SGLang:
Efficient Execution of Structured Language Model Programs:

Paper's Authors: Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, Ying Sheng

Presentation by: Anay Bhakat

Context

- What are Language Models(LMs)?
    - Simply put, they are the programmatic usage of LLMs
    - 2 common properties:
        - Contain multiple LLM calls in their control flow
        - Receive structured input and produce structured outputs
- Issues with LMs
    - Tedious and difficult to program due to “not deterministic nature”
    - Executing LMs is inefficient due to redundant computation and unoptimized memory usage

What is SGLang?

- SGLang: Structured Generation Language for LLMs
  - 2 Main components
    - Front-End to simplify the programming of LMs
    - Backend improvement through runtime optimizations

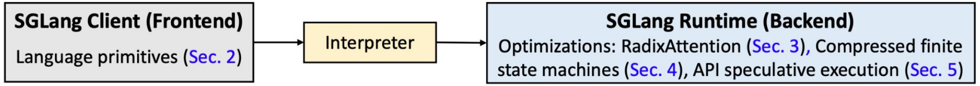


Figure 1: System architecture: An interpreter executes language primitives with optimized runtime.

Components of SGLang

Frontend

- Domain Specific Language Embedded in Python
  - Primitives for Generation:
    - ex) extend, Gen, select
  - Primitives For Parallelism Control:
    - ex) Fork, Join
- User-Friendly Abstraction to simplify interaction with LLMs

Backend

- 2 Main Components:
    - Interpreter and Compiler
- Handles:
    - Efficient program execution
    - Parallelism
    - Memory Management
    - Asynchronous Tasks

Interpreter

- Real-time execution engine:
    - Executes the code
- Manages prompt state as a stream:
    - Submits operations for asynchronous execution

Compiler

- Optimizes the program before execution:
  - Converts the program into a computational graph
  - Focuses on improving efficiency by identifying redundancies as well as opportunities for parallelism
- Benefits:
  - Parallelization of independent tasks
  - Memory Optimization
  - Optimizes for hardware and API only models

How the Optimizations Work

Radix Attention

Optimization 1

Radix Attention

- **Goal**: Maximize cache reuse and minimize redundant computations when handling multiple requests.
- **Strategy**: Use a **radix tree** to manage shared prefixes and a **DFS-based scheduling** for optimal cache usage.

