Fast Matrix Multiplications for Lookup Table-Quantized LLMs

Han Guo[⋆] **William Brandon**[⋆] **Radostin Cholakov**[†]
Jonathan Ragan-Kelley[⋆] **Eric P. Xing**[◇] **Yoon Kim**[⋆]
[⋆]Massachusetts Institute of Technology, [†]High School of Mathematics Plovdiv
[◇]Carnegie Mellon University, MBZUAI, Petuum Inc.
{hanguo,wbrandon,radi_cho,jrk,yoonkim}@mit.edu, epxing@cs.cmu.edu

Abstract

The deployment of large language models (LLMs) is often constrained by memory bandwidth, where the primary bottleneck is the cost of transferring model parameters from the GPU's global memory to its registers. When coupled with custom kernels that fuse the dequantization and matmul operations, weight-only quantization can thus enable faster inference by reducing the amount of memory movement. However, developing high-performance kernels for weight-quantized LLMs presents substantial challenges, especially when the weights are compressed to non-evenly-divisible bit widths (e.g., 3 bits) with non-uniform, lookup table (LUT) quantization. This paper describes FLUTE, a **f**lexible **l**ook**u**p **t**able **e**ngine for LUT-quantized LLMs, which uses offline restructuring of the quantized weight matrix to minimize bit manipulations associated with unpacking, and vectorization and duplication of the lookup table to mitigate shared memory bandwidth constraints. At batch sizes < 32 and quantization group size of 128 (typical in LLM inference), the FLUTE kernel can be 2-4× faster than existing GEMM kernels. As an application of FLUTE, we explore a simple extension to lookup table-based NormalFloat quantization and apply it to quantize LLaMA3 to various configurations, obtaining competitive quantization performance against strong baselines while obtaining an end-to-end throughput increase of 1.5 to 2 times.

1 Introduction

Large language model (LLM) deployment faces significant latency challenges due to the memory bandwidth constraints inherent in generative (token-by-token) inference. The primary bottleneck is the cost of transferring model parameters from the GPU's global memory to the registers, i.e., LLM inference is *memory-bound*. To overcome this "memory wall" (Gholami et al., 2024), practitioners have increasingly adopted weight-only quantization methods, wherein the parameters of an LLM are compressed to lower precision (e.g., 4 or 8 bits) than the precision in which they were trained (typically 16 bits). In addition to latency improvements, weight quantization can also drastically reduce GPU memory required for deployment.

Realizing practical speed-ups with weight-only quantization requires custom mixed-type matrix-matrix multiply (matmul) kernels which must (1) move a layer's quantized weights from GPU off-chip DRAM to on-chip SRAM, (2) *de*quantize the weights to floating-point (FP) format (on chip), (3) perform the FP matmul, and (4) write the results back to DRAM. Existing kernels such as bitsandbytes (Dettmers et al., 2023), Marlin (Frantar et al., 2024), and BitBLAS (Wang et al., 2024) demonstrate that this strategy can result in significant matmul speed-ups, e.g. up to four times faster when going from W16A16 to W4A16. However, these kernels are typically specialized to 4-bit quantization, and while some kernels support non-uniform, lookup table (LUT) quantization, they are generally slower than the uniform counterparts. Given the recent promising results with odd-bit (Shao et al., 2023; Ma et al., 2024b,a) and non-uniform (Guo et al., 2024; Kim et al., 2023) quantization methods, there is thus a need to develop flexible kernels that can support mixed-type matmuls with a wider range of settings.

This paper describes FLUTE, a **f**lexible **l**ook**u**p-**t**able **e**ngine for deploying weight-quantized LLMs, with a focus on the low-bit and non-uniform quantization setting. This setting raises several challenges. First, going beyond 8-bit quantization involves packing sub-8-bit matrices into supported data types, followed by unpacking during dequantization. Structuring the unpacked data to match GPU-native matmul formats is especially challenging when the weights are quantized to non-standard bit-widths. Second, while uniformly-quantized models can rely on assembly-level optimizations to convert from INT to FP through bit-level manipula-

Findings of the Association for Computational Linguistics: EMNLP 2024, pages 12419–12433
November 12-16, 2024 ©2024 Association for Computational Linguistics

tions, lookup table-based dequantization involves dynamic indexing, and a naïve implementation can lead to substantial overhead. Finally, typical matmul implementations which distribute the workload across a grid of parallel thread blocks become inefficient with small batches and low bit-width weights; this necessitates more sophisticated partitioning strategies to optimize hardware resource utilization.

