ChatGPT

Study Plan: Implementing FlexNanoVLLM from Scratch

Objective: Build up the knowledge and skills to implement a minimal **vLLM-style inference engine** (like *FlexNanoVLLM*) from scratch using PyTorch **FlexAttention**. This plan breaks down the journey into sequential exercises, each introducing key concepts and providing starter code. By completing these exercises, you will progressively gain the expertise needed to implement a FlexNanoVLLM-like system on your own.

Overview and Prerequisites

Before diving in, ensure you have a solid understanding of basic (non-optimized) Transformer inference. You should be comfortable with standard Transformer generation (using caching for past key/value states) and basic PyTorch operations. We will use **PyTorch 2.x** (for the FlexAttention API) and assume access to a *Qwen-3* model (e.g. **Qwen3-0.6B** from Hugging Face) as the base transformer implementation for experiments [1]. An **NVIDIA GPU** is needed to fully utilize FlexAttention and measure performance.

> **Note:** FlexNanoVLLM is a minimal high-throughput inference engine inspired by the vLLM project. It avoids custom CUDA kernels or FlashAttention, instead leveraging PyTorch's FlexAttention API for efficient **paged attention** [2] [3]. Essentially, it achieves ~90% of vLLM's performance in ~1000 lines of Python code [4]. This study plan will guide you through understanding and implementing the core ideas behind such an engine.

1. Revisit Standard Transformer Inference (Baseline)

Goal: Make sure you understand how standard (non-optimized) Transformer inference works, including the use of KV caching for decoding. In this exercise, you'll implement a simple autoregressive generation loop and compare it to Hugging Face's `generate()` for consistency.

Key Concepts to Learn

- How a Transformer model (like Qwen) generates text token-by-token using cached keys/values to avoid recomputation.
- Using Hugging Face Transformers' model API vs. manual loop.

Exercise 1: *Naive Autoregressive Generation with KV Cache*

1. **Load a Pre-trained Model:** Use Hugging Face to load a small model, e.g., `Qwen/Qwen3-0.6B` (or a similar GPT-style model). Make sure to load it in evaluation mode and on GPU.
2. **Manual Generation Loop:** Implement a loop that feeds input prompt tokens through the model to get logits for the next token, samples (or selects top-1) the next token, appends it to the prompt, and

repeats. Use the model's *past_key_values* (KV cache) to avoid re-computing attention for the entire sequence at each step.

3. **Compare with Built-in Generate:** For testing, use the model's `.generate()` on the same prompt and ensure your loop produces the same output sequence (with deterministic settings like `temperature=0`, greedy decoding).
4. **Understand the Output:** Confirm that your manual implementation can generate text correctly and that you understand how the model's internal cache works (the cache should grow with each new token, storing past keys/values for each attention layer).

Below is starter code for the manual generation loop using Hugging Face (for **greedy decoding** for simplicity):

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

# Load tokenizer and model (assumes Qwen3 or a similar small model)
model_name = "Qwen/Qwen3-0.6B"  # example model name
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).eval().to("cuda")

prompt = "Hello, my name is"
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
input_ids = inputs["input_ids"]
attention_mask = inputs["attention_mask"]

# Generate using manual loop
max_new_tokens = 20
output_ids = input_ids.clone()
past_key_values = None

for _ in range(max_new_tokens):
    # Model forward pass with past_key_values for faster decoding
    outputs = model(input_ids=output_ids[:, -1:],
                    attention_mask=attention_mask,
                    past_key_values=past_key_values, use_cache=True)
    next_token_logits = outputs.logits[:, -1, :]
    next_token_id = torch.argmax(next_token_logits, dim=-1)  # greedy pick
    # Append prediction
    output_ids = torch.cat([output_ids, next_token_id.unsqueeze(0)], dim=1)
    # Update past_key_values and attention_mask for next iteration
    past_key_values = outputs.past_key_values
    attention_mask = torch.cat([attention_mask, torch.ones((1,1)).to("cuda")],
dim=1)
# Decode the output ids to text
generated_text = tokenizer.decode(output_ids.squeeze(),
```