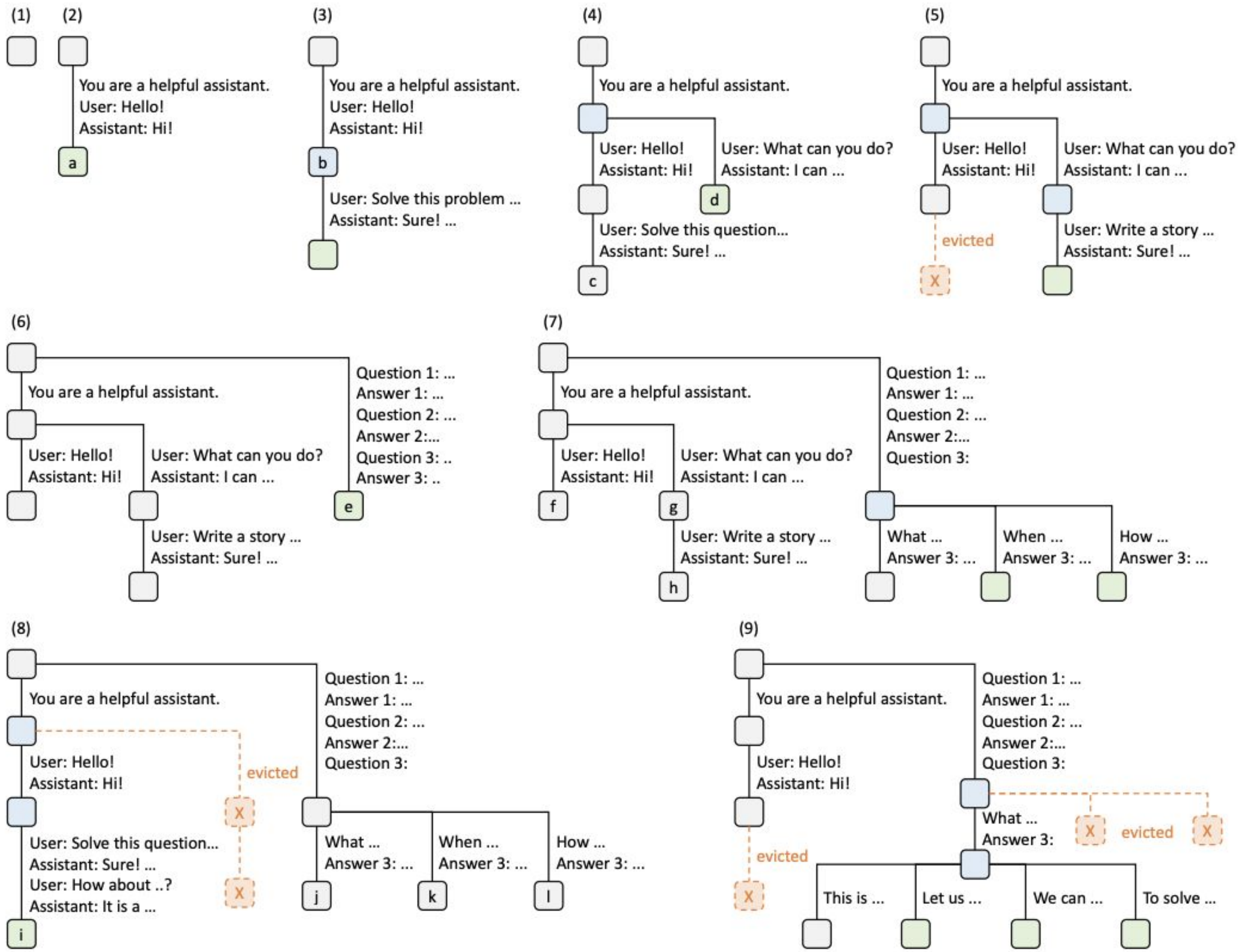
KV Cache and Radix Tree

KV Cache:

- Intermediate tensors generated during LLM inference are stored as **key-value (KV) pairs**.
- The **KV cache** depends on the prefix tokens of each request.
- **Reuse**: Requests with the same prefix can reuse the KV cache.
- Eviction Policy: **LRU**

Radix Tree:

- Efficient structure for managing shared prefixes across requests.
  - Space Efficient compared to a simple trie along with extended labeling capabilities allowing for increased efficiency
- Nodes represent sequences of tokens, allowing efficient KV cache reuse



(1)
(2)
You are a helpful assistant.
User: Hello!
Assistant: Hi!
a
(3)
You are a helpful assistant.
User: Hello!
Assistant: Hi!
b
User: Solve this problem ...
Assistant: Sure! ...
(4)
You are a helpful assistant.
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
d
User: Solve this question...
Assistant: Sure! ...
c
(5)
You are a helpful assistant.
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
evicted
User: Write a story ...
Assistant: Sure! ...
X
(6)
You are a helpful assistant.
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3: ..
Answer 3: ..
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
e
User: Write a story ...
Assistant: Sure! ...
(7)
You are a helpful assistant.
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3:
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
f
g
User: Write a story ...
Assistant: Sure! ...
h
What ...
Answer 3: ...
When ...
Answer 3: ...
How ...
Answer 3: ...
(8)
You are a helpful assistant.
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3:
User: Hello!
Assistant: Hi!
evicted
X
User: Solve this question...
Assistant: Sure! ...
User: How about ..?
Assistant: It is a ...
X
i
What ...
Answer 3: ...
When ...
Answer 3: ...
How ...
Answer 3: ...
j
k
l
(9)
You are a helpful assistant.
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3:
User: Hello!
Assistant: Hi!
What ...
Answer 3:
X
evicted
X
evicted
X
This is ...
Let us ...
We can ...
To solve ...

