Efficient Decode Attention: Flash-Decoding with Paged KV Caches

Emaan Heidari, Ara Esfarjani, Kamsi Nwabueze, Carlton Aikins, Emmanuel Ezirim

Code available on GitHub: emaanh/paged-attention.

***Abstract*—We implement and benchmark two core components of efficient LLM decode-stage inference on an NVIDIA A40 GPU: a split-K flash-decode attention kernel and a paged KV cache memory manager. The flash-decode kernel achieves up to 477 Gi/s versus 17 Gi/s for a parallel GPU baseline, a 28× speedup at large sequence lengths, by parallelizing over the token dimension rather than serializing at the softmax boundary. The paged cache eliminates internal memory fragmentation, supporting 4× more concurrent sequences in the same memory budget at 75% fragmentation, at a constant 1.8× kernel throughput cost from block-table indirection. Critically, this overhead is per-token-access, not per-page-boundary: page size controls memory waste but has no effect on throughput.**

I. Introduction & Motivation

LLM inference has two distinct computational phases with different performance characteristics. During *prefill*, the full prompt is processed in parallel; during *decoding*, tokens are generated one at a time. Each decode step requires attending over the full KV cache accumulated so far, making it memory-bandwidth bound rather than compute bound. As context lengths grow and serving systems handle more concurrent requests, two bottlenecks dominate: kernel efficiency and memory fragmentation.

a) The kernel bottleneck: FlashAttention and FlashAttention-2 [2], [3] are designed for prefill, tiling across the query matrix $\mathbf{Q} \in \mathbb{R}^{T\times d}$ to minimize HBM traffic. At decode time the query collapses to a single vector $\mathbf{q} \in \mathbb{R}^d$; rather than a full $QK^\top$ matrix, we need only:

$$\mathbf{o} = \text{softmax}\left(\frac{\mathbf{q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \tag{1}$$

where $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{T\times d}$ grow with each generated token. The only available parallelism is over the token dimension T. A naive multi-kernel pipeline serializes at the softmax boundary regardless of T, leaving most of the GPU idle. Flash-Decoding [4] resolves this by splitting T tokens across independent GPU blocks and merging partial softmax results, converting a serial bottleneck into a bandwidth-bound parallel scan.

b) The fragmentation bottleneck: Standard KV cache implementations reserve a contiguous block of GPU memory per sequence sized to `max_seq_len`. Memory unused by shorter sequences is unrecoverable. At 75% fragmentation, a realistic operating point when `max_seq_len` is set conservatively, three out of four bytes reserved for KV caches are wasted, directly limiting serving concurrency. PagedAttention [5] eliminates this by allocating memory in fixed-size pages, so each sequence consumes only what it needs. The cost is an extra global memory load per token to resolve the block-table indirection.

c) Motivation: These bottlenecks are not merely theoretical. PagedAttention has never been implemented in llama.cpp [7], despite a feature request open since 2023. Issue #10860 [8] documents the symptom directly: throughput drops from 110 to 50 tokens/s as parallel slots increase, a direct consequence of the unified KV cache forcing all sequences to share one contiguous memory region. A recent pull request [9] introduced per-sequence buffers, eliminating cross-sequence attention waste, but internal fragmentation remains unsolved. Before modifying a production codebase, precise kernel-level data on what paging actually costs is needed. This paper provides that data.

d) What we build and measure: We implement both techniques from scratch on an NVIDIA A40 and isolate their kernel-level tradeoffs in a controlled microbenchmark, providing fine-grained data that system-level evaluations aggregate away. Concretely:

1) Three attention implementations: a CPU reference, a parallel GPU baseline, and a split-K flash-decode kernel, benchmarked head-to-head across sequence lengths up to $T = 2{,}097{,}152$.
2) Two KV cache memory managers: `ContiguousPool` and `PagedPool`, sharing a common kernel interface so memory layout is the only variable.
3) Systematic sweeps over split-K block count, fragmentation level, and page size, with throughput and capacity measured independently.

e) Hypotheses: **H1:** Paged allocation will support 1.5-3× more concurrent sequences in the same memory budget by eliminating fragmentation. **H2:** The block-table indirection will add 15–25% per-request latency overhead; by Amdahl's Law this cost becomes negligible at high concurrency. Both hypotheses are evaluated quantitatively in Section VII.

