Less is More: Optimizing Function Calling for LLM Execution on Edge Devices

Varatheepan Paramanayakam[1], Andreas Karatzas[1], Iraklis Anagnostopoulos[1], Dimitrios Stamoulis[2]
[1]School of Electrical, Computer and Biomedical Engineering, Southern Illinois University, Carbondale, IL, U.S.A.
[2]Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, U.S.A.
Email: {varatheepan, andreas.karatzas, iraklis.anagno}@siu.edu, dstamoulis@utexas.edu

***Abstract*—The advanced function-calling capabilities of foundation models open up new possibilities for deploying agents to perform complex API tasks. However, managing large amounts of data and interacting with numerous APIs makes function calling hardware-intensive and costly, especially on edge devices. Current Large Language Models (LLMs) struggle with function calling at the edge because they cannot handle complex inputs or manage multiple tools effectively. This results in low task-completion accuracy, increased delays, and higher power consumption. In this work, we introduce Less-is-More, a novel fine-tuning-free function-calling scheme for dynamic tool selection. Our approach is based on the key insight that selectively reducing the number of tools available to LLMs significantly improves their function-calling performance, execution time, and power efficiency on edge devices. Experimental results with state-of-the-art LLMs on edge hardware show agentic success rate improvements, with execution time reduced by up to 70% and power consumption by up to 40%.**

I. Introduction and Motivation

Among the numerous advances in Generative AI, a powerful emerging paradigm is leveraging the reasoning capabilities of Large Language Models (LLMs) within agentic systems that can execute the appropriate API tools to complete user queries [1]. However, as function calling involves complex tasks, data handling, and API interactions, deploying such agents introduces significant hardware bottlenecks [2].

Recent approaches from both industry and academia – such as OpenAI's parallel tool-calling capabilities [3] and novel prompting techniques [4] – have contributed to system-efficient function calling. Unfortunately, most solutions focus on serverless GPT-based deployments, severely limiting their applicability on the edge. As highlighted by recent announcements from smart-device providers like Apple, there is a growing need for efficient on-device execution, especially in applications where user data cannot be sent to the cloud [5].

A common approach is to replace large billion-parameter models (e.g., 70B, 400B) with "*smaller*" LLMs that have fewer than 10B parameters (e.g., 1.5B, 3.8B, 7B, 8B) [5]. Even with fewer parameters, the default versions of these models still have significant overheads in terms of execution time and power consumption [2]. Libraries like HuggingFace [6] and Ollama [7] offer optimizations such as aggressive quantization, token pruning, and context window reduction to improve efficiency on edge hardware. However, these smaller, quantized/pruned LLMs often lack sophisticated function-calling abilities [8] and require costly refinement and fine-tuning to reach the performance of larger models [5]. Notably, when deploying "off-the-shelf" open-source LLMs **without** additional fine-tuning, there is a significant performance drop. As shown in Table I, the success rates on the Berkeley Function-Calling Leaderboard (BFCL) [9] and GeoEngine [10] benchmarks drop substantially when comparing quantized/pruned LLMs to their full-precision HuggingFace counterparts.

TABLE I: Success rate of Llama3.1-8b variations [7] on the BFCL [9] and GeoEngine [10] benchmarks.

Benchmark	Full precision	q4_0	q4_1	q4_K_M	q8_0
BFCL [9]	63.04%	20.43%	34.35%	39.57%	44.35%
GeoEngine [10]	63.91%	43.04%	59.57%	56.96%	53.04%

TABLE II: Execution of function-calling query using Llama3.1-8b-q4_K_M [7] on Nvidia Jetson AGX Orin [11].

Context window	# Tools	Successful	Exec. time (s)	Power (W)
16K	46	✗	30	27
16K	19	✓	20	26
8K	19	✓	17	22
Max drop			↓ 43%	↓ 19%

Furthermore, we would like to illustrate that enabling function calling on edge devices to support advanced functionality is a complex task, often resulting in high execution time and power consumption. Take, for example, the following query from the GeoEngine benchmark [10]:

Query example from GeoEngine benchmark [10]

Plot the fmow VQA captions in UK from Fall 2009

GeoEngine includes 46 tools that are provided to the LLM with each query, allowing the LLM to decide which ones to use. As shown in Table II, even though Llama3.1-8b-q4_K_M has a 16K context window that can fit all the tools, it fails to select the correct one. This occurs because of the large number of available options confusing the LLM. However, by reducing the number of tools provided, the model's reasoning ability improves. For instance, when only 19 tools are passed, the LLM chooses the correct tool and completes the query successfully. This reduction in tools helps the LLM focus and results in a higher success rate. Moreover, it significantly decreases execution time. In essence, providing **fewer** options enables the LLM to make more accurate and faster decisions. Therefore, it is crucial to develop a mechanism that *dynamically reduces the set of tools available to the LLM while adjusting*

978-3-9826741-0-0/DATE25/© 2025 EDAA

the context window accordingly to optimize both performance and energy efficiency on edge devices.

