# Familiar Flash: A Compact Native Runtime for Low-Latency Multimodal Inference on Jetson Orin Nano

Familiar AI*

**Abstract**

We present Familiar Flash, a compact native runtime for low-latency multimodal inference on Jetson Orin Nano. The runtime targets the Liquid 450M VLM and is designed for the regime where edge inference is limited not only by model size, but also by memory bandwidth, kernel-launch overhead, framework overhead, and startup cost. Familiar Flash does not use PyTorch, Transformers, or inference engines such as vLLM. It implements the inference path directly in C/CUDA: prompt prefill, autoregressive decode, vision encoding, multimodal embedding handoff, prefix caching, and runtime state management. Its decode path minimizes launches and approaches the Jetson Orin Nano memory-bandwidth ceiling, reaching about 130 generated tokens/s in batch-1 decode. Native prefill sustains about 12,000 prompt tokens/s, and prefix caching raises effective prompt throughput by 8–12x on repeated long-context prompts. This direct implementation also reduces deployment memory by about 2 GB. Runtime creation takes about 3.5 seconds, 9.5x and 87x faster than the cached- and new-model vLLM engine-creation paths. Together, these results show that practical VLM deployment depends not only on compact models, but also on runtime design, memory economics, and launch minimization.

## 1 Introduction

Interactive multimodal inference on Jetson Orin Nano is a hardware-limited systems problem, not a miniature version of datacenter serving. The goal is to approach the fastest useful runtime the device can sustain under its 102 GB/s LPDDR5 memory-bandwidth ceiling, 8 GB of unified memory, and much smaller SM and Tensor Core budget than datacenter accelerators [1, 45]. Decode is the clearest example: generating each token requires a forward pass that streams model weights and K/V cache state, so steady-state throughput is governed by memory traffic. The same constraint applies to memory footprint and readiness: a runtime should not reserve scarce memory simply to carry PyTorch, Transformers, vLLM, and their associated scaffolding, and it should not delay an interactive application with minutes of engine initialization before the model can respond.

Familiar Flash is a compact native runtime for the Liquid 450M VLM designed around this edge-inference regime [10, 11]. It owns the model execution path directly: tokenizer integration, prompt rendering, text prefill, autoregressive decode, native vision encoding, multimodal embedding handoff, prefix caching, and runtime statistics. Its implementation is organized around a Liquid-specific CUDA decode megakernel with CUDA Graph replay, a native CUDA/cuBLAS/FlashAttention prefill path [8, 9], and a vision encoder path that can consume GPU-resident image buffers directly.

The evaluation shows that this architecture changes the practical operating point for a compact VLM on Jetson Orin Nano. For decode, the relevant headline is proximity to the hardware limit: the measured long-context path implies about 98 GB/s of effective memory traffic against the 102 GB/s LPDDR5 ceiling, while delivering about 130 generated tokens/s in batch-1 decode. Native prefill sustains about 12,000 prompt tokens/s, and prefix caching raises effective prompt throughput by 8–12x on repeated long-context prompts. Familiar Flash also reduces deployment memory by about 2 GB and creates the native runtime in about 3.5 seconds, 9.5x and 87x faster than cached- and new-model vLLM engine creation. On an 8 GB device,

*Contact: Clement Wong.

the memory reduction is approximately 25% of total capacity, turning framework overhead into usable application headroom.

### 1.1 Main Contributions

This paper makes the following contributions:

1. **Compact native VLM runtime for Jetson Orin Nano.** Familiar Flash is a C/CUDA execution path for the Liquid 450M VLM that covers prompt rendering, prefill, decode, vision encoding, multimodal handoff, prefix caching, and runtime state for interactive batch-1 inference.
2. **Launch-minimized path for autoregressive decoding.** Familiar Flash implements the Liquid decode loop as a model-specialized CUDA execution path with resident decode work, fused state progression, a separate parallel LM-head kernel, and CUDA Graph replay for the repeated token loop.
3. **Native prefill pipeline with FlashAttention.** Familiar Flash processes prompt tokens through a native CUDA/cuBLAS/FlashAttention pipeline that writes cache state directly for decode, avoiding generic tensor staging and large prompt-square attention intermediates.
4. **Native vision and multimodal embedding path.** Familiar Flash implements GPU-resident image preprocessing, SigLIP2 vision encoding, projector packing, and multimodal projection into language-model image-token embeddings, including direct ingestion from CUDA image buffers.
5. **Native prefix caching for prompts and sessions.** Familiar Flash supports both system-prompt and live session-prefix caching by retaining K/V cache rows, convolution state, token history, image-token spans, and image embeddings, reducing repeated prefill work without changing decode-visible model state.
6. **Component evaluation against standard baselines.** We evaluate correctness against HuggingFace Transformers and compare decode, prefill, prefix caching, vision encoding, memory footprint, and startup latency against vLLM and Transformers on Jetson Orin Nano.

## 2 System Design

Figure 1 summarizes the inference pipeline described in this section.

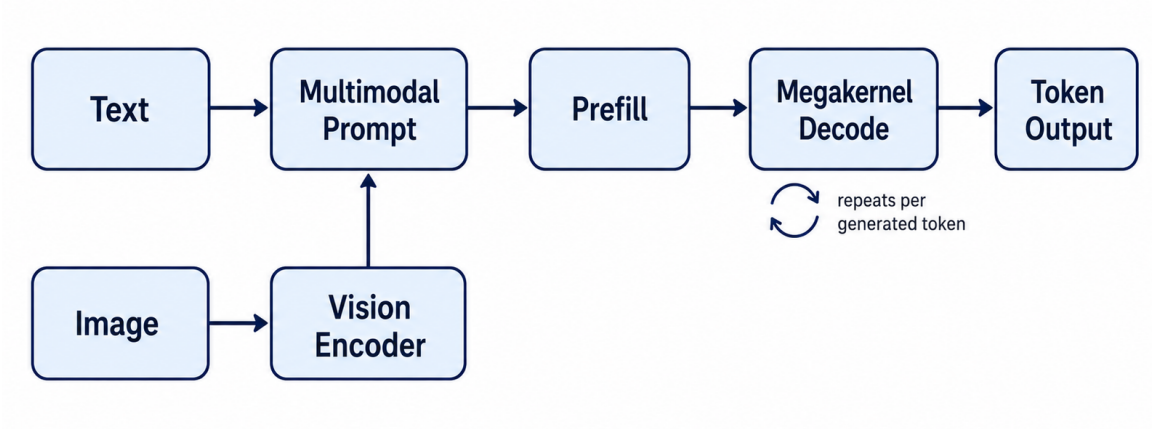


Figure 1: High-level Familiar Flash inference pipeline. Text inputs and image-derived embeddings are assembled into a multimodal prompt, prefilled once to initialize model state, and then advanced by the megakernel decode loop, which repeats for each generated token.

## 2.1 Decode Megakernel

Although the model uses the same autoregressive decoding process as other LLM/VLM runtimes, the decode phase is where Familiar Flash differs most from conventional inference stacks. For a single generated token, the Liquid 450M VLM executes a sequence of small matrix-vector operations, cache updates, normalization steps, attention scans, short-convolution state updates, and MLP projections. On Jetson Orin Nano, this workload is dominated by memory traffic, launch overhead, and framework scheduling rather than by large tensor-core GEMMs. Familiar Flash therefore implements decode as a resident, model-specialized CUDA execution path. A decode body kernel runs the Liquid layer stack for one token using device-side grid synchronization between dependent phases, while a separate LM-head kernel exposes enough parallelism for the vocabulary projection and fuses argmax, token storage, and position advance. This design removes repeated per-layer launch boundaries while preserving the larger parallel shape needed by the LM head.

