ChatGPT

Flex-Nano-vLLM: A Minimal vLLM-Style Inference Engine with FlexAttention

Introduction

Flex-Nano-vLLM is a minimal inference engine that implements ideas from the vLLM project using PyTorch's **FlexAttention** mechanism. It is designed for efficient serving of large language model (LLM) requests with high throughput and dynamic batching. The core idea is to manage the attention key/value cache in **pages** (fixed-size blocks of memory) and use a **paged attention** mechanism so that multiple requests can share the same attention computation without duplicating memory. This approach allows serving many concurrent sequences on a single GPU with minimal overhead, achieving performance close to the production vLLM system (within ~90% of its throughput).

Key concepts in Flex-Nano-vLLM:

- *FlexAttention & BlockMask:* FlexAttention is a PyTorch mechanism that generalizes attention via **block-sparse masking**. Instead of a standard attention mask, it uses a **BlockMask** object to define which query positions can attend to which key positions in blocks. This allows efficient skipping of "empty" attention blocks and flexible memory management. Each BlockMask contains structures (`kv_num_blocks`, `kv_indices`, etc.) indicating which blocks are present or masked out. Full blocks (completely visible) and partial blocks (partially visible, require runtime mask function) are distinguished for efficiency.

- *Paged Key-Value Cache:* Instead of storing each sequence's key/value cache contiguously (as in standard decoding), all sequences share a large **paged KV cache** tensor. Memory within this tensor is managed in fixed-size *pages* (e.g. blocks of 128 tokens). A **PageTable** keeps track of which pages belong to which sequence. This way, when some sequences finish, their pages can be recycled for new sequences, enabling dynamic batching of many requests without running out of memory.

- *Dynamic Batching & Preemption:* The engine supports adding new requests on the fly and reusing freed space from completed requests. If the cache is full and a new request needs space, the engine can **preempt** (temporarily evict) a newer request's pages to serve an older one that's still running. This ensures high GPU utilization by keeping as many requests as possible in flight.

In this tutorial-like report, we'll walk through the implementation of flex-nano-vLLM **block by block**. We assume you are an advanced Python programmer interested in building an LLM inference engine from scratch, so we'll focus on the core design and algorithms. Some sections of code will be presented and explained in detail, and a few parts will be left as "homework" exercises for you to consider implementing or researching further. The presentation mixes explanation with code (much like a Markdown notebook) to make it easy to follow along. Let's get started!

FlexAttention and BlockMask Basics

Before diving into page management, it's important to understand how **FlexAttention** works, since it's the backbone of our attention implementation. In PyTorch, `torch.nn.attention.flex_attention` provides an attention kernel that can utilize a `BlockMask` to efficiently handle sparse attention patterns.

BlockMask: In FlexAttention, instead of a boolean mask matrix, we use a `BlockMask` object that describes allowed query-key interactions in terms of blocks. By default, blocks are 128×128 (though configurable). A BlockMask holds several attributes (with shapes shown for a typical use):

- `seq_lengths`: a tuple indicating (Q_length, KV_length) of the full sequence.
- `kv_num_blocks` (shape `[B, H, Q_blocks]`): For each query block (per batch `B` and head `H`), how many key blocks are valid.
- `kv_indices` (shape `[B, H, Q_blocks, MAX_KV_blocks]`): The indices of those key blocks that are valid for each query block.
- `full_kv_num_blocks` and `full_kv_indices`: similar structures for any *full* (unmasked) blocks that can be treated as fully visible (to skip runtime mask checks).
- `mask_mod`: a callable that determines the mask at finer granularity for partial blocks at runtime (e.g., ensuring causal order within a block).
- `BLOCK_SIZE`: the tuple of block sizes (usually `(128, 128)` for Q and KV dimensions).

In simpler terms, the BlockMask encodes which blocks of keys each block of queries can attend to. **Full blocks** (completely visible) are stored in `full_kv_indices`, **empty blocks** (completely masked) are omitted entirely, and **partial blocks** are stored in `kv_indices` with a `mask_mod` function to mask individual positions as needed. This block-sparse representation allows FlexAttention to skip computation for empty regions and handle partial regions with minimal overhead.

For a **causal self-attention** (no sequence can attend to tokens after itself), the BlockMask will mark as valid only those key blocks up to the query's current block. During autoregressive decoding, one can **slice** the BlockMask to the latest query block and adjust the `mask_mod` so that each query position only attends to previous (or same) positions [1]. In code, this slicing is done via `block_mask[:, :, block_offset]` which effectively creates a new BlockMask for the next token query [2].

Creating BlockMasks: PyTorch provides a helper `create_block_mask(causal_fn, ...)` to generate a BlockMask for a given masking function (like causal lower-triangular mask). However, for performance, one can also construct BlockMask manually if the pattern is simple (as done in this project for causal masks). For example, a minimal causal BlockMask can be created with a lambda `mask_mod = lambda b,h,q_idx,kv_idx: q_idx >= kv_idx` to mask out future tokens [3].

FlexAttention Call: When calling `flex_attention(query, key, value, block_mask=..., ...)`, the BlockMask guides the attention computation. Importantly, FlexAttention allows the **batch dimensions of query and key/value to differ** as long as the BlockMask is set up consistently. This is crucial for our paged cache: we will use `query` of shape `(B, 1, ...)` (B sequences querying one new token each) but `key`/`value` of shape `(1, N, ...)` where `N` is the total length of the **paged cache**. By setting up a BlockMask of shape `[B, ..., KV_LEN=N]` that masks out all irrelevant positions for each query, we can have FlexAttention compute all B queries against the single concatenated key/value buffer correctly. This avoids having to expand the key/value tensor B times, which saves memory.

Summary: FlexAttention with BlockMask gives us the ability to treat a large shared key-value buffer as if it were separate per-sequence caches, by masking. The BlockMask and its `mask_mod` ensure that each sequence's query only "sees" its own keys. In the next section, we'll see how the **PageTable** uses this mechanism to map logical per-sequence positions to the physical shared buffer.

(For a deeper understanding of FlexAttention and BlockMask, including partial blocks and slicing, you may refer to Jonathan Chang's blog post, which explains these in detail. Now, let's move on to the paging system.)

Implementing Paged Attention: PageTable and KV Cache

The heart of flex-nano-vLLM is the **paged attention mechanism**, which involves a **PageTable** to manage memory pages and a **PagedKVCache** to store the model's key/value tensors. Together, these allow multiple sequences to share a single large KV buffer without interfering with each other.

