Efficiently Programming Large Language Models using SGLang

Lianmin Zheng[2,*] Liangsheng Yin[3] Zhiqiang Xie[1] Jeff Huang[4] Chuyue Sun[1] Cody Hao Yu[5] Shiyi Cao[2]
Christos Kozyrakis[1] Ion Stoica[2] Joseph E. Gonzalez[2] Clark Barrett[1] Ying Sheng[1,*]

[1]*Stanford University* [2]*UC Berkeley* [3]*Shanghai Jiao Tong University*
[4]*Texas A&M University* [5]*Independent Researcher*

Abstract

Large language models (LLMs) are increasingly used for complex tasks requiring multiple chained generation calls, advanced prompting techniques, control flow, and interaction with external environments. However, efficient systems for programming and executing these applications are lacking. To bridge this gap, we introduce SGLang, a Structured Generation Language for LLMs. SGLang is designed for the efficient programming of LLMs and incorporates primitives for common LLM programming patterns. We have implemented SGLang as a domain-specific language embedded in Python, and we developed an interpreter, a compiler, and a high-performance runtime for SGLang. These components work together to enable optimizations such as parallelism, batching, caching, sharing, and other compilation techniques. Additionally, we propose RadixAttention, a novel technique that maintains a Least Recently Used (LRU) cache of the Key-Value (KV) cache for all requests in a radix tree, enabling automatic KV cache reuse across multiple generation calls at runtime. SGLang simplifies the writing of LLM programs and boosts execution efficiency. Our experiments demonstrate that SGLang can speed up common LLM tasks by up to 5×, while reducing code complexity and enhancing control.

1 Introduction

Recent increases in the capabilities of LLMs have broadened their utility, enabling them to tackle a wider range of general tasks and act as autonomous agents [6, 40, 42, 61]. In such applications, LLMs engage in *multi-round* planning, reasoning, and interaction with external environments. This is accomplished through tool usage [49], multiple input modalities [1], and a wide-range of prompting techniques [31], like few-shot learning [5], self-consistency [63], skeleton-of-thought [37], and tree-of-thought [70]. All of these new use cases require multiple, often dependent, LLM generation calls, and critically, the control flow of these programs may depend on the output of LLMs [71]. *The emergence of these complex patterns signifies a shift in our interaction with LLMs, moving from simple turn-based chatting to a more sophisticated form of programmatic usage of LLMs (see Section 2.2).* Figure 1 shows an example of such an LLM program, where multiple generation calls are used to accomplish a task. The example uses SGLang, a new domain-specific language we introduce in this paper and explain in Section 4.

```
dimensions = ["Clarity", "Originality", "Evidence"]

@function
def essay_judge(s, essay):
  s += "Please evaluate the following essay. " + essay
  # Evaluate an essay from multiple dimensions in parallel
  forks = s.fork(len(dimensions)) ①
  for f, dim in zip(forks, dimensions):
    f += (
      "Evaluate based on the following metric: " +
      dim + ". End your judgement with the word 'END'")
    f += "Judgment: " + f.gen("judgment", stop="END") ②
  # Merge judgments
  for f, dim in zip(forks, dimensions):
    s += dim + ": " + f["judgment"] ③
  # Generate a summary and give a score
  s += "In summary," + s.gen("summary")
  s += "I give the essay a letter grade of " +
  s += s.gen("grade", choices=["A", "B", "C", "D"]) ④

ret = essay_judge.run(essay="A long essay ...") ⑤
print(ret["grade"])
```

Figure 1: The implementation of a multi-dimensional essay judge in SGLang utilizes the branch-solve-merge prompting technique [48]. This function uses LLMs to evaluate the quality of an essay from multiple dimensions, merges the judgments, generates a summary, and assigns a final grade. The highlighted regions illustrate the use of SGLang APIs. (1) `fork` creates multiple parallel copies of a prompt. (2) `gen` invokes an LLM generation and stores the result in a variable. (3) `[variable_name]` retrieves the result of the generation. (4) `choices` imposes constraints on the generation. (5) `run` executes a SGLang function with its arguments.

*Equal contribution.

To align with emerging trends, the community has seen the development and rising popularity of new interfaces such as LangChain [24], LMQL [4], Guidance [16], and DSPy [21]. These tools help users manage prompt templates and program pipelines with ease. However, their primary focus on front-end design often results in compromised runtime performance.

Conversely, while back-end inference engines like NVIDIA TensorRT-LLM [38], Hugging Face TGI [19], Orca [74], and vLLM [23] continue to reduce latency and improve throughput, their design largely remains anchored in traditional single-generation call interfaces. They typically support APIs similar to the OpenAI Completion API [39], where a user prompt is processed and a response is returned in a stateless manner.

While both front-end and back-end technologies are advancing rapidly, they are evolving along separate trajectories. The front end largely remains oblivious to the LLM inference processes, while the back end often lacks awareness of the application's structure and is unable to optimize across multiple calls. This disjointed development leads to numerous missed opportunities for performance optimization, such as KV cache sharing and parallelism, as discussed in Section 2.3.

In this paper, we present a holistic approach for an efficient LLM programming system by co-designing both the front-end language and the back-end runtime. The process involves three key challenges: (i) creating a flexible, domain-specific language with primitives suitable for LLM programming and ecosystem integration; (ii) ensuring efficient program execution via automatic parallelism, batching, caching, and sharing across multiple calls and multiple programs; and (iii) enhancing program performance through compiler optimizations like code movement and prefetching annotations.

On the front end, we introduce SGLang, a domain-specific language embedded in Python. It provides a set of primitives specifically designed to facilitate programming LLMs. These primitives enable the manipulation of prompts and generations (e.g., `extend`, `gen`, `select`) and the control of parallelism (e.g., `fork`, `join`). Furthermore, SGLang is compatible with Python's control flow and libraries. Leveraging these primitives along with native Python syntax, users can intuitively develop sophisticated prompting strategies and agents, while obviating the need for manual optimization. Moreover, the unified language enhances portability, enabling users to transition easily between different open-source and proprietary models, such as those from OpenAI and Anthropic.

We develop both an interpreter and a compiler for SGLang. The interpreter is responsible for managing the prompt state and executing SGLang programs. Specifically, it treats a prompt as a stream and submits primitive operations to this stream for asynchronous execution, ensuring proper control over synchronization and intra-program parallelism. Additionally, we can trace a SGLang program and utilize the compiler to compile it into a computational graph. This approach opens opportunities for more aggressive optimizations. We explore several classical compiler optimizations, e.g., code movement, in the context of LLM programming.

On the back end, we propose a novel technique called RadixAttention to automatically reuse the KV cache across multiple generation calls. In existing systems, the KV cache of a request is discarded after processing is completed, which prevents KV cache from being reused across multiple generation calls. Instead, our system maintains an LRU cache of the KV cache for all requests within a radix tree. This approach allows the runtime to efficiently achieve automatic sharing for various reuse patterns. Additionally, to increase the cache hit rate, we utilize a cache-aware scheduling policy.

In summary, we make the following contributions:

- We present SGLang, a domain-specific language with primitives for LLM programming, supporting parallelism, control flow, nested calls, and external calls.
- We develop an interpreter capable of executing SGLang programs in tandem with the Python interpreter and managing prompt states as asynchronous streams.
- We build a compiler capable of compiling programs into computational graphs and explore classical compiler techniques in the context of LLM programming.
- We propose RadixAttention, a novel technique for automatic KV cache reuse across multiple generation calls at runtime.
- Using SGLang, we implement various LLM applications, including agent control, logical reasoning, content generation, benchmarking, and long document processing. Our experiments demonstrate that SGLang can accelerate common LLM tasks by up to 5×, while reducing code complexity and enhancing control.