The Jetson Orin Nano hardware also constrains what a practical decode megakernel can be. NVIDIA lists the Jetson Orin Nano Super configuration as an Ampere GPU with 1024 CUDA cores and 32 Tensor Cores, paired with 128-bit LPDDR5 memory [1, 45]. That is a capable edge GPU, but it is not a datacenter accelerator with abundant SMs, Tensor Cores, or HBM bandwidth. Batch-1 decode also does not naturally expose the large matrix-matrix shapes that best exploit Tensor Cores. The Liquid decode kernel is therefore shaped around resident CTAs, shared-memory reuse of small activation vectors, streamed BF16 weight loads, online-softmax attention that avoids materializing score buffers, and an occupancy-selected launch geometry that keeps all grid-synchronized CTAs resident. These choices favor predictable memory movement and low launch count over generic scheduling flexibility.

The decode kernel is specialized to the Liquid 450M VLM language architecture rather than implemented as a generic transformer executor. The language stack uses a 1024-dimensional hidden state, 16 decoder layers, a 65,536-token vocabulary, grouped-query attention with 16 query heads and 8 key/value heads, and a hybrid layer schedule that mixes short-convolution layers with full-attention layers [12]. Familiar Flash encodes these dimensions and layer types directly, allowing the decode body to use fixed launch geometry, fixed scratch-buffer layouts, architecture-specific cache updates, and fused attention, short-convolution, and MLP phases instead of generic framework dispatch.

The decode path also uses CUDA Graph replay for the repeated step [7]. After prefill initializes the device-side state, each generated token follows the same fixed execution pattern: the resident decode body kernel advances the Liquid layer stack, and the LM-head kernel selects the next token and advances the position. Capturing this two-kernel sequence removes per-kernel host-side launch setup from the autoregressive loop. CUDA Graph construction has a one-time setup cost, but the runtime performs this during warmup, so steady-state generation benefits from lower per-token orchestration overhead.

## 2.2 Prefill Kernel

Prefill has a different optimization shape from decode. Decode advances one generated token at a time, while prefill processes the prompt or suffix as a sequence. Familiar Flash uses this extra sequence parallelism through a CUDA/cuBLAS/FlashAttention pipeline [8, 9]: token and image-token embeddings are laid out as contiguous BF16 rows, projections run as batched GPU matrix operations, attention writes directly into the K/V cache, and short-convolution state is advanced across the prompt. The result is the persistent model state consumed by decode. Compared with framework stacks that assemble prefill from generic tensor operators and runtime-managed tensors, Familiar Flash keeps prompt processing inside a fixed pipeline with preallocated buffers and cache layouts shared with decode.

The Jetson Orin Nano hardware makes bounded prefill memory and cache-aware attention important. Prefill can expose larger matrix shapes than decode, but the platform is still constrained by the same bandwidth ceiling, limited unified-memory budget, and limited SM residency compared with datacenter GPUs [1, 45]. Familiar Flash therefore avoids materializing full prompt-square attention intermediates where possible. For full-prompt text prefill, it uses a BF16 head-dim-64 causal GQA FlashAttention path, which computes attention with tiled online softmax rather than storing separate score and probability matrices [9]. For suffix prefill at nonzero cache offsets, the runtime retains a tiled cuBLAS fallback that streams over the

existing K/V cache with bounded scratch space [8].

The prefill pipeline is also specialized to the Liquid model's hybrid layer structure. Attention layers project the suffix into QKV rows, apply Q/K normalization and RoPE in a staging kernel, and write normalized BF16 K/V entries into the decode cache. Short-convolution layers scan the suffix while updating their three-token recurrent state. MLP layers use the packed gate/up projection and fused SiLU-product before the down projection. These choices keep prefill and decode coupled around the same cache layout and numerical boundaries, while still using sequence-level kernels where prompt length provides enough parallel work.

## 2.3 Vision Encoding

Vision encoding is the multimodal front end for Familiar Flash. Its job is to turn pixels into BF16 device rows that can be inserted into the language prompt before prefill. The runtime accepts a CUDA RGBA image buffer, applies raw CUDA resize and normalization into patch-major pixel rows, runs the SigLIP2 vision tower, applies post-layer normalization and 2x2 projector packing, and then runs the LFM multimodal projector. The output is a compact $[image\_tokens, 1024]$ BF16 embedding buffer, already in the language model's hidden width and ready to be merged into the multimodal prompt [13, 12].

This design is especially important on Jetson Orin Nano. GPU-resident image buffers avoid unnecessary CPU round trips before inference has even started. Familiar Flash keeps the pixel path on the device, stages pitched frames only when a contiguous view is needed, and sizes its vision workspace from the actual resize plan and requested image-token budget. The attention portion of the SigLIP2 encoder uses a BF16 head-dim-64 FlashAttention path that computes attention with tiled online softmax and a compact normalization workspace [14, 9]. This keeps the vision stage's scratch memory bounded and predictable without relying on large persistent attention-score tensors.

The implementation is also specialized to the Liquid vision architecture. The encoder is a SigLIP2 NaFlex tower with 16-pixel patches, 12 layers, 768 hidden channels, 12 attention heads, head dimension 64, and 3072-wide MLP blocks [14, 12]. Positional embeddings are interpolated from the model's base grid to the planned patch grid, so the same path handles different frame sizes and image-token caps. After the tower, Familiar Flash packs each 2x2 patch group into a 3072-wide projector input and projects it to 1024 dimensions, matching the language backbone [13]. These fixed architectural dimensions let the runtime preallocate compact vision workspaces and keep the image-to-token handoff in the same BF16 layout consumed by the language model.

## 2.4 Prefix Caching

Prefix caching in Familiar Flash caches model state rather than prompt text. After a stable prefix has been prefilled, the runtime retains the attention K/V rows and Liquid short-convolution recurrent state needed to continue computation from that point. A later request that extends the same prefix does not reprocess the entire prompt. Instead, Familiar Flash restores or reuses the cached prefix state, runs prefill only on the newly appended suffix at the correct absolute token positions, and then hands the resulting state directly to decode. This makes prefix caching part of execution rather than an external prompt-management layer.

Long system instructions, tool schemas, chat history, and image-token context can consume hundreds or thousands of prompt tokens. Re-prefilling those tokens on every request spends bandwidth and latency rebuilding the same model state. Familiar Flash turns that repeated work into a smaller suffix-prefill problem, using preallocated buffers and cache layouts that are already compatible with decode.

The runtime supports two cache modes because repeated prompts and conversations have different reuse patterns. A named system-prompt cache is stateless from the application's point of view: the stable prefix is prefilled once, snapshotted, and restored for each independent suffix. A live session cache is stateful: the runtime keeps the current conversation prefix resident and reuses it only when the next prompt is an exact token-prefix extension of the retained session. This distinction lets the runtime use snapshot/restore for reusable request templates while using resident state for linear multi-turn interaction, with explicit token-prefix validation in both cases.