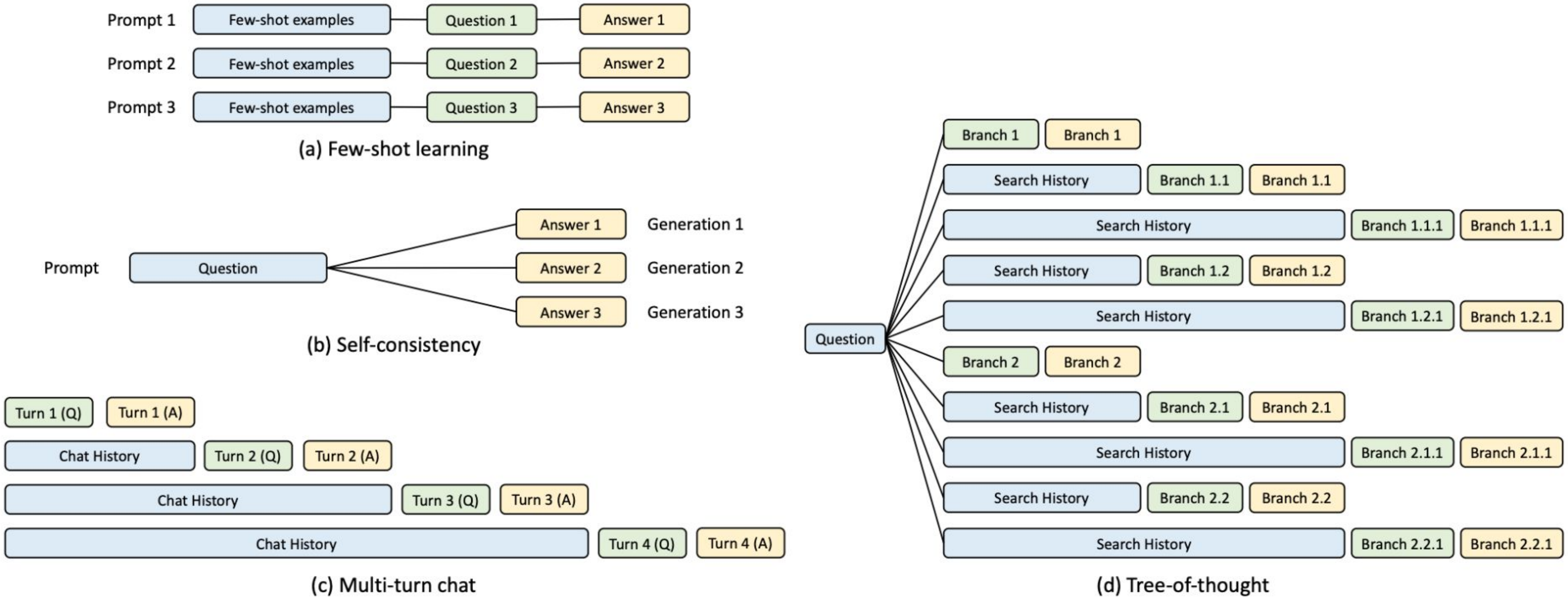


Figure 9: KV cache sharing examples. Blue boxes represent shareable prompt parts, green boxes indicate non-shareable parts and yellow boxes mark non-shareable model outputs. Shareable elements include few-shot learning examples, questions in self-consistency [53], chat history in multi-turn chat, and search history in tree-of-thought [56].

Pseudocode for Cache Aware Scheduling

1) Retrieve Requests form the Waiting Queue
2) Match Prefixes
3) Sort Requests by Matched Prefix Length
4) Select Requests for the Next Batch
5) Evict and Allocate Memory
6) Run the batch
7) Process Finished Requests

