# nano-vLLM: Continuous Batching and Scheduling

## Overview

**Continuous Batching** is a dynamic batching strategy where sequences can be added or removed at any iteration, rather than waiting for an entire batch to finish processing.

---

## The Scheduler

The scheduler manages two queues and enforces resource constraints:

```python
class Scheduler:
    def __init__(self, block_manager, max_num_seqs, max_num_batched_tokens):
        self.block_manager = block_manager
        self.max_num_seqs = max_num_seqs            # Max sequences per batch
        self.max_num_batched_tokens = max_num_batched_tokens  # Token budget

        self.waiting = deque()   # Sequences waiting to be processed
        self.running = deque()   # Sequences currently being processed
```

### The `schedule()` Method

Each iteration, the scheduler decides what to process next:

```python
```

```python
def schedule(self) -> tuple[list[Sequence], bool]:
    """
    Schedule sequences for the next iteration.

    Returns:
        (sequences_to_process, is_prefill)
    """
    scheduled_seqs = []
    num_seqs = 0            # Resets to 0 every iteration
    num_batched_tokens = 0

    # ─────────────────────────────────────────────────────────────
    # STEP 1: TRY TO ADD NEW SEQUENCES (PREFILL)
    # ─────────────────────────────────────────────────────────────
    while self.waiting and num_seqs < self.max_num_seqs:
        seq = self.waiting[0]

        # Check constraint 1: Token budget
        if num_batched_tokens + len(seq) > self.max_num_batched_tokens:
            break

        # Check constraint 2: Memory availability
        if not self.block_manager.can_allocate(seq):
            break

        # Both constraints satisfied — add this sequence
        num_seqs += 1
        self.block_manager.allocate(seq)
        num_batched_tokens += len(seq) - seq.num_cached_tokens
        seq.status = SequenceStatus.RUNNING
        self.waiting.popleft()
        self.running.append(seq)
        scheduled_seqs.append(seq)

    # If we added any new sequences, return them for prefill
    if scheduled_seqs:
        return scheduled_seqs, True  # True = prefill mode

    # ─────────────────────────────────────────────────────────────
    # STEP 2: PROCESS EXISTING SEQUENCES (DECODE)
    # ─────────────────────────────────────────────────────────────
    while self.running and num_seqs < self.max_num_seqs:
        seq = self.running.popleft()
```

```python
        # Check if we can append a new token
        while not self.block_manager.can_append(seq):
            # Need to free up space (preemption)
            if self.running:
                self.preempt(self.running.pop())
            else:
                self.preempt(seq)
                break
        else:
            # Can append — schedule this sequence
            num_seqs += 1
            self.block_manager.may_append(seq)
            scheduled_seqs.append(seq)

    assert scheduled_seqs  # Must have something to process

    # Restore sequences back to running queue
    self.running.extendleft(reversed(scheduled_seqs))

    return scheduled_seqs, False  # False = decode mode
```

---

## Key Advantages

### 1. Dynamic Batch Composition

**Traditional (Static) Batching:**

```
Batch 1: [Seq1, Seq2, Seq3, Seq4] → wait for all to finish
Batch 2: [Seq5, Seq6, Seq7, Seq8] → wait for all to finish
```

**Continuous Batching (vLLM):**

```
Iteration 1: [Seq1, Seq2, Seq3, Seq4]
Iteration 2: [Seq1, Seq2, Seq3, Seq4]  → Seq1 finishes
Iteration 3: [Seq2, Seq3, Seq4, Seq5]  → Seq5 added immediately
Iteration 4: [Seq2, Seq3, Seq4, Seq5]  → Seq2 finishes
Iteration 5: [Seq3, Seq4, Seq5, Seq6]  → Seq6 added immediately
```

The batch composition changes dynamically every iteration.

### 2. Priority-Based Scheduling

Every iteration follows this logic:

1. **Try to add new sequences** (prefill) — this has priority
2. **If no new sequences can be added**, process existing ones (decode)
3. **Remove finished sequences** immediately
4. **Loop**

This ensures new requests are prioritized when possible, while existing requests are never starved.