The same design extends to multimodal sessions. Familiar Flash tracks not only token history but also image-token spans and retained device image embeddings, so a text-only follow-up can continue from prior visual context without receiving the original image again. When the session approaches the context limit, the runtime performs a sliding rebuild over complete turns: old turns are dropped, surviving image embedding rows are compacted on device, image spans are remapped, and the shorter retained prefix is rebuilt once. Prefix caching therefore remains correctness-preserving across text turns, image-conditioned turns, and bounded-context conversations.

### 2.5 Runtime Integration

Familiar Flash exposes model execution through a C ABI with the native runtime as the ownership boundary. The production artifact is `liblfm_runtime.so`, which owns model weight loading, tokenization, text prefill, autoregressive decode, vision encoding, multimodal embedding handoff, prefix-cache state, and generation statistics. Applications can still use Python or C++ as control layers, but those layers do not schedule model execution through PyTorch, Transformers, or vLLM. The model path remains inside this boundary after initialization.

The C ABI is the stable integration contract. It provides runtime creation and teardown, tokenizer encode/decode, text generation, system-prompt and session-prefix cache operations, multimodal generation from projected BF16 image embeddings, native GPU-image vision encoding, and structured timing, memory, and session-state statistics. This boundary is intentionally narrow: callers pass token IDs, image-token spans, CUDA pointers, and output buffers, while the runtime owns cache layout, scratch allocation, kernel sequencing, and decode state progression.

Language bindings sit above this ABI as thin convenience layers. The Python wrapper uses `ctypes` to load the runtime library from an explicit path, an installed wheel, or a local build directory, then forwards calls into the C ABI. The C++ wrapper manages runtime and CUDA-buffer lifetimes automatically and provides text, system-prompt, and VLM chat-session helpers. These bindings format inputs, manage lifetimes, and expose metrics while leaving model execution inside the library.

Runtime integration also includes the prompt and tool-call surface needed by application tasks. The procedural chat-template renderer matches the model's ChatML-like template, expands image placeholders into exact image-token runs, validates image-token spans, and renders tool schemas into the system prompt. This keeps task prompts, multimodal token layout, and generated function calls aligned with the native tokenizer and runtime ABI. The result is a runtime interface that supports structured tool-calling workflows in the same native path used for ordinary text and multimodal generation.

## 3 Evaluation

We evaluate Familiar Flash along five axes: correctness, core generation speed, prefix reuse, multimodal cost, and deployment economics. Correctness is established by comparing outputs against Transformers, which serves as the reference implementation. Core generation speed is measured through decode and prefill sweeps against vLLM and Transformers. Prefix reuse is evaluated through system-prompt and live session-prefix caches. Multimodal cost is measured through vision encoding against framework and inference-engine baselines. Deployment economics are quantified through memory footprint and startup latency on Jetson Orin Nano.

### 3.1 Setup and Baselines

All measurements are run on Jetson Orin Nano using the same Liquid 450M VLM checkpoint across all runtimes. The device is treated as a memory-bandwidth-constrained target: NVIDIA specifies up to 102 GB/s LPDDR5 bandwidth for the current Jetson Orin Nano Super software configuration [1, 45]. The evaluation focuses on batch-1 inference, which is the operating point for interactive deployment: one live request, one model instance, and a latency-sensitive decode loop rather than server-style throughput batching.

Unless otherwise stated, generation uses BF16 model weights and greedy decoding, so the comparison isolates runtime behavior rather than sampling policy.

We compare Familiar Flash against two baselines. HuggingFace Transformers is used as the reference implementation for correctness because it follows the model's canonical architecture, tokenizer, chat template, vision encoder, and generation semantics [3]. vLLM is used as the optimized inference-engine baseline because it is a widely used production serving stack with highly optimized CUDA execution, KV-cache management, and decode scheduling [5, 6]. Transformers is therefore the correctness oracle, while vLLM and Transformers together form the performance baselines.

The benchmarks separate the model pipeline into the components that dominate interactive VLM inference: autoregressive decode, prompt prefill, prefix-cache reuse, vision encoding, memory footprint, and startup latency. A full application request combines several of these costs, but measuring them separately makes it clear where Familiar Flash improves the execution path and where platform limits become the controlling factor.

For correctness-oriented tests, Familiar Flash is compared against Transformers at the level of tokenizer/template behavior, multimodal image-token layout, native vision features, prefix-cache replay, and generated outputs. For performance-oriented tests, Familiar Flash is compared against vLLM and Transformers using the same model snapshot and matched prompt or context sizes wherever possible. This keeps the evaluation centered on runtime implementation differences rather than model or prompt differences.

## 3.2 Correctness Against Transformers

We use Hugging Face Transformers as the correctness oracle because it represents the canonical model implementation: tokenizer behavior, chat-template rendering, image-token expansion, vision encoding, multimodal projection, and generation semantics. The goal of this evaluation is not to prove bitwise identity for every floating-point intermediate, but to show that Familiar Flash preserves model behavior at the component boundaries that determine generated outputs. The correctness harness checks five boundaries:

1. **Tokenizer and prompt construction.** Familiar Flash's native tokenizer is tested against Transformers for encode and decode behavior across plain text, whitespace-sensitive strings, special-token handling, and batch inputs. The procedural chat-template renderer is also compared against the Transformers processor for system prompts, assistant history, tool schemas, image placeholders, multiturn VLM prompts, and expanded image-token spans. These tests verify that Familiar Flash and Transformers present the same token sequence to the model before any kernel-level execution differences can matter.

2. **Text generation.** Native decode is tested against Transformers-generated token IDs for fixed prompts, and the optimized megakernel decode path is compared against the fallback native decode path over longer generated sequences. This separates model correctness from kernel optimization: the fallback path validates the native model implementation, while the megakernel-vs-fallback tests verify that launch fusion, CUDA Graph replay, and resident decode execution do not change the generated sequence.

3. **Multimodal handoff.** Familiar Flash is tested with projected image embeddings aligned to the language model's expected hidden width and image-token layout, then compared against Transformers generation on the same prompt. This verifies that image-token placement, multimodal prompt assembly, and language-model consumption of visual embeddings remain compatible with the Transformers reference path.

4. **Native vision encoding.** The native SigLIP2 vision path and multimodal projector are compared against Transformers feature outputs for controlled image sizes and real image inputs. These tests report numerical feature differences rather than task labels, because the purpose is to validate the component implementation directly: resize and normalization, positional interpolation, vision-tower execution, 2x2 projector packing, and projection into the language hidden space.

5. **Prefix-cache replay.** Prefix-cache correctness is tested by comparing cached execution against full replay. System-prompt cache and session-prefix cache paths must produce the same continuation state as re-prefilling the retained prompt from scratch, including token-prefix validation and multimodal session state where image-token spans are retained. This validates that caching changes the amount of work performed, not the model state seen by decode.

We implemented the correctness harness around these boundaries and ran it against the evaluated Liquid 450M VLM checkpoint. The implemented tests passed, covering tokenizer/template behavior, text generation, multimodal handoff, native vision encoding, and prefix-cache replay. The test suite is available in the source repository[1] under the correctness and oracle test trees.

Beyond component-level correctness, we also evaluate end-to-end task behavior with VLMEvalKit [4]. We run the official Liquid 450M VLM checkpoint through Transformers and Familiar Flash on the same Jetson Orin Nano, using greedy decoding and exact-match evaluation. Table 1 shows that Familiar Flash preserves end-to-end model behavior across image-question answering, visual reasoning, science and diagram understanding, real-world QA, and multi-image perception benchmarks. These results complement the component tests: the component harness checks that individual runtime boundaries match the oracle, while VLMEvalKit verifies that those boundaries compose into comparable task-level behavior over thousands of complete VLM requests.