In this work, we introduce `Less-is-More`, a novel fine-tuning-free function-calling scheme for dynamic tool selection. Our **key insight** is that selectively reducing the number of tools available to the LLM significantly improves its decision-making ability. By presenting the LLM with fewer, *more relevant tools*, we reduce confusion, allowing the model to focus better and achieve higher accuracy. Additionally, this reduction improves execution time and power consumption, and it allows the use of smaller context windows, enhancing both execution speed and energy efficiency further. `Less-is-More` achieves these improvements without requiring any model retraining or fine-tuning, making it a practical solution for optimizing LLM performance on resource-constrained edge devices.

Overall, our main contributions are: ❶ We introduce `Less-is-More`, a new approach for dynamic tool selection that optimizes function-calling in LLMs without requiring any finetuning or retraining, making it a plug-and-play solution for all existing state-of-the-art LLMs. ❷ Our method boosts LLM task-completion success rates by reducing the number of tools provided to the LLM, which reduces tool-space complexity and enables the agent to make more accurate and efficient decisions during function calls. ❸ We achieve significant reductions in power consumption and response time for LLMs running on edge devices, improving overall performance and efficiency.

II. Related Work

Many recent approaches aim to improve the runtime efficiency of LLMs, such as quantization [12], pruning [13], KV token caching [14], knowledge distillation [15], token compression [16], in-memory execution [17], [18], and speculative decoding [19]. However, existing methods are evaluated primarily in GPT-style conversational flows or text generation tasks, limiting their applicability to function calling [20], where hardware-efficient deployment remains a bottleneck. As we demonstrated in Section I, when quantized models with pruned window sizes are used as agents, there is a significant performance drop. In contrast, our approach focuses on enhancing agentic capabilities and maintaining performance consistency across different models, whether quantized, fine-tuned, or otherwise.

Edge-LLM [21] employs adaptive layer voting for efficient execution on edge devices. Recent work extends these voting schemes to Mixture-of-Experts [22] and LLM cascading [23], dynamically selecting between more powerful and weaker models based on a local-remote paradigm [22]. While these methods are practical for certain tasks, they have limited applicability in function-calling scenarios where tool APIs often interact with user data, making remote execution unsuitable due to privacy concerns [24]. Last, tool- or query-caching [25], [26] has been explored for efficient function calling, based on the intuition of storing and reusing recently or frequently used LLM outputs. While these methods offer latency improvements, their effectiveness applies mainly to cloud-based environments where storage is not constrained.

Moreover, ToolLLM [27] employs a tree-based scheme to minimize the number of tools required for task execution, yet it requires LLM calls across the entire tool set, making it impractical for edge devices – where delay and power consumption are critical – while still suffering from the same limitations as the "off-the shelf" LLM baselines, such as limited window sizes. In [1], the authors employ a RAG-based tool search scheme to identify relevant tools, while TinyAgent [8] uses a transformer-based classifier for tool selection. Octopus [28] improves function calling by fine-tuning smaller LLMs with token-masking to address response misalignment. However, while these approaches enhance performance for the specific datasets they are trained on, they are not scalable across diverse function spaces, as they require extensive fine-tuning. In contrast, our method simplifies the task without the need for fine-tuning, allowing for easy adaptation to new tools.

Following OpenAI's parallel function-calling release [3], tool "compilation" [29], [30] has been explored to optimize agent performance by optimizing the number of tools executed per each LLM call. However, these methods require standalone LLM calls using full-precision GPT models to perform the compiler logic, making them impractical for edge deployments, where computational resources and power are limited. In contrast, our approach uses a much simpler and lightweight similarity-based mechanism, which can be handled efficiently by the deployed LLM itself. We enable this with an inexpensive, pretrained embedding tokenizer, eliminating the need for full-precision models and reducing both execution time and power consumption significantly.

III. Methodology

`Less-is-More` simplifies LLM function calling on edge devices by dynamically reducing the number of available tools, enhancing task-completion performance, execution speed, and power efficiency, without requiring fine-tuning or complex model adjustments. Figure 1 provides an overview of our approach, which consists of three main components: (**i**) a set of **Search Levels** constructed *offline* with varying tool-description granularity, (**ii**) a **Tool Recommender** which *at runtime* and for each user prompt generates tool descriptions needed to complete the task, and (**iii**) a **Tool Controller** which finalizes tool selection with respect to the selected **Level**.