```
skip_special_tokens=True)
print(generated_text)
```

Instructions: Run the above and verify it generates text. Compare `generated_text` with `model.generate()` on the same prompt (with `max_new_tokens=20`) to ensure they match. This exercise solidifies your understanding of caching and sequential decoding in a normal setting.

2. Introduction to FlexAttention and BlockMask

Goal: Learn what **FlexAttention** is and how to use it to implement custom attention patterns. Specifically, practice using **BlockMask** – the key feature that allows flexible "paged" attention masking in PyTorch 2.x [5] [6]. By the end of this, you should understand how to reproduce standard causal attention using FlexAttention, as well as more complex masks like sliding windows.

Key Concepts to Learn

- **FlexAttention API:** PyTorch's `torch.nn.attention.flex_attention` function and its arguments (query, key, value, mask functions, etc.) [7].
- **BlockMask:** an object representing a block-sparse attention mask. It divides sequences into blocks (default size 128) and categorizes blocks as *full*, *partial*, or *empty* for efficient masking [8].
- **Mask Functions (mask_mod):** user-defined functions that determine which queries can attend to which keys (for example, a causal mask function might forbid query positions from attending to future key positions).

Understanding BlockMask

A `BlockMask` encodes which blocks of keys each query block can attend to. For example, in causal self-attention: - Queries can attend to all key blocks **up to** their own position (including themselves), but not beyond. - This can be represented by a `mask_mod` function: `mask_mod(b, h, q, kv) -> bool` that returns True if query position `q` is allowed to attend key position `kv` (under some block indexing) [9] [10]. In causal masks, the condition is `q_idx >= kv_idx` [11] (query can attend only to earlier or same-position keys).

PyTorch provides a helper `create_block_mask(mask_mod, B, H, Q_len, KV_len, device)` to generate a BlockMask given: - `mask_mod` – a function that implements the attendance rule, - `B` (batch size) and `H` (number of heads) the mask should cover, - `Q_len` (query sequence length) and `KV_len` (key/value sequence length).

The `flex_attention(query, key, value, *, block_mask=None, mask_mod=None, score_mod=None, ...)` API will use these to apply the specified attention pattern. We can either supply a precomputed `block_mask` or directly give a `mask_mod` (FlexAttention will internally create the BlockMask).

Why FlexAttention? Normally, to customize attention (like implementing *paged attention* or other patterns), one would need to write custom CUDA kernels or use frameworks like Triton. FlexAttention leverages the PyTorch 2.0 compiler (`torch.compile`) to JIT generate optimized kernels from high-level mask/score

functions [7]. This means we can define complex attention behaviors in Python, and PyTorch will generate efficient fused kernels for us (including using FlashAttention under the hood for decoding when appropriate) [12] [13].

Exercise 2: *Using FlexAttention for Custom Masks*

In this exercise, you'll practice with FlexAttention on simple scenarios:

a. Causal Self-Attention with FlexAttention: Use FlexAttention to compute a causal attention output and verify it matches the output of a standard attention for a given query, key, value.

b. Sliding Window Attention: Use a mask that limits attention to a fixed window (e.g., each token can only attend to the last 1024 tokens before it). This simulates how you might restrict context in long sequences.

Steps:

1. **Prepare dummy data:** Create small random tensors for `query`, `key`, `value` of shape `[B, H, N, D]` (batch, heads, seq_len, model_dim). For testing causal attention, you can use `N=5` or `10` and small `D` (like 16) for clarity.
2. **Define a causal mask_mod:** e.g. `lambda b,h,q_idx,kv_idx: q_idx >= kv_idx`. Use `create_block_mask` to make a BlockMask for your `query` and `key` lengths.
3. **Call flex_attention:** Use `out = flex_attention(query, key, value, block_mask=my_block_mask)`. This should apply causal masking. Separately, compute the *ground truth* by doing a manual masked attention (or use PyTorch's `scaled_dot_product_attention` if available) on the same `query, key, value`.
4. **Verify outputs match:** Ensure the difference between FlexAttention output and the reference output is near zero.
5. **Sliding window mask:** Use Attention Gym's helper or write a mask_mod that implements a window (e.g., allow a query at position *q* to attend keys from *q - 1024* to *q*, and mask out keys older than that). Use `flex_attention` with this mask and inspect which keys each query attends (you can check the `block_mask.kv_indices` or compare outputs to expected masked outputs).

Below is some starter code illustrating part (a), causal attention with FlexAttention:

```
import torch
from torch.nn import functional as F
# (Make sure you're using PyTorch 2.5+ for flex_attention)
from torch.nn.attention import flex_attention, create_block_mask

B, H, N, D = 1, 1, 5, 16  # e.g., 1 batch, 1 head, seq_len 5, dim 16
torch.manual_seed(0)
query = torch.randn(B, H, N, D, device="cuda")
key   = torch.randn(B, H, N, D, device="cuda")
value = torch.randn(B, H, N, D, device="cuda")

# Define causal mask function: allow attending to current or previous positions
```

```
def causal_mask(b, h, q_idx, kv_idx):
    return q_idx >= kv_idx

# Create BlockMask for causal attention
block_mask = create_block_mask(causal_mask, B, H, Q_len=N, KV_len=N,
device="cuda")

# FlexAttention output with causal mask
out = flex_attention(query, key, value, block_mask=block_mask)  # shape: [B, H,
N, D]

# Reference: manual causal attention (no softmax for simplicity, just to compare
structure)
scores = torch.matmul(query, key.transpose(-1, -2))  # [B,H,N,N]
mask = torch.tril(torch.ones(N, N, device="cuda"))  # lower triangular causal
mask
scores = scores.masked_fill(mask==0, float('-inf'))
attn_weights = F.softmax(scores, dim=-1)
ref_out = torch.matmul(attn_weights, value)