Table 1: End-to-end VLMEvalKit correctness check against Transformers. Scores compare the canonical Transformers implementation with Familiar Flash on the same Liquid 450M VLM checkpoint.

| Benchmark | Transformers | FF (Ours) |
|---|---|---|
| MMBench | 0.562693 | 0.558824 |
| MMStar | 0.426000 | 0.429333 |
| RealWorldQA | 0.584314 | 0.562092 |
| AI2D | 0.623705 | 0.622409 |
| ScienceQA | 0.775905 | 0.768964 |
| SEEDBench | 0.675379 | 0.677628 |
| MMMU | 0.272222 | 0.280000 |
| BLINK | 0.432930 | 0.431352 |
| MMMB | 0.757576 | 0.755556 |

## 3.3 Decode Performance

Decode performance measures the steady-state autoregressive loop after prefill has completed. This is the latency-critical path for structured and interactive responses, where each generated token depends on the previous token and the runtime repeatedly reads the retained K/V cache. We benchmark greedy decoding over fixed context lengths and generate a short response-length suffix, matching regimes where tens of output tokens are typical rather than long-form text generation. The sweep generates 65 output tokens per prompt and reports the steady-state decode rate after prefill.

The primary metric is generated tokens per second. We compare Familiar Flash against vLLM decode and Transformers decode using the same Liquid checkpoint and matched prompt lengths. Table 2 reports the full decode sweep, and Figure 2 plots the same results as a context-length throughput curve. The context length is swept from short prompts to 4096 tokens so the benchmark captures both launch-dominated decode and long-context cache-read behavior. This sweep is important because a runtime can look strong at short context while degrading once attention must scan thousands of cached tokens per layer.

Familiar Flash maintains high decode throughput across the full context sweep. Throughput decreases only gradually from 132.27 generated tokens/s at 64 context tokens to 124.04 generated tokens/s at 4096 context tokens, even as each generated token scans a longer retained K/V cache. At 1024 context tokens,

[1] https://github.com/familiarmachines/megakernel.

Table 2: Batch-1 decode throughput by context length. Throughput is measured in generated tokens per second.

| Context tokens | FF (Ours) | vLLM | Transformers |
|---|---|---|---|
| 64 | **132.27** | 113.24 | 16.80 |
| 128 | 132.05 | 111.83 | 17.03 |
| 256 | 131.67 | 113.83 | 17.02 |
| 512 | 131.03 | 114.75 | 16.96 |
| 1024 | 130.07 | 114.09 | 16.97 |
| 1536 | 129.18 | 112.46 | 20.06 |
| 2048 | 127.67 | 111.11 | 21.09 |
| 2560 | 126.93 | 109.36 | 21.90 |
| 3072 | 126.14 | 107.89 | 21.05 |
| 3584 | 124.86 | 107.91 | 21.44 |
| 4096 | 124.04 | 106.93 | 22.17 |

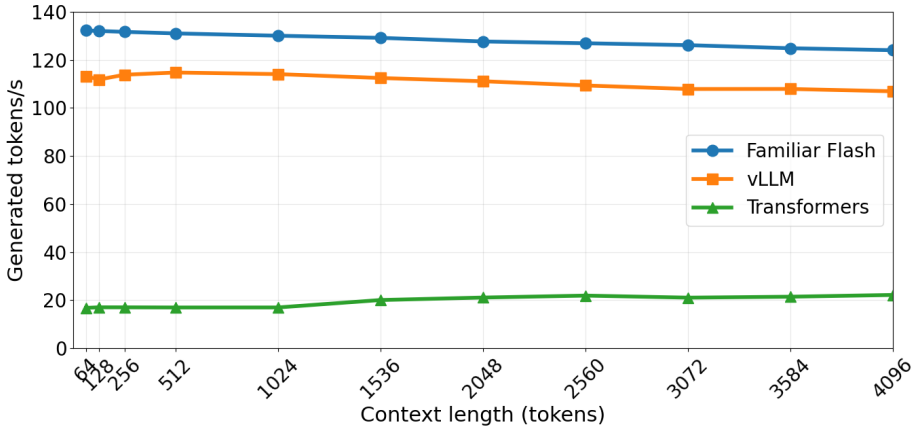


Figure 2: Batch-1 decode throughput across context lengths. Familiar Flash exceeds vLLM across the sweep while substantially exceeding Transformers decode throughput on the same device.

Familiar Flash reaches 130.07 generated tokens/s, 1.14x vLLM and 7.66x Transformers. At 4096 context tokens, Familiar Flash reaches 124.04 generated tokens/s, 1.16x vLLM and 5.60x Transformers.

The sweep shows that a compact native implementation can outperform a general serving engine at the batch-1 operating point when the decode path is shaped for one model and one device. vLLM remains a strong baseline because it is highly optimized for cache paging and attention scheduling [6]; Familiar Flash targets a narrower regime: predictable batch-1 execution for a compact VLM on Jetson Orin Nano. In that regime, it delivers higher measured decode throughput across the sweep while avoiding the PyTorch, Transformers, and vLLM engine stack.

## 3.4 Prefill Performance

Prefill performance measures the prompt-processing phase before autoregressive decode begins. During prefill, the runtime embeds the input sequence, executes the Liquid layer stack over all prompt tokens, writes attention K/V rows into the decode cache, advances short-convolution state, and produces the final hidden state used to generate the first output token. This phase is latency-critical for long prompts, multimodal prompts, tool schemas, and cached sessions that still need to process a newly appended suffix. The sweep measures prefill over context lengths from 64 to 4096 tokens and generates one output token so that prompt-processing throughput is isolated from long decode.

The primary metric is prompt tokens processed per second. We compare Familiar Flash against vLLM

prefill and Transformers prefill using the same model checkpoint and matched prompt lengths. Table 3 reports the full prefill sweep, and Figure 3 plots the same results as a context-length throughput curve. The context-length sweep runs from short prompts to 4096 tokens, exposing both small-prompt overhead and the larger sequence-level parallelism available at longer prompt lengths. Unlike decode, prefill can use more parallel work as the prompt grows, so throughput is expected to rise from very short contexts before flattening or declining once memory bandwidth and cache writes dominate.