The core idea behind `Less-is-More` is to *avoid presenting the LLM with all tools upfront*. **Without** providing any APIs at first, the LLM is instead instructed to reason about the number and type of tools it "believes" it requires to answer the user's query, with the **Recommender** generating "ideal" tool definitions based on its reasoning. Based on these LLM-generated descriptions, we employ "similarity search" to identify the actual tools that are "closest" to the LLM's suggestions, with the **Controller** determining the appropriate **Search Level**.

A. Constructing the *Search Levels* offline

To address varying levels of query complexity and performance demands, we consider three distinct **Search Levels**: either selecting from *individual* tools directly (**Search Level 1**), from *clusters* of tools (**Search Level 2**), or the *entire* tool set (**Search Level 3**). In constructing these search spaces, we aim to minimize the overhead induced at runtime for tool selection.

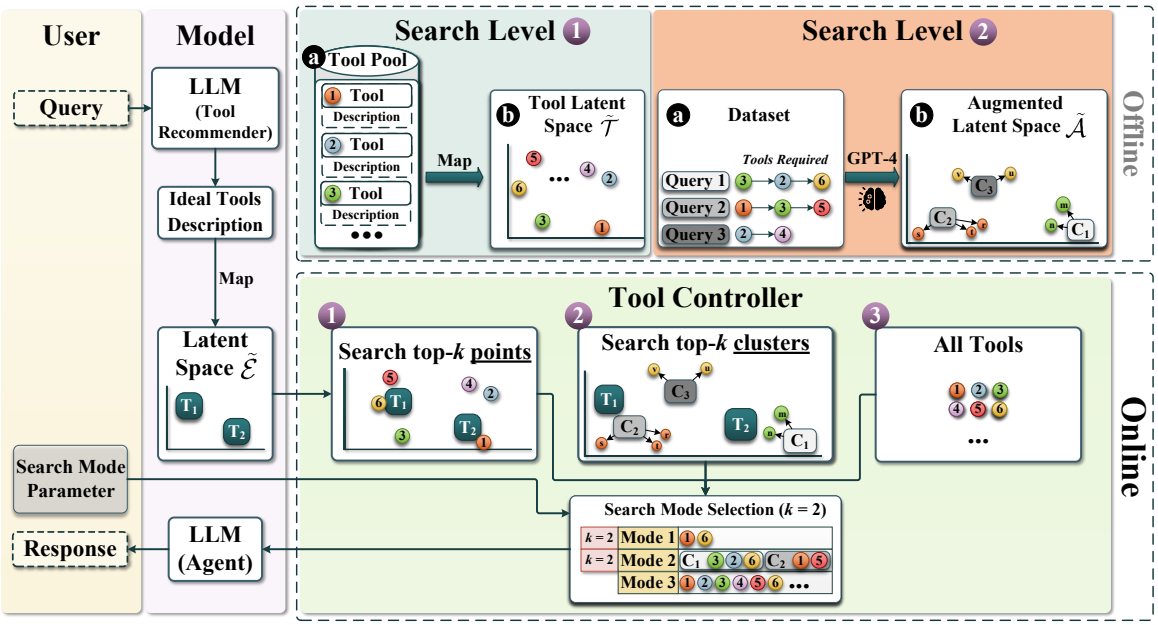

Fig. 1: `Less-is-More` considers three distinct tool representations (**Search Levels**): *individual* tools (**Level 1**), tool *clusters* (**Level 2**), or the *entire* tool set (**Level 3**), whose latent spaces are constructed *offline*. At runtime, given a query and **without** providing any tools, the LLM (**Recommender**) generates tool descriptions "ideal" for the task. Given the LLM-recommended tool embeddings, the **Controller** identifies the most relevant *real* tools based on their proximity in the latent space representations.

Drawing inspiration from RAG-based similarity search [31], we map all tool descriptions into embeddings, building a 768-dimensional latent space using a pre-trained embedding model based on the MPNet architecture [32].

Akin to RAG vectorstores [31], this embedding encodes textual API descriptions into numerical vectors that capture their semantic meaning with two key advantages: (**i**) for a given set of API tools, it requires only an **one-time** offline step to populate the search levels, placing tools with similar functions or characteristics close to each other in the latent space. Moreover (**ii**) at runtime, this spatial arrangement allows for efficient similarity measurements of tools to user queries, since the same tokenization can be applied to the LLM-generated "ideal" tool descriptions and match them against the stored "real" APIs. With this representation, `Less-is-More` can quickly identify which tools are most relevant to the prompted query based on their proximity in the latent space.