# Verify the outputs are close
print("Max difference:", (out - ref_out).abs().max().item())
```

Run the above to ensure `Max difference` is very small (close to 0). This confirms `flex_attention` with a causal BlockMask works as expected (essentially replicating standard attention).

Next, try part (b): - For a **sliding window**, PyTorch's Attention Gym provides `generate_sliding_window(window_size)` to get a `mask_mod` function [14]. You could use that or implement a simple version: e.g., allow `kv_idx` from `q_idx - W` to `q_idx` (within bounds). Create a BlockMask with that and use `flex_attention` to see the effect. (This is an open-ended exploration; you can print or visualize which indices attend by looking at `block_mask.kv_indices` content.)

Through this exercise, you become familiar with how FlexAttention uses BlockMasks and `mask_mod` to control attention patterns. You'll need this knowledge to implement **paged attention** in subsequent steps.

3. Understanding Paged Attention (vLLM's Approach to KV Caching)

Goal: Grasp the concept of **Paged Attention** – the key technique behind vLLM's high-throughput inference – and how it can be implemented with FlexAttention. Paged attention manages the KV cache by splitting it into fixed-size blocks (pages) and reusing them across different requests, enabling efficient memory management and high parallelism [3] [15].

Key Concepts to Learn

- **Page (Block) Size:** Typically a power-of-two (e.g. 128 tokens). The sequence length dimension is divided into pages of this length.

- **Page Table:** An indexing structure that keeps track of which *pages* of the KV cache belong to which sequence (similar to virtual memory page tables). This allows dynamically assigning pages to new sequences once old ones finish.
- **BlockMask in Paged Attention:** Instead of a full causal mask over a monolithic sequence, we use a BlockMask that marks entire pages as present or not for each sequence's query.
- *Full blocks:* completely filled with past tokens (except maybe the last partially filled page).
- *Partial block:* the current page where not all positions are filled yet (for a sequence still generating).
- *Empty blocks:* unused slots which should be masked out (not attended to) [8].

In vLLM, each decoding step involves one token per active sequence (so the query is of length 1 for each active sequence). **FlexNanoVLLM** and **nano-vLLM** mimic this by batching those 1-token queries together, and using a BlockMask to ensure each query token only attends to its own sequence's keys in the KV cache [16]. The BlockMask essentially acts as a **sparse attention mask** over a large combined KV buffer: - It prevents query tokens from different sequences from seeing each other's keys. - It skips over unused cache slots.

How BlockMask encodes pages: In FlexAttention, a BlockMask has tensors `kv_num_blocks` and `kv_indices` that indicate which key-value blocks are valid for each query block [17]. For example, `kv_indices` might list the page indices that a given query (block) should attend to. The optional `full_kv_indices` can be used to mark whole blocks that are fully valid without needing to call the `mask_mod` (saves compute) [8].

To make this concrete, let's consider a simple scenario: - Page size = 128. Each sequence's context and generation will use chunks of 128 token slots. - Suppose Sequence A is 250 tokens long (i.e., 2 full pages of 128 and 1 partial page with 250-256 = -6 unused slots in the last page), Sequence B is 90 tokens (1 full page, 1 partial page of 90). - If we allocate 3 pages for A and 2 pages for B in a common KV cache, we might have: - Pages 0-2 belong to A (with page 2 partially filled), - Pages 3-4 belong to B (page 4 partially filled). - For a new token query of A (in page 2) and B (in page 4) in a batch, the BlockMask should let A's query attend pages 0,1 (full) and 2 (partial up to 250th position), and B's query attend pages 3 (full) and 4 (partial up to 90th position), but not attend each other's pages. This is achieved by setting `kv_indices` appropriately for each batch element.

Exercise 3: *Simulate Paged KV Allocation and Masking*

In this exercise, you will simulate a simple page allocation and create a corresponding BlockMask manually (to solidify understanding before coding the real thing).

Steps:

1. **Define Page Parameters:** Decide on a page size (e.g. `P = 128`). Define a maximum number of pages for simplicity (e.g. `max_pages = 10` total available).
2. **Simulate Sequences:** Create two or three example sequences with different lengths. Determine how many pages each sequence would need (pages = ceil(seq_len / P)).
3. **Allocate Pages:** Assign each sequence a set of page indices. (For now, do this statically: e.g., Seq1 gets pages 0..m, Seq2 gets pages m+1..n, etc.)
4. **Create BlockMask Data:** Manually construct the `kv_indices` and `kv_num_blocks` tensors for a batch that contains one query token for each sequence:

5. `kv_num_blocks[batch_index, head_index, q_block_index]` should equal the number of (partial) blocks accessible to that query.
6. `kv_indices[batch_index, head_index, q_block_index, :]` should list the actual page indices that the query can attend.
7. Tip: Since each query here is just the last token of its sequence, you effectively have one query **block** per sequence. So `q_block_index=0` for each batch element (assuming one block of queries per sequence in this step).
8. Mark full vs partial blocks: For each sequence, all pages except the last are full; the last page is partial. In BlockMask, you might put full pages in `full_kv_indices` (so they are always attended fully) and the last page in `kv_indices` with a `mask_mod` that masks out the unused positions beyond the sequence length [18] [19].
9. **Verify logically:** Ensure that if you were to use this mask in flex_attention, each query would attend exactly its own sequence's keys. (We won't run flex_attention here, just simulate the data structures.)

This is a **pen-and-paper (or code comments) exercise** to map out how the data structures should look. You can represent `kv_indices` as a 2D list for each sequence. For example, you might end up with something like:

```
# Pseudo-structure for BlockMask representation (conceptual)
kv_indices = {
    "SeqA": [0, 1, 2, None, None, ...],  # attends pages 0,1,2; rest empty
    "SeqB": [3, 4, None, None, ...],     # attends pages 3,4; rest empty
}
```

Where `None` or an undefined value represents unused slots in the indices tensor for those queries.

No starter code provided here, since this is about reasoning through the structure. Write down (or code and print) the values you expect for `kv_num_blocks` and `kv_indices` for your example sequences. You might also sketch a small diagram of pages vs sequences to visualize it. This mental model will guide your implementation in code.

Outcome: You should now conceptually understand how a BlockMask can represent multiple sequences in a single attention call, isolating their attention to their own pages. This understanding is crucial for implementing the actual multi-sequence decoding loop.

4. Implementing a Paged KV Cache Structure

Goal: Start coding the core data structures needed for a paged attention inference engine. This includes a **Page Manager** (to allocate/free pages for sequences) and storing the actual KV tensors in a format compatible with FlexAttention.

Key Concepts to Learn

- **KV Cache Tensors:** In a naive implementation, each sequence has its own `[seq_len, D]` key and value tensors per head. In paged attention, we instead maintain a large preallocated tensor for keys and values (per head) and assign slices of it (pages) to different sequences on demand [20] [21].

- **Page Manager:** A component that tracks which pages are in use and which are free, and maps sequences to their list of pages. It might have methods like `allocate(num_pages)` and `free(sequence_id)`.

We will implement a simplified version: - Assume a fixed maximum number of concurrent sequences and a fixed total number of pages (for simplicity). - Use a **bitmap or list** to track free pages. - Use Python dictionaries to map `seq_id -> list_of_pages`.

Additionally, we need a way to store keys and values. A simple approach: - Maintain `keys` and `values` as tensors of shape `[Num_pages * P, H, D]` (if we flatten all pages for all heads). Alternatively, keep it as `[H, Num_pages * P, D]` or split by head. For clarity, we might manage per-head caches separately, shape `[Num_pages, P, D]` for each head, and then index into those.

When we perform attention with FlexAttention: - **Query**: will come from the model's output for the new token(s), shape `[Batch, H, 1, D]` (for decoding one token per sequence). - **Key** and **Value**: we can provide the entire cache (all pages) as shape `[1, H, KV_len, D]` where `KV_len = Num_pages * P` (concatenated all pages). But the BlockMask will ensure each query only "sees" the correct subset of this big KV tensor.

Memory Layout Consideration: The simplest is to treat the combined KV cache as if batch=1 (since FlexAttention can take a BlockMask that has B>1 to differentiate sequences). For instance:

```
# keys: shape [1, H, total_cache_length, D]
# values: shape [1, H, total_cache_length, D]
```

Where `total_cache_length = max_pages * P`. The block mask will map each query in the *batch dimension* to different segments of this length.

Important: Ensure that when a sequence finishes (reaches <EOS> or max tokens), you mark its pages as free (so they can be reused by new sequences). Our simplified plan might not implement full *preemption* (where running sequences get evicted and recomputed) [22], but we should handle finished sequences freeing memory.

Exercise 4: *Code a Simple PageManager and KV Cache*

Steps:

1. **Define PageManager class:** with fields like `page_size`, `total_pages`, an array or list `free_pages`, and a dict `allocations` mapping `seq_id` to list of pages.
2. Implement `allocate(n_pages)` that pops `n_pages` from `free_pages` and returns them (and marks them allocated).
3. Implement `free(seq_id)` that takes the pages for that sequence, and puts them back into `free_pages`.

4. **Initialize KV cache tensors:** e.g., `total_cache_length = total_pages * page_size`. Create `self.k_cache` and `self.v_cache` as PyTorch tensors of shape `[H, total_cache_length, D]` (we can add the batch dimension when calling flex_attention).
5. **Write functions to set and get slices:** e.g., a method to write a key/value tensor into a given page range, and another to read from it. When the model produces new key/value for a token, you'll use this to insert that into the global cache at the right location.
6. **Basic Test of Allocation:** Simulate allocating pages for a couple of sequences, write some dummy values into their slots, and ensure you can retrieve them correctly by indexing into the big `k_cache` / `v_cache`.

Below is a skeleton to get you started:

```
import torch

class PageManager:
    def __init__(self, num_pages: int, page_size: int, hidden_dim: int, n_heads:
int):
        self.num_pages = num_pages
        self.page_size = page_size
        self.hidden_dim = hidden_dim
        self.n_heads = n_heads
        self.free_pages = list(range(num_pages))  # all pages initially free
        self.allocations = {}  # seq_id -> list of allocated page indices
        # Combined KV cache (for simplicity, one tensor for keys and one for
values)
        total_cache_length = num_pages * page_size
        # We'll use shape [n_heads, total_cache_length, hidden_dim]
        self.k_cache = torch.zeros(n_heads, total_cache_length, hidden_dim,
device="cuda")
        self.v_cache = torch.zeros(n_heads, total_cache_length, hidden_dim,
device="cuda")

    def allocate_pages(self, seq_id: str, num_pages: int):
        assert num_pages <= len(self.free_pages), "Not enough free pages!"
        pages = [self.free_pages.pop(0) for _ in range(num_pages)]
        self.allocations[seq_id] = pages
        return pages

    def free_pages(self, seq_id: str):
        pages = self.allocations.pop(seq_id, [])
        # return pages to free list (optional: sort or maintain order)
        for p in pages:
            self.free_pages.append(p)
        # maybe sort free_pages for consistency
        self.free_pages.sort()

    def store_kv(self, seq_id: str, seq_k: torch.Tensor, seq_v: torch.Tensor):
```

```
        # seq_k, seq_v shape: [n_heads, seq_len, hidden_dim]
        pages = self.allocations[seq_id]
        # Assuming seq_len <= len(pages)*page_size
        # Copy seq_k and seq_v into the global cache at the positions for those
pages.
        for i, page in enumerate(pages):
            start = page * self.page_size
            end = start + self.page_size
            # Determine portion of seq_k/v to copy (if last page, maybe not
full)
            seq_start = i * self.page_size
            seq_end = min(seq_start + self.page_size, seq_k.shape[1])
            length = seq_end - seq_start
            if length <= 0:
                break
            self.k_cache[:, start:start+length, :].copy_( seq_k[:,
seq_start:seq_end, :] )
            self.v_cache[:, start:start+length, :].copy_( seq_v[:,
seq_start:seq_end, :] )