II. Related Work

a) Transformers and decode attention.: Vaswani et al. [1] introduced scaled dot-product attention. At decode time the query collapses to a single vector, making the operation memory-bandwidth bound (0.5 FLOPs/byte vs. the A40's ∼54 FLOPs/byte ridge point). Our work takes this roofline analysis as a starting point and measures how close two

concrete GPU implementations actually get to the hardware limit.

b) FlashAttention and FlashAttention-2.: Dao et al. [2], [3] tile the full query matrix $\mathbf{Q} \in \mathbb{R}^{T \times d}$ in shared memory to reduce HBM traffic during prefill and training. This tiling strategy is inapplicable to decoding, where $\mathbf{q}$ is a single vector and the available parallelism is over the token dimension T, not the query dimension. We use this distinction as motivation for why a separate decode-specific kernel is needed rather than reusing FA2.

c) Flash-Decoding.: Cai et al. [4] introduced split-K for single-query decode, dividing T tokens across independent GPU blocks and merging partial softmax results. Our flash-decode kernel directly implements this algorithm. Unlike the original work, which focuses on long-context inference in full serving systems, we isolate the kernel in a controlled single-GPU microbenchmark and sweep the split-K block count empirically to find the optimal value for the A40.

d) PagedAttention and vLLM.: Kwon et al. [5] introduced paged KV caches in vLLM, targeting multi-GPU production serving with continuous batching and preemption. Their evaluation measures system-level throughput and latency on real request traces. Our work is complementary: we isolate the single-GPU kernel-level cost of paged indirection and directly measure how page size, fragmentation level, and block-table lookup overhead affect throughput. This provides fine-grained data that system-level evaluations aggregate away.

e) Online softmax.: Milakov and Gimelshein [6] showed that softmax can be computed in a single pass without storing all scores, using running max and sum statistics. Flash-decoding depends on this technique to make per-chunk partial results composable without a global barrier. Our implementation follows their formulation directly.

III. Technical Approach

A. Attention Kernels

a) CPU reference.: A scalar C++ implementation of Eq. 1 used only for correctness checks. All GPU results are validated against it.

b) Parallel GPU baseline.: Three CUDA kernels connected by implicit barriers: (1) one thread per token computes the dot product $s_t = \mathbf{q} \cdot \mathbf{k}_t / \sqrt{d}$ using a strided loop over head dimensions; (2) a 128-thread block computes numerically stable softmax via parallel tree reduction over scores; (3) one thread per output dimension accumulates $\sum_t w_t \mathbf{v}_{t,i}$. Each kernel is well-optimized (tree reduction, coalesced memory access, shared memory for intermediate values), but the three-kernel pipeline introduces a fundamental bottleneck: softmax must complete over all T scores before output accumulation can begin, making the pipeline serial across that boundary regardless of T.

c) Flash-decode kernel.: Algorithm 1 shows the high-level structure. The key idea is splitting the token sequence across B GPU blocks that each process a chunk independently, then merging the results. Each block runs an online softmax [6] so it never needs to see scores from other blocks; there is no global barrier within the parallel phase.

Algorithm 1 Flash-Decode

```
Split T tokens across B GPU blocks (run in parallel)
for each block do
    for each token in this block's chunk do
        Score = dot(query, key_t) / sqrt(d)
        Accumulate into running softmax + weighted sum of values
    end for
    Emit partial result to global memory
end for
One final block merges all B partial results → output
```

B. KV Memory Layout

Both kernels are templated on a KV accessor type, so the same kernel code works for both contiguous and paged memory.