FLUTE addresses these challenges through a combination of (1) offline weight restructuring, (2) a shared-memory lookup table for efficient dequantization, and (3) Stream-K partitioning for optimized workload distribution. We compare FLUTE against existing kernels on standard LLM mixed-precision matmul settings where weights are quantized to 4 bits in groups of 128, and find that it outperforms existing non-uniform quantization kernels, and even matches the simpler uniform-quantization kernels in some cases. As an application of FLUTE, we experiment with quantizing LLaMA3—which has been found to be difficult to quantize (Huang et al., 2024)—using a variant of normal float (NF) quantization (Dettmers et al., 2023) which learns the quantization parameters based on calibration data. We find that we can achieve a 1.5 to 2 times increase in end-to-end throughput when integrated with frameworks such as vLLM (Kwon et al., 2023).

2 Background and Related Work

2.1 GPU Architecture and Memory Bandwidth Bottlenecks

GPUs are massively-parallel processors designed for throughput-oriented workloads containing large amounts of independent work. The hardware of a current-generation NVIDIA GPU consists of an array of many individual *streaming multiprocessors* ("SMs"), each consisting of 4 separate *warp schedulers* together with a single *shared memory* scratchpad accessible to all 4 warp schedulers. Each warp scheduler executes instructions on its own *functional units*, and is able to issue at most one instruction per cycle, which may then take multiple subsequent cycles to complete while the warp scheduler moves on to concurrently issue other instructions. At the hardware level, the instructions executed by a warp scheduler typically operate in a SIMD fashion over vectors of 32 data elements at a time. At the software level, CUDA asks the programmer to program at the level of individual logical *threads* executing scalar operations; threads are assigned sequential integer IDs, and every group of 32 consecutive threads are implicitly organized together into a single *warp*, corresponding to the GPU hardware's actual native unit of instruction execution.

Although GPUs are able to execute large numbers of instructions in parallel across the warp schedulers of their many SMs, the rate at which instructions can be executed is not always the bottleneck in realistic GPU workloads. Instead, the maximum achievable throughput of a GPU workload is often constrained by the speed of *data movement* between levels of the GPU's memory hierarchy. The memory resources of modern NVIDIA server-class GPUs consist of (roughly): (1) Tens of gigabytes of off-chip DRAM, referred to here as *global memory*; (2) Tens of megabytes of on-chip SRAM acting as a shared *L2 cache* accessible to all SMs; (3) Hundreds of kilobytes of local SRAM per SM, split into two configurably-sized portions, one acting as an *L1 cache* and the other an explicitly-addressed *local scratchpad*; and (4) Hundreds of kilobytes of local SRAM per SM, acting as *registers* for the threads running on that SM.

The read/write bandwidth of resources in this memory hierarchy can easily become the limiting factor for realistic GPU workloads. For example, an A100-80GB GPU supports a nominal peak throughput for 16-bit matrix-multiply instructions of $\approx 3 \times 10^{14}$ FLOP/s (aggregated across all SMs), but its main memory supports a nominal peak bandwidth of only $\approx 1.5 \times 10^{12}$ byte/s. This means that the speed of any kernel which performs fewer than roughly $(3 \times 10^{14})/(1.5 \times 10^{12}) = 200$ matrix-multiply FLOPs per byte of data accessed will necessarily be limited by the GPU's memory bandwidth, not by its compute throughput. Maximizing the ratio of FLOPs to bytes transferred, a quantity known as *arithmetic intensity*, is often the single most important consideration when designing high-performance kernels.

2.2 LLM Deployment Characteristics

Depending on the context, inference can be bottlenecked by compute throughput or memory bandwidth. For LLMs, training, large-prefill, and large-batch inference enjoy high arithmetic intensity as the sizes of matrices involved in the matmuls are large enough to saturate compute. Small-batch, token-by-token inference on the other hand involves narrower matmuls due to the smaller batch dimension, resulting in low arithmetic intensity. Reducing the amount of memory operations in this case can thus enable practical speed-ups, even

if the number of FLOPs remains the same (or is even slightly increased). This has led to much recent work on customized kernels which move the weights from main memory to on-chip SRAM while keeping them quantized/sparse (Dettmers et al., 2023; Kim et al., 2023; Frantar et al., 2024; Wang et al., 2024; Xia et al., 2024a), and then performing the actual matmuls in higher precision after dequantizing to `FP` on chip. `Marlin` implements this strategy for 4-bit uniform quantization and reports significant (up to 4×) matmul speed-ups even in moderate batch (16-32) settings. `bitsandbytes` (Dettmers et al., 2023) and `BitBLAS` (Wang et al., 2024) extend this to LUT-quantized LLMs, but do not allow for 3 bit-quantized weights. Moreover, existing LUT-quantization kernels generally underperform uniform-quantization kernels.