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)
end for
// 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
    end if
end for
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)
end if
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)
end for
return finished_requests
```

Computational Complexity(C)

$$C \geq \sum_{e \in \text{edges}(T)} |e|.$$

Variables:

- *T:* The radix tree of all requests in the batch.
- e : Each edge in the radix tree corresponds to a shared prefix between requests.
- |e|: The size of the **KV cache** associated with each edge.

Explanation:

- The **total computation** for processing all requests is at least equal to the sum of all KV cache sizes associated with the shared prefixes in the radix tree.
- Significance: Each shared prefix must be computed **at least once**, but by sharing the cache, we can avoid redundant computations.

Cache Hit Rate

The cache hit rate, defined as

$$\frac{\sum_{r \in R} \text{number of cached prefill tokens in } r}{\sum_{r \in R} \text{number of prefill tokens in } r},$$

equals $1 - \frac{C}{\sum_{r \in R} \text{number of prefill tokens}}$, reaches its upper bound, delivering optimality.

Explanation:

- **Numerator**: Total number of tokens that can be directly retrieved from the cache (without recomputation).
- **Denominator**: Total number of tokens across all requests in the batch.

Interpretation:

- The higher the cache hit rate the **less redundant computation** is required.
- An **optimal cache hit rate** means most tokens can be fetched from the cache, minimizing the need for recomputation.

DFS Order for Optimal Cache Usage

- Main Point from Theorem 3.1:
  - Processing requests in DFS Order on radix tree helps ensure optimal cache reuse
- Reason for using DFS:
  - When you process the **longest shared prefix** first, all requests that share this prefix will hit the cache continuously until the entire subtree has been processed.
  - This guarantees continuous cache hits and minimizes recomputation because no prefix needs to be recomputed until all results sharing the prefix are processed