Search Level 1 - Individual tools: The latent space, denoted as $\tilde{\mathcal{T}}$, consists of embeddings derived from the descriptions of all available API tools using the MPNet model. This mapping of tool descriptions into the latent space $\tilde{\mathcal{T}}$ is performed as a preprocessing step, offline and prior to any user interaction, ensuring efficient real-time query processing. At this finest tool granularity, **Level 1** prioritizes speed as it enables matching LLM-generated tool descriptions directly to individual tools for straightforward queries (e.g., single-function tasks). However, since this level relies only on the latent space closeness of tools, it might compromise accuracy for more complex queries that require interactions between multiple functions. To this end, we introduce a more complex search level to address the limitations of *standalone* tool-matching, as discussed next.

Search Level 2 - Tool clusters: Our goal is to construct a coarser latent space with clusters of tools. However, a clustering algorithm based on tool (text) descriptions would produce groups that poorly capture tool-usage patterns. For example, a task to translate a document and then open it in a browser would require document-related and UI-related tools, which rudimentary clustering will place in separate groups.

To this end, we draw inspiration from benchmark *augmentation* methods [24], [33]. In these approaches, GPT is first used to augment an existing dataset by generating queries contextually similar to those in the original pool. Subsequently, GPT generates additional queries containing tasks that are contextually proximate to the tasks present in the original queries. As both GeoEngine [10] and BFCL [9] provide a categorization of their benchmark question types (e.g., wiki questions, document tasks, math tools, etc.), we randomly sample 10 queries per category from their *training* sets. We then follow ToolQA [33] and we prompt GPT-4 `(Turbo 0125)` to generate queries with contextually proximate tasks and their respective solutions. For instance, in the previous example, where the original query involved opening a document, a task permutation might involve printing it instead. Note here that factual "correctness" for these generated queries is less critical, as they are not used for training or runtime decisions [33]. Instead, they only serve as "noisy" queries for clustering, and we measure their quality based on a similarity score (i.e., ROUGE score following [10], [33]), ensuring diverse tool combinations without redundancy.

We map these newly generated queries into a latent space using the MPNet model, thereby augmenting our initial representation. We denote this "augmented latent space" as $\tilde{\mathcal{A}}$. We then apply Agglomerative Clustering [34], i.e., a recursively clustering algorithm which starts by treating each query as its own cluster and then progressively merges the most similar clusters based on their similarity-distance in $\tilde{\mathcal{A}}$. With this

"augmented" clustering performed offline, the coarser tool groups in this **Search Level 2** capture synergistic relationships between tools prevalent in the underlying benchmarks.

Search Level 3 - Entire Tool Set: This corresponds to the default function-calling logic where the agent receives *all* APIs. We represent this directly with the complete tool set (text JSON format in LLM calls) without the need for a latent space, since no tool search is performed.

B. Tool *Recommender* and query latent space embedding

At runtime, given the user query and **without** providing any tools, the LLM (tool recommender) is prompted to generate descriptions of the "ideal" tools it believes would be necessary to complete the task. The LLM returns a structured response in a JSON format containing the functionality of each recommended tool in detail. For example, if a user prompts, "What's the weather like in New York and can you translate that information into French?" the LLM would generate definitions of a "weather_information()" and a "text_translation()" tool, explaining that one fetches weather data and the other translates text. Since no tools are executed yet (i.e., no tool APIs are appended to the input prompt), this **Recommender** step introduces negligible overhead compared to the subsequent function calling, as shown in our Results (Section IV). Using the same pretrained MPNet tokenizer, these LLM-recommended tool descriptions, alongside the corresponding user task, are mapped into 768-dimensional embeddings, denoted as $\tilde{\mathcal{E}}$, enabling efficient similarity tool-matching as described in the next step.

C. Tool *Controller*

Given the LLM-recommended query-tools embedding $\tilde{\mathcal{E}}$ and the tool space representations ($\tilde{\mathcal{T}}$ and $\tilde{\mathcal{A}}$), we efficiently identify the most relevant tools based on their proximity in the latent space. Following well-established RAG-based principles, our **Controller** runs a k-Nearest Neighbors (k-NN) search using FAISS similarity [35] against both **Search Level 1** and **Level 2**, retrieving the top-k tools – both individual tools and clusters – that closely match the LLM's tool-selection reasoning. We then compare the top-k similarity values and select the **Search Level** with the highest average score.