Table 3: Batch-1 prefill throughput by context length. Throughput is measured in prompt tokens per second.

| Context tokens | FF (Ours) | vLLM | Transformers |
|---|---|---|---|
| 64 | 6116.87 | 1395.36 | 1176.78 |
| 128 | 7530.68 | 2863.21 | 2355.71 |
| 256 | 9504.35 | 5450.12 | 4659.01 |
| 512 | 10906.49 | 8863.90 | 6875.50 |
| 1024 | 11827.94 | 9879.84 | 7578.88 |
| 1536 | **12097.64** | 10329.07 | 7516.23 |
| 2048 | 12020.49 | 10867.87 | 7491.26 |
| 2560 | 12013.14 | 11156.43 | 7153.78 |
| 3072 | 12048.88 | 11294.96 | 7200.50 |
| 3584 | 12017.27 | 11304.34 | 7016.96 |
| 4096 | 11835.48 | 11230.37 | 6791.55 |

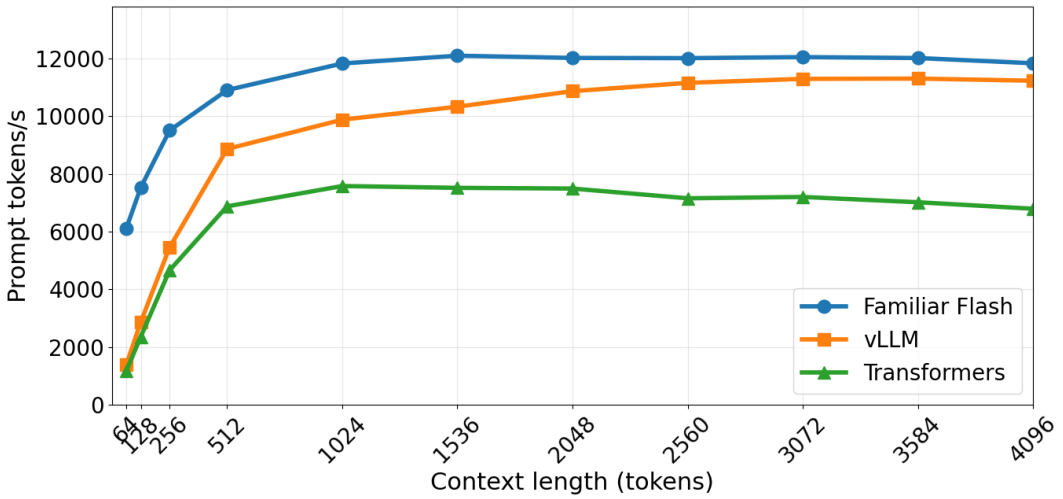


Figure 3: Batch-1 prefill throughput across context lengths. Familiar Flash reaches high sustained prompt-processing throughput while vLLM and Transformers remain lower over most of the sweep.

Familiar Flash reaches high sustained prefill throughput once the prompt is long enough to expose sequence-level GPU work. Throughput rises from 6116.87 prompt tokens/s at 64 context tokens to 11827.94 prompt tokens/s at 1024 context tokens, then remains near 12,000 prompt tokens/s through 4096 context tokens. At 1024 context tokens, Familiar Flash reaches 11827.94 prompt tokens/s, 1.20x vLLM and 1.56x Transformers. At 4096 context tokens, Familiar Flash reaches 11835.48 prompt tokens/s, 1.05x vLLM and 1.74x Transformers.

The shape of the prefill curve should be interpreted differently from decode. Decode throughput usually declines with context length because each generated token scans more cached tokens. Prefill throughput can increase with context length because the runtime has more prompt tokens to process in parallel and better amortizes fixed setup costs. Once the prompt is large enough, throughput becomes limited by memory traffic, cache writes, attention work, and available SM residency on Jetson Orin Nano. This makes the sustained long-context prefill rate a direct measure of prompt-processing efficiency on the device.

### 3.5 Prefix Caching Performance

Prefix caching performance measures the runtime benefit of reusing prefilled model state. Unlike ordinary prefill, where every request processes the full prompt from token zero, cached execution starts from a previously computed prefix state and processes only the newly appended suffix. This is common in systems with long instructions, repeated schemas, tool definitions, or multiturn interaction. The benchmark therefore reports both the requested prompt size and the number of tokens actually computed after the cache is applied.

We evaluate two cache modes, summarized in Table 4. The first is the system-prompt cache, where a stable prefix is prefilled once, snapshotted, and restored for independent requests. This models application prompts that repeatedly reuse the same instruction block, tool schema, or task definition. The relevant performance metric is effective throughput over the requested prompt length after cache reuse, rather than full-prompt prefill throughput from token zero. We therefore report the uncached full-prefill baseline, the cached effective throughput, and the throughput speedup.

The second mode is the session-prefix cache, where the runtime keeps a live conversation prefix resident and reuses it when the next prompt is an exact token-prefix extension. This models multiturn interaction: each new turn appends a small suffix to a growing retained context. Because the retained prefix is already resident, effective throughput is dominated by processing the new suffix rather than rebuilding the entire conversation state. This is why the session-cache throughput curve continues to improve as the retained context grows, until the session approaches the context limit and a sliding rebuild is required.

Table 4: Prefix-cache performance on synthetic text-only prompts. The benchmark uses synthetic token IDs, requested prefix lengths from 512 to 4096 tokens, a 64-token suffix, 16 generated tokens, two warmup steps, five measured steps, and EOS stopping disabled. The same run reports both system-prompt cache and live session-prefix cache behavior. The uncached baseline is the full-prefill throughput inferred from the prefill sweep at the same requested prompt length.

| Cache mode | Prompt toks | Cached toks | Uncached tok/s | Cached tok/s | Speedup |
|---|---|---|---|---|---|
| System Prompt | 576 | 512 | 11021.60 | 31530.55 | 2.86x |
| | 1088 | 1024 | 11861.67 | 58485.19 | 4.93x |
| | 2112 | 2048 | 12019.60 | 78086.29 | 6.50x |
| | 3136 | 3072 | 12044.91 | 104911.01 | **8.71x** |
| | 4160 | 4096 | 11835.46 | 96940.32 | 8.19x |
| Session Prefix | 896 | 831 | 11597.50 | 36405.01 | 3.14x |
| | 1408 | 1343 | 12030.18 | 57650.58 | 4.79x |
| | 2432 | 2367 | 12014.98 | 85519.38 | 7.12x |
| | 3456 | 3391 | 12025.18 | 126486.84 | 10.52x |
| | 4480 | 4415 | 11835.48 | 148187.35 | **12.52x** |

The two cache modes have different performance shapes. System-prompt cache includes snapshot restore cost for each independent request, so the measured benefit depends on how large the reusable prefix is relative to the appended suffix. Session-prefix cache avoids that repeated restore path and keeps the active prefix state resident, so its suffix-processing cost is less sensitive to retained context length. In both cases, the important result is that Familiar Flash reduces repeated prompt work without changing the decode-visible model state.

At a 4096-token requested prefix, Table 4 shows that the system-prompt cache raises effective prompt-prefill throughput from 11835.46 tokens/s to 96940.32 tokens/s, an 8.19x throughput increase. The live session-prefix cache is stronger at the same retained-prefix scale: effective throughput rises from 11835.48 tokens/s to 148187.35 tokens/s, a 12.52x throughput increase. In both modes, cached execution turns long repeated prompts into a small suffix-processing problem.

Prefix caching should be interpreted together with the prefill results. The prefill benchmark shows how quickly Familiar Flash can process prompt tokens when work must be done. The cache benchmark shows how much higher the effective prompt throughput becomes when the prompt has reusable structure. The

gain comes from removing repeated memory traffic and reusing already-computed model state when prompts are reused.

## 3.6 Vision Encoding Performance

Vision encoding performance measures the cost of converting an input image into projected image-token embeddings for the language model. This stage includes image resize and normalization, SigLIP2 vision-tower execution, post-layer normalization, 2x2 patch packing, and projection into the Liquid language backbone's 1024-dimensional hidden space. The benchmark isolates this stage from text prefill and decode so that the multimodal front-end cost can be compared directly across Familiar Flash, vLLM, and Transformers.