Optimal cache hit rate

- Goal:
  - We achieve this bound when each shared prefix is computed **only once** and the cache is reused for all requests that require it
- DFS Order:
  - Helps ensure that shared prefixes are processed together, keeping the cache active and minimizing recomputation.

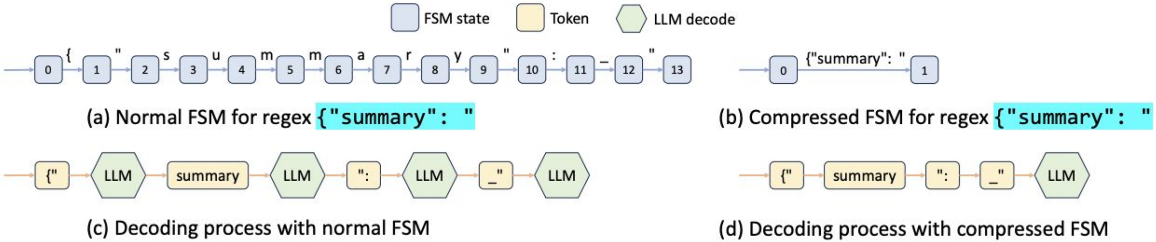
$$C = \sum_{e \in \mathrm{edges}(T)} |e|.$$

Compressed FSM

Optimization 2

Compressed FSM

- **Goal**: Speed up structured output decoding (e.g., JSON).
- Strategy: Use a compressed FSM to “batch” decode tokens to reduce number of decision points
- Benefits:
    - Increased Decoding Speed
    - Reduced Latency and Increased Throughput
    - Accuracy, Flexibility, and Optimized for generating structured output



(a) Normal FSM for regex `{"summary": "`

(b) Compressed FSM for regex `{"summary": "`

(c) Decoding process with normal FSM

(d) Decoding process with compressed FSM

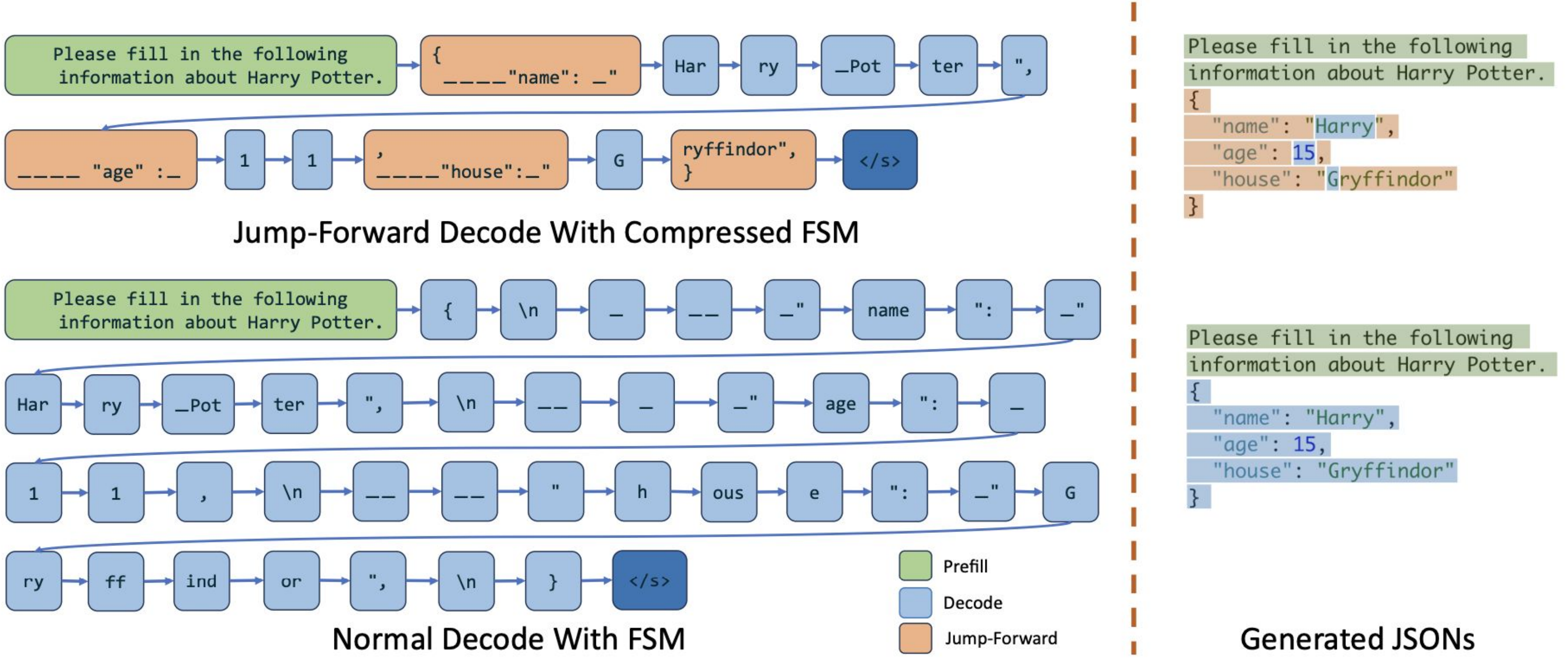


Figure 11: Comparison of decoding using Compressed FSM versus normal FSM: The left subfigure depicts the decoding process per forward pass, while the right subfigure explains the origins of various result components.

Efficient Endpoint Calling with API Speculation Execution

Optimization 3

API Speculative Execution

- Goal:
  - Optimize multi-call programs using API-only models
- Strategy:
  - Use speculative execution by generating additional tokens beyond condition and matching later
- Benefits:
  - Reduced API call latency as well as input token costs

How this works

- Example: pattern: s += context + "name:" + gen("name", stop="\n") + "job:" + gen("job", stop="\n")
  - A program asks the model to generate description of a character with a multiple call pattern
- <u>“Speculative Execution”</u>
  - Model ignores the stop and allows the model to generate a few extra tokens past the stop condition
  - These tokens are kept by the interpreter
  - If the model generates “job” in the first part, you save an additional api call

Programming Example

