Celty: SpMSpV GPU Kernel and SIMT Co-Design for Efficient Dual-Sparse LLM Inference

Ruokai Yin
Yale University
New Haven, USA

Priyadarshini Panda
University of Southern California
Los Angeles, USA

Abstract

Large Language Models (LLMs) increasingly rely on sparsity to cut inference cost, but most prior work exploits a single sparsity source and targets batched multi-user inference. Dual-sparsity, which pairs unstructured weight pruning with runtime activation sparsity, offers a compelling size–accuracy–latency tradeoff for single-user decoding, but forms a Sparse Matrix–Sparse Vector (SpMSpV) workload that existing GPU kernels handle poorly. We propose Celty, a co-designed sparse format, GPU kernel, and SIMT microarchitecture for SpMSpV in LLM inference. Celty's Run-Length Compressed CSC (RLC-CSC) format enables vectorized loading of compressed weight columns and exploits both sparsity sources to skip memory accesses, accumulating partial products in shared memory. The Celty Sparse SIMT Core then adds a pipelined RLC decoder that removes software index reconstruction and repurposes local register files for conflict-free accumulation, operating on the same compressed representation. The kernel alone achieves up to 2.8× over cuBLAS; with the Sparse SIMT Core, speedup reaches 5.3× over cuBLAS at 70% dual-sparsity.

Keywords

Large Language Model; Sparsity; SIMT

ACM Reference Format:
Ruokai Yin and Priyadarshini Panda. 2026. Celty: SpMSpV GPU Kernel and SIMT Co-Design for Efficient Dual-Sparse LLM Inference. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26), November 08–12, 2026, San Jose, CA, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831252.3833936

This work was supported in part by CoCoSys, a JUMP2.0 center sponsored by DARPA and SRC, the NSF (CAREER Award, Grant #2312366, Grant #2318152), the DARPA Young Faculty Award, the DoE MMICC center SEA-CROGS (Award #DE-SC0023198) and the Global Industrial Technology Cooperation Center (GITCC) program. Correspondence: ruokai.yin@yale.edu.

ICCAD '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3833936

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of tasks [1, 33], and the vision of personal intelligence—where a capable LLM serves as a dedicated assistant on a user's own local device—is rapidly gaining traction [19]. However, deploying increasingly capable LLMs on personal devices faces two fundamental bottlenecks: growing model weights strain limited device memory, and autoregressive decoding is bounded by memory bandwidth when serving a single user. Together, they cap the scale and quality of models that can be practically deployed outside data centers.

Sparsity—removing a subset of model weights—addresses both, shrinking the memory footprint and the data movement required per token. Structured pruning [24, 42] removes entire blocks or sub-modules and is straightforward to accelerate on GPUs, but degrades accuracy sharply, pushing the community toward unstructured weight pruning [7, 32].

While unstructured weight sparsity delivers on accuracy and compression, translating it into wall-clock GPU speedup remains a persistent challenge. Its irregular memory access pattern demands specialized kernels [6, 16, 35] and microarchitectural support [13, 16, 34], and most such efforts target batched multi-user inference, formulating the problem as Sparse Matrix–Dense Matrix (SpMM) multiplication on tensor cores, an assumption poorly suited to the single-user decoding central to personal-device deployment. And the gains remain modest: Flash-LLM [35] achieves only 1.3× speedup over cuBLAS at 70% sparsity on a 4096×4096 layer.

Activation sparsity, identified at runtime through magnitude thresholding [20, 38, 40], is a complementary source of sparsity that skips entire weight columns during decoding with minimal accuracy loss, directly addressing unstructured weight sparsity's acceleration weakness for single-user decoding. Because it is input-dependent, however, the full weight matrix must remain in GPU memory, offering no model size reduction.

Dual-sparsity [38] resolves this tension by pairing activation sparsity with moderate unstructured weight pruning: the weight sparsity compresses the stored model while activation sparsity drives runtime acceleration, and when both are present, even fewer weights need to be fetched per token. This combination represents a highly promising workload

for efficient single-user LLM decoding, formulated as Sparse Matrix–Sparse Vector (SpMSpV) multiplication. Yet existing SpMSpV kernels [15, 18, 36, 37] are designed for graph traversal workloads at near-99% sparsity, and no prior work has investigated efficient GPU kernel support for SpMSpV under the moderate dual-sparsity characteristic of LLM inference.

To bridge this gap, we propose the Celty SpMSpV GPU kernel, which employs a Run-Length Compressed CSC (RLC-CSC) sparse format whose strict one-to-one mapping between indices and non-zero values enables efficient vectorized loading of compressed weight columns and scatter-accumulation of partial products in shared memory. Although this kernel already exploits both sparsity sources effectively, profiling reveals that up to 40% of execution time is consumed by reconstructing row indices from the RLC-CSC format and by shared-memory bank conflicts during scatter-accumulation—bottlenecks that cannot be resolved at the kernel level alone. This motivates a co-design approach: the Celty Sparse SIMT Core integrates a lightweight pipelined RLC decoder that eliminates reconstruction overhead entirely in hardware and repurposes local register files as conflict-free partial-product buffers, operating directly on the same RLC-CSC format without any data layout change. Our contributions are summarized as follows:

(1) We propose the Celty SpMSpV GPU kernel[1], the first GPU kernel designed for dual-sparse LLM inference. We introduce the RLC-CSC sparse format that enables efficient vectorized loading of compressed weight columns and leverage both input and weight sparsity to skip unnecessary memory accesses during single-user decoding, with shared memory employed for partial-product accumulation. The Celty GPU kernel alone achieves up to 2.8× speedup over cuBLAS and 2.4× over Flash-LLM, the strongest sparse baseline.
(2) We propose the Celty Sparse SIMT Core, a lightweight micro-architectural enhancement that integrates a 3-stage pipelined RLC decoder to eliminate software-level index reconstruction and repurposes local register files for conflict-free partial-product accumulation. With less than 0.1% area overhead, the Sparse SIMT Core achieves up to 5.3× speedup over cuBLAS at 70% dual-sparsity and up to 2.8× over Coruscant's sparse tensor core on single-user decoding workloads.
(3) We evaluate Celty end-to-end on LLaMA-2-13B, demonstrating 2.28× decode speedup over cuBLAS at 50% dual-sparsity while maintaining a WikiText-2 perplexity of 7.2, compared to 4.9 for the FP16 dense model. To our knowledge, this is the first work to co-design the sparse format GPU kernel and SIMT microarchitecture for SpMSpV in LLM inference.

[1]Code is available at https://github.com/RuokaiYin/Celty.

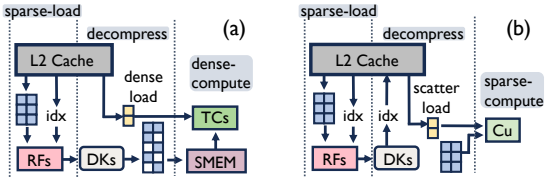

Figure 1: Illustration of (a) prior SpMM kernels. (b) prior SpMV kernels. DK stands for decompression kernel. TC stands for tensor core and Cu denotes the CUDA SIMT cores.

2 Background and Motivation

2.1 Sparsity for LLM Inference. LLM sparsity techniques span three categories, each offering a different tradeoff among accuracy, model size, and speedup.

Unstructured weight sparsity. Training-free pruning methods such as SparseGPT [7] and Wanda [32] remove individual weights based on importance scores derived from Hessian estimates or activation magnitudes. The irregular zero pattern preserves model accuracy well but demands dedicated sparse kernels [6, 16] for speedup, which remains modest at moderate sparsity [21].

Structured weight sparsity. Methods such as ShortGPT [24] and BlockPruner [42] remove entire transformer blocks or sub-modules. NVIDIA's sparse tensor cores, introduced in the Ampere architecture [27], natively accelerate a finer-grained 2:4 semi-structured pattern, but enforcing this fixed structure still incurs non-negligible accuracy degradation [7]. In both cases, the coarse or rigid pattern is straightforward to accelerate [24, 29] but limits achievable model quality.

Input activation sparsity. Recent work induces runtime sparsity through magnitude thresholding [20, 38] or SVD-based techniques [40], enabling column-skipping during decoding with minimal accuracy loss. However, the input-dependent pattern requires the full weight matrix to remain resident in GPU memory.

Dual-sparsity. DuoGPT [38] combines activation sparsity with moderate unstructured weight pruning, achieving a favorable tradeoff among speedup, model size, and accuracy for single-user decoding. This formulation motivates the SpMSpV workload targeted in this work. Throughout the paper, x% dual-sparsity refers to x% input activation sparsity plus x% unstructured weight sparsity.

2.2 GPU kernels for sparsity in LLMs. Most GPU kernel research on sparse LLM inference targets the sparse-matrix–dense-matrix (SpMM) operation [6, 8, 16, 35, 41], which corresponds to batched multi-user decoding on a pruned model. While batching improves GPU throughput, it introduces system-level complexity such as KV-cache management [17] and does not address the single-user decoding scenario.

Single-user decoding requires a Sparse Matrix–Dense Vector (SpMV) operation, equivalent to setting N=1 in SpMM. However, state-of-the-art SpMM kernels perform poorly in this regime. These kernels follow a *load-sparse-compute-dense* paradigm [6, 16]: sparse data is loaded into registers, decompressed into dense tiles via decompression kernel in shared memory (SMEM), and computed via tensor cores (Figure 1 (a)). While pipelining can hide the decompression latency at large N, at N=1 the tensor core computation is too short to mask the SMEM traffic overhead. Some existing SpMV kernels [23, 26] avoid this overhead entirely by decoding sparse indices via decompression kernel to scatter-load inputs from global memory and computing directly on SIMT cores (Figure 1 (b)). Other SpMV work [21] leverages tensor core as well to accommodate the computation overhead at very high sparsity regimes.

For dual-sparse workloads, the input vector is itself sparse, transforming the problem into a sparse-matrix–sparse-vector (SpMSpV) operation. The intersection of two irregular sparsity patterns leads to highly unpredictable memory access and workload distribution that existing kernels—designed for a single sparsity source—handle poorly. Prior SpMSpV kernels [15, 18, 36, 37] target optimization on graph algorithms such as BFS, at near-99% sparsity and do not translate to the moderate sparsity levels (30–70%), which is the standard sparsity regime for LLM inference.

2.3 Sparse SIMT Microarchitecture. NVIDIA's Ampere architecture introduced sparse tensor cores that natively compute the 2:4 semi-structured format [27]. To support unstructured sparsity, prior works [13, 16, 34] have proposed microarchitectural enhancements to the tensor core pipeline. However, all of these target SpMM operations with large N—either N≥ 2048 [13, 34] where the workload is strictly compute-bound and justifies the hardware overhead, or N≥ 8 [16] where sufficient pipeline depth hides the decompression latency. In single-user decoding, N is strictly 1, placing the workload firmly in the memory-bound regime, where tensor cores offer no advantage. Rather than enhancing the tensor core, we propose a microarchitectural modification to the SIMT core's CUDA core pipeline tailored to the SpMSpV workload, detailed in Section 4.