2.3 Weight-only Quantization in LLMs

Uniform quantization converts a group of full precision weights to lower-precision intervals of equal size through rounding. For example min-max quantization maps a group of weights $\mathbf{u}$ to integers $\{-2^{b-1}, \dots, 2^{b-1}-1\}$ via the function $\operatorname{clamp}\left(\operatorname{round}(\frac{1}{s}\mathbf{u}); -2^{b-1}, 2^{b-1}-1\right)$, where $s = \frac{\max(|\mathbf{u}|)}{2^{b-1}-1}$ is a scaling factor. Recent methods improve upon min-max quantization by using calibration data (Frantar et al., 2022; Lin et al., 2023; Shao et al., 2023; Ma et al., 2024b). When both the weights and activations are quantized uniformly, it is possible to use `INT` matmuls to enable speed-ups beyond the savings from reduced memory movement. However, activation quantization remains difficult due to the presence of outlier channels, which necessitate sophisticated mitigation strategies (Wei et al., 2022; Dettmers et al., 2022; Xiao et al., 2022; Zhao et al., 2023; Ashkboos et al., 2023, 2024; Nrusimha et al., 2024; Lin et al., 2024). Weight-only quantization thus remains a popular choice for LLMs. Moreover, if only the weights are quantized, it is possible to reduce quantization error further by applying quantization at a more fine-grained levels (e.g., a block of 128 weight values) than at row- or column-level.

Non-uniform quantization generalizes uniform quantization by mapping weights to potentially *unequal* intervals (Miyashita et al., 2016; Zhou et al., 2017; Zhang et al., 2018; Yang et al., 2019). Lookup table (LUT) quantization is a flexible variant of non-uniform quantization which can map intervals to arbitrary values via a lookup table (Cardinaux et al., 2020; Wang et al., 2022). LUT quantization needs to trade off the size of the lookup table and the granularity of the groups at which the weights are quantized. For example, SqueezeLLM (Kim et al., 2023) applies K-means clustering at the column (output channel) level to obtain the lookup table, while NormalFloat quantization (Dettmers et al., 2023) uses a tensor-level lookup table obtained from the quantiles of a Normal distribution that is multiplicatively modified through group-level parameters. While it is possible to perform matmuls with activations/weights that are quantized non-uniformly (e.g., through LUT-based matmuls (Xu et al., 2021; Park et al., 2022)), these methods cannot leverage specialized accelerators on modern GPUs which are typically optimized for `FP` matmuls. We thus seek efficient kernels which can simultaneously make use of quantized representations (to minimize memory movement) as well as GPU-native matrix multiplications in `FP`.

3 `FLUTE`: A Fast and Flexible Kernel for Mixed-Type Matrix Multiplications

Let $\mathbf{Q} \in \mathbb{Z}^{k\times n}$ be a quantized matrix obtained from quantizing the weight matrix $\mathbf{W} \in \mathbb{R}^{k\times n}$ using a lookup table $\mathbf{T}$. Concretely, given a lookup table $\mathbf{T} = [v_0, \dots, v_{2^b-1}]$ where b is the number of bits and each v_i is a floating-point number, each entry of $\mathbf{Q}$ is given by,

$$\mathbf{Q}_{ij} = \operatorname{quantize}(\mathbf{W}_{ij}; \mathbf{T}) = \arg\min_{c} |\mathbf{W}_{ij} - v_c|,$$

where $\mathbf{Q}_{ij} \in \{0, \dots, 2^b-1\}$. Now let $\widehat{\mathbf{W}} \in \mathbb{R}^{k\times n}$ be the *de*quantized matrix where

$$\widehat{\mathbf{W}}_{ij} = \operatorname{dequantize}(\mathbf{Q}_{ij}, \mathbf{T}) = \mathbf{T}[\mathbf{Q}_{ij}].$$

Our objective is to perform a fast matrix multiplication between a dense input activation matrix $\mathbf{X} \in \mathbb{R}^{m\times k}$ (typically stored in `FP16`) and $\widehat{\mathbf{W}}$.