### 3. Immediate Resource Reclamation

**Traditional:**

```
Seq1 finishes → memory sits idle → batch finishes → memory freed
```

**Continuous Batching:**

```
Seq1 finishes → immediately free blocks → next iteration adds Seq5
            (< 20ms delay)
```

---

## Walkthrough Example

**Setup:** 10 sequences arrive, max batch size = 7

### Iteration 1: First Prefill

| Step | Details |
|---|---|
| Scheduler | Tries to add new sequences from `waiting` |
| Result | Adds Seq1–Seq7 (hits max batch size of 7) |
| GPU | Prefills all 7 prompts in ONE forward pass |
| Queues After | `waiting: [Seq8, Seq9, Seq10]`, `running: [Seq1...Seq7]` |

### Iteration 2: Decode (Can't Add New)

| Step | Details |
|---|---|
| Scheduler | Tries to add Seq8 — can't allocate (no free memory) |
| Fallback | Decodes existing sequences instead |
| GPU | Generates 1 token for each of the 7 sequences |
| Result | Each sequence now has 1 generated token |

### Iterations 3–5: Continue Decoding

Each iteration:

- Try to add from `waiting` → still no memory available
- Decode all 7 sequences → generate one more token each
- Check for completion

**After Iteration 5:** Suppose Seq1 and Seq4 finish (EOS or max length).

| Step | Details |
|---|---|
| CPU | Removes Seq1 and Seq4 from `running` |
| CPU | Frees their memory blocks back to the pool |
| Queues After | `waiting: [Seq8, Seq9, Seq10]`, `running: [Seq2, Seq3, Seq5, Seq6, Seq7]` |

### Iteration 6: Add New Sequences (Prefill)

| Step | Details |
|---|---|
| Scheduler | Tries to add from `waiting` — now has free blocks! |
| Result | Allocates Seq8 and Seq9 using freed blocks |
| | Cannot allocate Seq10 (insufficient remaining blocks) |
| GPU | Prefills only Seq8 and Seq9 |
| Queues After | `waiting: [Seq10]`, `running: [Seq2, Seq3, Seq5, Seq6, Seq7, Seq8, Seq9]` |

Note: The other 5 sequences (Seq2, 3, 5, 6, 7) wait until next iteration.

### Iteration 7: Decode Mixed Batch

| Step | Details |
|---|---|
| Scheduler | Tries to add Seq10 — can't (no memory) |
| GPU | Decodes all 7 sequences in ONE forward pass |
| Mixed State | Seq2,3,5,6,7 are on their 6th generated token; Seq8,9 are on their 1st |
| Result | All sequences advance by one token |

---

## Appendix: Batching Implementation

### Prefill Preparation

The `prepare_prefill` method batches multiple sequences with different cache states:

```python
```

```python
def prepare_prefill(self, seqs: list[Sequence]):
    input_ids = []
    positions = []
    cu_seqlens_q = [0]    # Cumulative query lengths
    cu_seqlens_k = [0]    # Cumulative key lengths
    max_seqlen_q = 0
    max_seqlen_k = 0
    slot_mapping = []
    block_tables = None

    for seq in seqs:
        seqlen = len(seq)

        # Only process non-cached tokens
        input_ids.extend(seq[seq.num_cached_tokens:])
        positions.extend(list(range(seq.num_cached_tokens, seqlen)))

        seqlen_q = seqlen - seq.num_cached_tokens  # New tokens (query)
        seqlen_k = seqlen                          # Total context (key)

        cu_seqlens_q.append(cu_seqlens_q[-1] + seqlen_q)
        cu_seqlens_k.append(cu_seqlens_k[-1] + seqlen_k)
        max_seqlen_q = max(seqlen_q, max_seqlen_q)
        max_seqlen_k = max(seqlen_k, max_seqlen_k)

        # Build slot mapping for KV cache
        if not seq.block_table:  # warmup
            continue
        for i in range(seq.num_cached_blocks, seq.num_blocks):
            start = seq.block_table[i] * self.block_size
            if i != seq.num_blocks - 1:
                end = start + self.block_size
            else:
                end = start + seq.last_block_num_tokens
            slot_mapping.extend(list(range(start, end)))

    # Enable prefix caching if needed
    if cu_seqlens_k[-1] > cu_seqlens_q[-1]:
        block_tables = self.prepare_block_tables(seqs)

    # Convert to tensors and move to GPU
    input_ids = torch.tensor(input_ids, dtype=torch.int64, pin_memory=True).cuda(non_blocking=True)
    positions = torch.tensor(positions, dtype=torch.int64, pin_memory=True).cuda(non_blocking=True)
```

```
    cu_seqlens_q = torch.tensor(cu_seqlens_q, dtype=torch.int32, pin_memory=True).cuda(non_blocking=True)
    cu_seqlens_k = torch.tensor(cu_seqlens_k, dtype=torch.int32, pin_memory=True).cuda(non_blocking=True)
    slot_mapping = torch.tensor(slot_mapping, dtype=torch.int32, pin_memory=True).cuda(non_blocking=True)

    set_context(True, cu_seqlens_q, cu_seqlens_k, max_seqlen_q, max_seqlen_k,
                slot_mapping, None, block_tables)
    return input_ids, positions
```