3 Celty SpMSpV GPU Kernel Design

3.1 Celty SpMSpV Formulation

The central design goal of the Celty SpMSpV kernel (Figure 2) is to exploit input sparsity to skip loading the corresponding weight columns from global memory, thereby reducing memory traffic and latency. To achieve this, each thread must first inspect the input activation value b before deciding whether to issue weight-fetch instructions. This introduces a critical design consideration: if threads within a warp fetch different

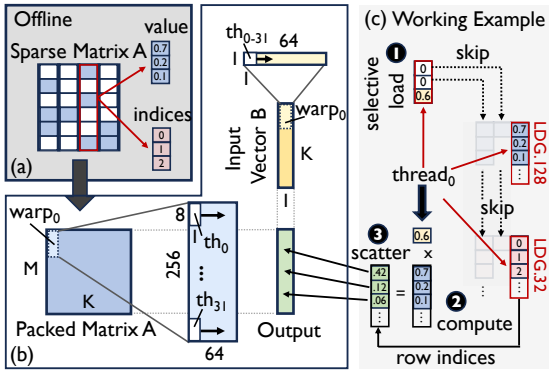

Figure 2: Celty SpMSpV kernel design. (a) Offline: weights are packed into RLC-CSC format, which run-length compresses the row indices. (b) Loop ordering: each warp is assigned a tile of 256 rows (M) and iterates over 64 columns (K). (c) Runtime example: zero-valued inputs trigger a column skip; for non-zero inputs, weights and indices are vector-loaded and scatter-accumulated into the output.

b values, the unstructured input sparsity pattern causes different threads to take different control-flow paths, resulting in warp divergence that negates any speedup. To avoid this, all 32 threads within a warp share the same input value b, and each warp processes multiple rows of A simultaneously. At each iteration, the warp loads b and checks whether it is zero; if so, the entire warp skips to the next column without issuing any memory access.

Essentially, this resembles a split-K formulation [12] where the K-dimension is distributed across warps. However, increasing K-parallelism also increases the cost of merging partial sums—either through atomic additions to global memory or a separate reduction kernel launch. To balance parallelism against atomic merge overhead, Celty assigns each warp a 2D tile (Figure 2(b)): the M-dimension is partitioned across warps, each covering 256 rows, while each warp iterates over 64 columns along K in an inner loop. We use 128-bit vectorized loads [22] for data loading: at FP16 precision, each thread loads 8 consecutive weights (16 bytes), so each 32-thread warp covers 256 rows. To guarantee alignment for vectorized access, we apply reverse offset memory alignment (ROMA) following [23].

3.2 Celty Sparse Format

With the kernel formulation established, we now describe the sparse format designed to complement it. Bitmap-based formats are a popular choice for storing unstructured pruned weights [6, 16], offering better compression than traditional Compressed Sparse Row (CSR) or Coordinate List (COO). However, bitmaps are fundamentally incompatible with our vectorized-loading strategy. The root issue is that the number

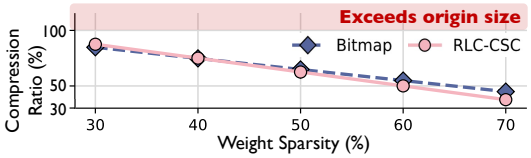

Figure 3: Compression ratio of bitmap vs. RLC-CSC across different sparsity on a 4096×4096 layer.

of set bits in a bitmap chunk is unpredictable: when a thread loads 32 bitmap bits, the number of corresponding non-zero values varies, making it impossible to issue a fixed-width vectorized load for the values. Allowing only threads with sufficient non-zeros to issue vectorized loads reintroduces warp divergence.

We instead adopt a Compressed Sparse Column (CSC)-based format, which provides a strict one-to-one mapping between indices and non-zero values—exactly matching the vectorized load granularity. Standard CSC uses a 32-bit row index per non-zero entry, yielding poor compression at the moderate sparsity levels (30–70%) typical of LLM inference [6, 16]. To address this, we compress each row index using n-bit run-length coding (RLC) [3, 4, 10, 23], where each RLC value encodes the number of zeros preceding the corresponding non-zero entry. Gaps exceeding $2^n - 1$ are handled by inserting zero-valued padding entries. For example, the sparse column $[0, 0, 4, 0, 0, 0, 0, 2, 8]$ is encoded with 2-bit RLC as a value array $[4, 0, 2, 8]$ and a run-length array $[2, 3, 0, 0]$. Figure 2(a) illustrates the offline packing process, where each column's non-zero values and their RLC-CSC indices are stored contiguously for vectorized access.

Celty uses 4-bit RLC-CSC, achieving approximately 63% compression efficiency on a 50% sparse 4096×4096 layer. Although bitmaps offer a theoretically lower bound of ~ 56% compression efficiency at the same sparsity, their deployment-time compression efficiency is ~ 64% because of the value-index decoupling discussed above. Figure 3 compares the two across sparsity levels: RLC-CSC is comparable to the bitmap implementation of [16] at deployment time while additionally being compatible with vectorized loading, which is the property the bitmap cannot provide.