2 Background

2.1 The Inference Process of LLMs

Most LLMs in use today, such as GPT-3 [5], PaLM [8], and LLaMA [58], are based on the autoregressive Transformer architecture [60]. The models predict the probability of the next token in a sequence based on the preceding tokens. During inference, the model first processes a sequence of input tokens through a forward pass. It then sequentially decodes output tokens, with each token depending on prior tokens. We refer to the process of taking a sequence of input tokens and generating a sequence of output tokens as a single generation call. Throughout this process, each token generates some intermediate tensors, which are used for decoding further tokens. These intermediate tensors, known as the "KV Cache", are named for the key-value pairs in the self-attention mechanism. An observation that will be important when discussing optimizations later is that the computation of the KV cache only depends on all previous tokens, so different sequences with the same prefix can reuse the KV cache of the prefix tokens and avoid redundant computation.

2.2 The Paradigm of Programming LLMs

As LLMs become more powerful, they are increasingly employed as versatile task solvers in complex systems. In these contexts, it is imperative that LLMs operate as dependable and intelligent components. This evolves the use of LLMs from single, straightforward generation calls to multiple, interdependent generation calls interleaved with control flow, leading to the emergence of the "Programming LLMs" paradigm. Programming LLMs refers to the development of computer programs that control the generation processes of LLMs. This concept can be exemplified in several ways.

First, advanced prompting techniques are examples of programming LLMs. An exemplary case is meta-reasoning [73]. To solve a problem, the LLM initially generates multiple solutions through chain-of-thought prompting. It then performs meta-reasoning across these solutions to derive the final resolution. Another example is skeleton-of-thought [37]. To answer a question, the LLM first generates a skeleton of the answer. It then issues multiple parallel calls to elaborate the skeleton into a detailed explanation. A more complex case is the tree-of-thought approach [70], which integrates LLM calls within tree search algorithms (e.g., breadth-first search).

Second, in LLM-based agents, the programmatic use of LLMs is crucial. Take the ReAct agent [71], for example. The behavior of the agent is controlled by loops and conditional statements. The agent performs one action per iteration. In each iteration, the agent chooses the action type (search or finish), generates action parameters, and processes observations from the external environment.

Third, simpler prompting forms also fall under the scope of programming LLMs. Techniques like few-shot prompting [5], chain-of-thought prompting [64], and retrieval-augmented generation [26] are basic elements in programming LLMs.

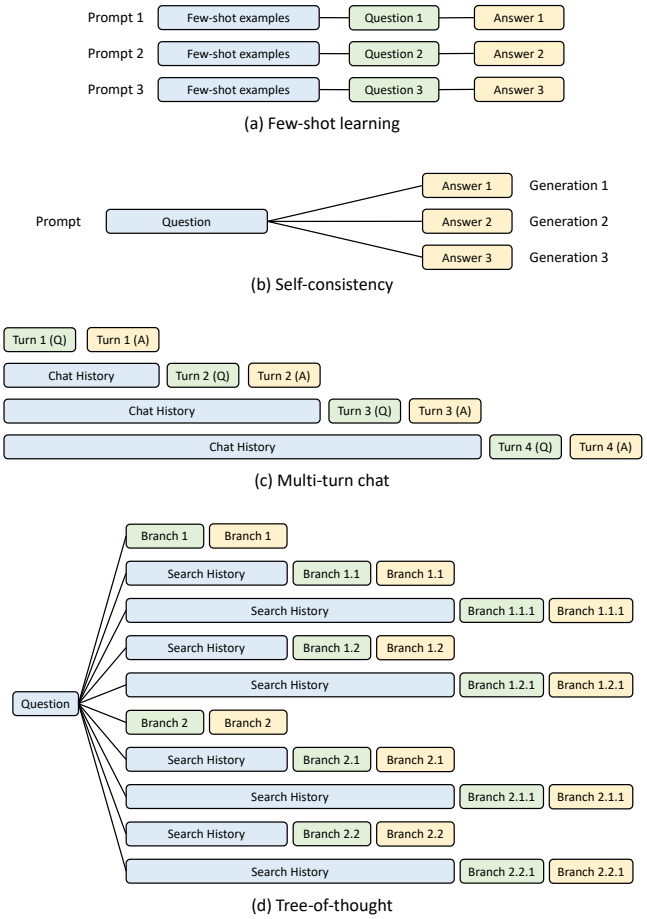

Figure 2: KV cache sharing examples. Blue boxes represent shareable prompt parts, green boxes indicate non-shareable parts, and yellow boxes mark non-shareable model outputs. Shareable elements include few-shot learning examples, questions in self-consistency [63], chat history in multi-turn chats, and search history in tree-of-thought [70].

2.3 Challenges and Opportunities

Designing programming systems for LLMs presents unique challenges and opportunities. A good system should help a programmer be productive, offering an intuitive and expressive interface for common programming tasks, including LLM state management, decoding process control, and smooth integration with external tools. Furthermore, it is essential to ensure performance by optimizing for low latency and high throughput execution. In particular, the rich structure of LLM programs and the unique properties of LLM inference provide plenty of opportunities for runtime optimizations. These include:

- **Caching:** Often in LLM programs, multiple text segments and generation calls are appended to a single prompt. Caching the computed KV cache for previous tokens across multiple chained calls can reduce redundant computation. This optimization, however, is neither free nor trivial, as it requires additional storage and more complex memory management.
- **Batching:** The decoding process in LLMs is primarily memory-bound, meaning increasing the batch size can significantly enhance throughput. Running a large number of LLM programs in a batch, and utilizing the contiguous batching technique [74] can be beneficial.
- **Sharing:** It is common in LLM programs to generate multiple outputs from a single prompt or to fork a new prompt from the current state. Basic prefix sharing has been investigated in vLLM [23]. More advanced sharing patterns like irregular tree-structured sharing can also be employed. Figure 2 shows four typical patterns of KV cache sharing across multiple calls; none of the existing systems can automatically handle all of them.
- **Parallelism:** An LLM program often includes multiple generation calls. By creating a dependency graph for these calls, we can identify independent calls and execute them in parallel. This creates intra-program parallelism, which adds additional parallelism to the straightforward

Language	Syntax	Primitives	Runtime Backends	Execution Modes
LMQL	Custom	extend, gen, select	Hugging Face Transformers, llama.cpp, OpenAI	Compiler
Guidance	Python	extend, gen, select	Hugging Face Transformers, llama.cpp, OpenAI, VertexAI	Interpreter
SGLang	Python	extend, gen, select, fork, join	SGVM, OpenAI, Anthropic, Hugging Face TGI*, llama.cpp*	Interpreter and compiler

Table 1: Comparison among LMQL, Guidance, and SGLang. The backends marked with * are experimental.

inter-program parallelism. For example, in Figure 1, the generation of multiple judgments can be parallelized.

- **Compilation:** Once a full program is available, it can be compiled into an optimized intermediate representation for more efficient execution. More aggressive optimizations can also be implemented, such as automatically adjusting the original prompts based on user-provided test cases. These optimizations may include expanding, compressing, rearranging parts of the prompts, and selecting different models for different calls.

To systematically explore these opportunities, it is essential to co-design the programming interface and runtime. However, inference engines like Orca and vLLM fall short, as they optimize only for single generation calls. This limitation leads to redundant computations and missed opportunities when executing LLM programs that involve multiple calls.

2.4 Existing LLM Programming Systems

Given the significance of LLM programming, specialized programming systems have been developed. The top layer comprises high-level application development libraries with pre-built modules, such as LangChain [24]. The middle layer features more flexible pipeline programming frameworks like DSPy [21]. At the bottom layer, there are domain-specific languages designed for controlling a single prompt, including LMQL [4] and Guidance [16].

