Leftovers for LLaMA

Ravi Kumar Singh
TCS Research
ravik.singh2@tcs.com

Likhith Bandamudi
TCS Research
likhith.bandamudi1@tcs.com

Shruti Kunde
TCS Research
shruti.kunde@tcs.com

Mayank Mishra
TCS Research
mishra.m@tcs.com

Rekha Singhal
TCS Research
rekha.singhal@tcs.com

ABSTRACT

In recent years, large language models (LLMs) have become pervasive in our day-to-day lives, with enterprises utilizing their services for a wide range of NLP-based applications. The exponential growth in the size of LLMs poses a significant challenge for efficiently utilizing these models for inference tasks, which require a substantial amount of memory and compute. Enterprises often possess multiple resources (workers, nodes, servers) with unused (leftover) capacity, providing an opportunity to address this challenge by distributing large models across these resources.

Recent work such as Petals, provides a platform for distributing LLM models in a cluster of resources. Petals require that users use their discretion to distribute blocks on a given cluster, consequently leading to a non-optimal placement of blocks. In this paper, we propose LLaMPS - a large language model placement system that aims to optimize the placement of transformer blocks on the available enterprise resources, by utilizing the leftover capacity of the worker nodes. Our approach considers leftover memory capacity along with available CPU cores, when distributing transformer blocks optimally across worker nodes. Furthermore, we enhance the scalability of the system by maximizing the number of clients that can be served concurrently. We validate the efficacy of our approach by conducting extensive experiments using open-source large language models - BLOOM (1b, 3b, and 7b parameters), Falcon, and LLaMA. Our experiments demonstrate that LLaMPS facilitates optimal placement of transformer blocks by utilizing leftover resources, thus enabling enterprise-level deployment of large language models.

CCS CONCEPTS

• **Computing methodologies** → **Distributed artificial intelligence**.

KEYWORDS

LLMs, Leftover capacity, Distributed inference, Optimal block placement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0444-4/24/05…$15.00
https://doi.org/10.1145/3629526.3645045

ACM Reference Format:
Ravi Kumar Singh, Likhith Bandamudi, Shruti Kunde, Mayank Mishra, and Rekha Singhal. 2024. Leftovers for LLaMA. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3629526.3645045

1 INTRODUCTION

Large Language Models (LLMs) have become pervasive, finding widespread applications in businesses, such as natural language processing [11] and recommender systems [9] [15] for inference tasks. LLMs [14] [10] have been instrumental in enabling decision-making and facilitating various aspects of day-to-day business operations in enterprises. With the continuous evolution of LLMs, a notable challenge that has emerged is their increasing size. As LLMs grow in scale [5], they require a substantial amount of memory and compute resources for effective deployment. These requirements may pose a constraint in enterprises having limited infrastructure, impeding their ability to fully leverage the potential of LLMs for inference tasks.

Many businesses deploy LLM-powered chatbots [16] [4] for customer support. These chatbots require substantial memory and computational resources for handling natural language queries efficiently. Recommender systems [13] powered by LLMs, like those used by streaming services, need to process vast amounts of user data and perform complex language-based recommendations, demanding considerable computational resources. Besides LLMs have been steadily increasing in size over time to improve their performance, and this growth has led to greater resource requirements. Larger models generally require more memory and computational power for efficient inference. As businesses grow and their workloads increase, they may find it challenging to scale their inferencing infrastructure to meet the demand. This can lead to performance bottlenecks and delays.

To address these challenges enterprises do have multiple options such as leveraging cloud-based LLM services to help mitigate some of the infrastructure and resource constraints. Cloud providers offer pre-configured LLM models and scalable infrastructure, which may not necessarily be a cost-effective solution.

Enterprises however have a latent opportunity comprising of multiple worker (server) nodes having leftover capacity in terms of memory and processing cores. We believe this capacity can be effectively utilized to deploy LLMs in a distributed manner. There are some efforts in the literature such as Deepspeed [18], Petals [6] etc that enable the distribution of transformer blocks across multiple workers. DeepSpeed enables distributed fine-tuning and inference

of transformer blocks by on the underlying GPU cluster, by utilizing maximum possible resources. Petals [5] is one such effort that enables the distribution of the transformer blocks across multiple workers for fine-tuning or inference. Petals utilize leftover capacity from the workers in a cluster to seamlessly enable the distribution of transformer blocks across worker nodes. However, Petals requires that users use their discretion to distribute blocks on a given cluster, consequently leading to a non-optimal placement of blocks. This may also lead to sub-optimal system performance in terms of inference latency.

In this paper, we address the above-mentioned challenges by proposing a system that enables optimal placement of transformer blocks in a given cluster. Our contributions are as follows:

- We propose an Optimal block Placement Algorithm (OPA), used in LLaMPS s.t., multiple models may be served on a single server, while optimally utilizing the leftover capacity of workers in the cluster.
- We evaluate the efficacy of LLaMPS on an enterprise CPU-cluster on multiple variants of open source transformer-based large language models, namely BLOOM [23], LLaMA [21] and Falcon [20].

The rest of the paper is structured as follows. Section 2 provides a background for our proposed work. We present the LLaMPS system in Section 3. Our experiment setup is discussed in Section 4. We present results, ablation studies, and related work in Sections 5, 6 and 7. We conclude with directions for future work in Section 8.

2 BACKGROUND

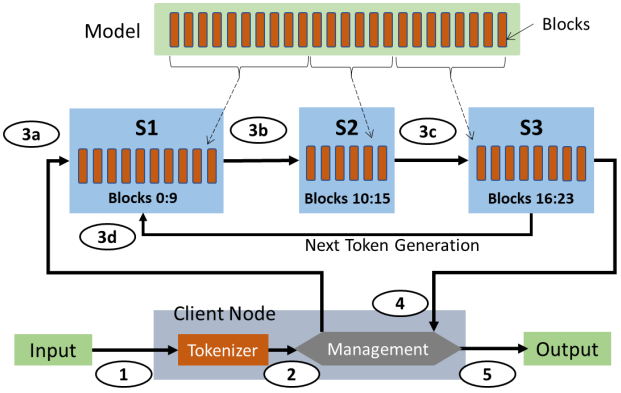

Figure 1: Inference

The resource constraints imposed by the massive scale of transformer models render them unsuitable for deployment on a single server. Consequently, this has been an active area of research for exploring solutions to harness the potential of these models in a distributed manner. One approach to address the resource challenge is to leverage distributed frameworks that enable the allocation of transformer blocks across multiple servers. These frameworks aim to maximize the computational resources available for distribution, thus achieving low-latency inference. However, this strategy, while effective in reducing response times, can lead to sub-optimal utilization of resources. An alternative school of thought advocates for making efficient use of the leftover capacity of servers, ensuring that no computational power goes to waste. One such framework is Petals, which capitalizes on the untapped resources of servers.