3.3 SMEM-Based Scatter-Accumulation

As discussed, when a warp encounters a non-zero input activation, it must compute and accumulate partial products from the corresponding weight column into the output vector. Figure 2(c) illustrates this process with a working example for a single thread. The key challenge is determining the correct output row for each loaded weight. Each thread loads 8 weights along with 8 RLC-CSC indices, but because these indices are run-length encoded, threads within a warp must cooperate to reconstruct the absolute row positions via an intra-warp prefix sum over the run-lengths (Algorithm 1);

Algorithm 1 Intra-warp prefix sum over RLC run-lengths

Inputs: *lane* and **rlc_arrays**
$size \leftarrow len(\textbf{rlc\_arrays}) - 1$
Initialize $sum \leftarrow 0$, $all \leftarrow$ `0xFFFFFFFF`
Initialize $\textbf{prefix}[0 : size] \leftarrow 0$
for $i \leftarrow 0$ **to** *size* **do**
  $sum \leftarrow sum + \textbf{rlc\_arrays}[i]$
end for
for $i \leftarrow 0$ **to** $\log_2 32 - 1$ **do**
  $pos \leftarrow 2^i$
  $prev\_sum \leftarrow$ SHFL_UP_SYNC(all, sum, pos)
  if $lane \geq pos$ **then**
    $sum \leftarrow sum + prev\_sum$
  end if
end for
for $i \leftarrow size$ **downto** 0 **do**
  $\textbf{prefix}[i] \leftarrow sum$
  $sum \leftarrow sum - \textbf{rlc\_arrays}[\text{i}]$
end for
Outputs: **prefix** {row index adds the entry's ordinal; see code}

the absolute row index follows by adding each entry's ordinal. The reconstructed indices then serve as scatter addresses for accumulating partial products.

Naively writing partial sums directly to global memory incurs severe overhead from atomic operations, write contention, and high access latency. To mitigate this, each thread block accumulates into a local shared-memory (SMEM) buffer, which is flushed to global memory only after all assigned columns have been processed. This reduces the frequency of expensive global writes, yielding on average 14.4× speedup over direct global-memory accumulation on the A5000 GPU. However, SMEM bank conflicts during scatter-accumulation and the inherently higher access latency compared to register files limit the achievable performance, motivating the hardware-level solution presented in Section 4.

4 Celty Sparse SIMT Microarchitectural Enhancements

Profiling the SMEM-based Celty SpMSpV kernel reveals two performance bottlenecks that cannot be resolved at the software level: SMEM bank conflicts during scatter-accumulation and the overhead of reconstructing row indices from RLC-CSC (Algorithm 1). We address each with a targeted microarchitectural enhancement to the GPU's SIMT core.

4.1 Register-File-Based Accumulation

In the SMEM-based kernel, scatter-accumulation writes partial products to a shared-memory buffer indexed by the reconstructed row positions. Because a single SMEM bank group (32 banks) is shared across four SIMT cores, concurrent write requests from multiple warps frequently collide on

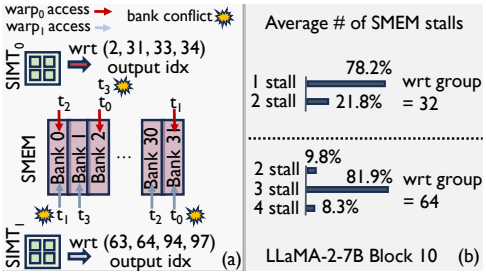

Figure 4: (a) Shared-memory bank conflicts during scatter-accumulation: threads from different SIMT cores targeting the same bank serialize writes. (b) SMEM stall distribution profiled at 50% dual-sparsity.

the same bank, serializing the writes and stalling the pipeline. Figure 4(a) illustrates this mechanism, and Figure 4(b) quantifies the impact: on a representative LLaMA-2-7B layer at 50% dual-sparsity, over 90% of iterations incur 3 or more stall cycles when the write group size reaches 64.

To eliminate this bottleneck, we repurpose the abundant local register files within each SIMT core as partial-product buffers. The output vector is partitioned into four equal segments, one per SIMT core within an SM sub-partition. For example, for $M = 5120$, each SIMT core reserves 1280 registers as a local accumulation buffer. During computation, after the RLC-CSC indices are decoded into row positions, each thread writes its partial products directly to the register file at the corresponding output position, bypassing SMEM entirely. These registers are reserved prior to kernel launch and shared among threads within each SM sub-partition[2].

Since register-file access has both lower latency and higher bandwidth than SMEM, this approach eliminates bank-conflict stalls and reduces accumulation latency. As demonstrated in our ablation study (Section 6.4), even partial register allocation (e.g., 50%) yields significant speedup at low dual-sparsity where SMEM pressure is highest, while full register-resident accumulation provides up to 2.40× improvement over the pure SMEM baseline. Importantly, the register-accumulation ratio can be tuned to balance performance against occupancy constraints, making the approach practical across different layer dimensions and sparsity levels. In the rare case that a target output position falls outside the locally partitioned register range, SMEM serves as a fallback buffer to ensure correctness; in practice, our profiling shows that such spills occur negligibly across all evaluated workloads.

Here we additionally provide an analysis on register accumulation's impact on the occupancy. From compiled register usage (`ptxas -v`), on the A5000 (sm_86, 65536 registers/SM, 48 max warps/SM), the Celty kernel uses 40 registers/thread (1280/warp), allowing the full 48 warps/SM. Reserving the registers for accumulation reduces the budget: for M=5120, 5120 reserved registers leave 60416, supporting 47 warps (97.9%, a 2.1% drop in occupancy). Across all evaluated layer sizes the reduction stays under 7%.

[2]Details on inter-thread register sharing can be found in prior work on register-file virtualization, such as [14].

4.2 Hardware RLC Decoder

While the RLC-CSC format reduces index storage and enables efficient vectorized loading, current GPUs do not natively support it. The kernel must therefore perform intra-warp prefix-sum reconstruction (Algorithm 1) at every iteration before scatter-accumulation can proceed. Notably, this cost is largely masked by SMEM stalls in the baseline kernel, but once register-file accumulation eliminates the SMEM bottleneck, reconstruction emerges as the dominant remaining overhead. As shown in Figure 5, it can occupy up to 40% of total kernel execution time, representing a substantial optimization opportunity.