Intuitively, for simpler queries that require only a single tool (where the LLM would likely recommend just one "ideal" tool description), similarity scores would be higher against standalone tool descriptions. In contrast, LLM recommendations involving multiple tools are more likely to match a tool cluster. Finally, we proceed with function calling, invoking the LLM with only the subset of tools identified by the **Controller**.

We note that prior work has leveraged similarity-based tool selection [1], [27], but their search is conducted against the entire tool ontology – which closely resembles running only **Level 1** in our method. In contrast, to our knowledge, we are the first to consider varying levels of tool embedding granularity without resorting to complex tree-based search schemes, enabling function calling that balances task complexity, tool diversity, and edge hardware overhead trade-offs.

To handle LLM errors at runtime, we employ a "fallback" mechanism, as is common in LLM "compilers" [29]. In our prompt, we also instruct the LLM to signal a failure by returning an error message if the function-calling step fails after retrying. If an error occurs, the next tool-calling attempt defaults to "vanilla" function-calling with *all* APIs provided (**Level 3**). Similarly, to account for any **Recommender** mistakes, if both average top-k scores are below 0.5, indicating low confidence in selecting either Level 1 or Level 2, we default to presenting all tools (Level 3) to the LLM.

IV. Experimental Results

We evaluate our method using two benchmarks: the BFCL benchmark [9] for general function calling and GeoEngine [10] for application-specific function calling. BFCL [9] mainly involves single function calls for each query, even when the query contains unrelated sub-questions, while GeoEngine focuses on geographic applications requiring sequential function calls, where each call depends on the previous result. In contrast to GeoEngine, BFCL is simpler because it handles each sub-question independently, without needing to process information sequentially across multiple function calls. Interestingly, we found that in BFCL **Search Level 1** yields higher tool-matching scores, whereas for GeoEngine it is **Search Level 2** with better tool selection. We conducted our experiments using mini-batches of 230 queries from each benchmark, along with 51 functions from BFCL and 46 functions from GeoEngine.

We used the NVIDIA AGX Orin [11] board as our edge device. We focused on four key metrics: (i) Tool Accuracy [1]: the frequency with which the LLM selected the correct tool from the available options, (ii) Success Rate [10]: it shows whether the LLM not only chose the correct tool but also used it properly, such as providing the correct input types according to the function's requirements; (iii) Normalized Execution Time: the average time taken to complete a query, adjusted relative to the baseline, and (iv) Normalized power consumption: the average power used by the device for processing each query, adjusted relative to the baseline.

The LLMs we evaluated are: (i) Hermes2-Pro-8b [36], an advanced LLaMA variant optimized for natural language understanding and function calling; (ii) Llama3.1-8b [37], a state-of-the-art model fine-tuned for language tasks; (iii) Mistral-8b [38], known for speed and efficiency; (iv) Phi3-8b [5], specialized for specific tasks; (v) Qwen2-1.5b [39], a smaller model optimized for performance on limited resources; and (vi) Qwen2-7b, a larger version for handling more complex language tasks. For each model, we evaluated four quantization variants: `q4_0` (4-bit quantization for memory efficiency), `q4_1` (an optimized version with improved accuracy), `q4_K_M` (introducing mixed-precision for balanced performance), and `q8_0` (8-bit quantization for higher precision but increased memory use). For `Less-is-More` (LiS), we tested with $k = 3$, and $k = 5$. We compared our method to the default execution, where LLMs access all tools, and to Gorilla [1], which uses similarity-based methods to identify the most likely tool. We also attempted to compare against ToolLLM [27], but its tree-based exploration could not fit on the board.

We determined the minimum context window required for each model to fit all tools and handle communication with

Fig. 2: Performance comparison of our method (LiS) at varying k values against the default approach, measured by Success Rate, Tool Accuracy, Normalized Execution Time, and Normalized Power for the BFCL [9] benchmark.

the LLM, both with and without `Less-is-More`. For the default models used via Ollama [7], the context window was set to $16k$, while Gorilla and `Less-is-More` reduced this to $8k$ across all k values. For the default models, we also tested context windows larger than $16k$. While there was no significant improvement in success rate, execution time increased noticeably, which is why we chose the $16k$ value.