Listing 1: KV accessor types (`kv_layout.hpp`)

```
struct ContiguousKV {
    const float *K, *V;
    __device__ float key(int token, int dim, int d)
        { return K[token * d + dim]; }
};

struct PagedKV {
    const float *K_pool, *V_pool;
    const int *block_table; int page_size;
    __device__ float key(int token, int dim, int d) {
        int page = block_table[token / page_size];
        return K_pool[(page*page_size + token%page_size)*d
              + dim];
    }
};
```

`ContiguousKV` computes the address directly. `PagedKV` first loads the physical page index from the block table, one extra global memory read per token. This single extra load is the source of the paged kernel's throughput penalty.

C. KV Cache Pools

Both pools share the same interface:

```
class KVPool {
public:
    virtual int  admit(const float* K, const float* V, int
         actual_len) = 0;
    virtual void release(int slot) = 0;
    virtual void append_token(int slot, int len, const
         float* k, const float* v) = 0;
    virtual void decode(int slot, const float* q, float*
         out) = 0;
};
```

a) ContiguousPool.: Reserves `max_seq_len` $\times$ d contiguous floats per slot on `admit`. Simple and fast, but wastes memory whenever `actual_len < max_seq_len`.

b) PagedPool.: Divides GPU memory into fixed-size pages of `page_size` tokens. A free list tracks available pages. Each sequence gets only as many pages as it needs; a per-slot *block table* records which physical pages hold which logical tokens. Worst-case waste is `page_size` $-$ 1 tokens per sequence, regardless of `max_seq_len`.

IV. Parallelization Design

a) Thread decomposition.: The flash-decode partial kernel launches B blocks of 128 threads. Block b processes tokens $[bT/B,\ (b{+}1)T/B)$ with no overlap, so blocks have no data dependencies on each other during the parallel phase.

b) Handling the softmax dependency.: Standard attention needs all T scores before it can normalize. We break this by using online softmax: each block maintains a running maximum and sum, updating them tile-by-tile. No cross-block communication is needed until the partial kernel finishes. The reduce kernel then merges the B partial results in a single block of d threads.

c) Synchronization.: Within a block, `__syncthreads()` is called after loading the query into shared memory and after each parallel reduction (max and sum). Between the partial and reduce kernels, CUDA's sequential-launch guarantee provides the barrier; no explicit fence is needed.

d) Load balance.: Token chunks differ by at most one token ($\lfloor T/B \rfloor$ vs. $\lceil T/B \rceil$), keeping blocks within one memory transaction of each other. The reduce kernel processes $B \times d$ values total, which is negligible compared to the partial kernel's $2Td$ bytes.

V. Implementation Details

a) Platform.: NVIDIA A40 (48 GB GDDR6, 696 GB/s HBM bandwidth, 37.4 TFLOPS FP32) on the USC CARC Discovery cluster. NVCC 12, C++17, CMake 3.29. Google Benchmark v1.8 for timing.

b) Kernel parameters.: `FLASH_TILE = 128` threads/block. `FLASH_BLOCKS = 128` maximum split-K blocks (determined empirically, see Section VII-A). `PAGE_SIZE = 16` tokens/page by default; configurable at runtime.

c) Shared memory.: Each block allocates $(d+2\times128)\times 4 = 1536$ bytes of shared memory: 128 floats for the cached query plus two 128-float reduction buffers. This is well within the A40's 48 KB per SM.

d) Correctness.: All GPU kernels are validated against the CPU reference using Google Test (gtest) for $d \in \{8, 64, 128\}$, $T \in \{4, 32, 256, 1024\}$, with max absolute error $< 10^{-4}$. Pool tests cover admit, release, recycling, overflow, token append across page boundaries, and multi-slot independence. All tests pass.