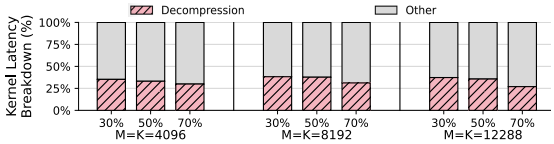

Figure 5: Percentage of RLC-CSC reconstruction latency in total kernel execution time on A5000 across different dual-sparsity levels and layer shapes.

We propose to eliminate this overhead through a lightweight microarchitectural addition to the SIMT core: a hardware RLC decoder that reconstructs absolute row indices directly from the RLC-CSC stream, removing the need for software prefix-sum instructions and intra-warp shuffle communication. The modification is minimal and allows the SIMT core to operate in both its original dense mode and the Celty sparse mode.

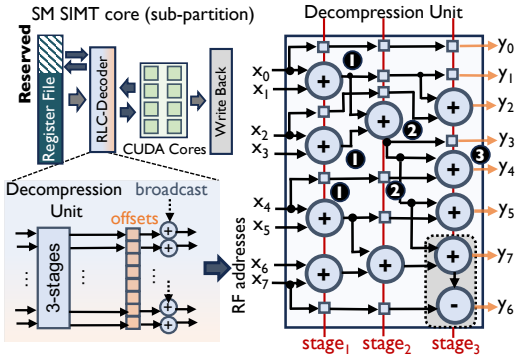

Figure 6: Celty Sparse SIMT Core microarchitecture. The RLC decoder reconstructs row indices; the output offsets address the register-file accumulation buffer.

Figure 6 illustrates the proposed RLC decoder, a 3-stage pipelined unit that ingests 8 RLC-CSC indices per cycle. Once

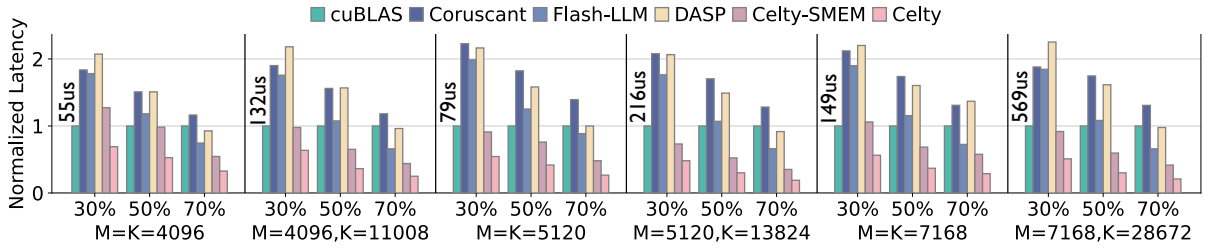

Figure 7: Comparison of kernel latency across workloads and sparsity.

the pipeline is full, 8 reconstructed row indices are produced every cycle. The pipeline implements a modified parallel prefix adder tree that computes cumulative sums of the run-lengths, equivalent to Algorithm 1 but executed entirely in hardware. The seventh output (y_6) is back-calculated via a subtractor in parallel with the final adder-tree stage, eliminating the need for an extra forward-carry stage. To illustrate, the markers ❶–❸ trace the computation of y_4: stage one (❶) computes partial sums x_0+x_1 and x_2+x_3 while buffering x_4; stage two (❷) aggregates these into a block sum $\sum_{i=0}^{3} x_i$ with x_4 still buffered; stage three (❸) adds the forwarded x_4 to produce the final index y_4.

To maximize thread utilization, the decoder ingests its 8 inputs from different thread lanes rather than from a single thread. This requires the offline RLC compression stage to interleave non-zero values and their RLC-CSC indices across threads—for instance, thread 0 loads entries at positions [0, 32, 64, 96, 128, 160, 192, 224] in the non-zero sequence. Because scatter-accumulation is order-independent, interleaving does not affect correctness. The reconstructed offsets from the decoder are added to a broadcasted base pointer to form the register-file address for the current partial product, which is then fetched and forwarded to the FP ALU for the fused multiply-accumulate operation. Together with the kernel design in Section 3, the Sparse SIMT Core completes the Celty co-design stack, operating end-to-end on the same RLC-CSC format from storage through computation.

5 Experimental Setup

Workloads. We evaluate on linear layer dimensions drawn from three widely used LLM models: 4096×4096 and 4096×11008 (LLaMA-2-7B [33]), 5120×5120 and 5120×13824 (LLaMA-2-13B [33]), and 7168×7168 and 7168×28672 (OPT-30B [39]).

Baselines. We compare against two state-of-the-art SpMM kernels—Coruscant [16] and Flash-LLM [35]—and one SpMV kernel, DASP [21], alongside dense cuBLAS. All kernels are compiled with CUDA Toolkit 12.6 and GCC 12.2, and evaluated on a single NVIDIA A5000 GPU (Ampere, compute capability 8.6, 24 GB GDDR6). Since prior sparse kernels exploit only weight sparsity, we apply dual-sparsity (both input and weight) to Celty and weight-only sparsity to all baselines, ensuring matched weight sizes across all kernels. For the two SpMM baselines, the N dimension is padded to 8 to satisfy tensor-core alignment requirements.