SGLang, uniquely positioned between the middle and bottom layers, shares similarities with LMQL and Guidance. However, unlike LMQL and Guidance, which primarily focus on language design while often overlooking runtime efficiency, SGLang thoughtfully integrates front-end language features with back-end runtime optimizations. This approach significantly enhances runtime efficiency, sometimes by an order of magnitude or more. For its front-end, LMQL employs an SQL-like custom syntax, contrasting with the more intuitive Python syntax. Guidance initially used the Handlebars template language and later transitioned to a Pythonic language embedded within Python. The front-end language of SGLang, inspired by Guidance, is being developed concurrently with the new version of Guidance.

SGLang introduces new primitives for enhanced expressiveness and runtime efficiency. It also offers the flexibility of both interpreter and compiler modes. A comparison among LMQL, Guidance, and SGLang is presented in Table 1. We omit the comparison with LangChain and DSPy because they are at a higher level. It is possible to compile high-level pipelines from LangChain and DSPy to SGLang.

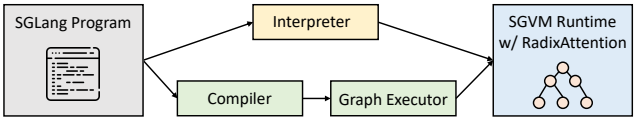

Figure 3: The overview of the system.

3 Overview

Figure 3 shows an overview of our system. At the user level, SGLang programs are written in Python with embedded SGLang primitives for managing prompts and generation calls. The syntax and primitives of SGLang will be introduced in Section 4.1. SGLang programs can be executed in two ways: either using an interpreter or a compiler. When using the interpreter, SGLang programs, which are also valid Python programs, are processed by both the Python and SGLang interpreters. The SGLang interpreter handles specific SGLang primitives. The design of this interpreter is explained in Section 4.2. Alternatively, a limited set of SGLang programs can be compiled, enabling further optimizations. Section 4.3 is dedicated to the SGLang compiler. Both the interpreter and the compiler work with the SGVM, which is our custom serving engine for LLM generation calls (Section 5). A key aspect of SGVM is its RadixAttention feature, which automatically reuses KV caches across multiple calls.

4 Language, Interpreter, and Compiler

4.1 Syntax and Primitives

SGLang is implemented as an embedded domain-specific language in Python.

Figure 1 shows an example of a SGLang program. This program evaluates the quality of an essay from multiple dimensions using the branch-solve-merge prompting method [48]. The function `essay_judge` takes two arguments: `s` and `essay`. The variable `s` manages the state of a prompt, allowing new strings and SGLang primitives to be appended using `+=`. In the function, it first adds the essay to the prompt, and then forks the prompt into three copies. These forks run in parallel, generating judgments from different dimensions. Finally, it merges these judgments, generates a summary, and assigns a letter grade to the essay.

Figure 4 presents another example. This program generates two tips in parallel for a given topic using the skeleton-of-thought prompting method [37]. It employs two functions: `expand`, which elaborates a short tip into a detailed paragraph, and `tip_suggestion`, which orchestrates the entire process.

```python
@function
def expand(s, tip):
    s += (
        "Please expand the following tip into a "
        "detailed paragraph: " + tip + "\n"
    )
    s += s.gen("paragraph")

@function
def tip_suggestion(s, topic):
    s += "Here are 2 concise tips for " + topic + ".\n"

    # Generate a skeleton of two short tips
    s += "1." + s.gen("tip_1", stop=["\n", ":", "."]) + "\n"
    s += "2." + s.gen("tip_2", stop=["\n", ":", "."]) + "\n"

    # Expand the tips into detailed paragraphs in parallel
    detailed_tip1 = expand(tip=s["tip_1"])
    detailed_tip2 = expand(tip=s["tip_2"])

    # Merge the results and generate a summary
    s += "Tip 1: " + detailed_tip1["paragraph"] + "\n"
    s += "Tip 2: " + detailed_tip2["paragraph"] + "\n"
    s += "In summary" + s.gen("summary")

state = tip_suggestion.run(topic="staying healthy")
print(state.text())
```

Figure 4: The SGLang program for parallel tip suggestion with skeleton-of-thought prompting.

It first generates two short tips and then calls `expand` to elaborate them in parallel. Finally, it generates a summary.

These examples illustrate the use of SGLang. In SGLang, we provide the following primitives: `gen` for calling LLM generation, `select` for letting the LLM choose the option with the highest probability from a list, `+=` or `extend` for extending the current prompt, `fork` for forking the current prompt state, and `join` for rejoining the forked prompt states. Users can interleave these primitives with arbitrary Python control flow and libraries.

4.2 Interpreter

The most basic way to execute a SGLang program is through an interpreter. In SGLang, a prompt is treated as an asynchronous stream. Executing primitives such as extend, gen, and select on a prompt leads to the interpreter submitting these primitives into the stream for asynchronous execution. These submission calls are non-blocking, allowing the Python interpreter to continue executing Python code without waiting for the LLM generation to finish. This process is akin to launching CUDA kernels to CUDA streams asynchronously. Each prompt is managed by a stream executor in a background thread, enabling users to achieve parallelism.

When fetching the generation results from a prompt, the fetch action will be blocked until the desired generation results are ready. This is implemented using event mechanisms, where a generation call sets an event upon producing its output. Thus, the interpreter effectively manages intra-program parallelism and synchronization.

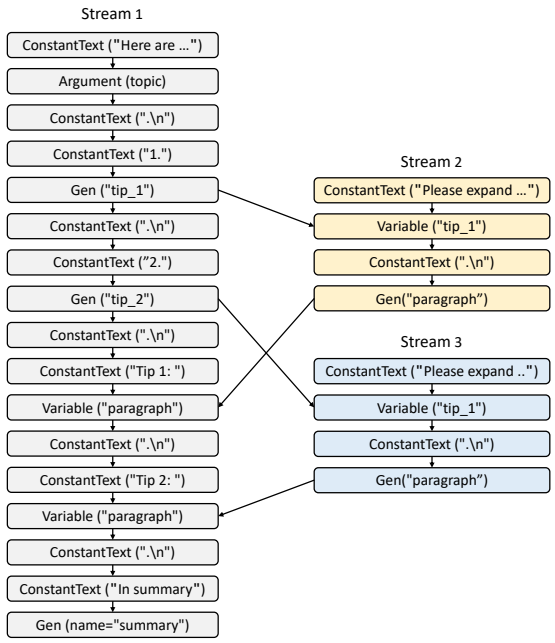

Figure 5: A computational graph for the program in Figure 4. The three streams correspond to three function calls.

4.3 Compiler

Another way to run SGLang programs is to compile them as computational graphs and run them with a graph executor. This opens the opportunity for more compilation optimizations, as we can rewrite the graph.

4.3.1 Tracing and Graph Executor

We design an intermediate representation (IR) for SGLang, representing SGLang program structures and operations as a computational graph. This graph includes nodes for primitive operators and edges for dependencies. See Figure 5 for the graph corresponding to the program in Figure 4. In the program, each call to a decorated function and or fork will create a new prompt state, or a stream.

There are several types of nodes. Each operand of `+=` and `+` in a SGLang program is represented by an IR node. These include `ConstantText`, `Argument`, `Gen`, `Select`, `Variable`, `Fork`, `GetForkItem`, and `Join`. There are two types of dependencies. The first is intra-stream dependency, where operations submitted into a stream using `+=` must be executed after all preceding operations in that stream. The second is inter-stream dependency, which occurs when one stream needs to fetch variable values from another stream, necessitating synchronization. Operations like fork manipulate multiple streams and thus introduce inter-stream dependencies.