Figure 2 shows the results for the BFCL benchmark. **Hermes2-Pro-8b**: Compared to the default model, the optimized versions achieved a significant increase to approximately 71% in success rate. Gorilla was also better than the default, but lower than LiS. LiS also improved tool accuracy to 89%, reflecting a better ability to select and use tools correctly. Notably, execution time was reduced by up to 80% on the device, and power consumption decreased by 45% at best, making this model much more efficient for edge deployment. **Llama3.1-8b:** The success rate increased from the baseline to 44.2% and tool accuracy reached 93.8%. These optimizations led to a 72% reduction in execution time and a 30% decrease in power consumption. **Mistral-8b**: For Mistral-8b, even though the optimizations did not result in any gain in success rate and tool accuracy, our method resulted in a 77% reduction in execution time and a 18% decrease in power consumption. Gorilla was the worst in success rate and tool accuracy mainly due to the limited capabilities of compressed Mistral. **Phi3-8b**: Our method improved the success rate to 55% and tool accuracy to 78%. Execution time was reduced by 55%, and power consumption decreased by 20%, enhancing the model's overall efficiency. **Qwen2-1.5b**: Although Qwen2-1.5b is a smaller model, the optimizations led to a noticeable increase in success rate to approximately 40% and tool accuracy to 76%. Execution time was reduced by 48%, and power usage dropped by 20%, showing that even lightweight models can benefit greatly from our method. **Qwen2-7b**: The larger Qwen2-7b model achieved

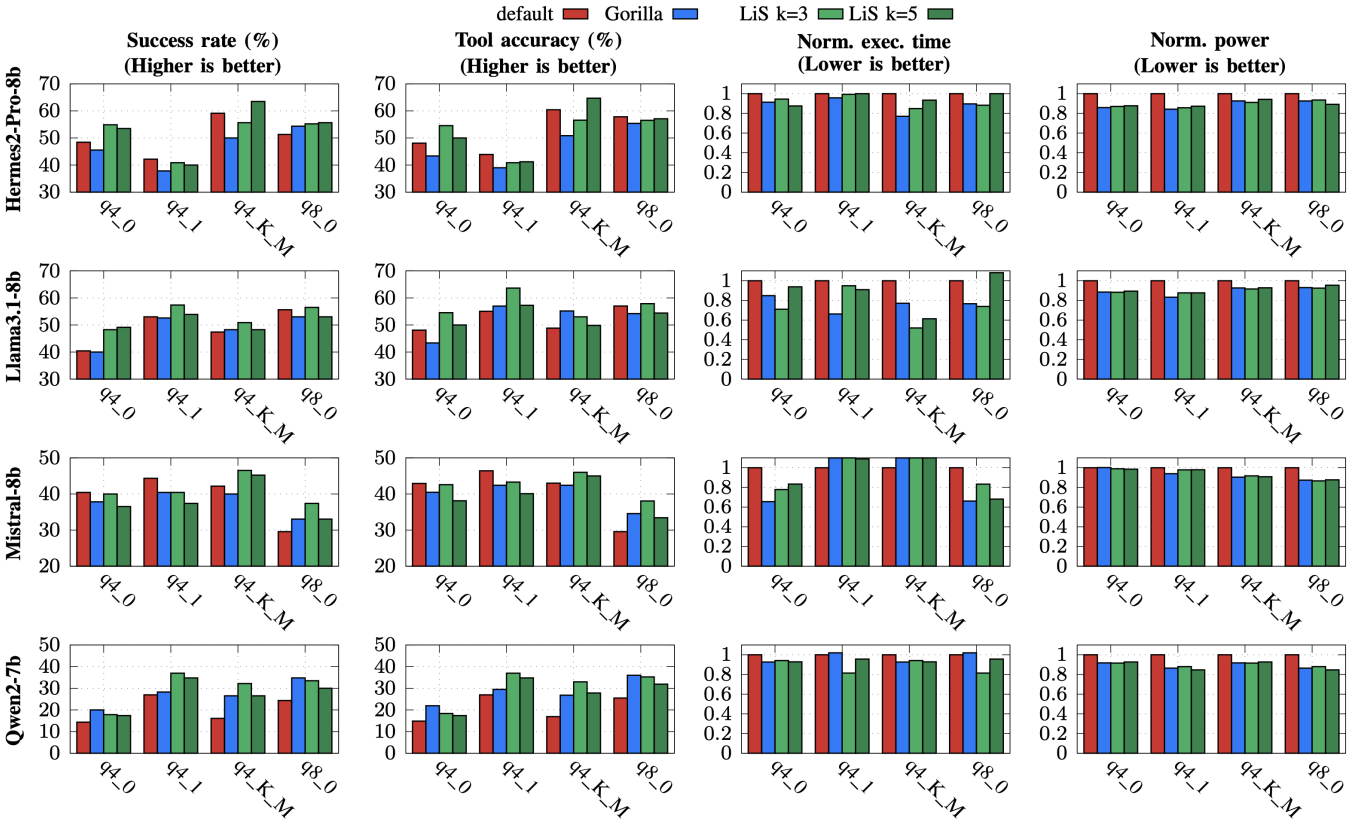

Fig. 3: Performance comparison of our method (LiS) at varying k values against the default approach, measured by Success Rate, Tool Accuracy, Normalized Execution Time, and Normalized Power for the GeoEngine [10] benchmark.