Hardware simulation and synthesis. For the Celty Sparse SIMT Core, we estimate GPU-level performance following the methodology of VectorSparse [43] and Coruscant [16]: we implement a modified version of the Celty kernel that removes the RLC reconstruction instructions and replaces SMEM-based scatter-accumulation with direct register-file access, while retaining the standard `fmaf_rn` computation. This modified kernel is executed on A5000 GPU, and the measured latency serves as an estimate of the performance achievable with the proposed Celty Sparse SIMT Core enhancements. We synthesize the RTL implementation of the RLC decoder using Synopsys Design Compiler at 400 MHz targeting 32 nm technology and scale to 7 nm using the methodology of [31].

6 Evaluation

6.1 Kernel Performance and Analysis

Figure 7 reports kernel latency across all evaluated layer dimensions and sparsity levels. The SMEM-based Celty kernel (Section 3) consistently outperforms the tensor-core baselines Coruscant (SpMM), Flash-LLM (SpMM), and DASP (SpMV) across all workloads and sparsity levels, as each requires an additional decompression stage whose SMEM traffic is not amortized at N=1. Against dense cuBLAS, Celty is slightly slower at low sparsity on smaller layers (roughly 1.2× at 30% for M=K=4096) but delivers speedup from 50% dual-sparsity onward for most shapes, averaging 1.49× and 2.19× at 50% and 70% and peaking at 2.80× on the 5120×13824 layer at 70%. On that layer it reaches 2.41× over Flash-LLM, 2.82× over DASP, and 2.84× over Coruscant, with margins largest at 30% where the baselines' decompression overhead is least amortized. The advantage stems from dual-sparsity's lower effective memory traffic, though gains remain bounded by the SMEM bank-conflict and RLC reconstruction overheads discussed above.

With the Celty Sparse SIMT Core enhancements, both bottlenecks are resolved in hardware. Averaged across all

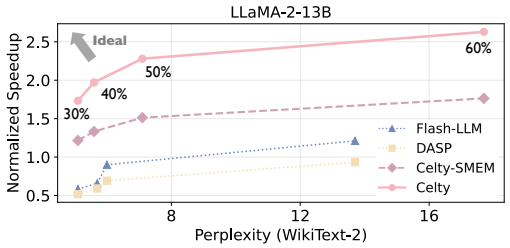

Figure 8: End-to-end decode latency vs. perplexity (WikiText-2) under different dual-sparsity configurations. Each point represents different sparsity.

evaluated layer shapes, Celty achieves 1.78×, 2.73×, and 4.06× speedup over dense cuBLAS at 30%, 50%, and 70% dual-sparsity, respectively. The maximum speedup reaches 5.27× on the 5120×13824 layer at 70% dual-sparsity.

6.2 End-to-End Model Performance

To evaluate the practical impact of Celty beyond individual layers, we measure end-to-end single-token decode latency on LLaMA-2-13B with 161 prefill tokens [30] under varying sparsity configurations, reporting the corresponding WikiText-2 [25] perplexity in Figure 8. For Flash-LLM and DASP, models are pruned using Wanda [32]; for Celty, we use DuoGPT's [38] dual-sparsity framework. All configurations use FlashAttention [5] for the attention layers.

As shown in Figure 8, Celty consistently improves end-to-end decode latency over dense cuBLAS across all evaluated accuracy–sparsity tradeoff points. In contrast, DASP and Flash-LLM fail to achieve speedup below 50% sparsity despite operating in a similar perplexity range. At 50% dual-sparsity, Celty reaches 2.28× end-to-end speedup while maintaining a WikiText-2 perplexity of 7.2, compared to 4.9 for the FP16 dense baseline. At higher sparsity, Celty reaches up to 2.63× speedup with increased but bounded perplexity degradation. These results confirm that dual-sparsity, paired with an efficient SpMSpV kernel and hardware support, delivers meaningful model-level latency reduction for single-user decoding.

6.3 Celty Sparse SIMT Core Evaluation

Area overhead. Table 1 reports the cumulative area overhead of the Celty Sparse SIMT Core scaled across GPU generations by the number of SIMT cores per chip. The overhead remains minimal across all generations, reaching at most 0.08% of additional area on Hopper. For comparison, we include the area overhead of prior sparse microarchitectural enhancements: DS-STC [34], RM-STC [13], and Coruscant-STC [16]. The Celty design is consistently the most area-efficient, owing to its lightweight RLC decoder and the reuse of existing register files for accumulation rather than introducing dedicated compute structures.

Table 1: Area overhead comparison to prior works. Available numbers are taken from respective paper. All area units are in mm^2. Overhead relative to full GPU is shown in parentheses for Celty.

GPU	DS [34]	RM [13]	Coruscant [16]	Celty-SIMT
V100 (815)	12.85	–	0.51	0.22 (**0.03%**)
A100 (826)	–	14.87	0.68	0.33 (**0.04%**)
H100 (814)	–	–	1.44	0.68 (**0.08%**)

Performance comparison. Figure 9 compares the Celty Sparse SIMT Core against Coruscant's sparse tensor core [16] on single-user decoding workloads. While Coruscant targets SpMM with $N \geq 8$ and decompresses sparse data into dense tiles before tensor-core execution, Celty operates directly on compressed RLC-CSC data at N=1. Across all evaluated layers and sparsity levels, Celty achieves on average 2.13× speedup over Coruscant-STC, with gains reaching up to 2.8× at 70% dual-sparsity.

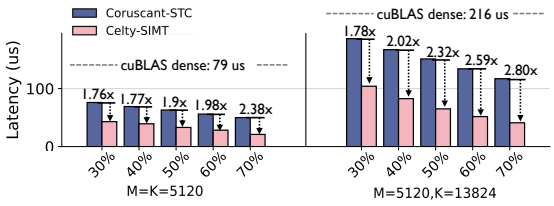

Figure 9: Comparison of sparse microarchitectural enhancements on 5120×5120 and 5120×13824 layers.