As shown in Figure 1 a typical enterprise may have servers s1, s2, and s3, each with different degrees of leftover capacity. Transformer model blocks are distributed across these servers. Server 1 hosts Blocks 0-9, Server 2 handles Blocks 10-15, and Server 3 manages Blocks 16-23. During an inference cycle, the input is initially tokenized at the client node (Steps 1 and 2), and the tokenized input is then relayed to server 1. The input traverses the allocated transformer blocks on server 1, and the intermediate output is sequentially passed to server 2, and so on, until it has traversed all the blocks of the transformer (Steps 3a, 3b, 3c). Finally, the output from the last server in the sequence is transmitted back to the client, where the required output is generated.

Petals, although making effective use of available block capacities, do not inherently guarantee an optimal distribution of these blocks across servers. In response to this limitation, we introduce an Optimal Placement Algorithm (OPA) within the LLaMPS framework, offering an enhanced method for strategically allocating blocks across servers. OPA's primary objective is to accommodate multiple models efficiently on a single server while exploiting any residual server capacity. The LLaMPS system can be seamlessly integrated with various open-source distributed frameworks, providing flexibility and compatibility. In this paper, we assess the performance of the Optimal Placement Algorithm within the LLaMPS framework when applied to the open-source Petals framework. To conduct our evaluation, we established a controlled lab environment within an enterprise setting, featuring a cluster composed of four CPU servers with heterogeneous capacities. Since Petals supports three different Large Language Models (LLMs), namely LLaMa, BLOOM, and Falcon; our analysis encompasses all three LLMs, offering a comprehensive assessment of the algorithm's efficacy across a range of scenarios.

3 LLAMPS

In this section, we present our system LLaMPS. As shown in Figure 2, the LLaMPS system takes the size of the transformer model as input. This input is crucial for accurately calculating the total memory requirement for accommodating the model on the available resources. LLaMPS leverages the residual resource capacity available within an enterprise cluster pool. We quantify this *leftover* capacity in terms of memory and number of cores (i.e., compute capacity). LLaMPS then runs the Optimal Placement Algorithm (OPA) to determine the optimal distribution of transformer blocks across nodes in the enterprise resource cluster. The optimal placement algorithm is designed to maximize the overall leftover capacity within the resource cluster to enable multiple clients to be served concurrently. Once the plan is ready, the transformer model is automatically chunked into blocks and the blocks are distributed to the identified servers. The application is then ready to start the process of inference. In contrast, Petals [5] distributed framework requires the user to decide the exact distribution of blocks across the available servers. This distribution may lead to a sub-optimal utilization of the enterprise resources. The Optimal Placement Algorithm of LLaMPS ensures that the leftover capacity of the enterprise

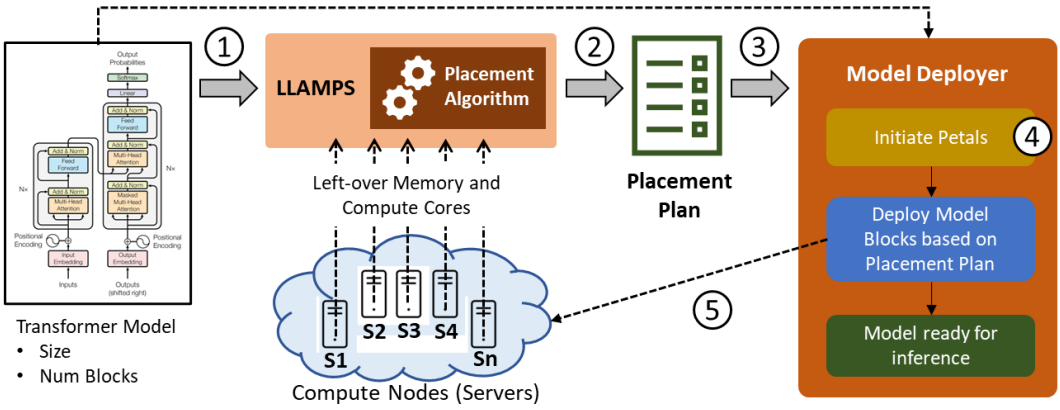

Figure 2: Architecture of LLaMPS

cluster is optimally utilized, subsequently serving multiple clients and enabling multiple models to be supported on individual worker resources in the enterprise cluster.

The Optimal Placement Algorithm is outlined in algorithm 1. As depicted in the Optimal Placement Algorithm 1, LLaMPS determines the available memory and cores on each worker in the enterprise cluster. Next, OPA calculates the leftover memory capacity of each worker (Step 2). Since we use the open-source Petals framework for the distribution of the transformer block, OPA needs to deduct the memory overhead imposed by Petals on the worker node. OPA is a generic approach that enables optimal distribution of transformer blocks on an enterprise cluster, hence the underlying distributed framework can be replaced by any other framework. It is expected that the memory overhead imposed by the framework will be deducted from the available memory at a worker node. Once the available memory and cores at each worker are determined, OPA then follows the Analytical Hierarchy Process (AHP) technique [12] - a popular approach for systematically determining weights based on the preferences of decision-makers. AHP helps in structuring the decision-making process and deriving relative weights through pairwise comparisons. Weights are based on preference scores. Since we compare OPA with GMA which is a memory-based approach, we have assigned higher preference to memory. In LLaMPS we have two objectives - memory and cores. Decision-makers assign a preference score to indicate how much one objective (memory) is more important than another (cores) (Step 6). After collecting preference scores, we normalize them to form a consistent comparison matrix (Step 7). Then, the weighted average of each objective is calculated based on the normalized comparison matrix (Step 8,9). The final weighted scores for each worker node are determined as shown in Step 10. Once the scores for all worker nodes are determined, the list of worker nodes is sorted in descending order based on the scores (Step 12). OPA then determines the number of blocks to be loaded on each server, by dividing the leftover memory at each worker by the size of each transformer block (Step 13). The blocks are then loaded on each server and the Petals framework is leveraged to perform downstream tasks.