A straightforward implementation of such mixed-type matrix multiplication uses separate kernels. The first kernel loads the quantized matrix $\mathbf{Q}$ from the GPU's off-chip global memory into its on-chip memory, performs dequantization, and writes back the dequantized matrix $\widehat{\mathbf{W}}$ back to the DRAM. The second kernel is a standard `FP` matmul kernel over $\mathbf{X}$ and $\widehat{\mathbf{W}}$. This separate-kernel can introduce substantial overhead since $\widehat{\mathbf{W}}$ is moved back and forth. We can achieve faster matmuls by *fusing* the dequantization and matmul kernels, where we dequantize on chip and immediately use the dequantized values for the matmul.

Algorithm 1 FLUTE (Simplified)

```
Require: X^g: inputs in HBM
         Q^g: quantized weight in HBM
         S^g: scales in HBM
         T^g: lookup table in HBM
         Y^g: outputs in HBM

  # ─── Terminology and Shapes (of X only for brevity) ───
  # tile: a block of matrix entries that fits into shared memory
  # fragment: a block of tile entries that fit into registers
  # g(lobal), s(shared), r(egisters) denotes where data reside
  # X^g: [M tiles, K tiles, fragments per tile, fragment size]
  # X^s: [fragments per tile, fragment size]
  # X^r: [fragment size]
  # ─── Offline Preprocessing and Host-side Code ───
  Q_1^g, Q_2^g ← reorder_and_split(Q^g)                    ▷ Section 3.1
  T_v^g ← make_vectorized_LUT(T^g)                         ▷ Section 3.2
  tile_scheduler ← StreamKTileScheduler(N_SMs)             ▷ Section 3.3
  # launching blocks proportional to SMs
  N_blocks ← tile_scheduler.num_blocks()
  # ─── Kernel Launches and Device-side Code ───
  parallel for block index b ← 1 to N_blocks do
     # partitioning works for this block
     tile_scheduler.initialize(b)
     # copy lookup table from global to shared memory
     copy^{g→s}(T_v^g, T_v^s)
     # initialize the register-backed accumulator
     Y^r ← 0
     # main loop (pipelined in practice, not shown here)
     while ¬tile_scheduler.done() do
        # copy data tile from global to shared memory
        # (X is shown but the same applies to Q_1, Q_2, S)
        i_tile, j_tile, k_tile ← tile_scheduler.get_tile_index()
        copy^{g→s}(X^g[i_tile, k_tile, :, :], X^s)
        for fragment index t_fragment ← 1 to N_fragments do
           # copy data fragment from shared memory to registers
           # (X is shown but the same applies to Q_1, Q_2, S)
           copy^{s→r}(X^s[t_fragment, :], X^r)
           # combine two bit-slices in registers (3-bit)
           Q_{1+2}^r ← combine(Q_1^r, Q_2^r)                ▷ Section 3.1
           # vectorized dequantization in registers
           Ŵ^r ← vec_dequantize(Q_{1+2}^r, S^r, T^s)        ▷ Section 3.2
           # i.e., Y^r ← Y^r + X^r Ŵ^r
           Y^r ← tensor_core_mma(Y^r, X^r, Ŵ^r)
        end for
        if tile_scheduler.end_of_output_tile() then
           Ŷ^r ← to_fp16(Y^r)                               ▷ Section 3.3
           # write output tile from Registers to HBM
           copy^{r→g}(Ŷ^r, Y^g[i_tile, j_tile, :])
        end if
        # update the internal counter of scheduler
        tile_scheduler.step()
     end while
  end parallel for
```

However, implementing a fused weight-only LUT-quantized matmul that leads to speed-ups presents several challenges. For one, high-performance matmul necessitates the use of specialized primitives, such as Tensor Cores, which have strict requirements regarding the types, shapes, and layout of data. Second, efficient dynamic indexing is crucial for LUT-based dequantization; however, GPUs do not natively support dynamic indexing of a lookup table in their fastest on-chip registers. Finally, with smaller input matrices arising from low-bit and low-batch deployment, achieving workload balance across SMs is vital for maintaining speed, thus necessitating sophisticated partitioning strategies. FLUTE addresses these challenges through a combination of offline restructuring of the quantized weight matrix (§3.1), vectorization and duplication of the lookup table to mitigate shared bandwidth constraints (§3.2), and Stream-K workload partitioning to minimize wave quantization (§3.3). Alg. 1 gives a simplified version of the FLUTE kernel, while Fig. 1 shows a high-level overview. (See Alg. 2 in the Appendix for more details).