Energy Efficiency comparison. For energy efficiency, we report the normalized energy cost comparison between the dense cuBLAS kernel running on A5000, Coruscant-STC kernel running on Coruscant-STC enhanced A5000, and Celty kernel running on Celty-Sparse-SIMT enhanced A5000. As shown in Figure 10, at 60% dual-sparsity, Celty reduces the energy cost by 1.85× compared to Coruscant-STC. Energy efficiency of Celty comes from the kernel's speedup at minimal cost (on A5000, Celty Sparse SIMT's total power overhead is 88.90 mW, which is 0.04% of the TDP).

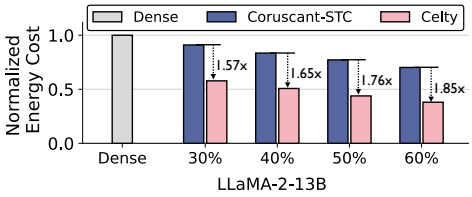

Figure 10: Energy cost of different GPU kernels (normalized to cuBLAS on A5000), each running on its own co-design-enhanced GPU.

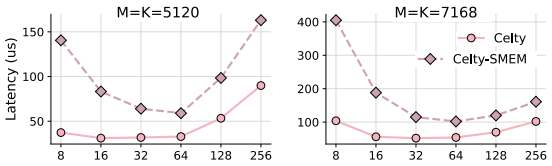

Figure 11: Kernel latency vs. Split-K parallelism at 50% dual-sparsity on 5120×5120 and 7168×7168 layers.

6.4 Ablation Study

Effect of Split-K parallelism. We study the impact of the inner-K-loop size on kernel performance by varying the Split-K parallelism. Figure 11 shows kernel latency across configurations for both 5120×5120 and 7168×7168 layers at 50% dual-sparsity. Increasing Split-K parallelism (i.e., decreasing the inner-K-loop size) reduces latency up to an inner-loop size of 64, beyond which the cost of global atomic additions for partial-sum merging dominates. We fix the inner-K-loop size to 64 for all other experiments.

Comparison with Macko across sparsity. Macko [23] is an SpMV kernel that operates on an RLC-CSR format using SIMT cores. However, its execution flow dedicates each warp to a single weight row, an approach optimized for weight-only sparsity that cannot exploit activation sparsity. For a comprehensive evaluation, we provide the performance comparison of Celty against Macko across sparsity levels and layer dimensions in Figure 12. Both the SMEM-based Celty kernel and the Celty Sparse SIMT Core outperform Macko across all configurations, with the dual-sparsity advantage growing at higher sparsity: Celty achieves on average 2.3× and 2.5× speedup over Macko at 50% and 70% dual-sparsity, respectively.

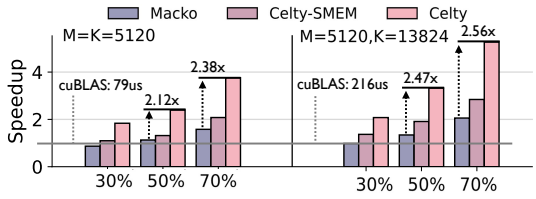

Figure 12: Comparison between Celty and Macko across sparsity levels and layer dimensions.

Accuracy under dual-sparsity. Table 2 reports WikiText-2 perplexity for LLaMA-2-7B and LLaMA-2-13B under different sparsity methods at 50% sparsity. We also report the average accuracy across three downstream tasks using LM Eval Harness [9]: 5-shot MMLU [11], 0-shot PiQA [2], and 0-shot Winogrande [28]. Structured pruning (ShortGPT) and semi-structured 2:4 pruning (SparseGPT) both incur severe perplexity degradation. Unstructured pruning (SparseGPT) preserves accuracy best, while dual-sparsity (DuoGPT) achieves 8.58 (7.17) perplexity on LLaMA-2-7B (13B), a moderate increase over the dense baseline of 5.47 (4.88) that confirms the sparsity levels targeted by Celty remain within an acceptable accuracy budget.

Table 2: Perplexity and downstream accuracy under different sparsity methods at 50% model sparsity.

Methods (Sparsity type)	LLM Model	Wiki2(↓)	Avg Acc(↑)
Dense	LLaMA-2-7B	5.47	63.3
SparseGPT (50% unstructured)	LLaMA-2-7B	6.51	60.1
SparseGPT (2:4)	LLaMA-2-7B	12.1	53.4
ShortGPT (50% structured)	LLaMA-2-7B	368.8	47.2
DuoGPT (50% dual-sparsity)	LLaMA-2-7B	8.58	55.7
Dense	LLaMA-2-13B	4.88	68.3
SparseGPT (50% unstructured)	LLaMA-2-13B	5.63	64.8
SparseGPT (2:4)	LLaMA-2-13B	9.56	56.9
ShortGPT (50% structured)	LLaMA-2-13B	191.3	48.0
DuoGPT (50% dual-sparsity)	LLaMA-2-13B	7.17	59.7

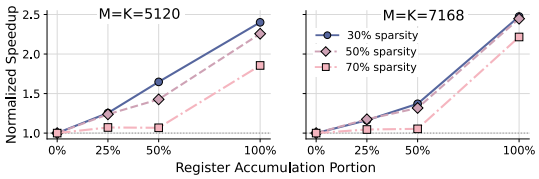

Figure 13: Normalized speedup of Celty Sparse SIMT Core as the register-accumulation portion varies from 0% (pure SMEM) to 100% (pure register) across dual-sparsity levels.