Algorithm 1 OPA - Optimal Placement Algorithm

Require:
- TB_s - Size of transformer model block
- P_o - Petals Framework Overhead
- S - Cluster of enterprise workers
- AM_S - Available memory for each worker
- AC_S - Available cores for each worker
- $Nb(S)$ - Number of blocks on a worker

Ensure: Optimal block placement on servers

1: **for** each worker in the cluster $S_i \subseteq S$ **do**
2: $LM(S_i) \leftarrow (AM_{S_i} - P_o)$ ▷ Left over memory capacity
3: $AC(S_i) \leftarrow$ Available cores of worker
4: **end for**
5: **for** each worker in the cluster $S_i \subseteq S$ **do**
6: Assign preference scores for memory and cores
7: $M[Scores_i] \leftarrow$ Normalized matrix of scores
8: $W_{m_i} \leftarrow$ Weighted memory score using $M[Scores_i]$
9: $W_{c_i} \leftarrow$ Weighted core score using $M[Scores_i]$
10: $[W_{Score_i}] \leftarrow (W_m \times LM(S_i)) + (W_{c_i} \times AC(S_i))$
11: **end for**
12: $S[W_{Score_i}]_{sorted} \leftarrow$ List of sorted servers based on scores
13: Assign blocks to each server from $S[W_{Score_i}]_{sorted}$ list s.t. $Nb(S_i) \leftarrow LM(S_i)/TB_s$
14: Distribute blocks on each server for inference

Algorithm 2 GMA - Greedy Memory Algorithm

Require:
- TB_s - Size of transformer model block
- P_o - Petals Framework Overhead
- S - Cluster of enterprise workers
- AM_S - Available memory for each worker
- $Nb(S)$ - Number of blocks on a worker

Ensure: Optimal block placement on servers

1: **for** each worker in the cluster $S_i \subseteq S$ **do**
2: $LM(S_i) \leftarrow (AM_{S_i} - P_o)$ ▷ Left over memory capacity
3: **end for**
4: $[LM(S_i)]_{sorted} \leftarrow$ List of sorted servers by leftover memory
5: Assign blocks to each server from $[LM(S_i)]_{sorted}$ list s.t. $Nb(S_i) \leftarrow LM(S_i)/TB_s$
6: Distribute blocks on each server for inference

Table 1: Regression Model Comparison: Predicting Petals' Memory Overhead

Model	MSE	R-Squared	Petals' Memory Overhead Equation
Linear Regression	263.7	0.773	$1.88 \cdot$ Number of Blocks $+ 0.60 \cdot$ Model parameters $- 15.05$
LGBM Regression	722.25	0.394	$73443.43 \cdot$ Number of Blocks $+ 148798.14 \cdot$ Model parameters
Polynomial Regression	6.74	0.994	$0.03 \cdot$ Number of Blocks $\cdot$ Model parameters $+ 0.65 \cdot$ (Model parameters)$^2 + 0.14$
Decision Tree Regression	19.75	0.998	Decision Tree

Algorithm 2 depicts GMA - Greedy Memory Algorithm. The GMA determines the leftover memory capacity (Step 2). A user is more likely to use a greedy memory approach to select resources for block placement. Hence we have used GMA as the baseline algorithm in our experiments. However, GMA does not guarantee optimal block placement while OPA does. Then it sorts the list of servers in the descending order of memory capacity (Step 5). The blocks are then assigned to the servers in the sorted order of the list based on the leftover memory capacity of each worker (Step 5). GMA does not take into consideration the cores when assigning blocks which may lead to a sub-optimal placement of blocks on the servers, subsequently affecting system performance adversely. We present detailed experiments and ablation studies to compare the OPA algorithm of LLaMPS with GMA.

4 EXPERIMENT SETUP

We designed multiple experiments to validate the efficacy of the optimal placement algorithm of LLaMPS. We aimed to place transformer blocks in an enterprise cluster while optimally utilizing the leftover capacity of the workers in the cluster. We believe that the optimal placement of blocks enables serving a maximum number of possible clients simultaneously within an enterprise. We now present some details of our experiment setup. Our experiment setup comprises multiple large language models, and worker nodes having heterogeneous configurations within an enterprise cluster.

4.1 Large Language Models

Our experiments are conducted on 3 different open-source models, namely BLOOM [23], LLaMA [21] and Falcon [20]. BLOOM is a multi-lingual large language model that has the ability to generate text in 46 natural languages and 13 programming languages. The BLOOM model comes in multiple variants. In this paper, we have conducted our experiments on 3 variants - namely 560 million, 3 billion, and 7 billion parameter versions of BLOOM. The multiple variants enable us to deeply evaluate the efficacy of our proposed system. Another model we have used to evaluate LLaMPS is LLaMA (Large Language Model Meta AI), which is a family of LLMs released by Meta AI [8]. LLaMA has multiple variants and we conduct our experiments on the 70 billion version of the LLaMA model. Additionally, we also conduct experiments on Falcon [20] which is a generative large language model. Falcon comes in multiple variants and we evaluate LLaMPS on the 40 billion version of Falcon. Through our experiments on multiple variants of multiple open-source models in the literature, we reinforce the efficacy of our proposed system LLaMPS.

4.2 Heterogeneous Enterprise Cluster

We create a set-up in our enterprise lab, utilizing leftover capacities of servers utilized by members of the labs. We pick 5 CPU-based servers and the configuration of these servers is depicted in Table 2. All servers have heterogeneous configurations in terms of operation system, OS version, cores, and memory, which is a typical real-world setup in any enterprise. We leverage this leftover capacity to optimally distribute blocks of large language models such that the number of users can be maximized. Table 2 provides details of the different servers and also distinguishes between the "Client" and four different servers ("Server 1," "Server 2,","Server 3" and "Server 4") and their respective configurations.

4.3 Distributed Framework Overhead

The LLaMPS system functions on the Petals distributed framework, utilizing its services for various tasks, including the creation and initialization of the Distributed Hash Table (DHT) and the loading of transformer blocks. The overhead and impact of these factors on OPA's performance within LLaMPS led us to conduct a series of experiments for a systematic analysis. In this investigation, Petals serves as the open-source foundation for LLaMPS, supporting block distribution and other DHT-related operations. It is important to note that although LLaMPS currently leverages Petals, it is not limited to using this particular framework and can seamlessly integrate with any alternative distributed framework for execution.

To predict the memory overhead linked to Petals (**Petals' Memory Overhead**), we formulated an equation with two variables: the *number of blocks* and *number of model parameters*. This predictive model proves to be valuable in the Block Placement algorithm.

Table 1 displays the outcomes of four regression models—Linear Regression, LGBM Regression, Polynomial Regression, and Decision Tree Regression—applied to 150 data points. Our experiments reveal that the Decision Tree Regression model outperforms the other three, demonstrating an R-squared value of 0.998 and an MSE of 19.75. Consequently, we employed the Decision Tree Regression model for predicting Petals' Memory Overhead. Choosing simpler machine learning models other than DDN and LLMs is a more cost-effective and resource-efficient approach.