    def get_page_indices(self, seq_id: str):
        # Return the list of page indices for this sequence
        return self.allocations.get(seq_id, [])
```

Instructions: Fill in any missing parts (like handling partial last pages in `store_kv`). Create a `PageManager` (e.g., `mgr = PageManager(num_pages=10, page_size=128, hidden_dim=hidden_dim, n_heads=num_heads)`). Allocate pages for two dummy sequences, and simulate storing some keys/values (perhaps use random tensors for `seq_k` / `seq_v` of a certain length). Then, verify that those values appear in the global `mgr.k_cache` at the expected positions. This will give you confidence that your page allocation and KV placement logic works.

5. Prefill Phase Implementation

Goal: Implement the **prefill phase** of inference using your paged cache. The prefill (or initialization) phase takes the initial prompt tokens of each sequence, runs them through the model to generate initial keys and values (KV cache), and prepares everything for iterative decoding.

In vLLM and similar systems, the prefill is often done in one batch per request to maximize efficiency. That is, if a sequence has N prompt tokens, we encode all N in one forward pass (rather than token-by-token) and populate the KV cache with those results [23]. We then use the final hidden state as the starting point for generation.

Key Concepts to Learn

- **Batched Prefill:** Executing the model on the input prompts possibly with padding. We need to capture the keys and values for each token in the prompt for caching.

- **Integrating with PageManager:** After running the model, we must take the output *keys and values* from each Transformer layer and store them in the global page cache via the PageManager. (This may require hooking into the model's forward pass to extract those, or running the model layer-by-layer. For simplicity, you might modify the model's attention module to store into our cache directly, but that can be complex. A simpler approach: run the model normally, then copy data into the cache.)

Approach: We will do a simpler version: use the model's `.forward` to get outputs and **manually copy** the KV cache. We know HuggingFace models return `past_key_values` for each layer. Each element of `past_key_values` is a tuple `(key, value)` for that layer, typically of shape `[batch, head, seq_len, head_dim]`. We can reshape those and use `PageManager.store_kv` for each sequence.

Exercise 5: *Prefill Implementation*

Steps:

1. **Allocate Pages for New Sequences:** Suppose we get one or more sequences with their prompt tokens (say we have their tokenized `input_ids`). For each sequence, determine how many pages it needs for the prompt (use `ceil(len(prompt) / page_size)`) and allocate that many pages via `PageManager.allocate_pages(seq_id, n_pages)`. Keep track of how many *tokens* of the last page are actually used (this will inform the mask for that page).
2. **Run Model Forward:** Concatenate all sequences' prompts into a batch and run a forward pass through the model to get `past_key_values`. You might need to pad the prompts so they have equal length in the batch. Alternatively, process each sequence separately (simpler but less efficient).
3. **Populate Global KV:** For each sequence *i* in the batch:
4. Extract its `past_key_values` from the model output. (In HuggingFace, `outputs.past_key_values` is a list of length = number of layers, each an (key, value) pair. The shapes might be `[batch, head, seq_len, head_dim]`.)
5. For each layer and each head, take the key and value for sequence *i* (index into batch dimension), reshape to `[head, seq_len, head_dim]` (since batch is 1 for that seq), and use `PageManager.store_kv(seq_id, seq_key, seq_value)`. This will copy the prompt's KV into the correct global positions for that sequence.

6. **Initial logits/output:** Also take the model's final layer output (logits for next token) for each sequence. This will be used to pick the first generated token in decode phase.

7. **Testing Prefill:** Try a simple test: Suppose sequence A = "Hello", sequence B = "Hi". Run prefill for both together (they have different lengths). Verify that:

8. Each sequence got a unique set of pages.
9. The keys/values in those pages correspond to the prompt encoding (you might not easily verify content without deeper checks, but at least sizes and nonzero vs zero in expected places).
10. You have the next-token logits to begin decoding.

Below is a conceptual code snippet focusing on steps 1-3:

```
# Assume `model`, `tokenizer`, and `PageManager` (mgr) are already set up as
before.
prompts = ["Hello, how are you?", "Hi there!"]  # example prompts for two
sequences
inputs = tokenizer(prompts, return_tensors='pt', padding=True)
input_ids = inputs["input_ids"].to("cuda")
attention_mask = inputs["attention_mask"].to("cuda")
batch_size = input_ids.shape[0]

# Allocate pages for each sequence
seq_ids = [f"seq{i}" for i in range(batch_size)]
for seq_id, ids in zip(seq_ids, input_ids):
    prompt_length = (ids != tokenizer.pad_token_id).sum().item()
    n_pages = (prompt_length + mgr.page_size - 1) // mgr.page_size
    mgr.allocate_pages(seq_id, n_pages)

# Run the model's forward to get outputs and past_key_values
with torch.no_grad():
    outputs = model(input_ids=input_ids, attention_mask=attention_mask,
use_cache=True)
past_kvs = outputs.past_key_values  # list of (key, value) for each layer