The primary metrics are wall-clock vision latency and image tokens processed per second. Table 5 reports results by image size and image-token count because these two factors determine both vision-tower work and the amount of multimodal context handed to the language model. Where possible, the comparison uses the same preprocessed pixel tensor or an equivalent preprocessing path so that the measured difference reflects encoder/runtime execution rather than resize or normalization drift.

Table 5: Vision encoding performance on synthetic image inputs. The test uses synthetic images at 128x128, 320x240, and 640x480, and the default 256 image-token cap. Familiar Flash results use the standalone SigLIP2/projector benchmark with preprocessed tensor input for the table. Framework baselines use Torch 2.10.0, Transformers 5.5.4, and vLLM 0.19.1.

| Backend | Image size | Image tokens | Vision ms | Image tok/s |
|---|---|---|---|---|
| Transformers | 128x128 | 64 | 27.336 | 2341.23 |
| | 320x240 | 80 | 25.710 | 3111.64 |
| | 640x480 | 234 | 43.141 | 5424.02 |
| vLLM | 128x128 | 64 | 25.398 | 2519.88 |
| | 320x240 | 80 | 24.827 | 3222.30 |
| | 640x480 | 234 | 52.131 | 4488.69 |
| FF (Ours) | 128x128 | 64 | 18.971 | 3373.56 |
| | 320x240 | 80 | 20.582 | 3886.88 |
| | 640x480 | 234 | 44.597 | 5247.01 |

As shown in Table 5, Familiar Flash encodes 64 image tokens at 128x128 in 18.971 ms, 1.34x faster than vLLM and 1.44x faster than Transformers. At 320x240, Familiar Flash encodes 80 image tokens in 20.582 ms, 1.21x faster than vLLM and 1.25x faster than Transformers. At 640x480, Familiar Flash encodes 234 image tokens in 44.597 ms, 1.17x faster than vLLM and within 3.4% of Transformers.

Familiar Flash is designed for the case where the image is already GPU-resident, because it avoids the framework tensor setup and backend handoff normally required before the vision tower runs. The same implementation owns preprocessing, encoder execution, projector packing, and the final BF16 embedding buffer consumed by text prefill. This reduces latency and memory movement at the point where a multimodal request first enters the model pipeline.

Decode and prefill measure the language side of inference, while vision encoding measures the image side. Reducing the vision front-end cost matters because every image request pays it before text prefill can begin, and because avoidable image-tensor setup competes with the same memory budget used by model weights, K/V cache, and decode buffers.

## 3.7 Memory Footprint

Memory footprint measures the deployment cost of keeping the Liquid 450M VLM ready for inference on Jetson Orin Nano. Capacity is not the main decode-speed bottleneck: adding memory alone would not make larger models fast when decode repeatedly streams weights and cache state through a 102 GB/s memory system. Runtime overhead still matters because memory consumed by Python frameworks, engine state,

CUDA allocators, graph buffers, and cache infrastructure reduces the headroom available for model context, vision buffers, concurrent perception models, and application state. We therefore compare the resident memory cost of Familiar Flash against the vLLM-based runtime using the same Liquid checkpoint and runtime workload.

The comparison separates memory into inferred buckets rather than reporting only a single RSS number. Model weights and logical K/V cache capacity are common to both runtimes, so they are treated as shared deployment cost. The remaining memory is attributed to application/Python/framework scaffolding, inference-engine/runtime overhead, CUDA working buffers, graph or eager execution state, and launcher overhead where applicable. Table 6 reports this bucketed comparison.

Table 6: Inferred memory-cost comparison on Jetson Orin Nano. The common model and logical K/V payload are separated from runtime scaffolding and engine overhead.

| Inferred bucket | vLLM | FF (Ours) |
|---|---|---|
| Common BF16 model weights + logical K/V | ∼887 MiB | ∼887 MiB |
| App/Python/framework scaffolding | ∼963 MiB | ∼509 MiB |
| Engine/runtime overhead | ∼1873 MiB | ∼360–370 MiB |
| Comparable system-visible RSS | ∼3723 MiB | **∼1762 MiB** |

The main result is that Familiar Flash removes a large fraction of the non-model deployment cost. In the vLLM configuration, the process carries the parent Python runtime plus the vLLM engine stack: PyTorch allocator state, CUDA/runtime buffers, paged-KV infrastructure, scheduler and block-manager state, graph or eager buffers, and serving-runtime scaffolding. Familiar Flash replaces that stack with a compact runtime using preallocated model buffers, decode state, graph replay state, and vision/prefix-cache workspaces. Comparable system-visible RSS decreases from about 3723 MiB to about 1762 MiB, saving roughly 1961 MiB, or 1.9 GiB. On an 8 GB Jetson Orin Nano, this is about 24% of total device memory.

This memory reduction changes the practical deployment envelope. Saving roughly 2 GB on an 8 GB Jetson Orin Nano is not a claim that capacity is the main speed bottleneck; it removes non-model overhead that competes with context, image buffers, concurrent perception models, application state, and live-operation safety margin. The result is a runtime that is faster in core kernels and easier to deploy on a constrained edge device.

## 3.8 Startup Latency

Startup latency measures the time from application launch to model readiness. On an edge device, this cost matters because the runtime may be started by a service manager, restarted after failure, updated in the field, or launched only when an interactive workload is needed. We therefore measure not only process startup, but the full path to a warmed inference runtime: model discovery, weight loading, runtime initialization, cache or graph setup where applicable, warmup generation, and the point at which the application reports the model as ready. Table 7 reports startup profiles parsed from application logs for three cases: vLLM with a cached model, vLLM while loading a new model snapshot, and Familiar Flash.

The headline gap is in runtime or engine creation. Familiar Flash creates the native runtime in 3.465 seconds, compared with 32.839 seconds for cached-model vLLM and 301.373 seconds for new-model vLLM, making runtime creation 9.5x and 87.0x faster respectively. From app start to model ready, Familiar Flash reaches readiness in 7.793 seconds, compared with 44.104 seconds and 315.398 seconds for the two vLLM profiles, making end-to-end readiness 5.7x and 40.5x faster respectively.

The primary metric is time-to-ready. This is stricter and more deployment-relevant than import time or process-start time because a model server that has started but is still compiling kernels, loading workers, initializing CUDA state, or warming graph execution cannot yet serve the workload. For Familiar Flash, startup is short because the runtime loads a compact set of model files, initializes fixed buffers, and warms the fixed decode execution path.

The difference is especially important for Jetson-class deployment. A startup path measured in seconds

Table 7: Startup profile from application logs. Times are seconds. Each profile includes startup through model readiness and model warmup.

| Phase | vLLM new | vLLM cached | FF (Ours) |
|---|---|---|---|
| App start to runtime configuration | 1.100 | 1.100 | 1.158 |
| Runtime configuration to model initialization | 5.538 | 3.569 | 0.001 |
| Runtime/engine creation | 301.373 | 32.839 | **3.465** |
| Model warmup/readiness | 7.398 | 6.606 | 3.180 |
| App start to model ready | 315.398 | 44.104 | 7.793 |

or minutes changes how a system can be operated: services must stay resident, updates are more disruptive, and recovery from failure is slow. A startup path under eight seconds makes the model behave more like an embedded runtime component. Combined with the memory-footprint result, startup latency shows that Familiar Flash's advantage is not only per-token speed, but also the operational cost of bringing the multimodal model online.