Three factors within the distributed framework contribute to the overall overhead: the overhead determined by the Decision Tree Regressor model, and the Attention cache's size. The size of the attention cache is calculated as twice the model's hidden size multiplied by 4096 times the tensor size (4 bytes for CPU). Together, these factors collectively constitute the **Petals Framework Overhead**, quantified as:

$$\text{Petals Memory Overhead} + \text{Attention Cache size} \quad (1)$$

Table 2: Server Configuration Information

Server	OS Version	Kernel	Total Memory(GB)	Cores
Client	Ubuntu 18.04.6	5.4.0	62.8	48
Server 1	Ubuntu 18.04.6	5.4.0	252	56
Server 2	Ubuntu 22.04.3	5.19.0	992	88
Server 3	CentOS 7	3.10.0	504	56
Server 4	CentOS 7.8	3.10.0	256	56

LLaMPS system uses this formula to calculate Petals Framework Overhead for block placement.

5 EXPERIMENTS

We performed various experiments on GMA and OPA by varying the model parameters, servers, clients, cores, batch size, and token length.

Given below are the details of various parameters used in experiments.

- **Model)**: The experiments were conducted using the "bloom-560m," "bloom-3b," "bloom-7b1," "falcon-40b," and "llama-70b" models.
- **Block Distribution**: The "Block_distribution" varied between "[24]" for "bloom-560m" and "[30]" for "bloom-3b" and "bloom-7b1." In the "falcon-40b" and "llama-70b" models, the block distribution was specified as "(32,28)" and "(35,25,20)" respectively.
- **Memory**: The "Memory" column indicates the memory configurations used for each experiment. For example, "[12, 14, 14, 14]" in "bloom-560m" refers to the memory allocated on a single server.
- **Cores**: The "Cores" column specifies the number of cores allocated to each server. It varied for different models and experiments.
- **Selected Servers**: This column lists the selected servers for each experiment. It may include server configurations and the number of servers, like "[14,2]" or "(35,25,20)".
- **Clients**: C1, C2, and C4 represent a number of clients being 1, 2, and 4.
- **Block Execution Time C1, C2, C4**: The "Block Execution Time" columns represent the execution time for different clients (C1, C2, C4) under both the "GMA " and "OPA" approaches.
- **Batch Size**: Batch size refers to the quantity of input data grouped together. In the context of text generation with a transformer model, a batch size of 1 corresponds to generating text based on a single input sentence. Conversely, a batch size of 'n' involves generating text for 'n' input sentences concurrently, sending all 'n' sentences to the transformer simultaneously.
- **Token**: The "number of tokens" refers to the quantity of output tokens produced by a transformer model. For instance, in text generation using a transformer model, having 100 tokens means generating a sequence of 100 words.

5.1 Varying the cores -> pick best the core

Assigning servers based on a combination of memory and cores is a basic premise of OPA in LLaMPS. In our experiments across various versions of the Bloom model (Figure 3), a notable observation emerged: the optimal performance was achieved at 8 cores, i.e., additional cores beyond 8 did not give any significant performance improvement. To validate our observations, we systematically conducted experiments with varying configurations, namely 2, 4, 8, 16, 32, and 56 cores. As shown in Figure 3, beyond the 8-core threshold, the block execution stabilized, revealing a clear knee point in the performance curve. Our results are consistent across multiple flavors of the BLOOM model and also support multiple concurrent clients.

5.2 Model fits on a single server

Table 3 depicts experiments conducted for varying flavors of BLOOM, LLaMA, and the Falcon models. We compare our approach with the GMA for the choice of servers selected for block distribution. We measure the block execution time in each case for single and concurrent clients. We experimented with OPA block placement algorithms on a single server, adjusting client loads across 1, 2, and 4 concurrent clients. These trials were conducted on three distinct models. Notably, when we refer to '1 server,' it implies that all transformer blocks are positioned on a single server, while '1 client' signifies a lone user sending input queries to the server. Conversely, '2 concurrent clients' and '4 concurrent clients' denote two and four concurrent users sending input queries to the server. The servers are named as S1, S2, S3 and S4. There server tuple is represented as S(available memory in GBs, Available cores). Our aim is to ensure the optimal distribution of transformer blocks such that the number of clients served is maximized.

(1) For the **Bloom-560m model**, we utilized four servers—S1, S2, S3,and S4—with heterogeneous memory and core specifications. GMA opted for S2 to distribute all 24 blocks, whereas OPA selected server S4. Comparing block execution times using a batch size of 16 and token size of 10, inference using GMA took 3.09 seconds for 1 client, 4.87 seconds for 2 concurrent clients, and 9.73 seconds for 4 concurrent clients. Conversely, inference using OPA resulted in execution times of 2.02 seconds for 1 client, 3.01 seconds for 2 concurrent clients, and 4.56 seconds for 4 concurrent clients. Notably, the block execution times for OPA with 1, 2, and 4 concurrent clients were lower than GMA with a single client. Therefore, OPA exhibited better performance for 1 and 2 concurrent clients compared to GMA for 1 client, while OPA across

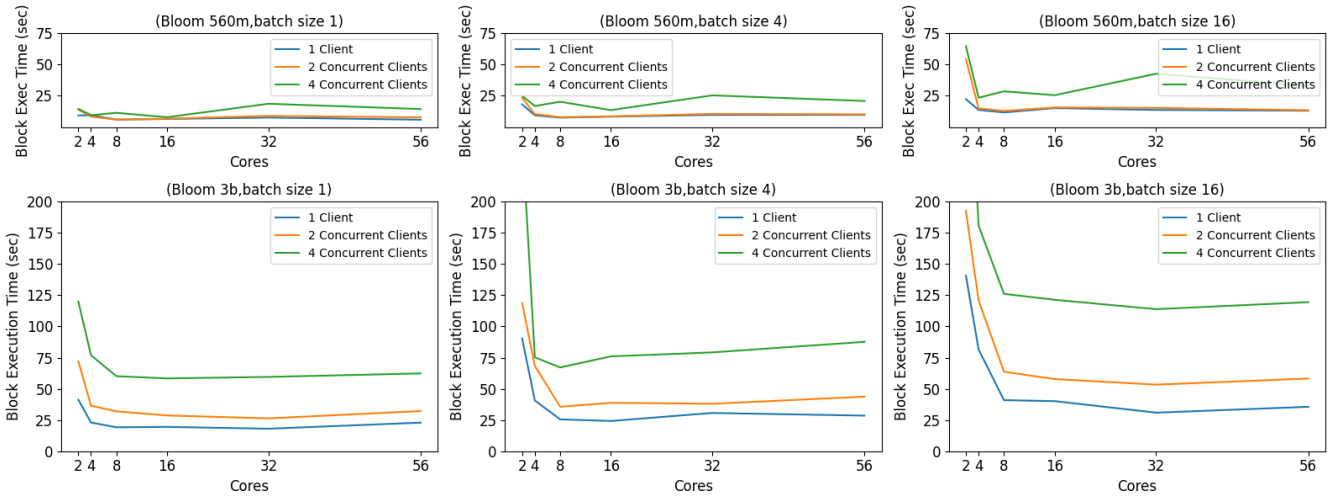

Figure 3: Knee Point on varying Cores

1, 2, and 4 concurrent clients outperformed GMA with 2 concurrent clients.

