2.1 Context Windows and the Memory Wall

The agent's state — what it remembers from past steps and what it uses to make the next decision — is the bridge between hardware constraints and agent behavior. This section is about the memory wall: why it exists, what it means in numbers, and how to budget for it in the agent loop.

The two key state mechanisms are **KV cache** (the prefill and decode phases' attention memory) and **context window** (the prompt that feeds the next prefill). They're distinct costs with different scaling properties, and conflating them is a common design mistake.

The KV Cache and Its Footprint

The KV (key-value) cache is the core optimization of autoregressive LLM inference: instead of recomputing the attention keys and values for every token position on every decoding step, you compute them once and keep them in memory. On the second token, you use the KV from token 1 plus the new KV for token 2. On the third token, you use KVs from tokens 1–2 plus the new one. This is why decode is so much faster than prefill — you're amortizing the work.

The KV cache lives in DRAM and is dimensioned by **[batch_size, num_heads, seq_len, head_dim]**. For a typical transformer:

- `batch_size`: 1 on NPU (Chapter 1.3)
- `num_heads`: 16 (common)
- `seq_len`: grows from 1 to context_length as you decode
- `head_dim`: 64 (common)

Per-token KV cache footprint = batch × num_heads × head_dim × 2 (K + V) × dtype_bytes.

For M2M-100 1.2B (16 heads, 64 head_dim) at FP16 (2 bytes):

- Per token per layer: 1 × 16 × 64 × 2 × 2 = **4,096 bytes = 4 KB per token per layer**
- M2M-100 1.2B has 24 encoder + 24 decoder layers; the decoder keeps **48 × 4 KB = 192 KB per token**
- At 128-token context: 128 tokens × 192 KB = **24.6 MB per inference batch**

But M2M-100 is encoder-decoder, so there's a second KV cache: the encoder output, which the decoder's cross-attention reads at every step. The encoder KV is computed once (during prefill) and reused throughout decode, so it doesn't grow with seq_len, but it's identical in size to the self-

attention KV of the decoder at any given encoder context length.

Full M2M-100 decoder KV footprint at T=128 token context and encoder context L=128:

- Self-attention KV: 24 layers × 128 tokens × 4 KB = **12.3 MB**
- Cross-attention KV (encoder output): 24 layers × 128 source tokens × 4 KB = **12.3 MB**
- **Total: ~25 MB per sequence** (FP16)

Now compare to **Phi-3-mini-3.8B**, which uses GQA (grouped-query attention) with 8 KV heads instead of 16:

- Per token per layer: 1 × 8 × 64 × 2 × 2 = **2,048 bytes = 2 KB per token per layer**
- 32 layers × 2 KB = **64 KB per token**
- At 128-token context: 128 × 64 KB = **8.2 MB** (before any encoder overhead)

So Phi-3-mini saves 3× on KV footprint per token, because it halves the KV head count. M2M-100 has full MHA and pays the bandwidth price.

The Attention Wall

The attention wall is simple to state: **at some context length, the KV cache's bandwidth demand exceeds what the NPU can sustain**. On Lunar Lake with 136.5 GB/s platform bandwidth, and given the 18% utilization we saw in Chapter 1.3, the per-NPU effective bandwidth is roughly 136.5 × 0.18 ≈ **~25 GB/s available**.

For M2M-100 decoder at FP16:

- 192 KB per token (self + cross attention, 48 decoder+encoder layers)
- At **6.10 tok/s**: 192 KB × 6.10 = **~1.17 MB/s** of KV cache bandwidth

This is well below the 25 GB/s ceiling, so the M2M-100 KV cache isn't the bottleneck yet. The wall appears at much larger context lengths or larger models.

The working hypothesis from Chapter 2.1 is that the KV cache wall appears somewhere between 2K and 8K tokens for typical 8B models on Lunar Lake, depending on model architecture. Intel's validated 8K context "preview" on Lunar Lake is right at that edge. The wall doesn't mean you *can't* have 8K; it means you're committing to recompute, sliding windows, or multi-GPU distribution to stay above a latency floor.