To generate the graph, we use tracing to run the program with abstract arguments and construct the graph dynamically. This method is limited to programs without data-dependent control flow, a limitation we plan to address in future work. Once constructed, the graph can be executed

through a graph executor, eliminating the need to reinterpret the original Python program. This results in benefits like graph rewriting optimizations, reduced runtime overhead, and program serialization. For execution, stream executors are launched for each data stream, dispatching IR nodes to the streams in topological order.

4.3.2 Compilation Optimizations

We explore two compilation optimizations for SGLang IR. The first one is noteworthy as it integrates GPT-4 into compiler optimization processes. The second one is more conventional. We anticipate more classical compiler techniques can also be applied, such as auto-tuning and instruction selection.

Code Movement for Improving Prefix Sharing. This optimization aims to improve prefix sharing by reordering nodes in the graph to increase the length of the constant prefix. It does not strictly preserve the original computation, classifying it as an aggressive optimization. For instance, changing the prompt from "Here is a question + {question}. Please act as a math expert and solve the given question" to "Please act as a math expert and solve the given question. Here is a question + {question}." results in a longer sharable prefix. This optimization is interesting because traditional program analysis cannot achieve it, due to the presence of natural language instructions in SGLang. Instead, we prompt GPT-4 to reorder graph nodes. We write a prompt with several examples to teach GPT-4 the concepts of SGLang IR, and we find that GPT-4 can successfully apply this optimization for some simple SGLang programs.

Prefetching Annotations. This involves automatically adding prefetching nodes to the graph, signaling the runtime to move the cached KV cache from the CPU to the GPU if the cache exists for a long prompt. This can reduce the latency for pipelines with long prompts.

5 Runtime with RadixAttention

The primary challenge in optimizing the execution of SGLang programs lies in reusing the KV cache across multiple calls and programs, as discussed in Section 2.3. While some existing systems can handle KV cache reuse in certain scenarios, this often requires manual configurations and ad-hoc adjustments. However, even with manual configurations, no existing system can automatically handle all scenarios due to the variety of possible reuse patterns.

This section introduces RadixAttention, a novel technique for automatic KV cache reuse during runtime. Instead of discarding the KV cache after finishing a generation request, our approach retains the KV cache for both prompts and generation results in a radix tree. This structure enables efficient prefix search, reuse, insertion, and eviction. We implement a Least Recently Used (LRU) eviction policy, complemented by a cache-aware scheduling policy, to enhance the cache hit rate. Furthermore, RadixAttention is compatible with existing techniques like continuous batching and paged attention.

5.1 RadixAttention

A radix tree is a data structure that serves as a space-efficient alternative to a trie (prefix tree). Unlike typical trees, the edges of a radix tree can be labeled with not just single elements, but also with sequences of elements of varying lengths. This feature significantly boosts the efficiency of radix trees. In our system, we utilize a radix tree to manage a mapping. This mapping is between sequences of tokens, which act as the keys, and their corresponding KV cache tensors, which serve as the values. These KV cache tensors are stored in a non-contiguous, paged layout, where the size of each page is equivalent to one token.

Considering the limited capacity of GPU memory, we cannot retrain infinite KV cache tensors, which necessitates an eviction policy. To tackle this, we implement an LRU eviction policy that recursively evicts leaf nodes. In the continuous batching setting, we cannot evict nodes that are being used by the currently running batch. Therefore, each node maintains a reference counter indicating how many running requests are using it. A node is evictable if its reference counter is zero.

The front end always sends full prompts to the runtime and the runtime will automatically do prefix matching, reuse, and caching. Figure 6 illustrates how the radix tree is maintained when processing several incoming requests. The tree structure is stored on the CPU and the maintenance overhead is small.

5.2 Cache-Aware Scheduling

To increase the cache hit rate, the scheduler must be cache-aware. Otherwise, when a large batch of requests arrives, the scheduler may switch between different requests, leading to cache thrashing. Algorithm 1 presents the algorithm for cache-aware scheduling in RadixAttention with contiguous batching. The key idea is to sort the requests by matched prefix length, as opposed to adhering to a first-come-first-serve schedule. There are further opportunities for the implementation of more advanced scheduling strategies that assign priority based on the weights of subtrees. Essentially, this would involve counting the total number of requests within each sub-tree(prefix-sharing trees) and even considering requests that have been historically evicted. The weighted priority can fully utilize the tree-shape locality of some LLM tasks such as tree-of-thought [70], making the least redundant evictions.

5.3 Other Optimizations

The development of new CUDA kernels is essential to enhance the efficiency of KV cache reuse. Existing systems mostly implement CUDA kernels for two main functions: prefill and decoding. The prefill kernel assumes that the lengths