(2) For the **Bloom-3b model**, similar to the Bloom-560m model, four servers (S1, S2, S3, and S4) were utilized, each with distinct memory and core configurations. GMA selected S2 to distribute all 30 blocks, whereas OPA utilized the S4 server. The block execution times using GMA were 17.01 seconds for 1 client, 28.43 seconds for 2 concurrent clients, and 59.46 seconds for 4 concurrent clients. However, with OPA, the execution times were notably lower: 7.12 seconds for 1 client, 10.12 seconds for 2 concurrent clients, and 20.15 seconds for 4 concurrent clients. Once again, OPA exhibited better performance for 1 and 2 concurrent clients compared to GMA for 1 client, while outperforming the GMA across 1, 2, and 4 concurrent clients.

(3) Lastly, for the **Bloom-7b model**, with the same set of servers (S1, S2, S3, and S4) and their corresponding memory and core specifications, GMA opted for S2 to distribute all 30 blocks, whereas OPA selected the use of the S4 server. The block execution times using the GMA were 45.02 seconds for 1 client, 83.21 seconds for 2 concurrent clients, and 179.01 seconds for 4 concurrent clients. However, employing OPA resulted in significantly lower execution times: 20.1 seconds for 1 client, 34.5 seconds for 2 concurrent clients, and 67.44 seconds for 4 concurrent clients. Similar to the previous models, OPA demonstrated superior performance for 1 and 2 concurrent clients compared to GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to GMA with 2 concurrent clients.

We were unable to conduct experiments with Falcon and LLaMA models as the leftover capacity of any single server was insufficient to load all the blocks of Falcon and LLaMA. In the next set of experiments, we focus on scenarios where the leftover capacity of servers is insufficient even to load the smallest version of the BLOOM model.

5.3 Models fits on 2 servers

As shown in Table3 we experimented with OPA block placement algorithms across two servers, varying client loads with 1, 2, and 4 concurrent clients. These trials encompassed two different models, including three versions of the Bloom model, and the Falcon model. The leftover capacity was insufficient to fit LLaMA on any two given servers and hence LLaMA was not a part of this experiment. When a single server couldn't accommodate all transformer blocks, we utilized two servers. Additionally, with a batch size of 16 and a fixed token length of 10, the OPA algorithm indicated an increase in block execution time as the number of concurrent clients increased, specifically with 2 concurrent clients.

In Table 3

(1) We allocated blocks of the Bloom 560m model using both GMA and OPA methods across servers S1, S2, S3, and S4. S1 has 13.5 GB memory with 8 cores, S2 has 12 GB memory with 2 cores, while S3 and S4 possess 12 GB memory with 4 and 8 cores, respectively. The GMA picked S1 (14 blocks) and S2 (10 blocks), while OPA selected S1 and S4. Block execution times with the GMA were 3.73 seconds for 1 client, 4.49 seconds for 2 concurrent clients, and 8.82 seconds for 4 concurrent clients. In contrast, OPA resulted in notably lower execution times: 2.75 seconds for 1 client, 2.88 seconds for 2 concurrent clients, and 3.11 seconds for 4 concurrent clients. Similar to prior models, OPA demonstrated superior performance for 1 and 2 concurrent clients compared to the GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to the GMA with 1 client.

(2) For the Bloom 3b model, we distributed its blocks using both GMA and OPA methods across servers S1, S2, S3, and S4. S1 offers 17.5 GB memory with 8 cores, S2 has 16 GB memory with 2 cores, while S3 and S4 possess 16 GB memory with 4 and 8 cores, respectively. The GMA selected S1 (16 blocks) and S2 (14 blocks), while OPA chose S1 and S4. Block execution times with the GMA were 18.11 seconds for 1 client,

Table 3: OPA vs GMA on 1,2,3 servers

Bloom 560m - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(12,2), S2(14,2), S3(14,4), S4(14,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S2)	[24]	3.89	4.87	9.73
OPA	16	10	(S4)	[24]	2.02	3.01	4.56
Servers: (GB,cores) [S1(13.5,8), S2(12,2), S3(12,4), S4(12,8)]							
GMA	16	10	(S1,S2)	[14,10]	3.73	4.49	8.82
OPA	16	10	(S1,S4)	[14,10]	2.43	2.59	3.65
Servers: (GB,cores) [S1(12.15,8), S2(11.65,8), S3(11.15,2), S4(11.15,4), S5(11.15,8)]							
GMA	16	10	(S1,S2,S3)	[12,8,4]	2.75	3.01	3.65
OPA	16	10	(S1,S2,S5)	[12,8,4]	2.68	2.88	3.11
Bloom 3b - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(23,8), S2(23.5,2), S3(23.5,4), S4(23.5,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S2)	[30]	17.01	28.43	59.46
OPA	16	10	(S4)	[30]	7.12	10.12	20.15
Servers: (GB,cores) [S1(17.5,8), S2(16.7,2), S3(16.7,4), S4(16.7,8)]							
GMA	16	10	(S1,S2)	[16,14]	18.11	28.62	54.44
OPA	16	10	(S1,S4)	[16,14]	8.6	9.97	19.13
Servers: (GB,cores) [S1(16.7,8), S2(15,2), S3(13.3,2), S4(13.3,4), S5(13.3,8)]							
GMA	16	10	(S1,S2,S3)	[14,10,6]	10.93	12.09	14.11
OPA	16	10	(S1,S2,S5)	[14,10,6]	10.28	11.23	13.91
Bloom 7b1 - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(42,8), S2(44,2), S3(44,4), S4(44,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S2)	[30]	45.02	83.21	179.01
OPA	16	10	(S4)	[30]	20.1	34.5	67.44
Servers: (GB,cores) [S1(31,8), S2(30,2), S3(30,4), S4(30,8)							
GMA	16	10	(S1,S2)	[16,14]	46.78	70.12	160.12
OPA	16	10	(S1,S4)	[16,14]	23.08	24.98	42.01
Servers:(GB,cores) [S1(30,8), S2(26,8), S3(22,2), S4(22,4),S5(22,8)]							
GMA	16	10	(S1,S2,S3)	[14,10,6]	27.79	29.12	38.35
OPA	16	10	(S1,S2,S5)	[14,10,6]	22.12	23.59	36.12
Falcon-40b - (batch size=16, tokens=10)							
Servers:(GB,cores) [S1(98.3,8), S2(88,2), S3(88,4), S4(88,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S1,S2)	[32,28]	491.2	536.12	840.02
OPA	16	10	(S1,S4)	[32,28]	211.37	305.12	682.01
Servers: (GB,cores) [S1(93.2,2), S2(67.5,8), S3(42,2), S4(42,4), S5(42,8)]							
GMA	16	10	(S1,S2,S3)	[30,20,10]	260.12	266.72	352.12
OPA	16	10	(S1,S2,S5)	[30,20,10]	258.19	263.89	345.26
Llama2-70b - (batch size=16, tokens=10)							
Servers: (GB,cores) [S1(155,8), S2(113.5,2), S3(83.5,2), S4(83.5,4),S5(83.5,8)]							
Approach	**batch size**	**token length**	**Selected Servers**	**Block Distrbn**	**1 client**	**2 clients**	**4 clients**
GMA	16	10	(S1,S2,S3)	[35,25,20]	396.11	408.12	798.12
OPA	16	10	(S1,S2,S5)	[35,25,20]	279.13	304.11	588.76