# For each sequence in batch, store its keys and values into the global cache
n_layers = len(past_kvs)
for layer_idx, (layer_keys, layer_values) in enumerate(past_kvs):
    # layer_keys shape: [batch, heads, seq_len, head_dim]
    # For each sequence in batch:
    for seq_idx, seq_id in enumerate(seq_ids):
        seq_k = layer_keys[seq_idx]    # shape [heads, seq_len, head_dim]
        seq_v = layer_values[seq_idx]  # shape [heads, seq_len, head_dim]
        # Store this layer's KV for the sequence's pages (you might extend
PageManager to handle per-layer storage)
        mgr.store_kv(f"{seq_id}_layer{layer_idx}", seq_k, seq_v)
```

Instructions: The above is a simplified view. In practice, you might include the layer index in the `seq_id` when storing, or maintain separate PageManagers per layer. A straightforward approach is to include the layer dimension in the `k_cache` / `v_cache` (e.g., have `k_cache[layer][head][position]`). For simplicity, you could create a list of PageManagers, one per layer.

Test the prefill by checking that `mgr.allocations` has entries for each sequence ID and that their allocated pages count matches the expected number. You can also check one of the stored key tensors: pick a position from the middle of a prompt and verify that the key at that position in the global cache equals the key computed by the model for that prompt (if you can extract it).

After this step, you have an initialized paged KV cache for all active sequences, and initial logits for the next token of each sequence.

6. Decoding Phase Implementation (Iteration with FlexAttention)

Goal: Implement the **decode loop** where, at each step, you generate new tokens for all sequences and update the KV cache using FlexAttention for efficiency. This is the heart of the vLLM-style system: using one forward pass per iteration for *all active sequences' new tokens*, rather than one per sequence.

Key Concepts to Learn

- **Batching across sequences:** Even if sequences start and end at different times, the engine can batch their decoding steps. For now, we assume all sequences start together (simpler), but you should design the logic such that if one sequence finishes earlier (EOS token), it can be removed from the batch and others continue.
- **Using FlexAttention for new tokens:** Instead of running a full model forward for one token (which would recompute attention over the entire sequence), use FlexAttention to attend the single new token query to the existing KV cache. This requires constructing the query vectors and using the BlockMask appropriately.
- **Two-phase inference:** Each iteration might involve two parts:
- **Attention computation for new token:** The new token's query at each layer attends to past keys/values (which reside in our global cache). We use `flex_attention` for this, which will leverage the optimized *FlexDecoding* kernel because our query length is 1 and key length is large [24] [13] .
- **Feedforward and output logits:** After attention, the model will apply the MLP (feed-forward network) for that layer, then move to the next layer. Ultimately we get logits for the vocabulary to pick the next token.

In practice, you might integrate with the model's code by replacing its attention mechanism with a custom one that pulls from our global cache. To keep the exercise manageable, we can do the decoding loop manually layer by layer: - We have the model's weights (Wq, Wk, Wv, Wo for each attention layer, etc.). We can manually compute the new query's key/value at each layer: - For layer `ℓ`, take the hidden state (initially the embedding of the last input token for each sequence or the output from previous layer for that token), compute Q, K, V = linear projections. - K and V for the *new token* we will append to the global cache (the new token occupies the next slot in its sequence's last page, or a new page if the last page was full). - Use `flex_attention(new_query, full_K_cache, full_V_cache, block_mask=...)` to get the attention output for this new token query. `full_K_cache` / `full_V_cache` here are the big combined ones in our PageManager. The BlockMask should be constructed such that: - For each sequence (batch element), it allows the query to attend exactly the pages that sequence has *so far*, including the current new token's page (as partial). - This means you'll update the BlockMask as the sequence grows (the partial page gains one more valid token). - The result of flex_attention is the aggregated context for the new token at layer `ℓ`. Then apply the output linear projection (Wo) to get the layer's output for that token. - Pass this as input to layer `ℓ+1`. - After the final layer, apply the LM head to get logits and sample the next token for each sequence. - Insert those new tokens' key/value into the cache (the `store_kv` function) and update the BlockMask (if a new page got allocated when an old page filled up, etc.). - Repeat until max tokens or all sequences hit EOS.

This is quite complex to implement fully, but we can outline a simplified version for the exercise.

Exercise 6: *Decode One Step with FlexAttention*

We'll do a single decoding step for all active sequences using FlexAttention, assuming the prefill is done.

Steps:

1. **Prepare query hidden state:** Take the final hidden state of the model for each sequence's last token (from prefill) as the initial hidden state for decoding step 1. (If you don't have it saved, you can get it by running the model one more time for the last token, or by storing it during prefill.)
2. **Loop over each Transformer layer:**
3. Compute the query, key, value for the new token via that layer's weights (use the model's `attn` weights if accessible, or reuse its `forward` in a partial way).
4. Retrieve the global K and V cache for **that layer** (from your PageManager or equivalent structure).
5. Build a BlockMask for this layer's attention:
    - The mask should have shape `[Batch, Heads, 1_block, KV_blocks]`. Each batch element's single query block can attend to the blocks in its allocation. Use the `PageManager.get_page_indices(seq_id)` for each sequence to know which page indices to include.
    - Use a `mask_mod` or manual BlockMask creation. One strategy: use `BlockMask.from_kv_blocks(...)` if available to directly create from indices. Or simpler, use `create_block_mask` with a custom `mask_mod` that checks if `kv_idx` belongs to that sequence's pages and is <= that sequence's current length.
    - PyTorch's `create_block_mask` might not directly take different per-sequence conditions easily; it might be simpler to manually assemble the `kv_indices` tensor. However, an easier path: since we have separated sequences by batch index, we can design `mask_mod` to encode page logic by embedding sequence info in `kv_idx`. (Alternatively, call `flex_attention` separately per sequence in a loop for clarity – but that loses the batching benefit.)
6. Call `flex_attention(query, K_cache, V_cache, block_mask=...)` to get the context vector for each sequence's new token at this layer.
7. Apply the layer's output projection and add any residual/normalization as needed (depending on model architecture).
8. **Compute next token logits:** After the last layer, take the output hidden states (one per sequence) and apply the language model's final projection to get logits. Determine the next token for each sequence (e.g., argmax for now or sampling).
9. **Update KV cache:** For each sequence:
10. Get the new token's key/value (from each layer's computation in step 2 – those were the `K` and `V` for the new token). Use `PageManager.store_kv` to put them in the global cache at the appropriate position (which will be `page * P + offset` corresponding to the new token's position).
11. If a sequence's last page became full with this new token, and the sequence is not finished, allocate a new page for it for future tokens.
12. Update the BlockMask or the info needed to generate it next time (the sequence's length increased by 1, and possibly its page count increased by 1 if a new page was added).

13. **Repeat:** Loop back to step 1 for the next decoding iteration (with updated hidden states). Continue until you reach the desired number of tokens or EOS. In a full implementation, you would manage sequences finishing at different times: if one sequence outputs an EOS, you would remove it from the batch and maybe recycle its pages immediately or after the step.

Since a full coding of this is lengthy, here's a **pseudo-code snippet** focusing on one layer's decode with FlexAttention to illustrate the idea:

```
# Pseudo-code for one decoding iteration (all sequences)
batch = len(seq_ids)  # active sequences
new_tokens = []  # will store next token for each sequence

# Assume we have `hidden_states` from last iteration (or prefill output) of
shape [batch, seq_len=1, hidden_dim]
hidden_states = last_hidden_states  # for first iteration, this is from prefill
(each seq's last token)

for layer_idx in range(n_layers):
    # Obtain weights for this layer (from the model) - pseudocode
    Wq, Wk, Wv, Wo = model.layers[layer_idx].attn.Wq, ...  # etc.
    # Compute Q (for the new token) [batch, heads, 1, head_dim]
    Q = (hidden_states @ Wq.T).view(batch, n_heads, 1, head_dim)
    # Compute K_new, V_new for the new token
    K_new = (hidden_states @ Wk.T).view(batch, n_heads, 1, head_dim)
    V_new = (hidden_states @ Wv.T).view(batch, n_heads, 1, head_dim)
    # Store K_new, V_new into global cache at the proper position for each seq
    for seq_idx, seq_id in enumerate(seq_ids):
        mgr.store_kv(f"{seq_id}_layer{layer_idx}", K_new[seq_idx],
V_new[seq_idx])
    # Prepare full K_cache, V_cache for this layer (combine all pages)
    K_cache_full = mgr.k_cache  # shape [n_heads, total_cache_length, head_dim]
(for this layer)
    V_cache_full = mgr.v_cache  # similar shape
    # We might need to add batch dimension of 1 around K_cache_full for
flex_attention
    K_cache_full = K_cache_full.unsqueeze(0)  # shape [1, n_heads,
total_cache_length, head_dim]
    V_cache_full = V_cache_full.unsqueeze(0)
    # Construct BlockMask for this layer's attention
    block_mask = create_block_mask(
        lambda b,h,q_idx,kv_idx: mask_rule(b, h, kv_idx),  # define mask_rule
appropriately
        batch, n_heads, Q_len=1, KV_len=mgr.num_pages * mgr.page_size,
device="cuda"
    )
    # Use FlexAttention to get context for the new token
    context = flex_attention(Q, K_cache_full, V_cache_full,
```