VI. Experimental Setup

a) Hardware.: NVIDIA A40 (48 GB GDDR6, 696 GB/s peak HBM, 37.4 TFLOPS FP32), 1 CPU core, 8 GB host RAM.

b) Benchmarks.:

- **Kernel comparison:** kernel-only and end-to-end (E2E) throughput for CPU reference, parallel GPU baseline, and flash-decode across $T \in [4096, 2097152]$, $d = 128$.
- **Block count sweep:** flash-decode latency at block counts $\{16, 32, 64, 128, 256, 512, 1024\}$ across the same T range.
- **Capacity report:** sequences fitting a 4 GB budget at varying `max_seq_len`, `actual_len=512`.
- **Page size sweep:** paged pool throughput at $N = 64$, $T = 512$ across page sizes $\{16, 32, 64, 128, 256, 512\}$.

c) Metrics.: *Kernel throughput* (Gi/s) isolates GPU computation time. *E2E throughput* (Gi/s) includes host-to-device and device-to-host memory transfer. *Latency* (μs) is the raw wall time per benchmark iteration.

VII. Results

A. Block Count Sweep

Figures 1 and 2 show flash-decode latency across all seven block counts and peak throughput at $T = 2{,}097{,}152$. At $T = 4096$, all configurations behave identically; divergence appears at $T = 8192$ for the most under-parallelized configurations. By $T = 2{,}097{,}152$, 16 blocks is 4.5× slower than 128 blocks. The optimal window is 128–256 blocks: they tie within 0.5% (476.7 vs. 474.4 Gi/s). Above 256, throughput degrades as chunks become too small for efficient HBM streaming and the reduce kernel overhead grows.

Result: $B = 128$ is optimal on the A40. We adopt this as the default.

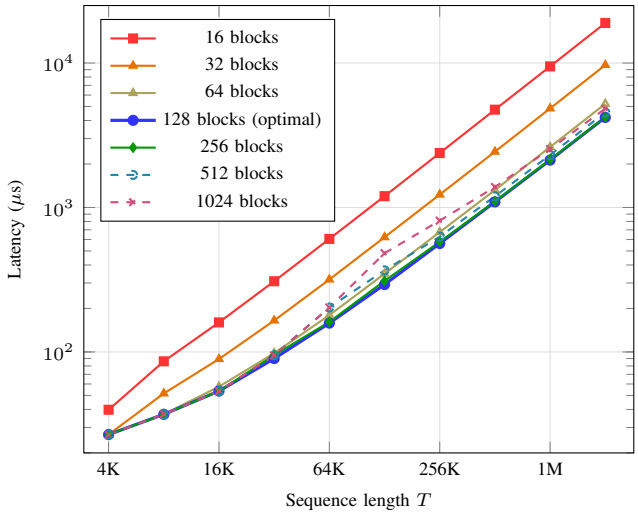

Fig. 1: Flash-decode latency vs. sequence length for all seven block counts ($d = 128$). Lower is better. 128 and 256 blocks are near-optimal (within 0.5%); fewer blocks under-parallelizes, more blocks over-shards into inefficient chunks.

B. Kernel Throughput: Flash-Decode vs. Baseline

Figure 3 shows kernel-only throughput for flash-decode and the parallel baseline. Two trends stand out. Flash-decode throughput grows steadily from 146 to 476 Gi/s as T increases; the kernel gets better at utilizing HBM bandwidth as sequences grow longer. The baseline does the opposite: throughput drops from 33 to 17 Gi/s. The culprit is not the implementation quality (it uses tree reduction and coalesced access), but the algorithmic structure: the softmax kernel runs on a single 128-thread block regardless of T, creating an $\mathcal{O}(T)$ serial stage that grows linearly with sequence length.