# 4 Discussion

The evaluation shows that runtime overhead becomes a first-order systems problem when multimodal inference is moved onto a bandwidth-limited edge GPU. On larger accelerators, framework and serving-engine overhead can be amortized by batching, high memory bandwidth, and abundant device memory. On Jetson Orin Nano, decode performance is governed primarily by how efficiently the runtime uses limited bandwidth and avoids launch overhead. More memory capacity by itself does not remove the batch-1 decode bandwidth bottleneck, because each generated token still has to stream weights, K/V cache rows, and activation state through the same memory system. Familiar Flash's main result is therefore not any single kernel speedup, but the combination of low-latency generation, compact memory use, and fast startup in one implementation.

This shifts how interactive VLM inference should be judged. The runtime is usually serving one live request at a time, with short generated outputs, repeated prompts, and occasional image-conditioned context. In that setting, launch count, cache layout, prefix reuse, and general-purpose inference-stack overhead are as important as peak arithmetic throughput. Familiar Flash is designed around that operating point rather than around high-throughput multi-user batching.

The component results also compose naturally. Vision encoding determines how quickly an image can enter the language pipeline. Prefill determines how quickly prompt and image-token context become state consumed by decode. Prefix caching removes repeated prefill work when prompts or sessions have reusable structure. Decode then determines the per-token latency of the final structured response. Optimizing only one of these phases would leave a different bottleneck exposed; the integrated implementation improves the whole interactive path.

The broader implication is that compact VLMs are only part of the edge-deployment story. A small model can still be impractical if the surrounding runtime consumes too much memory, takes minutes to start, or pays unnecessary runtime overhead on every token. Familiar Flash treats the runtime as part of the model system. This is why memory footprint and startup latency are evaluated alongside tokens per second: on Jetson-class devices, deployability is a performance metric.

# 5 Limitations

Familiar Flash is intentionally specialized. It targets the Liquid 450M VLM architecture and the interactive batch-1 inference regime. This specialization is what enables the compact C/CUDA execution path, but it also means the runtime is not intended to be a general-purpose replacement for Transformers, vLLM, or other serving engines across arbitrary models.

The evaluation is also centered on Jetson Orin Nano. This is the intended deployment target and the most important stress case for the work, but it does not establish the same tradeoffs across larger NVIDIA GPUs, other embedded accelerators, or CPU-only systems. Some design choices that are valuable on a small memory-bandwidth-constrained GPU may be less important on hardware with more SMs, more memory bandwidth, or larger batch sizes.

The native implementation has a higher maintenance cost than framework-based execution. Model architecture changes, tokenizer changes, vision-tower changes, or new cache semantics require corresponding runtime updates. This is a deliberate tradeoff: Familiar Flash gives up some generality to reduce overhead and expose predictable performance in the targeted deployment regime.

Correctness is evaluated at model-contract boundaries rather than by requiring bitwise identity for every internal floating-point value. This is the appropriate standard for BF16 GPU execution, FlashAttention-style kernels, and backend-specific operation ordering, but it still requires disciplined testing. Hugging Face Transformers remains the oracle for tokenizer, template, generation, vision, and task behavior; any future model or runtime change should be revalidated against that reference.

# 6 Related Work

HuggingFace Transformers provides the standard reference path for model execution, tokenizer behavior, chat templates, vision preprocessing, and generation semantics [3]. In this work, Transformers is used as the correctness oracle: Familiar Flash is expected to preserve model behavior while replacing dynamic framework execution with a compact implementation specialized to one Liquid VLM and one edge deployment regime.

vLLM is a high-performance LLM serving engine with strong support for scheduling, KV-cache management, and optimized decode execution [5, 6]. It is an important performance baseline because it represents a mature production inference stack. Familiar Flash targets a narrower but different operating point: low-latency batch-1 multimodal inference on Jetson Orin Nano, where startup time, memory footprint, and framework overhead are as important as steady-state serving throughput.

FlashAttention and related memory-efficient attention methods show the value of computing attention with tiled online softmax rather than materializing full score and probability matrices [9]. Familiar Flash uses the same broad principle in its prefill and vision paths, but combines it with cache layout, multimodal projection, prefix reuse, and decode state.

Recent megakernel and persistent-kernel systems make a closely related systems observation: low-latency LLM decode is often limited by orchestration, launch overhead, tail effects, memory movement, and cache behavior rather than peak dense linear algebra alone. Hazy Research's Llama megakernel work, ThunderMLA, Mirage persistent kernels, Event Tensor, Ada-MK, Kog's monokernel design, ClusterFusion, and deep kernel-fusion work all explore ways to fuse or persistentize larger parts of LLM inference [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. These are the closest conceptual references for Familiar Flash. The main difference is target and scope: prior work primarily focuses on datacenter GPUs, text-only LLMs, attention-specific kernels, compiler-generated persistent kernels, or serving-engine integration, while Familiar Flash is hand-specialized for a Liquid VLM on Jetson Orin Nano.

Other low-latency or small-batch inference work focuses on CUDA Graphs, dispatch overhead, decode attention kernels, prefix-aware attention, or serving-system specialization. llama.cpp CUDA Graph optimization, TensorRT-LLM graph tuning, SGLang graph execution, hybrid JIT/CUDA-Graph optimization, WebGPU dispatch analysis, CPU-GPU coupled transformer inference, FlashForge, prefix-aware attention, FlashDecoding++, FlashInfer, and FlashMLA all address important pieces of the latency problem [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41]. Familiar Flash differs by combining decode, prefill, prefix reuse, vision encoding, multimodal embedding handoff, and application-facing runtime state in one compact VLM implementation.

Jetson and edge-VLM work provides the deployment context. Existing examples show LiquidAI and other VLMs running on Jetson through HuggingFace Transformers, vLLM wheels built for Jetson Orin, VLM sparsification and offloading on Jetson Orin Nano, distributed transformer inference across Jetson Orin Nano devices, and broader edge-LLM characterization [42, 43, 44, 45, 46, 47]. These efforts establish

that compact LLMs and VLMs can run on Jetson-class devices. Familiar Flash addresses a lower-level runtime question: how much latency, memory, and startup overhead can be removed when the Liquid VLM inference path is implemented natively rather than routed through heavyweight framework or serving-engine stacks.

Other native or compiler-oriented inference systems, including TensorRT-LLM, ONNX Runtime, TVM-derived approaches, and lightweight local runtimes, also reduce framework overhead through specialization, compilation, or custom kernels [15, 16, 17, 18]. Familiar Flash sits in this broader family but makes a different design tradeoff: it gives up generality in order to provide a compact, model-specific, launch-minimized implementation for a constrained edge GPU.

To our knowledge, none of the above systems combines a Liquid-specific decode megakernel, native prefill, native SigLIP2/Liquid vision encoding, multimodal embedding handoff, system and session prefix caching, and application-facing C ABI integration into a PyTorch/Transformers/vLLM-free runtime for Jetson Orin Nano.

# 7 Conclusion

Familiar Flash demonstrates that a compact multimodal model can be made practical on Jetson Orin Nano when the runtime is optimized as aggressively as the model. By moving tokenizer-aligned prompt handling, prefill, autoregressive decode, vision encoding, multimodal handoff, and prefix caching into C/CUDA, Familiar Flash removes much of the overhead that normally surrounds VLM inference.

The result is an edge-native VLM execution path with strong component performance, lower memory footprint, and much faster startup than heavyweight framework stacks. More broadly, the work suggests that future edge VLM systems should treat runtime design, memory economics, and launch minimization as central model-deployment problems, not secondary implementation details.