28.63 seconds for 2 concurrent clients, and 54.44 seconds for 4 concurrent clients. However, OPA resulted in significantly lower execution times: 8.6 seconds for 1 client, 9.97 seconds for 2 concurrent clients, and 19.13 seconds for 4 concurrent clients. Similar to previous models, OPA exhibited superior performance for 1 and 2 concurrent clients compared to the GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to the GMA with 2 concurrent clients.

(3) In the case of the Bloom 7b1 model, we utilized both GMA and OPA methods to distribute its blocks across servers S1,

S2, S3, and S4. S1 has 31 GB memory with 8 cores, S2 has 30 GB memory with 2 cores, while S3 and S4 have 30 GB memory with 4 and 8 cores, respectively. The GMA opted for S1 (16 blocks) and S2 (14 blocks), whereas OPA chose S1 and S4. Block execution times with the GMA Approach were 46.78 seconds for 1 client, 70.12 seconds for 2 concurrent clients, and 160.12 seconds for 4 concurrent clients. However, OPA resulted in notably lower execution times: 23.1 seconds for 1 client, 24.98 seconds for 2 concurrent clients, and 42.01 seconds for 4 concurrent clients. As with previous models, OPA showcased better performance for 1 and 2 concurrent clients compared to the GMA for 1 client, and across 1, 2, and 4 concurrent clients compared to the GMA with 1 client.

(4) For the Falcon model, we employed both GMA and OPA methods to distribute all 60 blocks across servers S1, S2, S3, and S4. S1 offers 98.3 GB memory with 8 cores, S2 has 88.2 GB memory with 2 cores, while S3 and S4 possess 88.2 GB memory with 4 and 8 cores, respectively. The GMA selected S1 (14 blocks) and S2 (10 blocks), whereas OPA chose S1 and S4. Notably, for Client 1, GMA exhibited a block execution time of 491.2 seconds, while OPA showed 211.37 seconds for Client 1 and 305.5 seconds for Client 2, signifying an improvement over Client 1 using GMA. This observed trend remained consistent across other models as well.

5.4 Model fits on 3 servers

Next, we experimented with OPA block placement algorithms across three servers, varying the client loads between 1, 2, and 4 concurrent clients. These trials encompassed three different models, including three versions of the Bloom model, the Falcon model, and the LLama 2 model. When a leftover capacity of a single or two servers was insufficient to accommodate all transformer blocks we designed this experiment using 3 servers. Notably, when employing the OPA algorithm with 2 concurrent clients, the block execution time increased with a higher number of concurrent clients, all the while maintaining a batch size of 16 and a fixed token length of 10.

In the context of the Falcon model presented in Table 3, we distributed blocks using both GMA and OPA methods across servers S1, S2, S3, S4, and S5. These servers possess varying memory and core configurations, with GMA selecting S1, S2, and S3, while OPA opted for S1, S2, and S5. Specifically, for Client 1 in GMA, the block execution time recorded was 260.12 seconds. However, utilizing OPA, the block execution time was 258.19 seconds for Client 1 and 263.89 seconds for Client 2, demonstrating comparable performance to that of Client 1 in GMA.

(1) bloom 560m: Utilizing both GMA and OPA techniques, we allocated all 24 blocks among servers S1, S2, S3, S4, and S5. S1 has 12.15 GB memory and 8 cores, S2 holds 11.65 GB memory with 8 cores, while S3, S4, and S5 share 11.15 GB memory but differ in core count—2, 4, and 8 cores respectively. GMA selected S1, S2, and S3, whereas OPA favored S1 (12 blocks), S2 (8 blocks), and S5 (4 blocks). Under the GMA, block execution times for 1 client were 2.75 seconds, 2 concurrent clients took 3.01 seconds, and 4 concurrent clients demanded 3.65 seconds. Meanwhile, employing OPA resulted in execution times of 2.68 seconds for 1 client, 2.88 seconds for 2 concurrent clients, and 3.11 seconds for 4 concurrent clients. OPA showcased superior performance across 1, 2, and 4 concurrent clients in contrast to the GMA.

(2) bloom 3b: All 30 blocks were allocated across servers S1, S2, S3, S4, and S5 using both GMA and OPA methodologies. S1 possesses 16.7 GB memory and 8 cores, S2 holds 15 GB memory with 8 cores, while S3, S4, and S5 share 13.3 GB memory, differing in core count—2, 4, and 8 cores respectively. GMA favored S1, S2, and S3, while OPA distributed 14 blocks to S1, 10 blocks to S2, and 6 blocks to S5. Block execution times under GMA for 1 client were 10.93 seconds, 12.09 seconds for 2 concurrent clients, and 14.11 seconds for 4 concurrent clients. With OPA, execution times were 10.28 seconds for 1 client, 11.23 seconds for 2 concurrent clients, and 13.91 seconds for 4 concurrent clients. OPA demonstrated superior performance across 1, 2, and 4 concurrent clients compared to the Greedy Approach.

(3) bloom 7b1: Distributing all 30 blocks across servers S1, S2, S3, S4, and S5 was achieved using GMA and OPA methods. S1 boasts 30 GB memory and 8 cores, S2 holds 26 GB memory with 8 cores, while S3, S4, and S5 share 22 GB memory but differ in core count—2, 4, and 8 cores respectively. GMA selected S1, S2, and S3, whereas OPA allocated 14 blocks to S1, 10 blocks to S2, and 6 blocks to S5. Using GMA, block execution times were 27.79 seconds for 1 client, 29.12 seconds for 2 concurrent clients, and 38.35 seconds for 4 concurrent clients. Meanwhile, OPA showed execution times of 22.12 seconds for 1 client, 23.59 seconds for 2 concurrent clients, and 36.12 seconds for 4 concurrent clients. OPA showcased superior performance across 1, 2, and 4 concurrent clients compared to the GMA.