PageTable Data Structures

The `PageTable` class is a modified version of the one in PyTorch's Attention Gym (with improvements and bug fixes) [4] [5]. Its main responsibilities are to allocate/free pages and to translate between **logical indices** (positions within an individual sequence) and **physical indices** (positions in the big shared buffer). Here are the key data structures in `PageTable.__init__`:

```
# In PageTable.__init__()
self.n_pages = n_pages          # total pages available
self.page_size = page_size      # number of tokens per page
self.max_batch_size = max_batch_size
# Main mapping from logical (batch, block) -> physical page
self.page_table = -torch.ones((max_batch_size, n_pages), dtype=torch.int64,
device=device)
self.page_table[0, :] = 0  # reserve page 0 for padding/no-op
# CPU copy of page allocations for quick access (list of allocated pages per
batch)
self.page_table_cpu = [[] for _ in range(max_batch_size)]
# Capacity (in tokens) allocated per batch index
self.capacity = [0 for _ in range(max_batch_size)]
# Free lists for pages and batch indices
self.free_pages = list(reversed(range(1, n_pages)))          # physical pages
available (exclude 0)
self.free_batch_idx = list(reversed(range(1, max_batch_size)))  # batch slots
available (exclude 0)
# Mapping from (batch, physical_page) -> logical_page
self.physical_to_logical = -torch.ones((max_batch_size, n_pages),
dtype=torch.int64, device=device)
```

[6] [7]

A brief explanation of these fields:

- `page_table[batch_idx, block_idx] = physical_page_idx`: This 2D tensor is the **core page table**. For each logical batch and each logical block index within that batch, it stores the physical page number where that block's data is stored. Unused entries are `-1`. Page 0 is reserved and set to 0 (this simplifies some indexing in code by having a dummy page 0) [8].
- `page_table_cpu`: A Python list that mirrors page allocations for each batch index (used to quickly free pages on the CPU side without GPU memory copies [9]).
- `capacity[batch_idx]`: The current number of **tokens** that batch index has space for (always a multiple of page_size). Essentially `capacity = (number of pages allocated) * page_size` [10].
- `free_pages`: A stack (list) of physical page indices that are free. Initialized with all pages 1..N-1 (since 0 is reserved) [11].
- `free_batch_idx`: A stack of available batch indices (1..max_batch_size-1, with 0 reserved) [11].
- `physical_to_logical[batch_idx, phys_page] = logical_block_idx`: The inverse mapping of `page_table`. This helps translate a physical KV index back to a logical index (used for masking, as we'll see) [12].

  Note: Batch index 0 and page index 0 are reserved for special use (padding/no-op) to simplify certain operations. By avoiding real data at index 0, the code can use index 0 for out-of-bound or dummy references safely.

Allocating and Reserving Pages

When a new sequence request comes in, we need to allocate a fresh batch slot (logical batch index) and assign it some pages to store its tokens. The methods `allocate()` and `reserve()` handle this:

- `allocate()`: Grabs a free batch index from `free_batch_idx`, resets its capacity and mappings, and returns the batch index.

```
def allocate(self) -> int:
    batch_idx = self.free_batch_idx.pop()                # get a free batch slot
    self.capacity[batch_idx] = 0                         # no tokens allocated yet
    self.physical_to_logical[batch_idx, :] = -1          # reset mappings
    self.page_table[batch_idx, :] = -1                   # reset page table
entries
    return batch_idx
```

[13]

When a batch is allocated, it starts with 0 capacity and no pages assigned (all entries -1). The `allocate` method also clears any stale mappings from previous use of that batch index.

- `reserve(batch_idx, seq_len)`: Ensures that the given batch has capacity for at least `seq_len` tokens. This may allocate additional pages if needed. Key steps in `reserve` include calculating how many new pages are required and updating the mappings. Here's a simplified version of the `reserve` logic:

```
def reserve(self, batch_idx_int: int, batch_idx_tensor: Tensor, seq_len: int) ->
bool:
    if seq_len <= self.capacity[batch_idx_int]:
        return True  # already have enough capacity
    # Compute number of additional pages needed
    needed = seq_len - self.capacity[batch_idx_int]
    num_pages_to_allocate = _cdiv(needed, self.page_size)  # ceiling-divide
    if num_pages_to_allocate > self.pages_available:
        return False  # not enough free pages (or raise RuntimeError in non-dry-
run)
    # Determine which new physical pages to use
    start_page_idx = self.capacity[batch_idx_int] // self.page_size
    end_page_idx   = start_page_idx + num_pages_to_allocate
    allocated_pages_list = self.free_pages[-num_pages_to_allocate:]
    allocated_pages = torch.tensor(allocated_pages_list, device=self.device)
    # Update page_table for these logical blocks
    self.page_table[batch_idx_tensor, start_page_idx:end_page_idx] =
allocated_pages
    # Map physical pages back to logical
    self.physical_to_logical[batch_idx_tensor, allocated_pages] = torch.arange(
        start_page_idx, end_page_idx, device=self.device)
    # Update CPU metadata and free lists
    self.page_table_cpu[batch_idx_int] += allocated_pages_list
    self.free_pages = self.free_pages[:-num_pages_to_allocate]   # remove
allocated pages
    self.capacity[batch_idx_int] += num_pages_to_allocate * self.page_size
    return True
}
```

In words, `reserve` first checks if the sequence already has enough capacity. If not, it computes how many pages are needed (`num_pages_to_allocate`). It then verifies that many pages are available in `free_pages`. If not, the reservation fails (the real code raises an error in this case [14], unless we're doing a "dry run" check). If pages are available, it:

1. **Allocates pages:** Determines the logical page range for the new pages (`start_page_idx` to `end_page_idx`). It pops the needed number of physical page IDs from the end of the `free_pages` list (this acts like a stack) [15] [16].

2. **Updates mappings:** Inserts the new physical page IDs into the `page_table` at the appropriate positions for this batch [17]. Also updates `physical_to_logical` for the inverse mapping of each allocated physical page [18].
3. **Updates metadata:** Stores the allocated pages in the CPU list for that batch (`page_table_cpu`) [16], removes them from `free_pages` [16], and increases the batch's `capacity` by `page_size * num_pages_to_allocate` [16].

After `reserve`, the sequence can safely use positions up to `capacity` tokens without running out of allocated space. This design of allocating pages *only when needed* (and possibly incrementally) means we don't over-reserve memory for sequences that might end early.

> **Homework Exercise:** The `reserve` method in this minimal engine always allocates new pages sequentially and does not support complex cases like **prefix sharing** between sequences (where two requests share the same beginning tokens to save memory). In a full vLLM, a hashing scheme can detect identical prefixes and avoid duplicating them. Consider how you might extend `PageTable` to support prefix sharing – what additional mapping or reference counting would you need? (Hint: you'd need a way to let two logical sequences map to the same physical pages until they diverge.)

- `erase(batch_idx)`: Frees all pages used by a finished batch so they can be reused. This simply takes all pages listed in `page_table_cpu[batch_idx]` and pushes them back onto the `free_pages` list (making sure to reverse the order, since we treat it like a stack) and marks the batch index as free:

```python
def erase(self, batch_idx: int) -> None:
    self.free_batch_idx.append(batch_idx)
    allocated_pages_cpu = self.page_table_cpu[batch_idx]
    self.free_pages.extend(reversed(allocated_pages_cpu))
    self.page_table_cpu[batch_idx] = []
    # (page_table and physical_to_logical entries for this batch will be reset
on next allocate)
```

[19] [20]

After `erase`, the batch index can be recycled (put back into `free_batch_idx`), and all its pages are available for new allocations. The actual GPU memory in those pages is not zeroed out here (that would be wasted effort), but since the PageTable mappings are cleared on next allocate, any stale data will be overwritten when pages get re-used.

Updating the Key/Value Cache

Now that we have a paging system, how do we actually store and retrieve the model's key and value vectors in this paged structure? This is handled by the **PagedKVCache** class and the `assign` method of PageTable.

