MiniSGLang Study Notes

Day 4 — Forward Pass, Sampling, Prefill vs Decode

1. The _forward() Method

The core execution step per batch. Loads token IDs from the token pool, runs the model, writes the output token back, then updates decode state.

scheduler.py — _forward()

```
def _forward(self, forward_input: ForwardInput) -> ForwardOutput:
    self._load_token_ids(forward_input)
    batch, sample_args = forward_input.batch, forward_input.sample_args
    if ENV.OVERLAP_EXTRA_SYNC:
        self.stream.synchronize()
    forward_output = self.engine.forward_batch(batch, sample_args)
    self._write_token_ids(forward_input, forward_output)
    self.decode_manager.filter_reqs(forward_input.batch.reqs)
    return forward_output
```

The engine chooses between CUDA graph replay (fast path for fixed batch shapes) or a direct model forward call:

engine.py — forward_batch()

```
def forward_batch(self, batch: Batch, args: BatchSamplingArgs) -> ForwardOutput:
    assert torch.cuda.current_stream() == self.stream
    with self.ctx.forward_batch(batch):
        if self.graph_runner.can_use_cuda_graph(batch):
            logits = self.graph_runner.replay(batch)
        else:
            logits = self.model.forward()
```

llama.py — LlamaForCausalLM.forward()

```
def forward(self) -> torch.Tensor:
    # reads batch.input_ids from global context set by ctx.forward_batch()
    output = self.model.forward(get_global_ctx().batch.input_ids)
    logits = self.lm_head.forward(output)
    return logits
```

2. Load & Write Indices (CPU Pre-computation)

Previously computed inside the GPU kernel. Now **_prepare_batch()** builds them on CPU and bundles them into **ForwardInput** before the GPU runs.

scheduler.py — _prepare_batch() [index section]

```
load_indices = self._make_2d_indices(
    [(r.table_idx, r.cached_len, r.device_len) for r in batch.padded_reqs]
)
```

```
write_indices = self._make_2d_indices(
    [
        (r.table_idx, r.device_len, r.device_len + 1)
        if r.can_decode() # write next token slot
        else self.dummy_write_2d_pos # chunked req: throw away
        for r in batch.reqs
    ]
)
```

scheduler.py — _load_token_ids() / _write_token_ids()

```
def _load_token_ids(self, input: ForwardInput) -> None:
    input.batch.input_ids = self.token_pool.view(-1)[input.load_indices]

def _write_token_ids(self, input: ForwardInput, output: ForwardOutput) -> None:
    self.token_pool.view(-1)[input.write_indices] = output.next_tokens_gpu
```

scheduler.py — _make_2d_indices()

```
def _make_2d_indices(self, ranges: List[Tuple[int, int, int]]) -> torch.Tensor:
    # Example: table shape (3,4), ranges [(0,1,3),(2,0,2)] -> [1,2,8,9]
    STRIDE = self.token_pool.stride(0)
    needed_size = sum(end - begin for _, begin, end in ranges)
    indices_host = torch.empty(needed_size, dtype=torch.int32, pin_memory=True)
    offset = 0
    for entry, begin, end in ranges:
        length = end - begin
        offset += length
        torch.arange(
            begin + entry * STRIDE,
            end + entry * STRIDE,
            dtype=torch.int32,
            out=indices_host[offset - length : offset],
        )
    return indices_host.to(self.device, non_blocking=True)
```

3. Sampling

Important: BatchSamplingArgs is NOT passed into the model. The model only produces logits. Sampling runs after the forward pass and controls how a single token is picked from those logits.

engine.py — Sampler.prepare()

```
def prepare(self, batch: Batch) -> BatchSamplingArgs:
    params = [r.sampling_params for r in batch.reqs]
    if all(p.is_greedy for p in params):
        return BatchSamplingArgs(temperatures=None) # fast greedy path

    MIN_P = MIN_T = 1e-6
    ts = [max(0.0 if p.is_greedy else p.temperature, MIN_T) for p in params]
    top_ks = [p.top_k if p.top_k >= 1 else self.vocab_size for p in params]
    top_ps = [min(max(p.top_p, MIN_P), 1.0) for p in params]
```

```
    temperatures = make_device_tensor(ts, torch.float32, self.device)
    top_k = make_device_tensor(top_ks, torch.int32, self.device) if any(...) else None
    top_p = make_device_tensor(top_ps, torch.float32, self.device) if any(...) else None
    return BatchSamplingArgs(temperatures, top_k=top_k, top_p=top_p)
```

Sampling strategies:

Strategy	Behavior	Activated when
Greedy	argmax(logits) — deterministic	all reqs have is_greedy=True
Temperature	logits / T — higher T = more random	temperature > 0
Top-K	Sample only from the top K logit values	top_k >= 1
Top-P	Sample from smallest set with cumul. prob>=P	top_p < 1.0