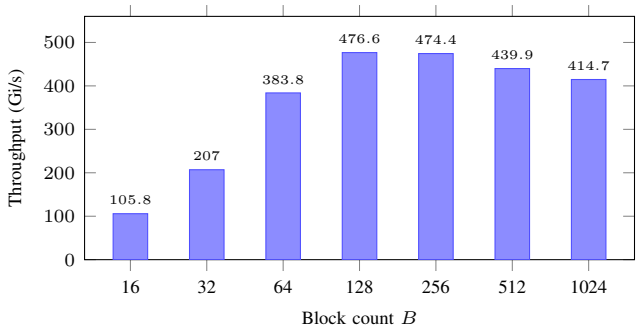

Fig. 2: Peak flash-decode throughput at $T = 2{,}097{,}152$ per block count ($d = 128$). The 128–256 window is near-optimal; 16 blocks reaches only 106 Gi/s, leaving 78% of peak HBM bandwidth unused.

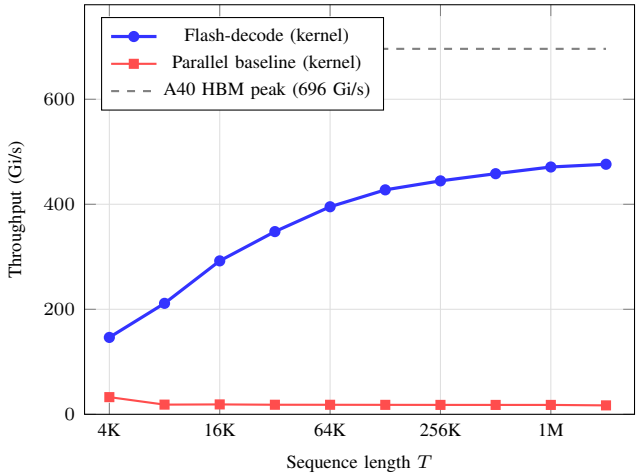

Fig. 3: Kernel-only throughput vs. sequence length. Flash-decode scales toward HBM bandwidth; the parallel baseline drops because its single-block softmax becomes an $\mathcal{O}(T)$ serial bottleneck at large T.

C. End-to-End Throughput

Figure 4 shows E2E throughput, which includes host-to-device transfer of the full KV data and device-to-host transfer of the output. This represents the case where KV data lives on CPU, which is slower than a real serving system where the KV cache lives permanently on GPU.

The CPU reference shows a two-stage decline: throughput holds at ∼3.3 Gi/s for $T \leq 16384$ while the KV data fits in L3 cache, then drops sharply to 2.1 Gi/s at $T = 32768$ as the KV footprint (∼33 MB) spills out of L3, and plateaus at 1.18 Gi/s for all larger T once the reference is fully DRAM-bound. Flash-decode E2E throughput grows from 4.9 to 10.1 Gi/s, much lower than its kernel-only 476 Gi/s because PCIe transfer dominates at all sequence lengths. The baseline E2E is 1.6× slower than flash at large T.

D. Memory Capacity Under Fragmentation

Figure 5 shows how many sequences fit in a 4 GB KV budget for `actual_len=512`, $d = 128$. The paged

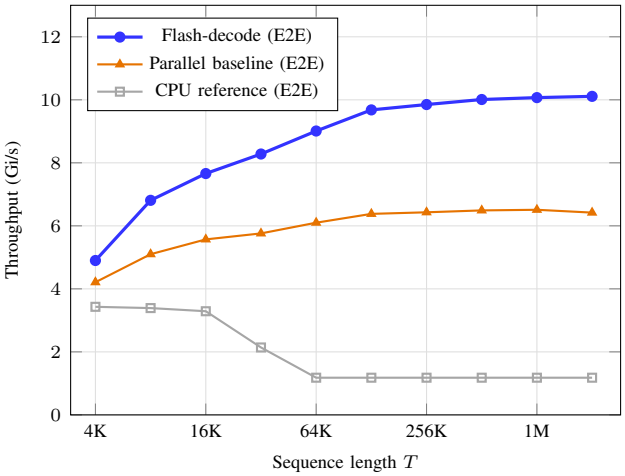

Fig. 4: End-to-end throughput including PCIe transfer. Flash-decode is 1.5× faster than the baseline at large T; the CPU reference saturates its memory bus and degrades sharply. The large gap between E2E and kernel-only (Fig. 3) is explained by PCIe transfer in Section VIII.

pool (page=16) holds 8192 sequences regardless of the configured maximum; the contiguous pool halves each time `max_seq_len` doubles.