```python
@function
def multi_dimensional_judge(s, path, essay):
  s += system("Evaluate an essay about an image.")
  s += user(image(path) + "Essay:" + essay)
  s += assistant("Sure!")

  # Return directly if it is not related
  s += user("Is the essay related to the image?")
  s += assistant(select("related", choices=["yes", "no"]))
  if s["related"] == "no": return

  # Judge multiple dimensions in parallel
  forks = s.fork(len(dimensions))
  for f, dim in zip(forks, dimensions):
    f += user("Evaluate based on the following dimension:" +
      dim + ". End your judgment with the word 'END'")
    f += assistant("Judgment:" + gen("judgment", stop="END"))

  # Merge the judgments
  judgment = "\n".join(f["judgment"] for f in forks)

  # Generate a summary and a grade. Return in the JSON format.
  s += user("Provide the judgment, summary, and a letter grade")
  s += assistant(judgment + "In summary," + gen("summary", stop=".")
              + "The grade of it is" + gen("grade"))

  schema = r'\{"summary": "[\w\d\s]+\.", "grade": "[ABCD][+-]?"\}'
  s += user("Return in the JSON format.")
  s += assistant(gen("output", regex=schema))

state = multi_dimensional_judge.run(...)
print(state["output"])
```

Handle chat template and multi-modal inputs

Select an option with the highest probability

Fetch result; Use Python control flow

Runtime optimization: KV Cache Reuse (Sec. 3)

Multiple generation calls run in parallel

Fetch generation results

Runtime optimization: API speculative execution (Sec. 5)

Runtime optimization: fast constrained decoding (Sec. 4)

Run an SGLang program

Figure 2: The implementation of a multi-dimensional essay judge in SGLang utilizes the branch-solve-merge prompting technique [40]. Primitives provided by SGLang are shown in red.

Evaluation Processes

Models Evaluated

Dense Models(**Llama-2**):

- Llama-2 models ranging from 7B to 70B parameters.
- Tests focused on LLM inference efficiency across different model sizes.

Sparse Models(**Mixtral**)

- A sparse model evaluated for performance under SGLang, focusing on how sparsity affects throughput and latency.

Multi-Modal Models:

- **LLaVA** (Image): Tested on tasks involving image inputs with language prompts.
- **LLaVA-NeXT** (Video): Multi-modal model tested for video input tasks.

API-Access Models(**OpenAI GPT-3.5**):

- Evaluated for black-box API access, focusing on optimizing API call efficiency and speculative execution for reducing latency and token usage.

Hardware

AWS EC2 Instances:

- G5 Instances with NVIDIA A10G GPUs.
- Additional experiments on A100G GPUs for higher-end tasks.
- Optimized for large-scale model inference and high-throughput workloads.

GPU Selection:

- NVIDIA A10G(24GB) GPUs:
  - Designed for both inference and graphics-intensive workloads, balancing cost and performance
- NVIDIA A100(80GB) GPUs:
  - Designed for deep learning and large model training

Baselines

SGLang was compared with state of the art baselines to showcase its performance

1. **vLLM**:
   - A high-throughput inference engine designed to maximize the efficiency of large language model (LLM) inference.