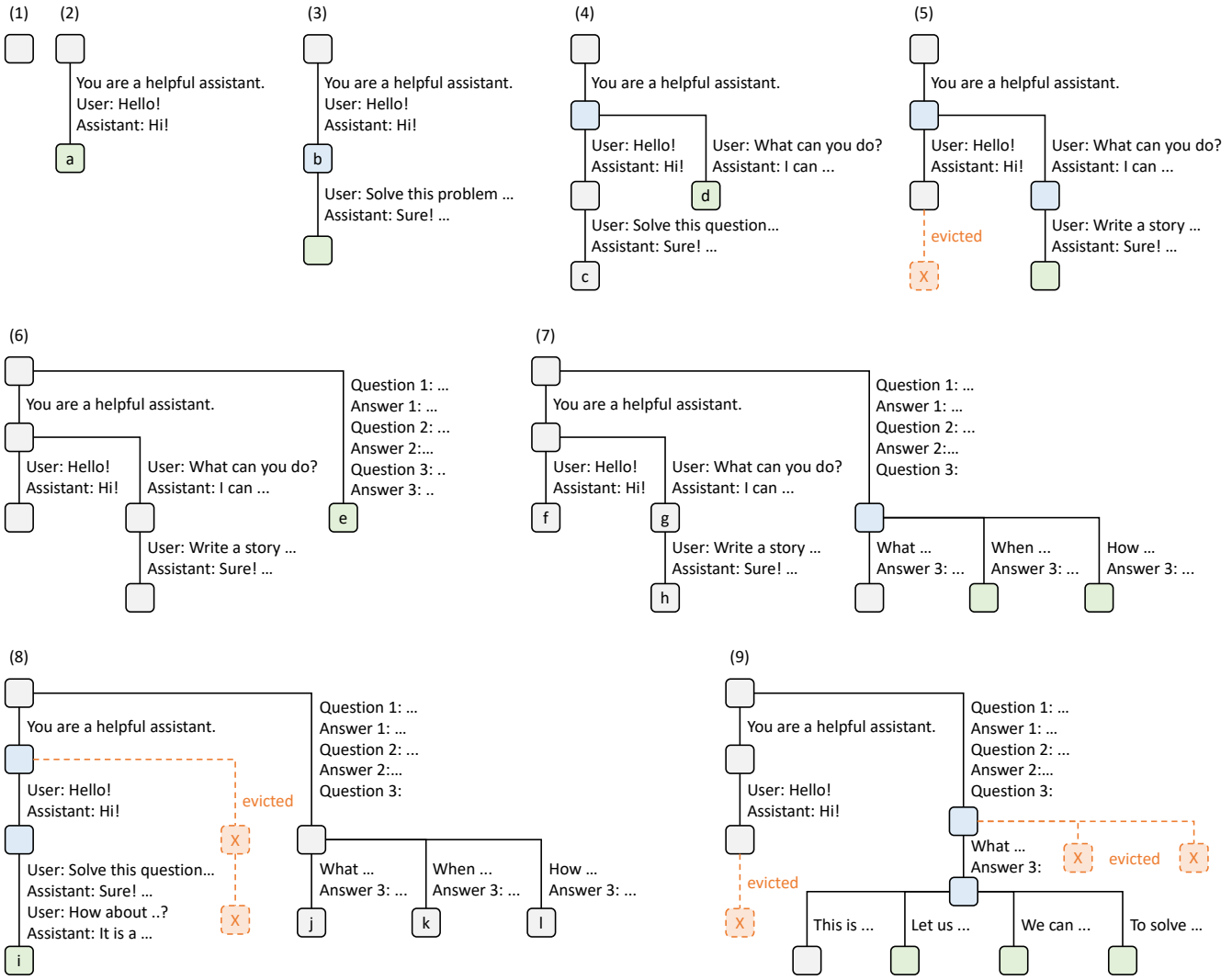

Figure 6: Examples of RadixAttention operations with an LRU eviction policy, illustrated across nine time points. The figure demonstrates the dynamic evolution of the radix tree in response to various requests. These requests include two chat sessions, a batch of few-shot learning inquiries, and a self-consistency sampling. Each tree edge carries a label denoting a substring or a sequence of tokens. The nodes are color-coded to reflect different states: green for newly added nodes, blue for cached nodes accessed during the time point, and red for nodes that have been evicted. In step (1), the radix tree is initially empty. In step (2), the server processes an incoming user message "Hello" and responds with the LLM output "Hi". The system prompt "You are a helpful assistant", the user message "Hello!", and the LLM reply "Hi!" are consolidated into the tree as a single edge linked to a new node. In step (3), a new prompt arrives and the server finds the prefix of the prompt (i.e., the first turn of the conversation) in the radix tree and reuses its KV cache. The new turn is appended to the tree as a new node. In step (4), a new chat session begins. The node “b” from (3) is split into two nodes to allow the two chat sessions to share the system prompt. In step (5), the second chat session continues. However, due to the memory limit, node "c" from (4) must be evicted. The new turn is appended after node "d" in (4). In step (6), the server receives a few-shot learning query, processes it, and inserts it into the tree. The root node is split because the new query does not share any prefix with existing nodes. In step (7), the server receives a batch of additional few-shot learning queries. These queries share the same set of few-shot examples, so we split node ’e’ from (6) to enable sharing. In step (8), the server receives a new message from the first chat session. It evicts all nodes from the second chat session (node "g" and "h") as they are least recently used. In step (9), the server receives a request to sample more answers for the questions in node "j" from (8), likely for self-consistency prompting. To make space for these requests, we evict node "i", "k", and "l" in (8)."

Algorithm 1 Cache-Aware Scheduling for RadixAttention with Continuous Batching.

Input: The radix tree T, the memory pool P, the current running batch B, the waiting queue Q.
Output: Finished requests and updated system state.

```
// Get all requests from the waiting queue
requests ← Q.get_all_requests()
// Search for the prefix matching for all waiting requests
for req ∈ requests do
    req.prefix_node, req.prefix_len ←
            T.match_prefix(req.input_tokens)
// Sort the requests according to the matched prefix lenghts
requests.sort()
// Select requests for the next batch
available_size ← T.evictable_size() + P.available_size()
current_size ← 0
new_batch ← []
for req ∈ requests do
    if req.size() + current_size < available_size then
        new_batch.append(req)
        delta ← T.increase_ref_counter(req.prefix_node)
        available_size ← available_size + delta
Q.remove_requests(new_batch)
// Insert requests into the current running batch
B.merge(new_batch)
// Allocate new memory and do eviction if necessary
needed_size ← B.needed_size()
success, buffer ← P.alloc(needed_size)
if not success then
    T.evict(needed_size)
    success, buffer ← P.alloc(needed_size)
B.run(buffer)
// Process finished requests
finished_requests ← B.drop_finished_requests()
for req ∈ finished_requests do
    T.decrease_ref_counter(req.prefix_node)
    T.insert(req)
return finished_requests
```

of the query and key are identical. Conversely, the decode kernel assumes that the query length is one. However, in the case of KV cache reuse, these kernels cannot be directly applied. We introduce a new "extend" kernel for this computation, capable of reading the KV cache from both contiguous buffers and non-contiguous memory pools. This fused extend kernel operates without extra copy overhead.

6 Evaluation

SGLang and SGVM are implemented in approximately 9K lines of Python code. SGVM reuses some components from LightLLM [35], an LLM serving system based on PyTorch [43] and Triton [57].

We evaluate the performance of SGLang across a diverse range of LLM workloads. Subsequently, we conduct case studies and ablation studies to demonstrate the effectiveness of specific components.

6.1 Setup

Models. We test the Llama 2 chat models [58], CodeLlama models [47], and vision language model LLaVA [29] models of various sizes. The number of parameters ranges from 7B to 33B. We use float16 precision for Llama and CodeLlama models, and 4-bit quantized precision for LLaVA models.
Hardware. Most experiments are conducted on AWS EC2 G5 instances unless otherwise stated. The GPUs on G5 instances are NVIDIA A10G GPUs, each with 24 GB of memory. The CPUs are AMD EPYC processors. For most experiments, we run 7B models with a single A10G GPU. To see how our system handles larger models, we also test a 33B model on an A100 GPU (80GB).
Baseline. We compare SGLang against both high-level programming systems with their respective languages and default runtimes, as well as low-level inference engines with standard OpenAI-like Completion APIs. The baselines include:

- Guidance [16], a language for controlling LLMs. We use Guidance v0.1.4 with the llama.cpp backend.
- LMQL [4], a query language for LLMs. We use LMQL v0.7.3 with the Hugging Face Transformers backend.
- vLLM [23], a high-throughput inference engine. We use vLLM v0.2.2 and its default API server.

Metrics. We report two performance metrics: throughput and latency. For throughput, we run a sufficiently large batch of program instances to compute the maximum throughput, comparing the number of program instances executed per second (programs per second, p/s). For latency, we execute a single program at a time without batching and report the average latency for multiple instances. Unless otherwise stated, most of our optimizations do not change the computation results.

While our primary focus is on runtime efficiency, we also conduct several case studies to compare the number of lines of code for productivity.

6.2 End-to-End Benchmark

6.2.1 Few-Shot In-Context Learning

We start with the most basic few-shot learning prompting. We evaluate the maximum throughput of systems during batch processing of few-shot learning requests using three benchmarks, each with 5 shots. Results are shown in Figure 7.

MMLU [18] is a multi-choice knowledge benchmark where the LLM outputs a single token (A/B/C/D) for answer selection. LMQL exhibits the worst performance due to its slow backend and complex pre- and post-processing. Guidance,

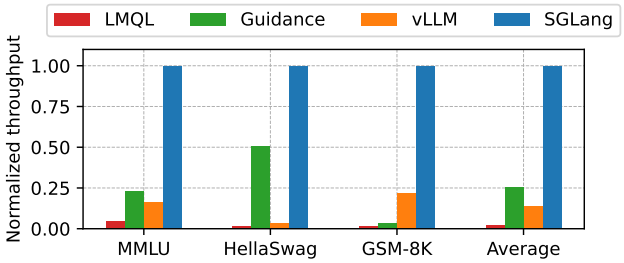

Figure 7: Throughput on few-shot in-context learning tasks.