PagedKVCache: Each Transformer attention layer will have its own PagedKVCache (for keys and values of that layer). In `Inference.__init__`, right after creating the PageTable, the code attaches a PagedKVCache to every attention layer in the model:

```
for layer in self.model.model.layers:
    layer.self_attn.kv_cache = PagedKVCache(
        self.page_table,
        n_heads=self.model.model.config.num_key_value_heads,
        head_dim=self.model.model.config.head_dim,
        dtype=self.model.dtype,
    ).to(self.device)
```

[21]

The `PagedKVCache` is basically a container for two large tensors: `k_cache` and `v_cache`, each of shape **[1, H, N, D]** where `H` is number of heads, `N = n_pages * page_size` is the total number of token slots, and `D` is the head dimension. The first dimension is a dummy batch dimension (set to 1 because we treat the whole cache as one big batch) [22]. These caches are registered as buffers (so they reside on the GPU) [22]. Initially, they are all zeros.

The `PagedKVCache.update(...)` method is used to write new key/value vectors into the cache via the PageTable. Its logic is:

```
def update(self, input_pos, k_val, v_val, batch_idx=None):
    assert batch_idx is not None, "batch_idx required for paged cache"
    if batch_idx.ndim == 1:
        # Decode step (each sequence adds one token)
        return self.page_table.assign(batch_idx, input_pos, k_val, v_val,
                                      self.k_cache, self.v_cache)
    else:
        # Prefill step (1 x L shape inputs)
        return self.page_table.assign_prefill_no_paging(batch_idx, input_pos,
k_val, v_val,
                                                        self.k_cache,
self.v_cache)
```

[23]

Depending on whether `batch_idx` is a 1D tensor (meaning we're doing a decoding step for B sequences in parallel) or a 2D tensor (meaning we have a single combined prefill batch of length L), it calls either `assign` or `assign_prefill_no_paging` on the PageTable. We'll explain both:

- `PageTable.assign(batch_idx_tensor, input_pos, k_val, v_val, k_cache, v_cache)`: This handles writing new KV values for one decoding step (one token per sequence in the batch). The parameters are:
- `batch_idx_tensor`: shape `[B]` (the batch indices for B sequences).
- `input_pos`: shape `[B, S]` (the positions of the new tokens in each sequence; for decode S=1 typically).
- `k_val, v_val`: the newly computed key and value tensors for those tokens, shape `[B, H, S, D]`.
- `k_cache, v_cache`: the big cache buffers.

The method calculates where in the big buffer to store each `k_val` and `v_val` slice. **This is the core of logical->physical mapping for the KV cache.** A snippet of `assign`:

```
logical_block_idx = input_pos // self.page_size        # [B, S] logical block
indices
logical_block_offset = input_pos % self.page_size      # [B, S] offset within
block
physical_block_idx = torch.gather(
    self.page_table[batch_idx], 1, logical_block_idx.to(torch.int64)
).to(torch.int32)                                      # [B, S] physical page
indices
addr = (physical_block_idx * self.page_size + logical_block_offset).view(-1)  #
[B*S] physical positions

# Reshape k_val, v_val to collapse batch and seq dims into one
k_val = k_val.permute(1, 0, 2, 3).contiguous().view(1, H, B*S, D)
v_val = v_val.permute(1, 0, 2, 3).contiguous().view(1, H, B*S, D)
# Write into cache at the computed addresses
k_cache[:, :, addr, :] = k_val
v_cache[:, :, addr, :] = v_val
return k_cache, v_cache  # (for convenience, though not used elsewhere)
```

[24] [25]

Let's break that down. For each sequence (batch entry) and each new token position: - Compute `logical_block_idx` = token_position // page_size, and `logical_block_offset` = position % page_size. - Look up the physical page: `physical_block_idx = page_table[batch_idx, logical_block_idx]`. This uses `torch.gather` to vectorize the lookup for all B sequences and their (possibly different) block indices [26]. - Compute the absolute address in the cache tensor: `addr = physical_block_idx * page_size + offset` for each token [27]. - Reshape the new key/value tensors so that all B*S tokens are in one dimension (this makes the assignment into the cache easier, because `addr` is a flat index). - Write the key and value into the cache at those addresses.

This effectively “appends” the new token’s key/value to each sequence’s slot in the paged cache. For example, if sequence with batch_idx=5 just got a token at position 256 and its page_table says logical block 2 of batch 5 is at physical page 17, then we will write the token’s key/value into `k_cache[0, :, pos, :]` where `pos = 17*page_size + (256 mod page_size)`.

- `PageTable.assign_prefill_no_paging(...)`**:** This is a simpler version for the **prefill** (initial input) stage. In prefill, we might input a whole sequence of length L at once (or even pack multiple sequences together). The “no paging” here means we are not doing any prefix sharing or dynamic assignment – we allocate all the needed pages upfront for the entire sequence. The function basically does a similar mapping: it calculates `physical_kv_idx = page_table[batch_idx, input_pos_block_idx] * page_size + input_pos_offset` for each token in the input, and then writes the `k_val` and `v_val` into those positions [28]. In practice, the code uses the fact that each sequence’s tokens are contiguous in a few pages after allocation, so it writes them in one go.

Important: The assign functions rely on the PageTable having been properly allocated/reserved **before** writing. For example, before calling `assign` for a new decode token at position X, you must have ensured via `reserve()` that the page covering position X is allocated to that sequence. The `Inference` code does this, as we will see, by reserving pages whenever a sequence’s length is about to exceed current capacity.

At this point, we have a way to store and retrieve keys/values for all sequences in one big buffer, and we know which pages belong to whom. Next, we need to use this information to create the correct BlockMask for attention.

Converting Logical to Physical: BlockMask Transformation

We have logical BlockMasks (for example, a causal mask per sequence that assumes a contiguous per-sequence memory layout) and we need to convert them to a BlockMask that applies to the **physical** layout (the big combined buffer). This is done by `PageTable.convert_logical_block_mask(block_mask, batch_idx)`:

```
def convert_logical_block_mask(self, block_mask: BlockMask, batch_idx: Tensor) -
> BlockMask:
    B, H, ROWS, MAX_BLOCKS_IN_COL = block_mask.kv_indices.shape
    # Ensure block sizes match the page size
    if block_mask.BLOCK_SIZE[1] != self.page_size:
        raise RuntimeError("BlockMask block size must equal page size")
    if batch_idx.ndim == 0:
        batch_idx = batch_idx.view(1)
    assert batch_idx.shape[0] == B

    page_table = self.page_table[batch_idx]  # shape [B, n_pages]
    def transform(num_blocks, indices):
        if num_blocks is None:
            return None, None
        new_kv_num_blocks = num_blocks.clone()
        new_kv_indices = torch.zeros((B, H, ROWS, self.n_pages),
dtype=torch.int32, device=indices.device)
        new_kv_indices[:, :, :, :MAX_BLOCKS_IN_COL] = torch.gather(
            page_table, 1, indices.view(B, -1).long()
        ).view_as(indices).to(torch.int32)
        return new_kv_num_blocks, new_kv_indices

    new_kv_num_blocks, new_kv_indices = transform(block_mask.kv_num_blocks,
block_mask.kv_indices)
    new_full_kv_num_blocks, new_full_kv_indices =
transform(block_mask.full_kv_num_blocks, block_mask.full_kv_indices)
    new_mask_mod = self.get_mask_mod(block_mask.mask_mod, batch_idx)
    seq_lengths = (block_mask.seq_lengths[0], self.n_pages * self.page_size)
    return BlockMask.from_kv_blocks(new_kv_num_blocks, new_kv_indices,
                                    new_full_kv_num_blocks, new_full_kv_indices,
                                    block_mask.BLOCK_SIZE, new_mask_mod,
                                    seq_lengths=seq_lengths)
```