an increase in success rate to 68% and tool accuracy to 87%, remarkably better than its default performance. Execution time was reduced by up to 70%, and power consumption improved by 27%, enabling this model to handle more complex tasks efficiently on edge devices. Overall, our method improved all four metrics—success rate, tool accuracy, execution time, and power consumption—across all the LLMs. The improvements in success rate and tool accuracy were achieved by reducing the number of tools available to the LLM, which minimized confusion and allowed it to make better decisions. Additionally, by speeding up the LLM's decision-making process and using a smaller context window, we further reduced execution time and achieved significant gains in power efficiency.

Figure 3 depicts the results on the GeoEngine benchmark which is more complex than BFCL. Given this complexity, the Phi3 and Qwen2-1.5b models exhibited a low default success rate of approximately 10%, making their power consumption and execution time measurements unreliable. To maintain fairness, we have excluded these two LLMs from our analysis. **Hermes2-Pro-8b**: Our optimizations improved the success rate to 63% and enhanced tool accuracy to 64%. Additionally, we achieved a 15% reduction in execution time and a 6% decrease in power consumption. **Llama3.1-8b**: The success rate improved to 56%, similarly to the tool accuracy. Execution time was reduced by 40% at best case, and power consumption decreased by approximately 12%, significantly enhancing the model's efficiency for edge devices. **Mistral-8b**: The optimizations resulted in a success rate increase to 46%, with tool accuracy improving to 47%. Execution time was 10% higher for some variations, compensated by the big increase in success rate. Power consumption decreased by 9%, making Mistral-8b more power-efficient. **Qwen2-7b**: The success rate improved to 35%, with tool accuracy reflecting a similar trend. The optimizations resulted in a 21% reduction in execution time and a notable decrease in power consumption of about 13%. Overall, Gorilla struggled to improve the success rate in most cases as it only checks tool similarity, while GeoEngine requires sequential function calls where each depends on the previous result.

V. Conclusion

In this paper, we introduced `Less-is-More`, a novel method for optimizing function calling in Large Language Models (LLMs) deployed on edge devices. By reducing the number of available tools and using hierarchical search levels, our method improves success rate, execution speed, and power efficiency without the need for extensive fine-tuning. Experimental results demonstrate significant improvements in success rate, tool accuracy, execution time, and power consumption compared to default approaches, particularly for complex queries. Overall, our approach provides a practical solution for deploying LLMs on resource-constrained devices, opening new possibilities for efficient and scalable edge AI applications.

Acknowledgments

This work is supported by grant NSF CCF 2324854.

REFERENCES

[1] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *arXiv preprint arXiv:2305.15334*, 2023.

[2] Z. Yuan, Y. Shang, Y. Zhou, Z. Dong, C. Xue, B. Wu, Z. Li, Q. Gu, Y. J. Lee, Y. Yan *et al.*, "Llm inference unveiled: Survey and roofline model insights," *arXiv preprint arXiv:2402.16363*, 2024.

[3] OpenAI, "Function calling documentation," https://platform.openai.com/docs/guides/function-calling.

[4] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[5] M. Abdin *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," *arXiv preprint arXiv:2404.14219*, 2024.

[6] HuggingFace, "Function calling," https://huggingface.co/models?other=function+calling, 2024.

[7] Ollama, "Tool support," https://ollama.com/blog/tool-support, 2024.

[8] L. E. Erdogan, N. Lee, S. Jha, S. Kim, R. Tabrizi, S. Moon, C. Hooper, G. Anumanchipalli, K. Keutzer, and A. Gholami, "Tinyagent: Function calling at the edge," *arXiv preprint arXiv:2409.00608*, 2024.

[9] F. Yan, H. Mao, C. C.-J. Ji, T. Zhang, S. G. Patil, I. Stoica, and J. E. Gonzalez, "Berkeley function calling leaderboard," https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html, 2024.

[10] S. Singh, M. Fore, and D. Stamoulis, "Geollm-engine: A realistic environment for building geospatial copilots," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

[11] L. S. Karumbunathan, "Nvidia jetson agx orin series," 2022.