4. Prefill vs Decode

Both run in the same scheduler loop via _schedule_next_batch(), which tries prefill first, then falls back to decode:

scheduler.py — _schedule_next_batch()

```
def _schedule_next_batch(self) -> ForwardInput | None:
    batch = (
        self.prefill_manager.schedule_next_batch(self.prefill_budget)
        or self.decode_manager.schedule_next_batch()
    )
    return self._prepare_batch(batch) if batch else None
```

prefill.py — PrefillManager.schedule_next_batch()

```
def schedule_next_batch(self, prefill_budget: int) -> Batch | None:
    if len(self.pending_list) == 0:
        return None
    adder = PrefillAdder(token_budget=prefill_budget, ...)
    reqs, chunked_list = [], []
    for pending_req in self.pending_list:
        if req := adder.try_add_one(pending_req):
            pending_req.chunked_req = None
            if isinstance(req, ChunkedReq):
                pending_req.chunked_req = req
                chunked_list.append(pending_req)
            reqs.append(req)
        else:
            break
    self.pending_list = chunked_list + self.pending_list[len(reqs):]
    return Batch(reqs=reqs, phase="prefill")
```

decode.py — DecodeManager.schedule_next_batch()

```
def schedule_next_batch(self) -> Batch | None:
    if not self.runnable:
        return None
```

```
    return Batch(reqs=list(self.running_reqs), phase="decode")
```

Comparison:

	Prefill	Decode
Source	pending_list	running_reqs
Input tokens	All prompt tokens (or a chunk)	1 token per req *
Output	KV cache built	One new token sampled
After pass	Promoted to running_reqs	Stays in running_reqs
Phase label	"prefill"	"decode"

** Why 1 token for decode: load_indices uses (table_idx, cached_len, device_len). For a decode req, device_len = cached_len + 1 because after each step the previous output token was written to token_pool and device_len advances by exactly 1. So the range [cached_len, device_len) always covers exactly 1 token.*

5. Chunked Prefill Deep Dive

A 2000-token prompt with prefill_budget=512 is split across 4 forward passes. Each chunk writes KV cache entries that persist for all future chunks.

Round	Chunk	cached_len after	What happens
1	[0 : 512]	512	KV for tokens 0-511 written. chunked_req set on pending_req.
2	[512 : 1024]	1024	Attends over 0-511 (already cached) + computes 512-1023.
3	[1024: 1536]	1536	Same pattern. KV accumulates in page table.
4	[1536: 2000]	2000	Final chunk. chunked_req=None. Promoted to decode via filter_reqs.

prefill.py — why chunked reqs go to the FRONT of pending_list

```
# After scheduling, in-progress chunked reqs jump to the front.
# If they went to the back, new reqs could keep getting scheduled first,
# leaving a half-prefilled req's KV pages allocated forever (memory leak).
self.pending_list = chunked_list + self.pending_list[len(reqs):]
# ^^ in-progress ^^ remaining new reqs
```

Each completed chunk leaves its KV cache in the page table. The next chunk's load_indices starts at cached_len (not 0), so only new tokens are loaded — but attention covers the full KV history via the page table.

6. Prefill → Decode Promotion via filter_reqs()

Called after every forward pass. Merges the just-processed batch into running_reqs, keeping only requests that can still generate tokens.

decode.py — DecodeManager.filter_reqs()

```
def filter_reqs(self, reqs: Iterable[Req]) -> None:
    self.running_reqs = {
```

```
        req for req in self.running_reqs.union(reqs)
        if req.can_decode()
    }
    # After a prefill's final chunk:
    # req.can_decode() returns True -> enters running_reqs
    # After a chunked prefill mid-chunk:
    # req.can_decode() returns False -> not added yet
    # After decode generates EOS or hits max_tokens:
    # req.can_decode() returns False -> removed from running_reqs
```

7. Normal vs Overlap Loop

normal_loop: sequential — schedule, run forward, process results. CPU overhead blocks GPU.

overlap_loop: GPU runs batch N while CPU processes results of batch N-1, using two CUDA streams synchronized via stream.wait_stream().

scheduler.py — overlap_loop()

```
def overlap_loop(self, last_data: ForwardData | None) -> ForwardData | None:
    blocking = not (last_data or self.prefill_manager.runnable
                    or self.decode_manager.runnable)
    for msg in self.receive_msg(blocking=blocking):
        self._process_one_msg(msg)

    forward_input = self._schedule_next_batch()
    ongoing_data = None
    if forward_input is not None:
        with self.engine_stream_ctx: # GPU stream
            self.engine.stream.wait_stream(self.stream)
            ongoing_data = (forward_input, self._forward(forward_input))

    # CPU processes LAST batch's results while GPU runs CURRENT batch
    self._process_last_data(last_data, ongoing_data)
    return ongoing_data
```

scheduler.py — normal_loop() [for comparison]

```
def normal_loop(self) -> None:
    blocking = not (self.prefill_manager.runnable or self.decode_manager.runnable)
    for msg in self.receive_msg(blocking=blocking):
        self._process_one_msg(msg)

    forward_input = self._schedule_next_batch()
    ongoing_data = None
    if forward_input is not None:
        ongoing_data = (forward_input, self._forward(forward_input))

    self._process_last_data(ongoing_data, None) # sequential, no overlap
```