[29] [30]

The conversion works as follows: - It takes a logical BlockMask (e.g. of shape `[B, H, 1, L_logical_blocks]` for a single-step query of length 1 block) and a tensor of batch indices (`batch_idx`) corresponding to that BlockMask's sequences. - It asserts that the block size for KV in the BlockMask matches the PageTable's page size (so that KV blocks align with pages) [31]. - It then uses the `page_table` mapping to replace every **logical block index** in the mask with the corresponding **physical page index**. This is done by the `transform` helper: - It creates a new `new_kv_indices` tensor of shape `[B, H, ROWS, n_pages]` (initially zero) and then fills the first part of it (same number of blocks as the original mask had) with the looked-up physical page indices [32]. - Essentially, if a logical BlockMask said "query block 0 can attend to logical key blocks [0, 1, 5]", after conversion it might say "query block 0 can attend to physical key blocks [2, 7, 9]" if those were the pages assigned. - It copies `kv_num_blocks` unchanged, since the count of blocks per query doesn't change – only their identities. - It does the same for `full_kv_indices` (fully visible blocks) if any [33]. - Next, it adjusts the mask function: `new_mask_mod = self.get_mask_mod(old_mask_mod, batch_idx)`. The mask_mod (if not None) is a function that FlexAttention calls for each query-key block pair to determine masking at a finer level. The original mask_mod expects **logical** indices. The new mask_mod needs to translate a physical KV index back to a logical one and then call the original function. The `get_mask_mod` does exactly that by capturing the current batch's mapping [34]. We'll discuss `get_mask_mod` in a moment. - Finally, it calls `BlockMask.from_kv_blocks` to construct a new BlockMask with the transformed indices and mask_mod, and an updated `seq_lengths` reflecting that the KV length is now `n_pages * page_size` (the length of the physical cache) [35].

This conversion allows us to prepare a BlockMask that tells FlexAttention about the *physical* layout of the KV cache. For example, suppose we have 2 sequences in batch, each with a BlockMask that says "attend up to your current position" (causal mask). After conversion, the combined BlockMask will say, for each sequence's query, which physical pages (and which blocks within them) to attend to.

Mask Functions and get_mask_mod: In a BlockMask, `mask_mod(b, h, q_idx, kv_idx)` is a callback that FlexAttention uses to determine if a specific query index can attend to a specific key index (within a partial block). In our case, the logical mask_mod is the causal condition `q_idx >= kv_idx` (for self-attention). However, after mapping to physical indices, we must ensure that: 1. Queries don't attend to keys from other sequences. 2. Within a sequence's own pages, maintain causality.

The PageTable's `get_mask_mod` function wraps an existing mask_mod to add these checks:

```
def get_mask_mod(self, mask_mod, batch_idx: Tensor):
    if mask_mod is None:
        mask_mod = noop_mask  # if no mask, treat as all-True mask
    def new_mask_mod(b, h, q_idx, physical_kv_idx):
        is_valid, safe_logical_kv_idx = self.get_logical_kv_idx(b,
physical_kv_idx, batch_idx)
        return torch.where(
            is_valid,
            mask_mod(b, h, q_idx, safe_logical_kv_idx),   # if the physical
index maps to a valid logical index, apply original mask_mod
            False                                          # if invalid (belongs
to no sequence), mask out
        )
    return new_mask_mod
```

[34]

Here, `get_logical_kv_idx(b, physical_kv_idx, batch_idx)` does the inverse of what `assign` did: - It takes a physical KV index and finds which logical batch and logical index it corresponds to. It uses the `physical_to_logical` mapping to get `logical_block_idx`, then reconstructs `logical_kv_idx = logical_block_idx * page_size + offset` [36]. - It also returns an `is_valid` mask indicating if that physical index was actually assigned (if not, `logical_block_idx` might be -1 meaning no sequence owns that index) [37].

Using this, `new_mask_mod` checks if a given physical KV index is part of the sequence corresponding to batch `b`. If yes, it converts it to the sequence's logical index (`safe_logical_kv_idx`) and calls the original `mask_mod` (e.g. the causal check). If not, it returns False (mask out) [38]. Also note, `batch_idx` here is the tensor mapping *physical batch index* to *logical batch index*. In our use, `b` (the batch parameter in mask_mod) is actually an index into the **physical key tensor's batch**. Since we pass the combined key tensor as batch size 1 (all keys in one group), `b` is usually 0 for all keys. But FlexAttention may conceptually treat different sequences separately for masking, so `batch_idx` helps route the check to the correct logical sequence.

The result of all this is that after `convert_logical_block_mask`, we have a BlockMask that we can feed directly to `flex_attention` along with the big `k_cache` / `v_cache` (which have batch dimension 1). FlexAttention will call our `new_mask_mod` for each query and key, which will ensure queries only see their own keys and obey causal order. This is the crux of how we achieve **paged attention**: through careful mapping and masking, multiple sequences' attentions are combined into one big attention operation.

Recap: We covered a lot of ground here. We saw how `PageTable` manages pages of the KV cache, how new tokens are assigned to the cache, and how we convert a per-sequence attention mask into one that works for the global cache. Next, we will put these pieces together in the **Inference Engine** that orchestrates decoding for multiple sequences at once.

(Feel free to take a moment to digest the paging mechanism. A good exercise is to trace through a simple scenario: say you have `page_size=128`*,* `n_pages=4`*, and two sequences A and B. Sequence A uses pages 1 and 2, sequence B uses page 3. If sequence A has a BlockMask that allows it to attend up to token 130 (which is in logical block 1), what will the physical BlockMask contain? Try to manually map A's logical blocks to physical pages and see how mask_mod ensures A doesn't see B's page.)*

The Inference Engine: Dynamic Batching and Generation Loop

With the low-level paging and attention in place, the remaining challenge is **feeding data to the model and managing multiple sequences**: scheduling new requests, decoding tokens, and handling completion – all in a way that keeps the GPU busy. This is handled by the `Inference` class (in `inference.py`), which provides a high-level `generate()` function.

Sequence Abstraction and Tokenization

The engine defines a simple `Sequence` class to represent an input/output sequence being processed:

```
class Sequence:
    def __init__(self, text: str):
        self.text = text
        self.input_ids = []
        self.input_length = None
        self._output_ids = []
        self._output_logits = []
        self._output_probs = []
        self.finished = False
        self.done = False
        self.batch_idx = None
    def add_next_token(self, token_id, logits, probs):
        self._output_ids.append(token_id)
        self._output_logits.append(logits)
        self._output_probs.append(probs)
    @property
    def output_ids(self):  # returns tensor of output token ids
        return torch.tensor(self._output_ids, dtype=torch.long)
    @property
    def total_length(self):
        return self.input_length + len(self._output_ids)
    @property
    def last_token_id(self):
        return self._output_ids[-1] if self._output_ids else None
```

(simplified for clarity) [39] [40]

A `Sequence` holds the original text and will accumulate generated tokens. The engine uses `sequence.input_ids` (the tokenized input prompt) and internal lists for outputs. We won't delve into all details, but basically after tokenization, `seq.input_ids` is a tensor of the prompt tokens and `seq.input_length` is set. The `add_next_token` method appends a generated token and its logit/prob to the sequence's history [41] .