(4) Falcon model: All 60 blocks were distributed across servers S1, S2, S3, S4, and S5 using GMA and OPA methodologies. S1 possesses 92.2 GB memory and 8 cores, S2 has 67.7 GB memory with 8 cores, while S3, S4, and S5 share 42 GB memory, differing in core count—2, 4, and 8 cores respectively. GMA selected S1, S2, and S3, whereas OPA opted for S1, S2, and S5. Under GMA, block execution times were 260.12 seconds for 1 client, 266.72 seconds for 2 concurrent clients, and 352.12 seconds for 4 concurrent clients. OPA demonstrated execution times of 258.19 seconds for 1 client, 263.89 seconds for 2 concurrent clients, and 345.26 seconds for 4 concurrent clients. OPA showcased superior performance across 1, 2, and 4 concurrent clients compared to the GMA Approach.

(5) LLama 2 model: Employing both GMA and OPA methods, we distributed all 80 blocks among servers S1, S2, S3, S4, and S5. S1 boasts 155 GB memory and 8 cores, S2 holds 113.5 GB memory with 8 cores, while S3, S4, and S5 share 83.5 GB memory, differing in core count—2, 4, and 8 cores respectively. S1, S2, and S3, while OPA favored S1 (35 blocks), S2 (25 blocks), and S5 (20 blocks). Block execution times under GMA for 1 client were 396.11 seconds, 408.12 seconds for 2 concurrent clients, and 798.12 seconds for 4 concurrent clients. Meanwhile, employing OPA resulted in execution times of 279.13 seconds for 1 client, 404.11 seconds for 2 concurrent clients, and 588.76 seconds for 4 concurrent clients.

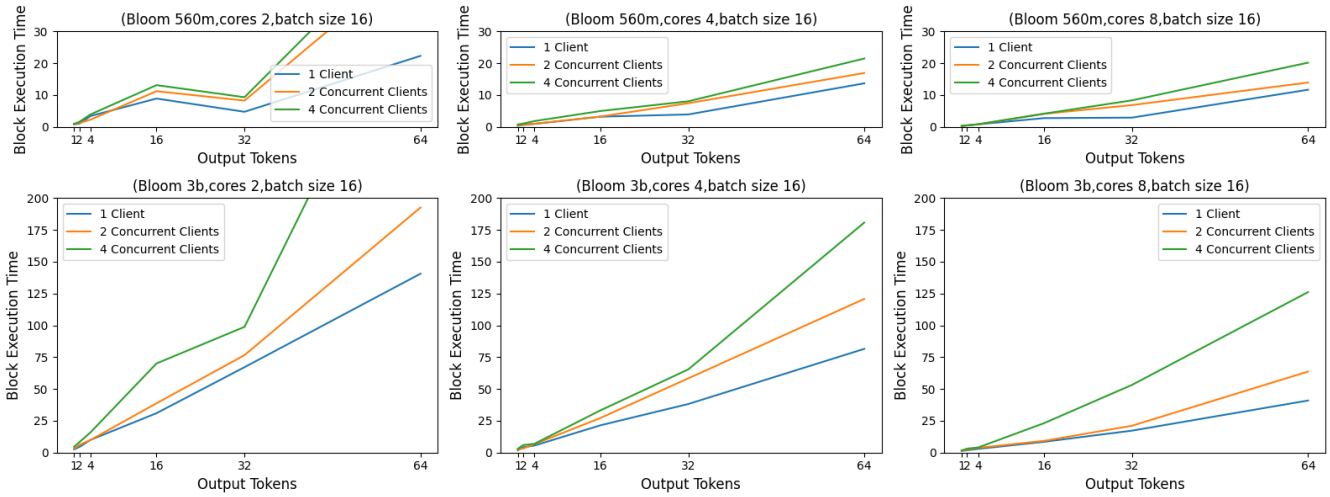

Figure 4: Block execution time for 1, 2, 4 concurrent clients

OPA demonstrated superior performance across 1, 2, and 4 concurrent clients compared to the GMA.

As observed from the results in Table 3, OPA outperforms GMA in almost all the cases, with a lower block execution time as compared to GMA. These results are consistent across different models. BLOOM, Falcon and LLaMA 2 with varying parameters and number of clients.

6 ABLATION STUDIES

In this section we discuss ablation studies to better understand and analyze the performance of OPA in LLaMPS. Figure 4 illustrates the behavior of OPA upon increasing the number of output tokens while keeping the batch size fixed. When the cores are limited to 2, the block execution time shows a steep increase as the number of output tokens increases. This is expected behavior as the compute increases, and so does the block execution time. This is particularly observed with an increase in concurrency (clients 2 and 4) and hence increase in a number of client requests. Upon adding more cores (4 and 8) we observe that the overall block execution time shows a decreasing trend as more compute becomes available. This trend remains consistent across the BLOOM 3b model, thus corroborating the efficacy of the OPA approach.

Previously, in Figure 3, we discussed the *knee-point* of the curve by varying the number of cores. These graphs illustrate that with increasing batch size, adding cores is beneficial till a certain point (i.e. 8 cores). Beyond 8 cores, we do not observe a significant improvement as some tasks performed during inference may not be parallelizable beyond 8 degrees. For example, in Bloom 560m with a batch size of 1, batch execution time is less than 20 seconds, for a batch size of 4, it is slightly over 20 seconds, and for a batch size of 16, it is less than 40 seconds. With the increase in cores, concurrency is more efficiently handled and this pattern holds for the Bloom 3b model as well.

Thus, OPA shows expected behavior across varying parameters (cores, batch sizes, number of output tokens) and the trend in the results is consistent across models with varying parameters. The efficacy of these results has also been validated on a heterogeneous cpu-based cluster setup in an enterprise.

7 RELATED WORK

In recent years, the deployment of large language models has seen widespread adoption owing to their versatile applications in real-world scenarios, predominantly in the field of natural language processing (NLP). As these models increase in size and complexity, their computational demands have necessitated distributed computing solutions, given their impracticality to fit on a single machine. Models such as OPT-175B[25], BLOOM-176B [23], and LLama-70b[22] demand substantial accelerator memory, exceeding 350 GB for inference. The computational requirements impose a barrier, necessitating multiple high-end GPUs for downstream tasks.