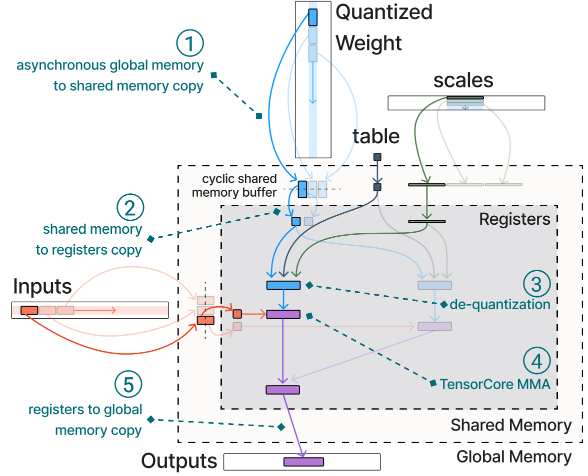

Figure 1: A simplified view of a kernel that fuses the dequantization and matmul steps. Each threadblock (group of threads) is responsible for computing one or more output tiles by performing the matrix product between specific rows of inputs and columns of weights. **(1)** The threadblock issues asynchronous copy instructions to fetch small chunks of input data (tiles) from global memory to shared memory. **(2)** As soon as a tile arrives in shared memory, it is further sliced into smaller chunks (fragments) and copied into registers. **(3)** Once all necessary components are in the registers, the quantized matrix undergoes dequantization. **(4)** The dequantized matrix and inputs are then processed by Tensor Cores using MMA (Matrix Multiply Accumulate) instructions. **(5)** Finally, the accumulated results are written back from the registers to the outputs in global memory.

3.1 Offline Matrix Restructuring

Modern GPUs feature specialized primitives (Tensor Cores)—distinct from general-purpose vector ALUs—which can substantially accelerate dense matrix multiplications. For example, A100's FP16 tensor core matmuls are 16× faster than FP32 vector matmuls. However, this acceleration comes at the expense of generality and programmability. Tensor Core MMA (matrix-multiply-accumulate) operations require the input matrices to adhere to specific layout specifications within the registers of 32 threads. The fused kernel needs to first load fragments of the quantized weight into registers, dequantize the matrix, and then perform the MMA operation between the input fragments and dequantized matrix fragments. This necessitates that the post-dequantization matrix layout meets the required specifications. While runtime data reordering is one approach, it introduces a substantial number of operations. Instead, we leverage the fact that $\mathbf{Q}$

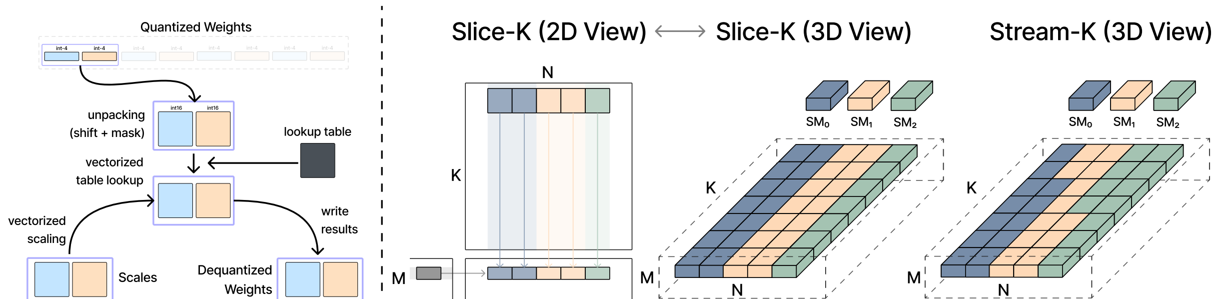

Figure 2: Vectorized Lookup Table Design (Left). Instead of dequantizing one element at a time, we vectorize the lookup table by creating another table that holds the values of all possible pairs of indices. This can look up two values simultaneously, followed by efficient vectorized scaling operations. **Stream-K Work Decomposition (Right).** In classic work decomposition, output tile production is independently assigned to threadblocks. Each threadblock processes one (or more) rows of the left operand and one (or more) columns of the right operand, slicing down the inner K dimension to compute the corresponding output tile (Slice-K). However, when the weight matrix is heavily quantized, the reduced size can lead to "stragglers" in Slice-K due to uneven workload assignment. Stream-K (Osama et al., 2023) addresses this by decomposing work at a finer granularity, enabling multiple threadblocks to collaboratively compute a single output tile.

(the quantized weights) are static during inference, allowing for offline weight reordering such that after dequantization, the weights are already laid out exactly in the expected format (Frantar et al., 2024; Xia et al., 2024b; Lin et al., 2024).