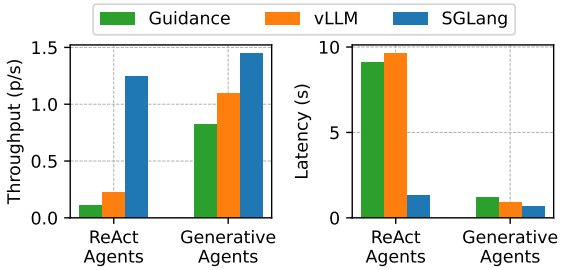

Figure 8: Throughput and latency on agent tasks.

being faster than LMQL, benefits from reduced processing overhead and some KV cache reuse. However, both systems support only a batch size of one. vLLM performs similarly to Guidance, but it lacks automatic KV cache reuse for the shared prefix. Although the vLLM paper discusses this feature, the released code does not support it due to the absence of a reliable high-performance kernel. Even if a high-performance kernel is implemented for vLLM, manual configuration is required to use this feature. Conversely, SGLang automatically achieves sharing with RadixAttention and attains a throughput that is 4.4× higher than its closest competitor.

HellaSwag [76] is a multi-choice benchmark where each option is a short string. Unlike MMLU, which decodes a single output token, HellaSwag requires an LLM to calculate the probability of each string choice by performing a forward pass. LMQL exhibits slowness due to high overhead. vLLM lacks a "select" primitive. Instead, it concatenates each choice with the prompt and computes the probability for the entire concatenated prompt for all choices, leading to redundant calculations. Guidance, although offering a "select" feature, is limited to a batch size of 1. SGLang outperforms others by 2× with automatic KV cache sharing and batching.

GSM-8K [9] is a benchmark that requires LLMs to generate a reasoning process and produce an integer answer. Since the benchmark demands longer output lengths, batching is crucial for enhanced throughput. However, LMQL and guidance currently lack support for batching, leading to reduced performance. vLLM is also slow due to its inability to reuse the KV cache for the shared prefix. In contrast, SGLang achieves a 4.5× increase in throughput, thanks to the integration of RadixAttention and contiguous batching.

Due to its overhead, LMQL often performs significantly worse than other systems in both short and long-generation tasks, so we will not include it in our future benchmarks.

6.2.2 LLM-Based Agents

We benchmark on two highly regarded agent frameworks: ReAct agent [71], which processes tasks in an interleaved manner by chaining multiple LLM calls with actions like web search; and generative agents [42], employed in a gaming environment where agents interact with each other and make decisions by invoking LLMs. Both original papers utilized OpenAI models. To accurately gauge potential performance gains, we mirrored the settings described in the papers and used GPT-3.5-turbo as the backbone model, and then traced the input and output of LLM calls to create a dataset. For ReAct agents, we used the HotpotQA [69] evaluation as reported in the paper. For generative agents, we rewrote the top five functions (out of forty traced) from the game using SGLang, which accounts for approximately 70% of all LLM calls. We then replayed these traces on our experimental platform.

As depicted in Figure 8, the ReAct agent, using SGLang, achieves a throughput that is 5.6× higher than that of vLLM and has only 13% of its latency. This improvement primarily results from the agent's process of appending current states (thoughts, actions, and observations) to the prompt for subsequent LLM calls. Our RadixAttention mechanism effectively identifies and minimizes redundant computations, a capability not present in the current inference engines. This mechanism also significantly reduces memory usage, allowing more questions to be addressed by a greater number of agents, thereby optimizing GPU utilization and enhancing throughput.

In the case of the generative agent task, the gaming setting's constraints permit only a single LLM call per simulation to maintain game logic consistency. In this case, SGLang still outperforms vLLM by 30% in both throughput and latency. This efficiency is due to the caching of agents' predefined routines, reducing redundant computations. The task prompt templates contain multiple arguments that vary at different speeds based on timestamps, resulting in complex prefix reuse patterns. Nevertheless, the SGLang runtime can automatically detect these patterns and reuse the KV cache.

6.2.3 Reasoning

We benchmark the performance of systems using advanced prompting techniques for reasoning tasks. These tasks involve solving GSM-8K problems in a zero-shot setting. Both throughput and latency metrics are reported in Figure 9 (a)(b).

Multi-chain reasoning [73] involves generating several solutions to one question and using meta-reasoning across these solutions for the final answer. This process allows for sharing the question and parallelizing the generation of multiple solutions. For each solution generation, we use various eliciting prompts such as "Think step-by-step" and "Take a deep breath". Unlike simple parallel sampling, this flexible approach with different eliciting prompts is not supported by

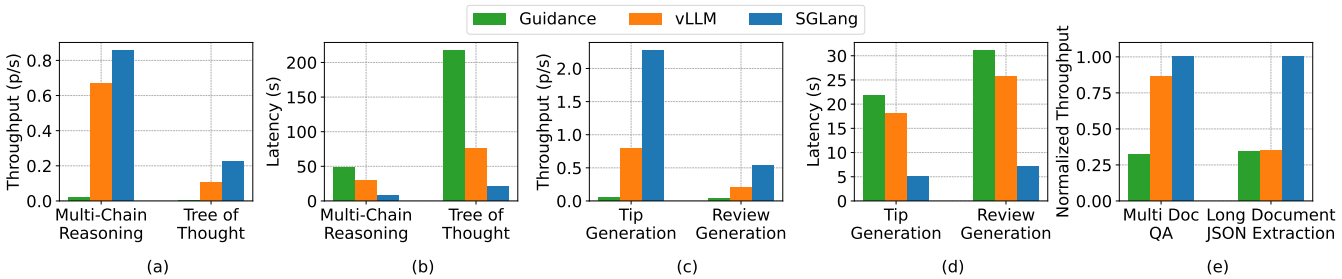

Figure 9: (a) Throughput on reasoning tasks. (b) Latency on reasoning tasks. (c) Throughput on content generation tasks. (d) Latency on content generation tasks. (e) Throughput on long-context tasks.

the vLLM API. Hence, vLLM exhibits lower throughput due to its inability to share prefixes flexibly. Guidance has lower throughput due to no batching. In contrast, SGLang allows users to intuitively implement this technique without worrying about manual optimizations, yet achieve high throughput. The improvement in throughput is not as substantial as in previous cases of few-shot learning, due to the zero-shot setting having a shorter shareable prefix.

Regarding latency, the SGLang interpreter can automatically resolve dependencies and submit requests in parallel. For a fair comparison, we directly translate the SGLang program into an equivalent Python program without explicitly using threading or asyncio to avoid added complexity. Consequently, the translated program does not parallelize parallelizable requests, resulting in higher latency.

Tree-of-thought [70] incorporates LLMs into the breadth-first tree-search algorithm. The LLMs must propose the next steps for branch exploration, execute these proposals, and evaluate the solutions. This method leads to complex sharing patterns beyond the capabilities of existing systems. Once again, SGLang surpasses others in terms of both higher throughput and lower latency in this benchmark.

6.2.4 Content Generation

We evaluate two advanced prompting methods for content generation: tip generation and article review. The skeleton-of-thought technique [37] first creates a brief outline of tips, then elaborates each into a detailed paragraph. The branch-solve-merge approach [48] assesses an article from multiple dimensions and then assigns a final grade. Results are shown in Figure 9 (c)(d). In both cases, there are opportunities for sharing and intra-program parallelism. SGLang systematically exploits them through an intuitive programming model, achieving higher throughput and reduced latency.

6.2.5 Long Document Processing

We benchmark the performance on long document processing tasks. When the document is long, KV cache reuse becomes more important as the forward pass to encode the prompt becomes significantly longer. We use CodeLlama models, which were fine-tuned on a context length of 16K tokens. On our setup, running a 7B model in fp16 precision on a 24 GB GPU, we can store at most 15K tokens. We select inputs exceeding 10K tokens in length, meaning the system can process only one request at a time at most. The first benchmark tests answering questions using multiple document segments from a paper, akin to a retrieval augmented generation setup. The second benchmark involves summarizing a city's information from its Wikipedia page and outputting a JSON object.