There have been efforts to distribute transformer blocks across multiple servers. Deepspeed [3] [17][18] is a distributed inference solution that supports large transformer-based language models. It enables parallel inference and achieves tensor parallelism by leveraging multiple GPUs. Deepspeed utilizes maximum available resources on multiple servers, enhancing overall performance and scalability.

Another approach by Deepspeed [3][17][18], namely Deepspeed Zero inference[19] involves "offloading" model parameters to more economical memory sources like RAM or SSD and subsequently executing them on the GPU layer by layer. While this strategy enables the deployment of LLMs on a single GPU, it introduces inherent trade-offs, such as increased latency due to the time-consuming offloading of layers. Additionally, the large size of layers and the substantial transfer overhead between the GPU and CPU further contribute to these challenges.

FastServe [24] is a system for distributed inference serving of LLMs. The system is built on NVIDIA FasterTransformer and exploits the autoregressive pattern of LLM inference to enable preemption at the granularity of each output token. FastServe utilizes

high-end GPUs and similar to DeepSpeed leverages maximum resources available to minimize the job (JCT) completion time.

Another avenue to enhance LLM accessibility involves leveraging public inference APIs for example deploying the LLM on Sagemaker [1] which is AWS [2] service later accessed model using API. However, the pricing structure of these APIs can render certain research projects prohibitively expensive. This economic constraint poses a limitation on the widespread use of LLMs for various research endeavors and potential applications.

Petals [7] works as a decentralized framework designed for fast inference of transformer-based models. It splits any given model into several blocks that are hosted on different servers. These servers can be spread out across continents, and anybody can connect their own GPU! In turn, users can connect to this network as a client and apply a chosen model to their data.

In our context, our paper introduces a novel perspective, focusing on the inference of large language models within enterprise setups. We seek to leverage the residual (leftover) capacity present on servers, harnessing untapped resources for efficient model inference. Our work explores the optimization of block assignment, which stays unexplored in related literature at this point. Through this exploration, we aim to push the boundaries of efficiency in large language model inference, ultimately maximizing the number of clients that can be accommodated, and thereby ensuring distributed inference of LLMs in a cost-effective manner for enterprises by utilizing leftover capacity.

8 CONCLUSIONS AND FUTURE WORK

Our paper introduces LLaMPS, a Large Language Model Placement System designed to address the challenge of efficiently deploying large language models (LLMs) within enterprise setups LLaMPS. Our approach focuses on the placement of transformer blocks, optimizing the utilization of enterprise resources by utilizing leftover capacity in worker nodes in an enterprise setup. The Optimal Placement Algorithm (OPA) maximizes the number of clients that can be served concurrently by optimally placing transformer blocks on residual resources. Through extensive experimentation with open-source large language models such as BLOOM (with 1b, 3b, and 7b parameters) Falcon and LLaMA, our results consistently demonstrate the efficacy of LLaMPS in facilitating optimal transformer block placement. By leveraging leftover resources, LLaMPS paves the way for enterprise-level deployment of large language models. LLaMPS can also maintain sustainability by optimally utilizing leftover capacity of machines within an enterprise; consequently saving the cost of buying extra machines/resources to host the entire LLM model. As a part of future work, we plan to investigate the feasibility of a system that dynamically identifies and utilizes leftover capacities across cloud instances. LLaMPS can be used in Retrieval Augmented Generation (RAG) applications for doing the inference or other downstream tasks using LLMs when latency is not critical.

REFERENCES

[1] [n. d.]. Amazon SageMaker. https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html.

[2] [n. d.]. AWS. https://aws.amazon.com/.

[3] 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.

[4] Debarag Banerjee, Pooja Singh, Arjun Avadhanam, and Saksham Srivastava. 2023. Benchmarking LLM powered Chatbots: Methods and Metrics. *arXiv preprint arXiv:2308.04624* (2023).

[5] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2022. Distributed Inference and Fine-tuning of Large Language Models Over The Internet. (2022).

[6] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2022. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188* (2022).

[7] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2023. Petals: Collaborative Inference and Fine-tuning of Large Models. arXiv:2209.01188 [cs.LG]

[8] Jennifer D'Souza. [n. d.]. A Review of Transformer Models. ([n. d.]).

[9] Wenqi Fan, Zihuai Zhao, Jiatong Li, Yunqing Liu, Xiaowei Mei, Yiqi Wang, Jiliang Tang, and Qing Li. 2023. Recommender systems in the era of large language models (llms). *arXiv preprint arXiv:2307.02046* (2023).

[10] A Shaji George and AS Hovan George. 2023. A review of ChatGPT AI's impact on several business sectors. *Partners Universal International Innovation Journal* 1, 1 (2023), 9–23.

[11] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. (2023).

[12] AE Hefnawy and Ahmed Soliman Mohammed. 2014. Review of different methods for deriving weights in the Analytic Hierarchy Process. *International Journal of the Analytic Hierarchy Process* 6, 1 (2014), 92–123.

[13] Xu Huang, Jianxun Lian, Yuxuan Lei, Jing Yao, Defu Lian, and Xing Xie. 2023. Recommender AI Agent: Integrating Large Language Models for Interactive Recommendations. *arXiv preprint arXiv:2308.16505* (2023).

[14] Nian Li, Chen Gao, Yong Li, and Qingmin Liao. 2023. Large language model-empowered agents for simulating macroeconomic activities. *arXiv preprint arXiv:2310.10436* (2023).

[15] Jianghao Lin, Xinyi Dai, Yunjia Xi, Weiwen Liu, Bo Chen, Xiangyang Li, Chenxu Zhu, Huifeng Guo, Yong Yu, Ruiming Tang, et al. 2023. How Can Recommender Systems Benefit from Large Language Models: A Survey. *arXiv preprint arXiv:2306.05817* (2023).

[16] Keivalya Pandya and Mehfuza Holia. 2023. Automating Customer Service using LangChain: Building custom open-source GPT Chatbot for organizations. *arXiv preprint arXiv:2310.05421* (2023).

[17] Samyam Rajbhandari, Conglong Li, Zhewei Yao, and Minjia Zhang. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International Conference on Machine Learning*. PMLR, 18332–18346.

[18] 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 & Data Mining*. 3505–3506.

[19] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, and Chen. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. (2023).

[20] Ah-Hwee Tan. 2004. FALCON: A fusion architecture for learning, cognition, and navigation. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, Vol. 4. IEEE, 3297–3302.

[21] Hugo Touvron, Thibaut Lavril, and Gautier Izacard. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[22] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[23] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Castagné, François Yvon, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[24] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. *arXiv preprint arXiv:2305.05920* (2023).

[25] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]