The above strategy is difficult to straightforwardly extend to the case of non-evenly-divisible bit widths (e.g., 3 bits). Kernels employ vectorized data access when loading data from global to shared memory. Hence each thread should access the quantized weight in granularity of 128 bits (or at least in powers of 2). While this could be addressed by padding, this would be inefficient. We instead split the (3-bit) quantized weight into two partitions (Xia et al., 2024b), or bit-slices: one containing the 1-bit portion and the other the 2-bit portion, and issue two separate vectorized (asynchronous) data copy instructions. Once the two bit-slices are loaded into the registers, we combine them before dequantization.

3.2 Vectorized Lookup in Shared Memory

During dequantiation each element c in the quantized array needs to access a 16-bit element $\mathbf{T}[c]$ from the lookup table. Each thread needs to access the lookup table using different indices, and such "non-uniform access" can degrade runtime performance when implemented naïvely. While storing the lookup table in on-chip shared memory can reduce expensive off-chip memory access, this still introduces significant traffic to shared memory. To reduce memory access instructions we "vectorize" the lookup operation by accessing two elements at a time (Fig. 2, left). We build an alternative lookup table for every possible *pair* of values, with each element containing a tuple of 16-bit values. The storage overhead from the vectorized lookup table is minimal compared to the rest of memory usage. For example, for 4-bit LUT the table has only 2^4 elements of 16-bit values, and thus the vectorized table has only 2^8 elements of 32-bit values, slightly more than 1KB of storage. This is a fraction of the 48KB-163KB of shared memory on modern GPUs.

Reducing bank conflicts. Shared memory is organized such that each successive 32-bit segment corresponds to a "bank", and there are 32 such banks. Each memory address in shared memory corresponds to the $\lfloor \frac{\text{addr}}{32} \rfloor \bmod 32$ bank. If threads in a warp access data from different banks, access is parallelized. However, if two threads access data from the same bank (but not the same address), the access is serialized. For the 4-bit and 3-bit vectorized tables, a simple implementation could thus cause up to 8-way bank conflicts (4-bit) or 2-way bank conflicts (3-bit). To mitigate this, we duplicate the 4-bit vectorized lookup table multiple times, placing copies in different memory banks, which allows threads to access values from different banks with reduced bank conflicts.

3.3 Stream-K Workload Partitioning

For high SM occupancy, standard matmul implementations block the computation using a data-parallel tiling of the output matrix, where a group of threads ("thread block") is assigned to compute the work on one output tile. This is shown on the left of Figure 2. As each thread block can only occupy one SM, it is important to avoid "wave quantization", which happens when the number of output tiles is not an even multiple of the number of processor cores. In this case the last wave uses only a subset of the cores, leaving the rest idle.

Wave quantization and workload imbalance are especially problematic in low-bit and low-batch

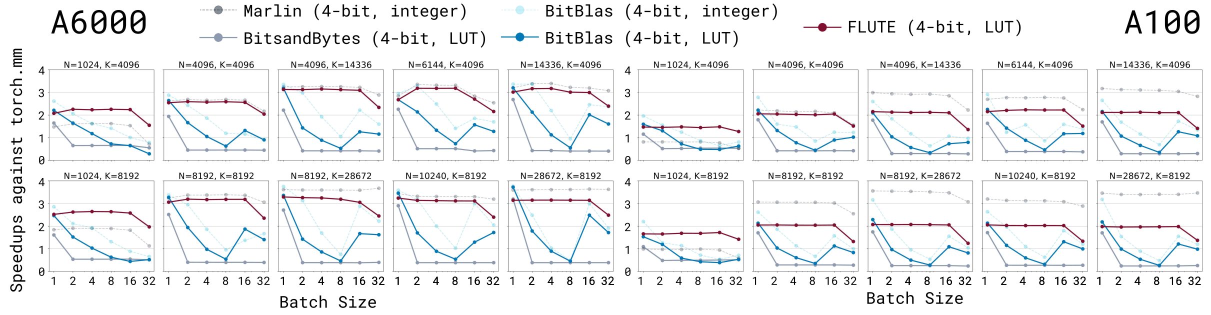

Figure 3: Runtime performance of FLUTE in the standard W4G128 setting where, the weights are quantized to 4 bits in groups of 128. We show speedup against 16-bit `torch.mm`. The matrix shapes for our benchmarks are selected based on those used in Llama-3-8B (top row) and Llama-3-70B (bottom row) models. For each M-N-K shape tuple, we generate three random sets of data, run each kernel on the data 100 times, and average. While our main comparisons are against other LUT kernels (`bitsandbytes`, `BitBLAS-NF4`), for reference we also include comparisons with kernels that only support uniform (integer) dequantization (`Marlin`, `BitBLAS`). These results are represented with dashed lines in our figures.

scenarios, which result in smaller input matrices (activations and quantized weights), thus making the effect of wave quantization more pronounced. To mitigate this, we implement a method known as Stream-K workload decomposition (Osama et al., 2023), which distributes the tiles such that each SM's computations can span beyond specific rows or columns. This method is depicted on in Fig. 2 (Right). Here, the 35 M-N-K tiles are more evenly divided among the 3 SMs than in the simpler Slice-K partitioning (Figure 2, middle), in which SM's computations do not span beyond rows/columns.