[12] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[13] X. Ma, G. Fang, and X. Wang, "Llm-pruner: On the structural pruning of large language models," *Advances in neural information processing systems*, vol. 36, pp. 21 702–21 720, 2023.

[14] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[15] Y. Gu, L. Dong, F. Wei, and M. Huang, "MiniLLM: Knowledge distillation of large language models," in *The Twelfth International Conference on Learning Representations*, 2024.

[16] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu, "Llmlingua: Compressing prompts for accelerated inference of large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 13 358–13 376.

[17] K. Alizadeh, I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," *arXiv preprint arXiv:2312.11514*, 2023.

[18] B. Kim, S. Cha *et al.*, "The breakthrough memory solutions for improved performance on llm inference," *IEEE Micro*, vol. 44, no. 3, pp. 40–48, 2024.

[19] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, Z. Zhang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi *et al.*, "Specinfer: Accelerating large language model serving with tree-based speculative inference and verification," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 932–949.

[20] B. Li, Y. Jiang, V. Gadepally, and D. Tiwari, "Llm inference serving: Survey of recent advances and opportunities," *arXiv preprint arXiv:2407.12391*, 2024.

[21] Z. Yu, Z. Wang, Y. Li, R. Gao, X. Zhou, S. R. Bommu, Y. K. Zhao, and Y. C. Lin, "Edge-llm: Enabling efficient large language model adaptation on edge devices via unified compression and adaptive layer voting," 2024.

[22] Q. J. Hu, J. Bieker, X. Li, N. Jiang, B. Keigwin, G. Ranganath, K. Keutzer, and S. K. Upadhyay, "Routerbench: A benchmark for multi-llm routing system," *arXiv preprint arXiv:2403.12031*, 2024.

[23] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," *arXiv preprint arXiv:2305.05176*, 2023.

[24] J. Y. Koh, R. Lo, L. Jang, V. Duvvur, M. C. Lim, P.-Y. Huang, G. Neubig, S. Zhou, R. Salakhutdinov, and D. Fried, "Visualwebarena: Evaluating multimodal agents on realistic visual web tasks," *ACL*, 2024.

[25] S. Singh, M. Fore, A. Karatzas, C. Lee, Y. Jian, L. Shangguan, F. Yu, I. Anagnostopoulos, and D. Stamoulis, "Llm-dcache: Improving tool-augmented llms with gpt-driven localized data caching," *arXiv preprint arXiv:2406.06799*, 2024.

[26] C. Hu, H. Huang, J. Hu, J. Xu, X. Chen, T. Xie, C. Wang, S. Wang, Y. Bao, N. Sun *et al.*, "Memserve: Context caching for disaggregated llm serving with elastic memory pool," *arXiv preprint arXiv:2406.17565*, 2024.

[27] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, dahai li, Z. Liu, and M. Sun, "ToolLLM: Facilitating large language models to master 16000+ real-world APIs," in *The Twelfth International Conference on Learning Representations*, 2024.

[28] W. Chen, Z. Li, and M. Ma, "Octopus: On-device language model for function calling of software apis," *arXiv preprint arXiv:2404.01549*, 2024.

[29] S. Kim, S. Moon, R. Tabrizi, N. Lee, M. W. Mahoney, K. Keutzer, and A. Gholami, "An llm compiler for parallel function calling," *arXiv preprint arXiv:2312.04511*, 2023.

[30] S. Singh, A. Karatzas, M. Fore, I. Anagnostopoulos, and D. Stamoulis, "An llm-tool compiler for fused parallel function calling," *arXiv preprint arXiv:2405.17438*, 2024.

[31] Anthropic, "Contextual retrieval," https://www.anthropic.com/news/contextual-retrieval, 2024.

[32] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "Mpnet: Masked and permuted pre-training for language understanding," *Advances in neural information processing systems*, vol. 33, pp. 16 857–16 867, 2020.

[33] Y. Zhuang, Y. Yu, K. Wang, H. Sun, and C. Zhang, "Toolqa: A dataset for llm question answering with external tools," *Advances in Neural Information Processing Systems*, vol. 36, pp. 50 117–50 143, 2023.

[34] scikit learn, "Agglomerative clustering," https://scikit-learn.org/stable/api/index.html, 2024.

[35] M. Douze, A. Guzhva *et al.*, "The faiss library," 2024.

[36] NousResearch, "Hermes-2-pro-llama-3-8b," https://huggingface.co/NousResearch/Hermes-2-Pro-Llama-3-8B.

[37] A. Dubey *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[38] A. Jiang *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[39] A. Yang *et al.*, "Qwen2 technical report," *arXiv preprint arXiv:2407.10671*, 2024.