# References


[1] NVIDIA. Jetson Orin Series. https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-orin/.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. https://arxiv.org/abs/1912.01703.

[3] HuggingFace. Transformers. https://github.com/huggingface/transformers. See also: https://huggingface.co/papers/1910.03771.

[4] OpenCompass. VLMEvalKit. https://github.com/open-compass/VLMEvalKit.

[5] vLLM Project. vLLM. https://vllm.ai/. Source: https://github.com/vllm-project/vllm.

[6] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. https://arxiv.org/abs/2309.06180.

[7] NVIDIA. CUDA Graphs. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html.

[8] NVIDIA. cuBLAS Documentation. https://docs.nvidia.com/cuda/cublas/index.html.

[9] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Re. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. https://arxiv.org/abs/2205.14135.

[10] Liquid AI. LFM2.5-VL-450M. https://docs.liquid.ai/lfm/models/lfm25-vl-450m.

[11] LiquidAI. LFM2.5-VL-450M Model Card. https://huggingface.co/LiquidAI/LFM2.5-VL-450M.

[12] LiquidAI. LFM2.5-VL-450M Configuration. https://huggingface.co/LiquidAI/LFM2.5-VL-450M/blob/main/config.json.

[13] Liquid AI. LFM2-VL: Efficient Vision-Language Models. https://www.liquid.ai/blog/lfm2-vl-efficient-vision-language-models.

[14] M. Tschannen, A. Gritsenko, X. Wang, M. Naeem, I. Alabdulmohsin, N. Parthasarathy, T. Evans, L. Beyer, Y. Xia, B. Mustafa, O. Bachem, X. Zhai, and J. Puigcerver. SigLIP 2: Multilingual Vision-Language Encoders with Improved Semantic Understanding, Localization, and Dense Features. https://arxiv.org/abs/2502.14786.

[15] NVIDIA. TensorRT-LLM Documentation. https://docs.nvidia.com/tensorrt-llm/.

[16] Microsoft. ONNX Runtime Documentation. https://onnxruntime.ai/docs/.

[17] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, M. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. https://arxiv.org/abs/1802.04799.

[18] ggml-org. llama.cpp. https://github.com/ggml-org/llama.cpp.

[19] Hazy Research. Look Ma, No Bubbles! Designing a Low-Latency Megakernel for Llama-1B. https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles.

[20] Hazy Research. Megakernels. https://github.com/HazyResearch/Megakernels.

[21] Hazy Research. ThunderMLA: FlashMLA, Faster and Fused-er! https://hazyresearch.stanford.edu/blog/2025-03-04-thundermla.

[22] Mirage Project. MPK: A Compiler and Runtime for Mega-Kernelizing Tensor Programs. https://arxiv.org/abs/2512.22219.

[23] Mirage Project. Mirage. https://github.com/mirage-project/mirage.

[24] Z. Jia. Compiling LLMs into a MegaKernel: A Path to Low-Latency Inference. https://zhihaojia.medium.com/compiling-llms-into-a-megakernel-a-path-to-low-latency-inference-cf7840913c17.

[25] Event Tensor authors. Event Tensor: A Unified Abstraction for Compiling Dynamic Megakernel. https://arxiv.org/abs/2604.13327.

[26] Ada-MK authors. Ada-MK: Adaptive MegaKernel Optimization via Automated DAG-based Search for LLM Inference. https://arxiv.org/abs/2605.11581.

[27] Kog AI. Building a Single-Kernel Latency-Optimized LLM Inference Engine on AMD MI300X GPUs. https://blog.kog.ai/building-a-single-kernel-latency-optimized-llm-inference-engine-on-amd-mi300x-gpus/.

[28] ClusterFusion authors. ClusterFusion: Expanding Operator Fusion Scope for LLM Inference via Cluster-Level Collective Primitive. https://arxiv.org/abs/2508.18850.

[29] Deep Kernel Fusion authors. Deep Kernel Fusion for Transformers. https://arxiv.org/abs/2602.11808.

[30] NVIDIA. Optimizing llama.cpp AI Inference with CUDA Graphs. https://developer.nvidia.com/blog/optimizing-llama-cpp-ai-inference-with-cuda-graphs/.

[31] NVIDIA. Tuning CUDA Graph Batch Sizes for Higher Output Throughput in TensorRT-LLM. https://nvidia.github.io/TensorRT-LLM/latest/blogs/tech_blog/blog20_Tuning_CUDA_Graph_Batch_Sizes_for_Higher_Output_Throughput.html.

[32] SGLang Project. CUDA Graph Optimization. https://sgl-project-sglang-93.mintlify.app/optimization/cuda-graph.

[33] Hybrid JIT-CUDA Graph authors. Hybrid JIT-CUDA Graph Optimization for Low-Latency Large Language Model Inference. https://arxiv.org/abs/2604.23467.

[34] WebGPU dispatch authors. Characterizing WebGPU Dispatch Overhead for LLM Inference Across Four GPU Vendors, Three Backends, and Three Browsers. https://arxiv.org/abs/2604.02344.

[35] Physical AI inference gap authors. Memory-Bound but Not Bandwidth-Limited: The Physical AI Inference Gap in Batch-1 LLM Decode. https://arxiv.org/abs/2605.30571.

[36] CPU-GPU coupled inference authors. Characterizing and Optimizing LLM Inference Workloads on CPU-GPU Coupled Architectures. https://arxiv.org/abs/2504.11750.

[37] FlashForge authors. CoDec: Prefix-Shared Decoding Kernel for LLMs. https://arxiv.org/abs/2505.17694.

[38] PAT authors. PAT: Accelerating LLM Decoding via Prefix-Aware Attention with Resource Efficient Multi-Tile Kernel. https://arxiv.org/abs/2511.22333.

[39] Y. Hong, J. Duan, C. Zhang, Z. Li, C. Fu, H. Wang, Z. Wang, and B. Ding. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. https://proceedings.mlsys.org/paper_files/paper/2024/file/5321b1dabcd2be188d796c21b733e8c7-Paper-Conference.pdf.

[40] FlashInfer Project. FlashInfer Attention Kernels. https://docs.flashinfer.ai/api/attention.html.

[41] DeepSeek AI. FlashMLA. https://github.com/deepseek-ai/FlashMLA.

[42] LearnOpenCV. Getting Started with VLM on Jetson Nano. https://learnopencv.com/vlm-on-jetson-nano/.

[43] thehighnotes. vLLM Jetson Orin Wheel. https://huggingface.co/thehighnotes/vllm-jetson-orin.

[44] VLM in a Flash authors. VLM in a flash: I/O-Efficient Sparsification of Vision-Language Model via Neuron Chunking. https://arxiv.org/abs/2511.18692.

[45] NVIDIA. Jetson Orin Nano Super Developer Kit. https://www.nvidia.com/en-au/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/.

[46] Jetson distributed transformer authors. Profiling-Driven Adaptive Distributed Transformer Inference on Embedded Edge Deployment. https://arxiv.org/abs/2605.25682.

[47] EdgeReasoning authors. EdgeReasoning: Characterizing Reasoning LLM Deployment on Edge GPUs. https://www.researchgate.net/publication/397280552_EdgeReasoning_Characterizing_Reasoning_LLM_Deployment_on_Edge_GPUs.