The `Inference.tokenize()` method simply runs the model's tokenizer on all input texts to produce the `input_ids` for each Sequence [42] .

Initialization and Prefill Phase

When `Inference` is created, it is given: - `model` (a loaded Transformer model, e.g. Gemma2), - `tokenizer` , - `max_batch_size` (max concurrent sequences), - `max_seq_length` (max total length per sequence), - `n_pages` and `page_size` (for the KV cache).

In `Inference.__init__`, it sets up the PageTable and attaches PagedKVCache to each model layer's attention (as we saw) [21] . It also preallocates some helper tensors: - `self.input_pos`: a tensor of shape `[max_batch_size]` to store the current input position for each batch index (this will be used when building BlockMasks for decoding) [43] . - `self.block_mask`: a **full** causal BlockMask for the *logical* space of size `L = max_seq_length`. This is created once by `page_table.create_causal_blockmask(B=max_batch_size, L=max_seq_length)` [44] . Essentially, this is a BlockMask that (if used without paging) would represent a standard causal mask for sequences up to max_seq_length tokens. The engine will slice and reuse this BlockMask for each decoding step (to avoid recreating masks repeatedly).

Prefill (Initial Forward Pass): Before starting the iterative decoding, the engine "prefills" the model with the input prompt tokens for each sequence. This serves two purposes: 1. It populates the KV cache with the embeddings for all prompt tokens. 2. It allows us to retrieve the model's outputs at the end of each prompt, which is where generation will begin.

In a naive approach, we could simply loop through each sequence, feed its prompt into the model, and store the keys/values. But that would be slow for many sequences. Instead, flex-nano-vLLM does **batched prefill**: it concatenates all prompts into one long sequence and does one forward pass! This is possible because the KV cache is paged and can intermix different sequences.

- The engine takes all `sequences` still waiting to be processed, concatenates their token IDs, and creates a tensor `input_ids` of shape `[1, L_total]` (where L_total is sum of lengths) [45] . It also creates a `batch_idx_tensor` of shape `[1, L_total]` which assigns a batch index to each token in that long sequence (e.g., if sequence A has length 50 and B has length 30, the first 50 positions of `batch_idx_tensor` will be A's batch index, next 30 positions will be B's index) [45] .

Similarly, it creates an `input_pos` tensor of shape `[1, L_total]` which is just a range of token positions for each sequence (e.g., 0..49 for A repeated for A's positions, 0..29 for B, etc.) [45]. Essentially, we "pack" multiple sequences into one batch dimension by differentiating them with a batch_idx marker in these tensors.

- It then calls `page_table.reserve` for each sequence to allocate enough pages for the entire prompt (ensuring the KV cache has space) – this is done inside the loop where sequences are allocated a batch_idx.

- Next, it creates a **BlockMask for prefill** using `page_table.create_prefill_blockmask_no_paging(batch_idx_tensor)`. This BlockMask encodes a *documented-packed causal mask*: tokens can attend to earlier tokens *if* they belong to the same sequence (document) [46]. In other words, it's a block mask that enforces causal order and also prevents attention across different sequence's tokens (since different sequences have different batch_idx entries in that tensor). The lambda used inside `create_prefill_blockmask_no_paging` is `document_mask = (docs[q_idx] == docs[kv_idx])` combined with `causal_mask` [46] – exactly to restrict attention to tokens of the same document (sequence).

- Finally, it runs a forward pass through the model with these tensors:

```
outputs = self.model.model(
    input_ids=input_ids,
    position_ids=input_pos + 1,          # position ids (1-based in Gemma2)
    flex_attn_block_mask=mask,           # the BlockMask we created
    flex_attn_input_pos=input_pos,       # positions for KV cache update
    flex_attn_batch_idx=batch_idx_tensor,  # batch indices for KV cache update
    flex_attn_kernel_options=self.kernel_options | {"FORCE_USE_FLEX_ATTENTION":
True}
)
logits = self.model.lm_head(outputs.last_hidden_state[:, logits_to_keep, :])
```

[47] [48]

A few things to note here: - We call `self.model.model(...)` which is the **base model** (Gemma2Model) without the LM head, to get hidden states, then apply `lm_head` to get logits. This is an implementation detail; one could also call the full model and get logits directly. - The `flex_attn_*` arguments we pass are how we hook into our modified Gemma2 model's attention. By providing `flex_attn_block_mask`, `flex_attn_input_pos`, and `flex_attn_batch_idx`, we signal the model's `Gemma2Attention` layers to use our PagedKVCache update and FlexAttention instead of standard attention. Inside the model, as shown earlier, the `Gemma2Attention.forward` checks `if self.kv_cache is not None and flex_attn_input_pos is not None: self.kv_cache.update(...)` [49], which will populate the KV cache for each layer, and then calls `flex_attention(..., block_mask=flex_attn_block_mask, ...)` [50] instead of normal attention. This mechanism ensures that: - Keys/values from this forward pass are written into the global cache at the right spots (via `kv_cache.update`). - Attention is computed correctly respecting the cross-sequence mask (via the block mask). - We use `FORCE_USE_FLEX_ATTENTION=True` in kernel options to force PyTorch to use FlexAttention even for short sequences (it sometimes might default to a faster flash-attn kernel if not forced) [51]. - `logits_to_keep` is a tensor that marks the index of the last token of each sequence in the packed input (essentially the cumulative sum of lengths minus one) [45]. We use this to select the logits corresponding to the end-of-prompt for each sequence (since that's where generation will start). The code above indexes `outputs.last_hidden_state[:, logits_to_keep, :]` and then applies the LM head, so `logits` ends up of shape `[num_sequences, vocab_size]` giving the logits for the next token for each sequence's prompt.

After prefill, each `Sequence` object now has its `batch_idx` (the slot it was assigned) and the KV cache contains all prompt tokens for all layers. We also got the initial logits for each sequence's next token. The engine immediately samples one token for each sequence from those logits:

```
next_token, logits, probs = sample(logits_1LV[0, :, :], to_cpu=True, greedy=...)
for b in range(len(batch)):  # batch here is the list of sequences we just
prefilled
    batch[b].add_next_token(next_token[b], logits[b], probs[b])
self.running.extend(self._check_done(batch))
```

[52] [53]

Here `sample()` is a helper that either picks the argmax token (greedy) or does random sampling from the probability distribution [54]. We then append the generated token to each sequence's outputs. `_check_done(batch)` will mark a sequence as finished if it hit end-of-sequence token, max length, or max_new_tokens, and free its pages via `page_table.erase` [55]. Any sequence not finished is moved into the `running` deque for decoding.

This completes one "prefill step," after which the prompt has been processed and one new token has been generated for each sequence. All sequences are now either finished (if the first generated token was EOS or if no generation was required) or in progress.

Decoding Steps (Autoregressive Generation Loop)

Once prompts are prefilled, the engine enters a loop to generate remaining tokens for sequences that are still running. Each iteration of this loop is a **decode step** where one token per sequence is generated (for those sequences being processed in that iteration).