   - Uses KV cache management to improve performance.
2. **Guidance**:
   - A programming system built to facilitate prompting for LLMs.
   - The evaluation uses the llama.cpp backend for this baseline.
3. **LMQL**:
   - A query language designed to improve the prompting process of LLMs.
   - Uses Hugging Face's Transformers backend for evaluation.
4. **OpenAI GPT-3.5**:
   - Evaluated as a black-box API model.
   - This serves as a baseline for models where API access is the only interaction point, without internal optimizations.

Metrics

- Throughput:
  - Run a large batch of program instances
  - Compute maximum throughput number of program instances per second
  - Unit: programs per second(p/s)
- Latency:
  - Singular program executed without batching and average latency for multiple instances is reported

Findings

Results on open weight models

SGLang Improvements:

- Improved throughput by upto **6.4x**
- Improved latency by upto **3.7x**

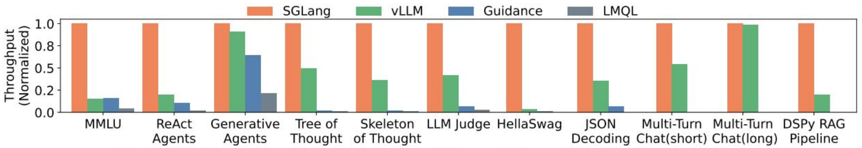


Figure 5: Normalized throughput on Llama-7B models. Higher is better.

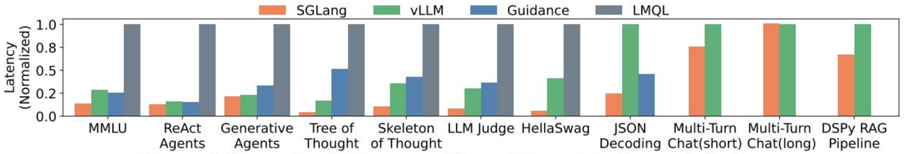


Figure 6: Normalized latency on Llama-7B models. Lower is better.

Speedup on Large Models with Tensor Parallelism

- Good speedup and throughput on large models as well

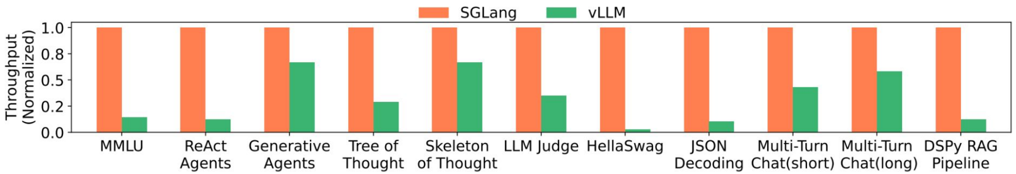


Figure 7: Normalized throughput on Mixtral-8x7B models with tensor parallelism. Higher is better.

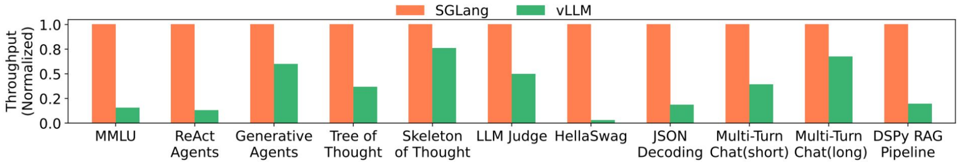


Figure 12: Normalized throughput on Llama-2-70B models with tensor parallelism. Higher is better.

Performance for LLaVa Video and Image Models

Table 2: Throughput comparison on multi-modal LLaVA image and video models.

Model	LLaVA-v1.5-7B (image)	LLaVA-NeXT-34B (video)
Author's original implementation	0.18 image/s	0.02 frame/s
SGLang	**1.15 image/s**	**0.10 frame/s**

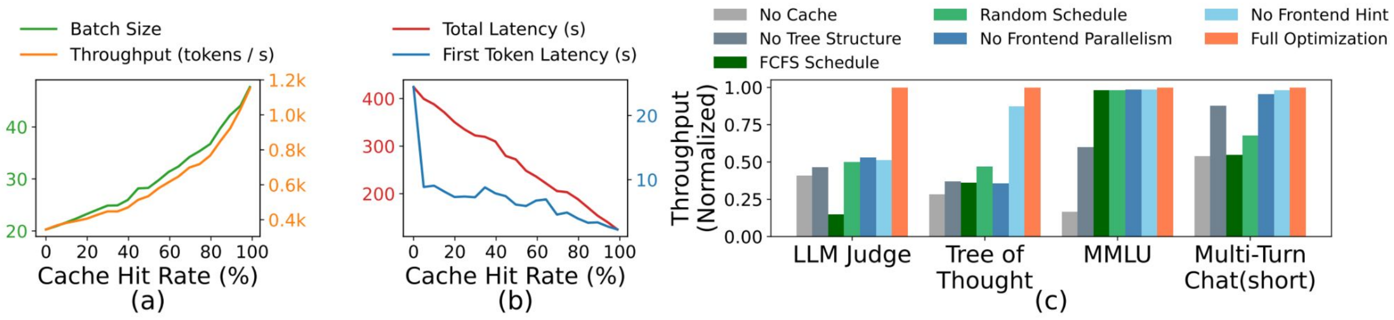


Figure 8: (a)(b) Cache hit rate ablation study. (c) RadixAttention ablation study.

Production Deployment Testing

- Deployed in Chatbot arena to “serve” open-weight models
  - 52.4% Radix Attention cache hit rate for LLaVa-Next-34B
  - 74.1% hit rate for Vicuna-33B
    - Reduces first-token latency by ~1.7x

Related Work + My Opinions

Related Work

- Multiple papers have used/considered KV cache however thai paper was the first to use KV cache as a tree based LRU cache
- Work's differentiators are in the novel runtime optimizations and is compatible with other frameworks and inference optimizations

My Take

- Radix Attention is very novel and using SGLang to simplify interfacing with models is very novel
- Improving cache efficiency and getting rid of redundant computations is huge
- A system that actually focuses on improving the experience when using black boxed APIs like GPT is super useful
- The code is public so developers can experiment and create their own features on top of SGLang

Thanks For Listening

Citation(all images are from the paper)

- Zheng, Lianmin, et al. "Efficiently programming large language models using sglang." *arXiv preprint arXiv:2312.07104* (2023).