Mixed precision accumulation and global reduction. In Stream-K, when multiple SMs compute the same M-N dimension across different K tiles, they must reconcile their partial sums in off-chip global memory. SMs that complete their share of K tiles write their partial sums to a global scratch space, allowing subsequent SMs to read, reduce, and write back these sums. For numerical stability, most kernels perform multiplications in `FP16` but accumulate results in `FP32`. However, writing to global memory in `FP32` results in significant traffic. We thus implement in-register accumulation in `FP32` and globally reduce partial sums in `FP16`.

4 Experiments

Our experiments consist of two settings: *kernel-level* experiments which compare FLUTE matmuls standalone against existing mixed-input matmul kernels (§4.1), and *end-to-end* experiments which assess whether practicals speed-ups are obtainable on realistic LLM workloads (§4.2).

4.1 Kernel Benchmarks

For each matrix size, we compile multiple instantiations of the kernel with various configurations, including different tile sizes, pipeline stages, and the number of lookup table duplicates, selecting the best-performing configuration based on benchmarking.[1] We compare FLUTE against a collection of weight-quantized matrix multiplication kernels, including those capable of flexible LUT-based dequantization such as `bitsandbytes` (Dettmers et al., 2023)[2] and `BitBLAS` (Wang et al., 2024).

LUT quantization method. There are many methods for LUT quantization; we follow the popular NormalFloat LUT quantization scheme (Dettmers et al., 2023), where a tensor-level table $\mathbf{T}$ is modified to be a group-level table via a scaling factor s_g for each group g, resulting in a group-level table $\mathbf{T}_g = [s_g \cdot v_0, \dots, s_g \cdot v_{2^b-1}]$. Here $s_g \in \mathbb{R}^+$ is a scalar that varies per group. This approach requires maintaining a tensor-level lookup table $\mathbf{T}$ and group-level scalars $\{s_g\}_{g=1}^{d^2/g}$, and thus incur almost the same memory overhead as uniform quantization (which also requires maintaining the group-level scalars). While we primarily focus on LUT kernels, for completeness we also compare against high-performance kernels specialized for uniformly quantized weights (`BitBLAS`[3] and `Marlin`[4] (Frantar et al., 2024)). These kernels do not require dynamic indexing into a lookup table and can perform dequantization in registers using highly tuned PTX assembly instructions that are not applicable to LUT-based dequantization.

Results. Figure 3 presents the results with the standard setting of 4-bit quantization and a group size of 128, where memory traffic is reduced by

[1] Concretely, we randomly generate three sets of input data based on the matrix shapes for 8B/70B LLMs, run the kernel on the data 100 times, and report the average performance on both A100 and A6000 GPUs.

[2] For most of the kernels, we pre-allocate the output memory buffer and use the `out` keyword to exclude the memory allocation time from our measurements. However, as of this writing, `bitsandbytes` still allocates memory in some cases. Our preliminary experiments indicate that this introduces an overhead of approximately 2.5%.

[3] https://github.com/microsoft/BitBLAS

[4] https://github.com/IST-DASLab/marlin

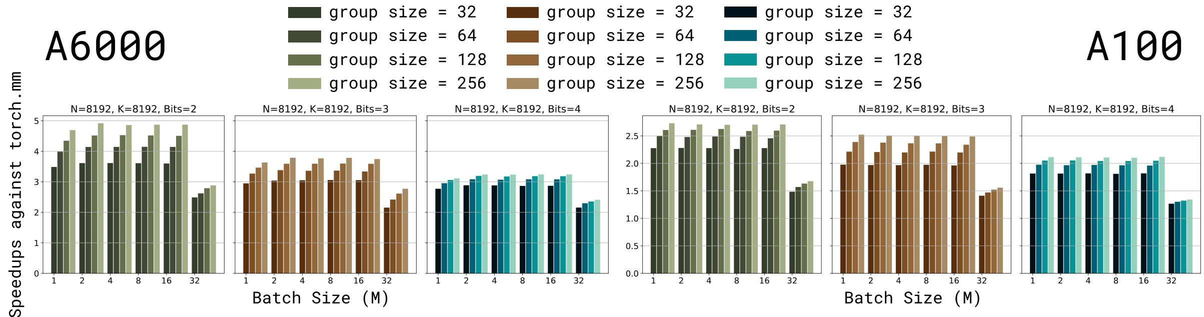

Figure 4: Runtime performance at various bit-widths and group sizes with N=K=8192. FLUTE consistently achieves speedups across different settings, including the in the 3-bit configuration.