```
block_mask=block_mask)  # [batch, n_heads, 1, head_dim]
    # Merge heads and apply output projection Wo
    context = context.view(batch, 1, n_heads * head_dim)
    hidden_states = context @ Wo.T  # [batch, 1, hidden_dim] -> this becomes
input to next layer
    # (Add layer norm, residual connection if applicable, or handle outside the
loop)
```

Instructions: This pseudo-code glosses over many details (like residual connections, layer norm, how to maintain separate caches per layer in the PageManager). For the exercise, **focus on a single layer**: assume no feed-forward or norm for now, and test that flex_attention returns expected results. For example, use a tiny model (one layer, one head) and manually verify that decoding one token via flex_attention yields the same result as doing the attention "the normal way." You can reuse some code from Exercise 2 to verify correctness.

The main result of this step is that you conceptually implement the decoding using the global KV cache and FlexAttention. Once you have this working for one step, extending to a full generation loop is mostly about repeating and handling book-keeping (which you can do as an additional challenge).

7. Testing and Benchmarking Your Implementation

Goal: Validate that your FlexNanoVLLM-like implementation is **correct** and evaluate its **performance** relative to a baseline.

Key Validation Checks

- **Correctness of output:** For a few test prompts, compare the generated text from your implementation with that from the standard `model.generate()` (with the same randomness settings). They should match or be very close. Minor differences can occur due to floating-point variations (as noted by Jonathan Chang [25]), but the overall output should generally align if no sampling randomness is introduced.
- **Consistency across calls:** Reset and generate the same prompt twice; ensure your engine yields the same result each time. This catches any stateful bugs (like not properly resetting or reusing pages).
- **Edge cases:** Test a prompt that is longer than one page to ensure page transitions work. Test multiple sequences of vastly different lengths together to see if padding and mask logic holds up.

Performance Measurement

Measure how fast your implementation generates tokens vs. a naive approach: - Time your `generate()` loop for a batch of, say, 4 sequences with 512 input tokens each, generating 128 new tokens each. - Time the same using `model.generate()` or a simple loop for each sequence. - If possible, compare to `nano-vllm` or `vLLM` on the same hardware for a rough reference. (Keep expectations reasonable: your pure Python implementation might be a bit slower, but it should be in the same ballpark if FlexAttention is working well.)

If you have incorporated `torch.compile` (via `torch.compile(flex_attention)` or compiling your step function), ensure to enable it and measure again. **FlexAttention** compiled with the FlexDecoding

backend can significantly speed up decoding of single tokens [26]. Similarly, you can try using CUDA Graphs to capture the decode loop if batch sizes remain constant (nano-vLLM did this for every 16 tokens of batch growth [27], but you can try a simpler fixed graph capture for your loop).

Exercise 7: *Validation and Profiling*

1. **Correctness Test:** Use a fixed prompt (or a few) and run both your implementation and the HuggingFace model to generate, say, 20 tokens. Compare results. If they differ, investigate where (perhaps print intermediate logits or check if the BlockMask might be off by one position).
2. **Throughput Test:** Use `time.time()` or `torch.cuda.Event` timing around the generation loop to compute tokens per second. Compare with a baseline. If you have access to vLLM or nano-vLLM, run their generation on the same prompt and measure for reference.
3. **Scaling Test (Optional):** Increase the number of concurrent sequences and see if your implementation scales as expected. (The benefit of vLLM is more pronounced with many concurrent requests.) Try 50+ short sequences in parallel generation and see if it's much faster than running them sequentially.

No code is explicitly provided here to avoid clutter, but you can use standard Python timing and reuse your generation function from earlier exercises.

Tip: You might find it useful to integrate parts of existing projects for validation: - The [**nano-vLLM** repo][48] has a simple API where you can plug in a model and compare outputs. - The [**flex-nano-vllm** repo][5] by Jonathan Chang is a great reference to compare your design and to debug issues. Since it's ~1000 LOC, you can read through it to see how it handles certain details (like mask creation, two-phase fusion of prefill+decode, etc.). For example, their testing procedure (comparing to HF outputs) is something you can emulate [28].

8. Further Study and Next Steps

By completing the above exercises, you should have a working understanding of how to implement a minimal vLLM-like inference engine with FlexAttention. You will have essentially created a prototype that: - Uses **FlexAttention** for efficient decoding, avoiding the need for custom kernels [2]. - Manages a **paged KV cache** for multiple concurrent sequences, akin to vLLM's core innovation [15]. - Delivers performance improvements by batching decoding steps and using PyTorch 2.x compilation features (you've touched on `torch.compile` and maybe CUDA graphs).

Next steps and enhancements: - **Unified Prefill+Decode:** In production vLLM (and as noted by Jonathan) the prefill and decode can be unified in one scheduling loop to reduce overhead [29]. You could experiment with processing new requests even while others are mid-generation. - **Memory preemption:** Implement evicting and recomputing cache pages if you run out of memory (allowing more concurrent sequences by trading off recomputation) [22]. - **Advanced mask patterns:** Explore using `score_mod` in FlexAttention for techniques like logit bias or more complex masking beyond causal (FlexAttention allows arbitrary score modification in the attention softmax [30]). - **Multi-GPU or Tensor Parallelism:** If you have larger models, splitting layers or heads across GPUs as nano-vLLM does (basic tensor parallelism) could be a topic to explore [31] [21]. - **Model compatibility:** Try integrating a different model architecture (with FlashAttention, multi-query attention, etc.) and adapt your engine. The flexibility of FlexAttention should handle many variants as long as you set up the masks appropriately [32].

Throughout this journey, refer to the resources and inspiration repositories: - **FlexNanoVLLM (Jonathan Chang)** – *FlexAttention based minimal vLLM* [2] – for a concise reference implementation. - **nano-vLLM (GeeeekExplorer)** – for a pure-Python/triton perspective and educational code structure [20] [21]. - **Attention Gym (PyTorch Labs)** – for more examples of creating masks and using FlexAttention in various scenarios [14].

By iteratively building and testing with these exercises, you will have gained a deep understanding of high-performance LLM inference. Good luck, and enjoy the process of building your own FlexNanoVLLM!

Sources:

- PyTorch FlexAttention tutorial and docs [24] [13] [6]
- FlexNanoVLLM (Jonathan Chang) repository and blog [2] [3]
- NanoVLLM blog (HuggingFace) [20] [21]
- Attention Gym repository (FlexAttention examples) [14]

---

[1] [15] [20] [21] [31] ⚡ nano-vLLM: Lightweight, Low-Latency LLM Inference from Scratch
https://huggingface.co/blog/zamal/introduction-to-nano-vllm

[2] [3] [32] GitHub - changjonathanc/flex-nano-vllm: FlexAttention based, minimal vllm-style inference engine for fast Gemma 2 inference.
https://github.com/changjonathanc/flex-nano-vllm

[4] [5] [6] [8] [9] [10] [16] [17] [18] [19] [22] [23] [25] [27] [28] [29] [30] vLLM from scratch with FlexAttention – Jonathan Chang's Blog
https://jonathanc.net/blog/vllm-flex-attention-from-scratch

[7] [11] [12] [13] [24] [26] FlexAttention Part II: FlexAttention for Inference – PyTorch
https://pytorch.org/blog/flexattention-for-inference/

[14] GitHub - meta-pytorch/attention-gym: Helpful tools and examples for working with flex-attention
https://github.com/meta-pytorch/attention-gym