Register vs. SMEM accumulation tradeoff. We vary the register-accumulation portion on Celty sparse SIMT kernel from 0% (pure SMEM) to 100% (pure register) in Figure 13. The gains are largest at low dual-sparsity, where denser scatter updates intensify SMEM bank conflicts: on a 5120×5120 layer at 30% dual-sparsity, only 50% register accumulation can yield 1.65× speedup over pure SMEM accumulation, reaching 2.40× at 100%. At 70% dual-sparsity, conflicts are less frequent and 50% register allocation provides only 1.07× improvement, yet full register accumulation still achieves 1.86×, confirming that SMEM access latency remains a bottleneck beyond conflicts alone. Crucially, this study shows that fully register-resident accumulation is not required—the partition ratio can be tuned to balance speedup against occupancy constraints.

7 Conclusion

We presented Celty, a co-designed sparse format, GPU kernel, and SIMT microarchitecture for dual-sparse LLM inference, formulated as SpMSpV operations. The Celty Sparse SIMT Core eliminates software reconstruction overhead and shared-memory bottlenecks through a hardware RLC decoder and register-file accumulation, achieving up to 5.3× speedup over cuBLAS with negligible area overhead and up to 2.63× end-to-end decode speedup on LLaMA-2-13B with bounded accuracy degradation.

References

[1] Sandhini Agarwal et al. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925* (2025).

[2] Yonatan Bisk et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 7432–7439.

[3] Yu-Hsin Chen et al. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[4] Yu-Hsin Chen et al. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[5] Tri Dao et al. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[6] Ruibo Fan et al. 2025. Spinfer: Leveraging low-level sparsity for efficient large language model inference on gpus. In *Proceedings of the Twentieth European Conference on Computer Systems*. 243–260.

[7] Elias Frantar et al. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*. PMLR, 10323–10337.

[8] Trevor Gale et al. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*.

[9] Leo Gao et al. 2021. A framework for few-shot language model evaluation. *Version v0. 0.1. Sept* 10 (2021), 8–9.

[10] Song Han et al. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[11] Dan Hendrycks et al. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[12] Adnan Hoque et al. 2024. Accelerating a Triton Fused Kernel for W4A16 Quantized Inference with SplitK work decomposition. *arXiv preprint arXiv:2402.00025* (2024).

[13] Guyue Huang et al. 2023. Rm-stc: Row-merge dataflow inspired gpu sparse tensor core for energy-efficient sparse acceleration. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 338–352.

[14] Hyeran Jeon et al. 2015. GPU register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*. 420–432.

[15] Haonan Ji et al. 2022. Tilespmspv: A tiled algorithm for sparse matrix-sparse vector multiplication on gpus. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[16] Donghyeon Joo et al. 2025. Coruscant: Co-Designing GPU Kernel and Sparse Tensor Core to Advocate Unstructured Sparsity in Efficient LLM Inference. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 232–245.

[17] Woosuk Kwon et al. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[18] Min Li et al. 2020. Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1842–1853.

[19] Yuanchun Li et al. 2024. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459* (2024).

[20] James Liu et al. 2025. Training-free activation sparsity in large language models. *The Thirteenth International Conference on Learning Representations* (2025).

[21] Yuechen Lu et al. 2023. Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[22] Justin Luitjens et al. 2025. *CUDA Pro Tip: Increase Performance with Vectorized Memory Access*. https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

[23] Vladimír Macko et al. 2025. MACKO: Sparse Matrix-Vector Multiplication for Low Sparsity. *arXiv preprint arXiv:2511.13061* (2025).

[24] Xin Men et al. 2024. Shortgpt: Layers in large language models are more redundant than you expect. *arXiv preprint arXiv:2403.03853* (2024).

[25] Stephen Merity et al. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[26] Yuyao Niu et al. 2021. Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 68–78.

[27] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[28] Keisuke Sakaguchi et al. 2021. Winogrande: An adversarial winograd schema challenge at scale. *Commun. ACM* 64, 9 (2021), 99–106.

[29] Fabrizio Sandri et al. 2025. 2SSP: A Two-Stage Framework for Structured Pruning of LLMs. *arXiv preprint arXiv:2501.17771* (2025).

[30] ShareGPT. 2023. ShareGPT. https://sharegpt.com/

[31] Aaron Stillmaker et al. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[32] Mingjie Sun et al. 2023. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695* (2023).

[33] Hugo Touvron et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[34] Yang Wang et al. 2021. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1083–1095.

[35] Haojun Xia et al. 2023. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *arXiv preprint arXiv:2309.10285* (2023).

[36] Lei Xu et al. 2024. HAM-SpMSpV: An optimized parallel algorithm for masked sparse matrix-sparse vector multiplications on multi-core CPUs. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 160–173.

[37] Carl Yang et al. 2015. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 841–847.

[38] Ruokai Yin et al. 2025. DuoGPT: Training-free Dual Sparsity through Activation-aware Pruning in LLMs. *Advances in Neural Information Processing Systems* 38 (2025), 146883–146912.

[39] Susan Zhang et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[40] Zhenyu Zhang et al. 2025. R-Sparse: Rank-Aware Activation Sparsity for Efficient LLM Inference. In *The Thirteenth International Conference on Learning Representations*.

[41] Ningxin Zheng et al. 2022. {SparTA}:{Deep-Learning} Model Sparsity via {Tensor-with-Sparsity-Attribute}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 213–232.

[42] Longguang Zhong et al. 2024. Blockpruner: Fine-grained pruning for large language models. *arXiv preprint arXiv:2406.10594* (2024).

[43] Maohua Zhu et al. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 359–371.