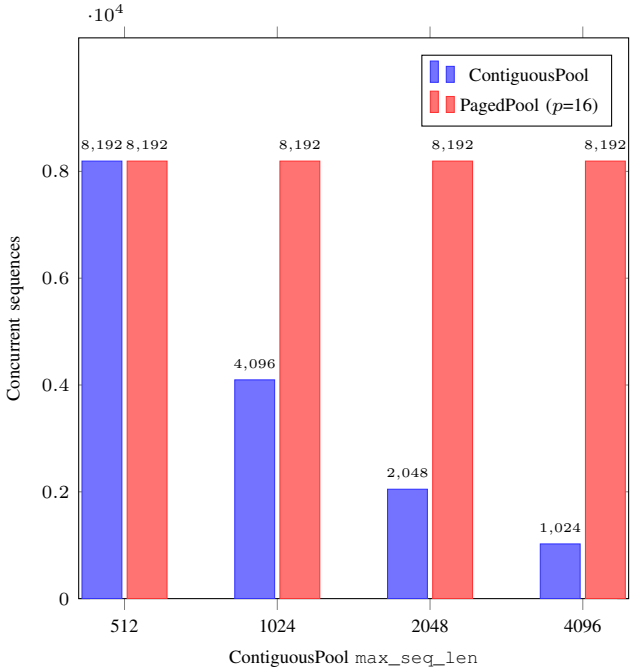

Fig. 5: Sequences fitting a 4 GB KV budget (`actual_len=512`, $d = 128$). At `max_seq_len=2048` (75% fragmentation), the paged pool holds 4× more sequences.

E. Page Size Sweep

Figure 6 shows paged pool throughput across page sizes for $N = 64$, `actual_len=512`. All page sizes from 16 to 256 give the same ≈12 Gi/s. Only page=512 (which reduces to a single page per sequence with no block-table indirection) recovers the contiguous baseline of 21.5 Gi/s.

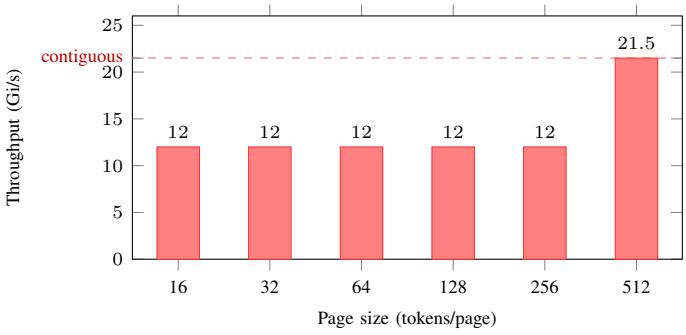

Fig. 6: Page size sweep ($N = 64$, `actual_len=512`, $d = 128$). Throughput is flat across all paged configurations. The page=512 bar reaches the contiguous baseline because each sequence fits in a single page with no indirection.

VIII. Analysis & Discussion

A. Decode Attention is Memory-Bandwidth Bound

For a decode step over T tokens at head dimension d, the kernel reads K and V ($2Td$ floats) and reads the query plus writes the output ($2d$ floats). Total bytes: $\approx 8Td$. Total floating-point operations: $\approx 4Td$ (dot products plus output accumulation).

$$\text{Arithmetic intensity} = \frac{4Td}{8Td} = 0.5 \text{ FLOPs/byte}$$

The A40's compute-to-bandwidth ratio is ≈ 54 FLOPs/byte. Since $0.5 \ll 54$, the kernel is firmly memory-bandwidth bound. This is why flash-decode's kernel throughput (Fig. 3) approaches but never exceeds the A40's 696 Gi/s HBM peak; we are reading data as fast as the hardware allows.