Context Window vs. KV Cache

A critical distinction: **context window is what the model can attend to; KV cache is what you must keep in memory**.

For a decoder-only model like Llama 3 70B:

- Context window: 8K tokens
- KV cache for full context: 70B parameters × 16 heads × 64 head_dim × 2 (K+V) × 2 bytes × 8K tokens ÷ (70B total params) = roughly 70–80 GB for a single sequence at full context.

That doesn't fit on a single Lunar Lake. The roofline says: if you want 8K context with 70B, you compress the model (quantize), shard it (multi-GPU), or use a sliding window (throw away old context).

For M2M-100 1.2B at 128 tokens, KV cache is 25 MB, which fits easily. At 2K tokens, it's about 400 MB (2K ÷ 128 × 25 MB). At 8K, it's 1.6 GB — still under the 4–8 GB weight budget, but now you're committing real DRAM.

The practical implication: **the agent's working-memory window (what it can see in a single prompt) is bounded by KV cache size, not by model capability**. An 8B model trained on 8K context can't actually use that context on NPU if the KV cache doesn't fit.

Implications for Agent Design

Three consequences flow from this:

1. Bounded context is a feature, not a limitation. If your agent loops (agent thinks → acts → observes), and the context window is fixed at, say, 1K tokens, then the agent's working memory is fixed. Every observation older than 1K tokens falls off the window. This forces a design choice: either the agent uses only recent observations (myopic), or long-term memory lives outside the model in a vector store or database (Chapter 2.3).

2. KV cache reuse is precious. In the M2M-100 pattern (encoder-decoder), the encoder is computed once; the KV cache is reused throughout decode. In a chatbot where the user query is short but the response is long, this is efficient. In a long-conversation scenario where both sides grow, every new user message requires a re-encode. This is why copy-on-write KV cache techniques (keeping separate buffers for user messages that don't change) matter.

3. The sliding-window technique (Phi Silica's N=64 approach from Chapter 1.3) is a deliberate trade: throw away the oldest tokens' KVs to free DRAM, then recompute them if you need to backtrack. On NPU where compute is cheaper than bandwidth (relatively speaking), this is a valid trade. On GPU where compute is expensive relative to DRAM, it usually isn't.

How Intel's "8K Validated Preview" Works

Intel's announcement that Lunar Lake supports "8K context" (Chapter 1.2's static-shape discussion) is narrowly true: the compiler can emit a static-shape graph for 8K, and it runs without crashing. What's not guaranteed is latency.

The 8K window likely uses chunked prefill (process 1K chunks at a time) and either sliding-window KV for decode or hybrid compute-cache layering (let the CPU assist with KV management). The "preview" designation means it's not validated for production; the team is still characterizing it.

For agent design, treat 8K as the ceiling, not the target. A 1K–2K working memory is reliable; 4K–8K requires careful modeling and testing; beyond 8K requires either multi-GPU or architectural workarounds.

What This Section Bought You

You should now understand:

- **KV cache footprint scales with [seq_len, num_heads, head_dim, layers, dtype]** — M2M-100 1.2B at 128 tokens is ~25 MB
- **Full MHA (M2M-100) vs. GQA (Phi-3-mini) creates a 3× KV bandwidth difference** — attention architecture is destiny
- **The attention wall appears at 2K–8K tokens** on Lunar Lake depending on model size
- **KV cache growth is the per-token latency problem**; context window is the per-prompt problem
- **Encoder KV reuse** (encoder-decoder models) is a structural advantage
- **Sliding-window KV** trades compute for bandwidth — a valid move on NPU
- **8K context on Lunar Lake is validated-preview, not production**; design for 1K–2K working memory
- **Long-term memory for the agent lives outside the model** — in SQLite, vector stores, or filesystems

The next section turns to the agent's reasoning loop: given bounded context and bounded KV cache, what patterns actually work for multi-step agents?

Previous: *Chapter 1: Foundations* **Next:** *2.2 KV Cache Engineering*

Revision #4
Created 2026-05-12 15:58:32 UTC by Admin
Updated 2026-05-12 19:16:59 UTC by Admin