script provides a mechanism for users to customize the execution of jobs on their own cluster’s job scheduler. A Chameleon Cloud interface is provided.