BlockMask for Decoding

For each decode step, we need to compute attention for the *new token* of each active sequence, given all the past tokens (which are already in the KV cache). We again use FlexAttention with a BlockMask, but now each sequence only has a query length of 1 (the new token) and a key length equal to the number of tokens so far.

The `Inference.get_decoding_block_mask(batch_idx)` constructs a **logical** BlockMask for this scenario, for the given batch of sequences:

```
def get_decoding_block_mask(self, batch_idx: Tensor):
    # batch_idx: [B] (logical batch indices of sequences in this step)
    B = batch_idx.shape[0]
    input_pos = self.input_pos[batch_idx]  # [B] current pos (index of last
token each seq had before this new one)
    def causal_offset(off):
        # returns a mask_mod that shifts the causal mask by offset
        def offset_mask(b,h,q_idx,kv_idx):
            return q_idx + off[b] >= kv_idx
        return offset_mask
    block_mask = self.block_mask  # the precomputed full causal mask
[max_batch, ...]
    input_block_idx = input_pos // block_mask.BLOCK_SIZE[0]  # which block each
seq's last token is in
    kv_num_blocks    = block_mask.kv_num_blocks[batch_idx, :,
input_block_idx].view(B, 1, 1)
    kv_indices       = block_mask.kv_indices[batch_idx, :,
input_block_idx].view(B, 1, 1, -1)
    full_kv_num_blocks, full_kv_indices = None, None
    if block_mask.full_kv_num_blocks is not None:
        full_kv_num_blocks = block_mask.full_kv_num_blocks[batch_idx, :,
input_block_idx].view(B, 1, 1)
        full_kv_indices    = block_mask.full_kv_indices[batch_idx, :,
input_block_idx].view(B, 1, 1, -1)
    seq_length = (1, block_mask.seq_lengths[1])  # 1 query, full length keys
logically
    mask = BlockMask.from_kv_blocks(
        kv_num_blocks, kv_indices, full_kv_num_blocks, full_kv_indices,
        BLOCK_SIZE=block_mask.BLOCK_SIZE,
        mask_mod=causal_offset(input_pos),  # adjust causal mask for each seq's
current offset
        seq_lengths=seq_length
    )
    return mask, input_pos
}
```

[56] [57]

Let's interpret this: - `self.block_mask` is the full causal mask for a sequence of length max_seq_length. By indexing into it with a specific `input_block_idx` (the block index of the *current last token* for each sequence), we effectively slice out the portion of the mask that covers from the beginning up to that block [57]. This gives us a BlockMask where `Q_LEN = 1` block (the next query block) and `KV_LEN` covers all blocks up to the current one for that sequence. - We extract `kv_num_blocks` and `kv_indices` for just that portion for each sequence [57]. If any full blocks are present (earlier blocks that were completely unmasked), we extract those too. - We then set `mask_mod = causal_offset(input_pos)`. Here `causal_offset` builds a function that returns True if `q_idx + offset[b] >= kv_idx`. Why this offset? Because our BlockMask slice starts at a block boundary, we need to ensure within the current block, queries don't attend beyond their current length. For example, if a sequence has length 130 (which is block 1 and offset 2 if block_size=128), we want the new query at position 130 to only see keys $\leq$ 129. The offset (which is the input_pos for that sequence) ensures the mask_mod will mask out keys with index greater than `q_idx + offset` [56]. - Finally, we assemble a new BlockMask via `BlockMask.from_kv_blocks` representing the logical (per-sequence) mask for each sequence's next token attention [58].

This `mask` is still in **logical space** (each sequence's own indices). So immediately after, the code does:

```
mask = self.page_table.convert_logical_block_mask(mask, batch_idx)
```

[59]

This gives the **physical BlockMask** corresponding to the current decode step, mapping each sequence's attention to the global KV cache.

Now we're ready to do the actual attention step.

Single Decode Step Computation

`Inference._decode_step(batch_idx, input_ids)` performs one step of generation for the given batch of sequences (where `input_ids` are the latest token IDs of those sequences). Internally, it builds the BlockMask as above, converts it, and then calls the model:

```
outputs = self.model(
    input_ids=input_ids.view(B, 1),
    position_ids=(input_pos + 1).view(B, 1),
    flex_attn_block_mask=mask,
    flex_attn_input_pos=input_pos.view(B, 1),
    flex_attn_batch_idx=batch_idx.view(-1),
    flex_attn_kernel_options=self.kernel_options,
)
return outputs.logits  # shape [B, 1, vocab_size]
```

[60]

This is very similar to the prefill call, except now `B` could be smaller (if some sequences finished) and we only input one token for each sequence (`input_ids.view(B,1)`). We pass the appropriate flex_attn arguments for the model to use the KV cache: - `flex_attn_block_mask=mask` (the physical mask for current step), - `flex_attn_input_pos` (the positions of the *previous* token, since that's what we just added to cache), - `flex_attn_batch_idx` (the batch IDs).

Inside the model, each layer's attention will do `self.kv_cache.update(flex_attn_input_pos, key_states, value_states, flex_attn_batch_idx)` which writes the new keys/values for this token into the cache [49], and then `flex_attention(query, key_cache, value_cache, block_mask=mask, ...)` which computes attention outputs [50]. Because of how we constructed `mask` and the `mask_mod`, each sequence's query attends to exactly its own past (and nothing more), so the result is the correct attention output for that sequence's new token.

The `_decode_step` returns the logits for the new token(s). Typically it's shape `[B, 1, vocab_size]`. The engine then samples a token for each sequence from these logits (just like after prefill):

```
next_token, logits, probs = sample(logits_BLV[:, -1, :], to_cpu=True,
greedy=...)
for i in range(B):
    batch[i].add_next_token(next_token[i], logits[i], probs[i])
```

[61]

(They use `logits_BLV[:, -1, :]` which is the last token's logits for each sequence – the `-1` index is redundant here since there's only one, but in some cases the model might produce a block of logits.)

After adding the token, the engine again calls `_check_done(batch)` to mark any sequences that have finished and free their pages [62].

CUDA Graphs (Optimization)

One thing you might notice: each decode step involves launching a number of GPU operations (attention, KV cache updates, etc.) and doing it for varying batch sizes as sequences finish. To reduce overhead, flex-nano-vLLM uses **CUDA Graphs** to capture and replay the decode step for fixed batch sizes. The code for this is in `capture_decode_cudagraph()` and `decode_step()` methods [63] [64]. The idea is: - Before generation, warm up the model and capture CUDA graphs for batch sizes 1, 2, 4, 8, ... up to max_batch_size (in increments of 16 in this implementation) [65] [66]. - Store these graphs in a dictionary `self.graphs` keyed by batch size [67]. - During decoding, if `self.cudagraph_captured` is True (user requested graph optimization), then `decode_step` will **reuse** the captured graph for the current batch size instead of launching kernels one by one [68] [69]. It does this by copying the current `input_ids` and `batch_idx` into preallocated memory, and then calling `graph.replay()` [70]. The graph was captured to produce outputs in a preallocated `outputs` buffer, which is then sliced to get the results.

This technique can significantly improve throughput by reducing per-step CPU overhead and ensuring kernels are launched in a fixed sequence. However, it adds complexity and only works when the sequence of operations is identical between steps (hence capturing a few representative batch sizes).

For our understanding, you don't need to replicate the exact CUDA graph logic; you can treat it as an optional optimization. If implementing from scratch, you might first get a working version without CUDA graphs, then add this as an advanced improvement.