B. Why the Baseline Degrades with Sequence Length

The score and output kernels both scale well: one thread per token and one thread per output dimension respectively, with tree reduction and coalesced access throughout. The bottleneck is the softmax kernel, which is constrained to run on a single 128-thread block to maintain a global view of all T scores. Its runtime is $\mathcal{O}(T)$ on 128 threads, so it takes twice as long each time T doubles. At $T = 2097152$ the softmax dominates, explaining why baseline kernel throughput *decreases* with T (from 33 to 17 Gi/s) while flash-decode increases. The lesson is not that the baseline is poorly implemented (it is not), but that a three-kernel pipeline with a global barrier after scoring cannot scale beyond what a single block can do.

C. Block Count: Why 128 is Optimal

Below 64 blocks, there are not enough blocks to keep all 84 of the A40's SMs busy, and each block processes too large a chunk for coalesced streaming. Above 256, two effects compound: chunks become too small for efficient HBM transactions, and the reduce kernel must merge more partial results. The 128–256 block range is the sweet spot where SM occupancy is saturated and chunks remain large enough for streaming. Figures 1 and 2 confirm this: 128 and 256 blocks both achieve ∼475 Gi/s (within 0.5%), while 512 and 1024 blocks degrade by 8% and 13% respectively.

D. CPU Reference: L3 Cache Spill

The CPU throughput curve in Fig. 4 exposes the host memory hierarchy directly. At $T \leq 16384$, the combined KV footprint ($2 \times T \times d \times 4$ bytes $\approx$ 16 MB) fits within the CPU's L3 cache, giving a stable ∼3.3 Gi/s. At $T = 32768$ the footprint reaches ∼33 MB, evicting data from L3 and dropping throughput to 2.1 Gi/s. By $T = 65536$ the reference is fully DRAM-bound and plateaus at exactly 1.18 Gi/s through $T = 2097152$, the CPU's effective main-memory bandwidth ceiling for this streaming access pattern.

E. E2E vs. Kernel-Only: The PCIe Wall

Flash-decode achieves 476 Gi/s kernel throughput but only 10.1 Gi/s E2E. The difference is PCIe transfer. At $T = 2097152$, the KV matrices are $2 \times 2097152 \times 128 \times 4 \approx 2$ GB. PCIe 4.0 x16 peaks at ∼16 GB/s, making this transfer alone take ∼125 ms. The actual E2E time is 198 ms, so PCIe consumes ∼63% of wall time.

This is only relevant for scenarios where KV data lives on the CPU. In a real serving system, the KV cache is permanently on GPU, so the kernel benchmark is the right performance indicator, not E2E.

F. Paged KV: Why Throughput is 1.8× Lower

The paged kernel is slower because `PagedKV::key()` requires loading `block_table[token/page_size]` from global memory before it can compute the KV address. This creates a load-use dependency: the memory pipeline stalls waiting for the page index before issuing the actual KV load. For a memory-bandwidth bound kernel, any stall on the critical path directly reduces throughput.

G. Page Size Has No Effect on Throughput

If page-boundary crossings were the problem, larger pages would reduce the frequency of block-table lookups and should improve throughput. But Fig. 6 shows throughput is completely flat from page=16 to page=256. The overhead is not from how *often* a new page is entered; it is from the block-table load that happens on *every single token access*, regardless of page size.

Practical takeaway: page size only controls memory waste, not performance. Pick the smallest page that keeps per-sequence waste acceptable; there is no throughput penalty for doing so.