For the first benchmark, we employ the fork/join primitives in SGLang to implement parallel-context window [46]. By using this technique, we can encode multiple documents separately in parallel and concatenate the resulting KV cache, without letting them attend to each other. This approach reduces the quadratic computational cost of self-attention for long prompts. While this optimization alters the computation, it does not negatively impact accuracy in our benchmark. For the second benchmark, we reuse the KV cache of the long prompts when decoding multiple fields of the JSON object. The results are shown in Figure 9 (e). SGLang achieves 1.2× and 2.9× improvement on these two benchmarks, respectively. Since the maximum batch size is one, latency improvements are the same as throughput improvements.

6.3 Case Study

6.3.1 Vision Language Models

We conduct a case study to demonstrate the extensibility of SGLang by supporting a new vision language model, LLaVA v1.5 [29], on a new hardware platform, Apple Silicon. To achieve this, we add a new primitive, `sgl.image`, to accept image inputs, and incorporate a new backend, llama.cpp [15], an inference engine compatible with a wide range of hardware.

We integrate `sgl.image` as a new IR node into the interpreter and reuse the SGVM protocol to build a server based on llama.cpp. This integration is achieved in around 400 lines of code. Additionally, the code that uses `sgl.image` can be executed on GPT-4 Vision through the OpenAI back-end without code modification. We create a cache for image tokens based on the image's hash ID, similar to how we maintain RadixAttention. We run the LLaVA-Bench and report the throughput using SGLang and the default llama.cpp server

on an M2 Ultra. The LLaVA-Bench contains a total of 24 images and 60 questions, with 2.5 questions on each image on average. SGLang increases the throughput of llama.cpp by 1.7×. The speedup is mainly affected by the number of questions per shared image and the output length.

6.3.2 Compiler Optimizations

We evaluate the effectiveness of two compiler optimizations. The first is GPT-4-assisted code movement for prefix sharing. We collect 20 prompt templates from the internet and implement them in SGLang. We utilize 5 of these templates for few-shot training examples and the remaining 15 as test cases. Our evaluation shows that, for 12 out of the 15 templates, GPT-4 successfully reorders the graph nodes without altering the semantics, as confirmed by manual inspection of the modified prompts. On average, this optimization results in a 60-token increase in the sharable prefix length, showcasing GPT-4's effectiveness. Failures in creating optimized prompt order come from an incorrect understanding of the semantic meaning behind the graph nodes. It is too aggressive and puts all constants upfront even when such ordering changes the original semantics. This case study aims to explore the use of GPT-4 for compiler optimizations. More work is needed to make these kinds of optimizations reliable in the future.

The second optimization involves prefetching annotation insertion. For frequently used long prompts, we cache them in CPU memory. The compiler automatically inserts prefetch nodes into the computational graph. During runtime, when the server encounters a prefetch node, it evicts nodes from the radix tree and swaps in the cached prefix. As the cost of swapping can be substantially overlapped, this reduces the first-token latency of the subsequent request. We test SGLang programs that contain function calls with a 4k prefix length. By prefetching the 4k prefix, the first token latency of the called function has been reduced from 1 second to 0.2 seconds, which is a 80% reduction.

6.3.3 Productivity

We conducted a case study to demonstrate SGLang's productivity improvement by comparing its Lines of Code (LoC) with OpenAI APIs. We designed a prompting flow for article analysis: if it's about a celebrity, the program generates a bio and introduction; otherwise, it summarizes the article. Implementing this flow in both OpenAI APIs and SGLang, we counted effective LoC, excluding comments and empty lines. The results show the OpenAI API required 206 lines, while SGLang needed only 91, marking a 55% reduction.

We delve into the implementations and summarize the insights regarding why SGLang is capable of using fewer LoC. First, whereas users must manually extract content from OpenAI API responses and chain them to form a multi-step flow, SGLang simplifies this process by leveraging the string concatenation operator. Second, since the flow involves several calls with restricted answers, such as true/false and integers, users are required to construct an appropriate logit bias to ensure OpenAI returns one of the expected answers. In contrast, SGLang provides primitives that automatically construct logit bias for this purpose, allowing users to merely specify the acceptable data type and choices. Third, in cases where skeleton-based prompting is used to reduce end-to-end latency, OpenAI API users need to initiate a thread pool for parallel request submission explicitly. However, SGLang's built-in primitive `fork` automatically parallelizes requests.

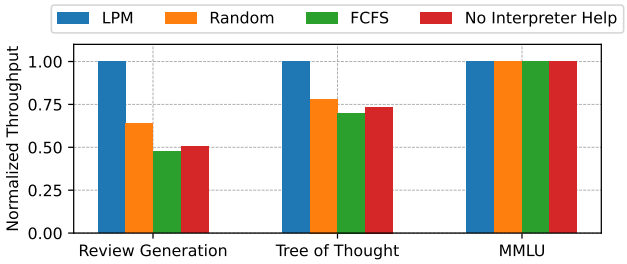

Figure 10: Performance of different scheduling policies.

6.3.4 Larger Models

While most of our experiments are conducted using a 7B model, our techniques also perform effectively on larger models. We test a 13B model on an A100 (40GB) and a 33B model on an A100 (80GB) using the ReAct agent benchmark, achieving a similar speedup (5×).

6.4 Ablation Study

6.4.1 Scheduling Policy

We evaluate the effectiveness of our cache-aware scheduling by comparing it with several other policies. The results are shown in Figure 10. "LMP", or longest matching prefix, is the default policy outlined in Section 5.2. "Random" employs a random order. "FCFS" adheres to a first-come-first-serve policy. "No interpreter help" omits the prefix hints from the interpreter, which is explained in detail below. Although RadixAttention can automatically detect shared prefixes, our runtime implementation is simplified: we only compare a pending request with the existing cache tree. We do not construct a tree index for all requests in the waiting queue. This approach may lead to suboptimal performance when there is a large batch of pending requests that share a prefix but have yet to be processed and inserted into the tree. To mitigate this, we use the interpreter to help with the scheduling. When the interpreter identifies a highly likely shared prefix (e.g., during a fork), it issues a forward request for this prefix first and stores it in the tree, and then it sends out the requests that can reuse the prefix. This method leverages hints from the interpreter to lessen the load on backend scheduling, underscoring the benefits of our co-design strategy. As depicted in Figure 10,

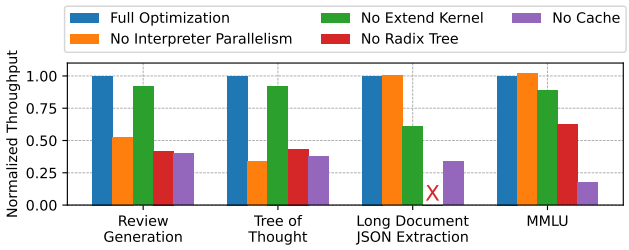

Figure 11: Optimization breakdown.

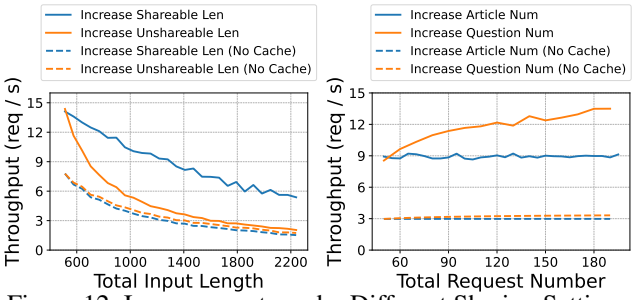

Figure 12: Improvements under Different Sharing Settings

"LPM" exhibits superior performance, while other policies struggle with a lower cache hit rate and reduced throughput. All policies perform similarly for simple tasks like MMLU.