(If you're up for a challenge, consider implementing CUDA graph capture for a simple operation and then generalizing it to a whole model forward pass. The tricky parts are managing any input-dependent control flow or memory writes. In flex-nano-vLLM, they handle it by padding the batch to a fixed size and reserving a dummy batch/page index 0 to avoid writing out of bounds.)

Dynamic Batching Scheduler

Finally, let's talk about how the `Inference.generate()` loop intermixes prefilling and decoding for multiple sequences efficiently. The design is such that at each step, the engine will try to utilize any free capacity to start new sequences (prefill) while others are mid-generation, and if memory is low it will juggle (preempt) sequences.

Key variables in `Inference`: - `self.waiting`: a deque of Sequence objects that have not been started yet (just tokenized, waiting for prefill). - `self.running`: a deque of Sequence objects that are currently in generation (some tokens generated but not finished). - `self.done`: a deque for completed sequences.

The main loop in `generate()` is:

```
while self.waiting or self.running:
    step_type = self.run_one_step()
    # ... logging/progress ...
```

[71] [72]

Each iteration calls `run_one_step()`, which decides whether to do a **prefill step** or a **decode step** (or both in some cases). Let's outline `run_one_step`:

1. **Prefill new sequences if possible:** While there are sequences waiting, and while the PageTable has enough free pages to accommodate the next waiting sequence's total length (input + potential output) and we haven't exceeded an optional `prefill_length_limit`, allocate a batch index for a waiting sequence [73]. Then:
2. Pop it from `waiting` (FIFO order) [74].
3. Use `page_table.allocate()` to get a new batch index for it [75].
4. Call `page_table.reserve(batch_idx, seq_len=seq.total_length)` to allocate exactly enough pages for its prompt (and here they did **not** reserve future output tokens; a comment notes they only reserve current seq length to maximize decode batch size, and they rely on dynamic reserve + preemption for growth [76]).
5. Set `seq.batch_idx` and collect the sequence in a `batch` list.
6. Keep track of the sum of prefill lengths to avoid batching too many tokens at once (to control latency spikes) [77].

This loop essentially accumulates as many waiting sequences as it can to prefill together, either limited by memory or by a configured length cap. Suppose we got `k` new sequences in `batch`.

1. **If any sequences were batched for prefill:** do a prefill step:

2. Call `logits_1LV = self.prefill_sequences(batch)`, which does the steps we discussed (one forward pass for all sequences in `batch`) [78].
3. Sample one token for each sequence and add to their outputs [53].
4. Append those sequences to `self.running` if they are not finished (after `_check_done`) [79].
5. Return `"prefill"` to indicate this step was a prefill step (the loop in `generate()` can log this) [80].

Prefill steps take precedence whenever new sequences are waiting and there is space, because we want to start processing new requests as soon as possible.

1. **If no prefill happened (i.e., either no waiting sequences or no space for them):** then proceed to decode for existing sequences:
2. First, for each sequence in `self.running`, check if it needs more pages for its next token. If its current `capacity` is >= current length, it's fine. If not, try `page_table.reserve` to allocate another page for it [81].
3. If `reserve` fails because not enough free pages, then we need to **preempt** a sequence. The strategy (as implemented) is: if we can't allocate a page for the *oldest* running sequence (which we popped from the left of the deque), then we put that sequence back (it can't continue now), and instead we take the **newest** running sequence (right end of deque) and preempt it [82]. Preempting means:
    - Move that newest sequence from running back to waiting (it will be resumed later) [83].
    - Free its allocated pages: `self.page_table.erase(seq.batch_idx)` [84].
    - Mark that sequence's batch index as free, so its pages can now be used by others.
    - Do *not* mark the sequence finished – it's just paused. (Its generated tokens so far stay in its Sequence object, and when it's resumed later, it will reallocate pages and **recompute** its KV cache from scratch for those tokens. This is a trade-off: preemption adds recomputation cost, but frees memory for higher priority sequences. In a more advanced system, you might choose which sequence to preempt based on some policy like newest or least progress.)
4. After ensuring all remaining `running` sequences have the needed pages (or have been preempted), the code collects all currently running sequences into a `batch` list for decoding [85].
5. It then builds the input tensors:
    - `batch_idx` (list of batch indices),
    - `input_ids` (list of last token IDs from each sequence),
    - `input_pos` (list of each sequence's current last token position, i.e., length-1). These are moved to GPU [85].
6. Calls `logits_BLV = self.decode_step(batch_idx, input_ids, input_pos)` which handles the CUDA graph or not and ultimately calls `_decode_step` as needed [86].
7. Samples the next token for each sequence and appends to outputs [61].
8. Updates `self.running` to only include those still not finished (using `_check_done`) [87].
9. Returns `"decode"` to indicate a decode step happened.
10. The loop then repeats. It will keep alternating between prefilling new sequences (when available and memory allows) and decoding existing ones. Notably, the design tries to **always use all available**

memory for decoding by not over-reserving upfront. If memory runs low, it frees some by preempting newer sequences. This way, older sequences (which presumably started earlier) get to finish, and newer ones wait a bit if needed – a reasonable fairness policy.

This dynamic batching approach ensures high throughput: at any given time, the GPU is busy either decoding some batch of sequences or prefilling new ones, until all sequences are done. By combining sequences in prefill and decode steps, the engine amortizes the cost of kernel launches over many sequences.

Implementation Challenges: Managing this scheduler is complex. Offloading sequences (preemption) means you need to later resume them. In this code, a preempted sequence is put back to `waiting`, and when it comes to be processed again, its prompt (and any tokens generated so far) will be re-prefilled from scratch before decoding resumes. This is correctness-critical: if you free a sequence's pages and then later allocate new pages for it, you must recompute its KV cache for the tokens it already had. The code doesn't explicitly show that second prefill – it happens naturally when that sequence comes to the front of the waiting queue again, it will go through the prefill logic as if it's a new request (but starting from where it left off). In practice, this means additional overhead, but if preemption is rare (only in extreme memory pressure), the trade-off can be worth it.

Another challenge is handling extremely long sequences that exceed `max_seq_length` or the assumed block_mask size. The code as given doesn't implement a sliding window or truncation for very long sequences (the author notes sliding window not covered in current usage [88]). Implementing **sliding window attention** (where you discard old tokens from KV cache to keep length bounded) would be another advanced feature you could attempt to add – it would require evicting oldest pages and adjusting the BlockMask accordingly.

Finally, ensuring that consecutive calls to `generate()` produce identical results (determinism) required a subtle fix: the `mask_mod` needed to incorporate the actual batch index mapping, not just assume `b` matches logical batch. The author discovered a bug where outputs differed in later calls because `mask_mod` was using physical index `b` incorrectly. The `get_mask_mod` we saw, which maps `b` to logical via `batch_idx`, fixes this. This highlights the importance of careful index bookkeeping in such a system.

Putting it All Together

Let's walkthrough a **simplified example** of how all these pieces interact:

- Say `max_batch_size=3` and we have 3 requests: A (length 100), B (length 50), C (length 80). We set `n_pages` such that maybe two sequences can fully fit but not all three simultaneously (to illustrate preemption).

- Initially, `waiting=[A,B,C]`, `running=[]`. Memory free = all pages.