H. H1: Serving Capacity (Confirmed, Exceeded)

At 75% fragmentation (`max_seq_len`=2048, `actual_len`=512), the paged pool holds 4× more concurrent sequences than contiguous in the same 4 GB budget. At 87% fragmentation (`max_seq_len`=4096), that grows to 8×. H1 predicted a 1.5-3× improvement; the actual gain exceeds the upper bound. H1 is **confirmed and exceeded**.

I. H2: Per-Request Overhead (Magnitude Wrong)

H2 predicted 15-25% latency overhead per request. We measured contiguous at 21.5 Gi/s and paged at 12.0 Gi/s; the paged kernel takes $21.5/12.0 = 1.79\times$ longer, a **79% latency overhead**, significantly larger than predicted. The Amdahl's Law argument in H2 (that overhead becomes negligible at high concurrency) is directionally correct: as more sequences run concurrently, the fixed 1.8× kernel cost is amortized against the proportionally larger serving capacity. But we did not benchmark concurrent batched decoding, so we cannot fully validate this claim.

IX. Conclusion

Two questions motivated this work: how are decode attention kernels optimized at the GPU level, and what are the real tradeoffs of paged memory management? Both are now answered concretely. Flash-decoding parallelizes over the token dimension T rather than serializing at the softmax boundary, converting a memory-bound pipeline bottleneck into one that scales with HBM bandwidth. Paged memory management eliminates fragmentation at a fixed per-token cost, independent of page size, meaning page size is purely a capacity knob with no performance consequence.

H1 confirmed and exceeded. Paging enables 4–8× more concurrent sequences in the same memory budget, beyond the predicted 1.5-3×. The gain is entirely in serving capacity; kernel throughput per token is unchanged.

H2 magnitude incorrect. The paged kernel is 79% slower than contiguous, not 15-25%, due to a load-use pipeline stall on every token access. The Amdahl's Law argument that this overhead becomes negligible at scale is directionally correct but requires batched evaluation to validate.

Unexpected finding. Page size has zero effect on throughput. The overhead is per-token-access, not per-page-boundary.

a) Limitations.: Decode calls are serialized across sequences. Production systems fuse all N sequences into one kernel call, which could significantly reduce the paged pool's relative overhead. All kernels use FP32; FP16 would halve KV memory and double effective bandwidth.

b) Future work.: The 79% paged kernel overhead merits deeper investigation. Concrete next steps include prefetching block-table entries to hide the load-use stall, using pinned memory for block tables to eliminate the per-decode `cudaMemcpy`, batched paged decode across all N sequences, and FP16 KV caches. The key open question is the crossover point where the paged pool's capacity advantage outweighs its per-token overhead under realistic workload distributions, the data needed to determine whether a PagedAttention implementation in llama.cpp is worth the engineering cost.

References

[1] A. Vaswani et al., "Attention is all you need," *NeurIPS*, vol. 30, 2017.

[2] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," *NeurIPS*, vol. 35, pp. 16344–16359, 2022.

[3] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," *ICLR*, 2024.

[4] T. Cai, F. Qin, Y. Guo, and T. Dao, "Flash-decoding for long-context inference," *ICML Workshop on Efficient Systems for Foundation Models*, 2023.

[5] W. Kwon et al., "Efficient memory management for large language model serving with PagedAttention," *SOSP*, 2023.

[6] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv:1805.02867*, 2018.

[7] llama.cpp contributors, "Feature request: PagedAttention," GitHub Issue #1955, June 2023. https://github.com/ggml-org/llama.cpp/issues/1955

[8] llama.cpp contributors, "Throughput degradation with parallel slots due to unified KV cache," GitHub Issue #10860. https://github.com/ggml-org/llama.cpp/issues/10860

[9] llama.cpp contributors, "Per-sequence KV buffers to reduce cross-sequence attention waste," GitHub PR #14363, July 2025. https://github.com/ggml-org/llama.cpp/pull/14363