6.4.2 Optimization Breakdown

We study the effectiveness of optimizations by disabling each one individually. Figure 11 illustrates the results. "No Extend Kernel" disables our optimized kernel, leading to degraded performance, particularly in long-context tasks. "No Interpreter Parallelism" turns off intra-program parallelism, resulting in reduced parallelism and cache locality. "No Cache" deactivates the cache, significantly deteriorating throughput across all benchmarks. "No Radix Tree" uses a list instead of a tree to manage the cache, leading to increased memory usage and copy overhead, and consequently, lower performance.

6.4.3 Improvements under Different Sharing Settings

We study how the improvement of prefix sharing is influenced by different parameters, including the lengths of shareable and unshareable inputs, and sharing structure configurations.

As shown in Figure 12, in the left subfigure, we establish a base setting with a shareable input length of 256, an unshareable input length of 256, and an output length of 256. We then gradually increase either the shareable input length or the unshareable length and observe the changes in throughput. The results demonstrate that the RadixAttention cache significantly enhances throughput compared to the "No Cache" scenario, with more notable improvements observed when the shareable length is longer.

In the right subfigure, we set a base scenario with 5 articles and 2 questions per article, where the article comprises 1000 tokens and each question 256 tokens. There are $5 \times 2 = 10$ requests in this base setting, with each article being shared by two questions. Subsequently, we increment either the number of articles or the number of questions per article. The findings indicate that the RadixAttention cache exhibits superior performance when the number of questions per article increases, as this creates more opportunities for sharing.

6.4.4 RadixAttention Overhead Analysis

We analyze the overhead of maintaining the radix tree at runtime. The tree structure is maintained on the CPU. In a typical scenario, which has a shareable input length of 256, an unshareable input length of 256, and 128 requests (organized into 16 subtrees, with 8 branches for each subtree), the forward pass in our setup takes 17.6 seconds. Meanwhile, the tree operation only takes 0.07 seconds. We also construct a request trace with no cache hits and observe the tree maintenance incurs only a 0.5% overhead. Therefore, the overhead is negligible. Furthermore, the overhead of tree maintenance can be further reduced by using a high-performance C++ implementation and by overlapping CPU and GPU operations.

7 Discussion and Limitations

Here are SGLang's limitations and future development plans:

- Our interpreter runs arbitrary Python code, but the tracing and compiler lack support for data-dependent control flows. Introducing new control flow nodes to the SGLang IR, similar to TensorFlow's dynamic flow [75], is required to address this.
- The current tokenization algorithm may create artifacts at prompt and generation boundaries. Implementing the token healing technique [16] could resolve this.
- We have not yet implemented grammar-constrained decoding. Efficient implementation would necessitate techniques from [22, 65] and careful batching.
- Plans include integrating more input and output modalities, such as incorporating text-to-image models into SGLang and exploring audio and video capabilities.

8 Related Work

Programming with Language Models The trend in programming LLM includes the development of better prompting techniques [3, 37, 64, 70], tools usage [44, 49, 50], agent frameworks [27, 41, 42, 53, 54, 67, 71], and programming systems [4, 16, 21, 24, 30, 65]. Among these, Guidance and LMQL are the most similar to our work. We discuss the differences between SGLang and existing LLM programming systems in Section 2.4 and evaluate their performance in Section 6. In short, the co-design approach of SGLang significantly enhances execution efficiency.

Inference Optimizations for LLMs. Common inference optimizations for LLMs include batching [74], memory optimizations [2, 23, 52], GPU kernel optimizations [10], model parallelism [45], parameter sharing [51], speculative execution [25, 34, 56], scheduling [17, 66], caching [55], quantization [11, 14, 28, 68], and sparsification [13]. These optimizations are complementary to our work and can be, or have already been, integrated into our backend.

Compilers for Machine Learning. Various domain-specific compilers for machine learning have been developed. These compilers are capable of optimizing a single tensor operator [7, 12, 33, 57, 62, 72, 77], a computational graph [20, 59], a distributed computation graph [32, 78], and classical algorithms [36]. Distinct from these more general-purpose or low-level compilers, SGLang is specifically designed for LLM applications and incorporates LLM-specific optimizations.

9 Conclusion

In this paper, we introduce SGLang, an efficient programming system for LLMs. SGLang greatly accelerates common LLM workloads by up to 5× with reduced code complexity and improved control. Through the co-design of the language and runtime, we aim to present a more integrated approach to optimizing LLM programming paradigms.

References

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[3] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*, 2023.

[4] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[9] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[11] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[12] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[13] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

[14] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Optq: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2022.

[15] Georgi Gerganov. Llama.cpp. https://github.com/ggerganov/llama.cpp. Accessed: 2023-11.

[16] Guidance AI. A guidance language for controlling large language models. https://github.com/guidance-ai/guidance. Accessed: 2023-11.

[17] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[18] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2020.

[19] Hugging Face. Text generation inference. https://github.com/huggingface/text-generation-inference. Accessed: 2023-11.

[20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[21] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[22] Michael Kuchnik, Virginia Smith, and George Amvrosiadis. Validating large language models with relm. *Proceedings of Machine Learning and Systems*, 5, 2023.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[24] LangChain AI. Langchain. https://github.com/langchain-ai/langchain. Accessed: 2023-11.

[25] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[26] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[27] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[28] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[29] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. *arXiv preprint arXiv:2310.03744*, 2023.

[30] Jerry Liu. LlamaIndex, November 2022.

[31] Xiaoxia Liu, Jingyi Wang, Jun Sun, Xiaohan Yuan, Guoliang Dong, Peng Di, Wenhai Wang, and Dongxia Wang. Prompting frameworks for large language models: A survey. *arXiv preprint arXiv:2311.12785*, 2023.

[32] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[33] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[34] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

[35] Model TC. Lightllm. https://github.com/ModelTC/lightllm. Accessed: 2023-11.

[36] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 899–917, 2020.

[37] Xuefei Ning, Zinan Lin, Zixuan Zhou, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Large language models can do parallel decoding. *arXiv preprint arXiv:2307.15337*, 2023.

[38] NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM. Accessed: 2023-11.

[39] OpenAI. Openai api reference. https://platform.openai.com/docs/api-reference. Accessed: 2023-11.

[40] OpenAI. Gpt-4 technical report, 2023.

[41] Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2023.

[42] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *In the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[44] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

[45] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[46] Nir Ratner, Yoav Levine, Yonatan Belinkov, Ori Ram, Inbal Magar, Omri Abend, Ehud Karpas, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. Parallel context windows for large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6383–6402, 2023.

[47] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[48] Swarnadeep Saha, Omer Levy, Asli Celikyilmaz, Mohit Bansal, Jason Weston, and Xian Li. Branch-solve-merge improves large language model evaluation and generation. *arXiv preprint arXiv:2310.15123*, 2023.

[49] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[50] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.

[51] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[52] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[53] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[54] Significant Gravitas. AutoGPT.

[55] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. Ugache: A unified gpu cache for embedding-based deep learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 627–641, 2023.

[56] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.

[57] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[58] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[59] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[61] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv: Arxiv-2305.16291*, 2023.

[62] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[63] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[65] Brandon T Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv e-prints*, pages arXiv–2307, 2023.

[66] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[67] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. 2023.

[68] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[69] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[70] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

[71] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[72] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 660–678, 2023.

[73] Ori Yoran, Tomer Wolfson, Ben Bogin, Uri Katz, Daniel Deutch, and Jonathan Berant. Answering questions by meta-reasoning over multiple chains of thought. *arXiv preprint arXiv:2304.13007*, 2023.

[74] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[75] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[76] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, 2019.

[77] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[78] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.