- **Step 1 (Prefill):** `run_one_step` sees `waiting` not empty. It tries to allocate A, B, C. Suppose A and B fit without hitting `prefill_length_limit`, but maybe allocating C would exceed memory. So it batches A and B:

- Allocates batch_idx 1 for A, reserves pages for 100 tokens.
- Allocates batch_idx 2 for B, reserves pages for 50 tokens.
- Leaves C in waiting because perhaps adding C's 80 would exceed memory (or it might add C too if memory allowed).
- Prefill batch = [A, B]. It runs them together through the model (packed input of length 150). Both get their KV caches filled and one token generated each.
- A and B now in running (unless one finished immediately which is unlikely as they just started).

- Returns "prefill".

- **Step 2 (Prefill or Decode):** Now `waiting` has [C], `running` has [A, B].

- The loop iterates again. It will attempt to prefill C (since waiting not empty). It checks `can_reserve` for C's 80 tokens. If enough pages are free (we allocated pages for A and B, but perhaps there's still room, especially if we allowed all three, but let's assume memory is tight).
- If there is space, it will prefill C similarly (allocate batch_idx 3, etc.). Now running = [A, B, C].
- If there was not space for C's entire prompt, it might hold off (though in this simple approach they pre-allocate prompt space fully).

- Let's assume it *did* prefill C as well in second step (or C might be done in a third step if needed). So now all three are running.

- **Subsequent Steps (Decode):** Now waiting is empty, running = [A, B, C].

- The engine moves to decode steps. It will prepare to decode next token for A, B, C together (batch of 3).
- It checks capacity for each: all have capacity exactly equal to prompt length from reserve. As soon as they try to generate token #2, each needs one more page (if prompt lengths were exactly multiples of page_size, maybe not immediately; but eventually they will).
- Suppose A needs a new page (say page_size=128, A=100 tokens fits in one page so far, adding one token still in page 1, okay; B 50 tokens in page 1, fine; C 80 tokens, fine too). So maybe none need new page at token 2.
- It then does decode step with batch [A, B, C], generates token 2 for each.
- Next iteration, still all running. After a few tokens, eventually one sequence might hit EOS or length limit. Or memory might become an issue if their lengths grow.
- If memory becomes an issue (say after some decoding, A grows beyond 128 tokens and needs a second page, and similarly C does, and free_pages gets low), the code might preempt one. It will try to reserve for A, B, C; maybe by the time pages are 90% used, it preempts the newest (which would be C, since running deque is [A (oldest), B, C (newest)]). Then C is moved back to waiting.

- Later, when either A or B finishes (freeing some pages), C will be resumed by allocate & prefill of its remaining tokens (which means recompute its KV from scratch up to the point it was preempted – that is a drawback but acceptable if preemption is rare).

- The loop ends when both waiting and running are empty, meaning all sequences done.

Throughout, metrics are collected (counts of running/waiting, etc.) for analysis, and the code can print timing stats at the end [89] [90].

Conclusion and Next Steps

Flex-Nano-vLLM demonstrates that with around 1000 lines of code, one can implement a fairly sophisticated LLM inference engine that achieves near state-of-the-art performance. We explored how it uses **FlexAttention** with custom BlockMasks to enable a **paged KV cache**, and how it implements a scheduler for dynamic batching and preemption.

To recap, the critical components we examined were: - **BlockMask and FlexAttention:** enabling masking of a single large attention buffer so multiple sequences don't interfere. - **PageTable:** managing fixed-size memory pages, allocation, and mapping between logical (per-sequence) and physical addresses. - **PagedKVCache:** a simple wrapper holding big key/value tensors and using PageTable to place new keys/values [24] [25]. - **Inference Engine:** orchestrating the process – tokenizing input, prefilling the model, then alternating between decoding steps and adding new requests, with careful memory management (reserving and freeing pages, and even offloading sequences when necessary).

There are several **implementation challenges** and possible extensions left as "homework" or future work:

- **Sliding Window / Long Context Support:** The current engine assumes sequences fit in the given `max_seq_length`. Implementing a sliding attention window (dropping oldest tokens and reusing pages) would be required for truly long streams or chatbot-like continuous conversation. This would involve updating the BlockMask creation to maybe use a rolling cache and adjust positions, which is non-trivial.

- **Prefix Sharing:** As noted, the engine does not attempt to detect and merge identical prefixes among different requests. In production systems, if two users ask the same question, it can save compute to run the prompt once for both. Adding this would require hashing prompts and a more complex page-table reference counting (so that two sequences can point to the same physical page until they diverge).

- **Multi-GPU or Multi-Node Scaling:** Flex-Nano-vLLM is single-GPU. Extending it to multiple GPUs (model parallel or data parallel for batch) would introduce challenges in how to partition the KV cache and coordinate page tables across devices. The nano-vLLM project (which inspired this) has some multi-GPU support that could serve as a reference.

- **Unified Prefill and Decode Steps:** The blog notes that vLLM can intermix prefill and decode in one scheduler step (doing both in parallel), whereas this simpler engine separates them, causing more sequential steps especially when many new requests arrive. An advanced exercise is to try to overlap the prefill of some requests with the decode of others in one combined operation, further increasing throughput (but requiring even more complex scheduling logic).

- **Error Handling and Robustness:** We glossed over some error cases (like running out of memory pages leading to exceptions). A robust server would need to handle these gracefully (maybe queue requests or spill to CPU). You might consider how to integrate such policies.

We encourage you to experiment with this codebase and even try implementing some of the above features as an exercise. Many components, like the `PageTable`, are self-contained – for instance, you could write a small script to unit test allocating, reserving, and freeing pages to ensure it behaves as expected. Likewise, verifying that BlockMask conversions are correct (e.g., by comparing the output of a single-sequence model vs. the paged model) is a great way to gain confidence in the implementation.

By understanding and building these pieces yourself, you'll gain insight into how high-performance LLM serving systems work under the hood. Happy coding!

Sources: This explanation is based on the flex-nano-vLLM repository [91] [92] and the accompanying blog post by Jonathan Chang, which provides a step-by-step reasoning of the design. Each code snippet above corresponds to actual code in the repository for authenticity. For further reading, you may refer to the original vLLM paper and the Attention Gym tutorial to deepen your understanding of the techniques used. Good luck with implementing your own inference engine!

---

[1] [2] vLLM from scratch with FlexAttention – Jonathan Chang's Blog
https://jonathanc.net/blog/vllm-flex-attention-from-scratch

[3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [46] GitHub
https://github.com/changjonathanc/flex-nano-vllm/blob/98f98ea9640e681fa4fec120b2a60c9ca3ed93cd/flex_nano_vllm/paged_attention.py

[21] [39] [40] [41] [42] [43] [44] [45] [47] [48] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [89] [90] GitHub
https://github.com/changjonathanc/flex-nano-vllm/blob/98f98ea9640e681fa4fec120b2a60c9ca3ed93cd/flex_nano_vllm/inference.py

[49] [50] [88] modeling_gemma2.py
https://github.com/changjonathanc/flex-nano-vllm/blob/98f98ea9640e681fa4fec120b2a60c9ca3ed93cd/flex_nano_vllm/modeling_gemma2.py

[91] [92] GitHub - changjonathanc/flex-nano-vllm: FlexAttention based, minimal vllm-style inference engine for fast Gemma 2 inference.
https://github.com/changjonathanc/flex-nano-vllm