### Decode Preparation

The `prepare_decode` method is simpler — each sequence contributes exactly one token:

```python
def prepare_decode(self, seqs: list[Sequence]):
    input_ids = []
    positions = []
    slot_mapping = []
    context_lens = []

    for seq in seqs:
        input_ids.append(seq.last_token)
        positions.append(len(seq) - 1)
        context_lens.append(len(seq))
        slot_mapping.append(
            seq.block_table[-1] * self.block_size + seq.last_block_num_tokens - 1
        )

    # Convert to tensors and move to GPU
    input_ids = torch.tensor(input_ids, dtype=torch.int64, pin_memory=True).cuda(non_blocking=True)
    positions = torch.tensor(positions, dtype=torch.int64, pin_memory=True).cuda(non_blocking=True)
    slot_mapping = torch.tensor(slot_mapping, dtype=torch.int32, pin_memory=True).cuda(non_blocking=True)
    context_lens = torch.tensor(context_lens, dtype=torch.int32, pin_memory=True).cuda(non_blocking=True)
    block_tables = self.prepare_block_tables(seqs)

    set_context(False, slot_mapping=slot_mapping, context_lens=context_lens,
                block_tables=block_tables)
    return input_ids, positions
```

---

## Worked Example: Batching Two Sequences with Prefix Caching

Consider two sequences with different cache states:

| Sequence | Total Tokens | Cached Tokens | New Tokens |
|---|---|---|---|
| Seq1 | 35 | 16 | 19 |
| Seq2 | 20 | 5 | 15 |

## Step-by-Step Construction

```python
```

```
# Initialize accumulators
input_ids = []
positions = []
cu_seqlens_q = [0]
cu_seqlens_k = [0]

# ────────────────────────────────────────────────────────────
# Process Seq1 (35 tokens, 16 cached)
# ────────────────────────────────────────────────────────────
seqlen = 35
num_cached = 16

input_ids.extend(seq1[16:])          # Tokens 16–34 (19 tokens)
positions.extend(range(16, 35))      # [16, 17, ..., 34]

seqlen_q = 35 - 16  # = 19 new tokens
seqlen_k = 35       # = 35 total context

cu_seqlens_q.append(0 + 19)          # → [0, 19]
cu_seqlens_k.append(0 + 35)          # → [0, 35]

# ────────────────────────────────────────────────────────────
# Process Seq2 (20 tokens, 5 cached)
# ────────────────────────────────────────────────────────────
seqlen = 20
num_cached = 5

input_ids.extend(seq2[5:])           # Tokens 5–19 (15 tokens)
positions.extend(range(5, 20))       # [5, 6, ..., 19]

seqlen_q = 20 - 5   # = 15 new tokens
seqlen_k = 20       # = 20 total context

cu_seqlens_q.append(19 + 15)         # → [0, 19, 34]
cu_seqlens_k.append(35 + 20)         # → [0, 35, 55]
```

### Final Batched Tensors

```
input_ids:   [seq1[16:35], seq2[5:20]]  → 34 tokens total
positions:   [16..34, 5..19]            → 34 positions

cu_seqlens_q: [0, 19, 34]
          ├─ Seq1 queries: indices 0–18  (19 vectors)
```

```
    └─ Seq2 queries: indices 19–33 (15 vectors)

cu_seqlens_k: [0, 35, 55]
    ├─ Seq1 context: 35 tokens
    └─ Seq2 context: 20 tokens
```

#### How FlashAttention Uses This

The model computes Q, K, V projections for all 34 input tokens:

```python
Q = model.q_proj(hidden_states)  # Shape: [34, num_heads, head_dim]
K = model.k_proj(hidden_states)  # Shape: [34, num_heads, head_dim]
V = model.v_proj(hidden_states)  # Shape: [34, num_heads, head_dim]
```

FlashAttention then uses the cumulative sequence lengths to correctly route attention:

| Sequence | Query Vectors | Key Context | What Happens |
|---|---|---|---|
| Seq1 | 19 vectors (new tokens) | 35 tokens (16 cached + 19 new) | Each of 19 queries attends to all 35 keys |
| Seq2 | 15 vectors (new tokens) | 20 tokens (5 cached + 15 new) | Each of 15 queries attends to all 20 keys |

The key insight: **query length ≠ key length** when using prefix caching. FlashAttention handles this via the separate `cu_seqlens_q` and `cu_seqlens_k` arrays, allowing efficient batched attention even when sequences have different cache states.