4x (modulo the overhead coming from scales). FLUTE achieves favorable performance across a wide range of matrix shapes on both A6000 and A100, occasionally nearing the peak theoretical speedup (of 4x) on A6000. Other LUT-compatible kernels achieve similar speedups only with a batch size of 1, and their performance quickly degrades. FLUTE also compares favorably to Marlin, which is highly specialized for cases where the input is FP16 and the weight is uniform-quantized to INT4.

We further showcase the flexibility of FLUTE by experimenting with different group sizes not just in terms of its lookup-table design but also in supporting various bit-widths and group sizes. In particular, FLUTE can perform multiplications with 3-bit matrices (§3.1), a capability that the aforementioned alternatives do not support. The results in Figure 4 demonstrate consistent speed-ups over torch.mm across across a wide rage of settings.

4.2 End-to-End LLM Benchmarks

As an application of FLUTE, we experiment with quantizing LLaMA3-8B and LLaMA3-70B. The LLaMA3 family of models has been found to be more difficult to quantize than other open source models (Huang et al., 2024), and thus presents a testing ground for different quantization strategies.

For the LUT quantization method, we use a simple extension of NormalFloat (NF) quantization (Dettmers et al., 2023). Standard NF quantization calculates 2^{b-1} evenly-spaced values from $[\delta, \frac{1}{2}]$, and $2^{b-1}+1$ evenly-spaced values from $[\frac{1}{2}, 1-\delta]$, where $\delta = \frac{1}{2}(\frac{1}{30}+\frac{1}{32})$. This results in 2^b probability values $[p_0, \dots, p_{2^b-1}]$ where $p_0 = \delta, p_{2^{b-1}-1} = \frac{1}{2}$, and $p_{2^b-1} = 1-\delta$. These probabilities are converted into quantiles $[q_0, \dots, q_{2^b-1}]$ where $q_i = \Phi^{-1}(p_i)$ is the Gaussian quantile for p_i. The quantiles are then normalized to $[-1, 1]$ by $\tilde{q}_i = \frac{q_i}{q_{2^b-1}}$. Then, given a group of weights $\mathbf{u} = [u_1, \dots, u_B]$ and the absmax value $s = \max(|\mathbf{u}|)$ for that group, the weights u_j in this group are quantized to the nearest quantile, i.e., $c_j = \arg\min_{i \in \{0,\dots,2^b-1\}} \left|\tilde{q}_i - \frac{u_j}{s}\right|$. Given an NF-quantized matrix $\mathbf{Q} \in \{0, \dots, 2^b-1\}^{k \times n}$, the matmul kernel loads the tensor-level lookup table $\mathbf{T} = [\tilde{q}_0, \dots \tilde{q}_{2^b-1}]$, as well as the group-level scales $s_1, \dots s_{\frac{kn}{B}}$, and then dequantizes via $\mathbf{T}[\mathbf{Q}_{ij}] \cdot s_{(i \times j) \mod B}$.

Our simple extension builds upon the above by using calibration data to refine the scales, which has been found to be beneficial for uniform quantization (Shao et al., 2023). Since the lookup table consists of quantiles from $\mathcal{N}\left(0, \sigma^2\right)$ with standard deviation $\sigma = \frac{1}{\Phi^{-1}(1-\delta)}$, we can reformulate the quantization function as $c_j = \arg\min_{i \in \{0,\dots,2^b-1\}} |s\tilde{\sigma} q_i - u_j|$. For learning, we initialize $\tilde{\sigma} = \frac{1}{\Phi^{-1}(1-\delta)}$ and optimize this with gradient descent against the negative log-likelihood of calibration samples, where we use the straight-through estimator.[5] After learning, we can save $\frac{s\tilde{\sigma}}{\sigma}$ as the new scale, and hence the number of scalar values to be loaded for dequantization remains unchanged. We use use 128 examples of length 2048 from WikiText-2 training as our calibration dataset.

We conducted end-to-end evaluations by integrating the FLUTE kernels into two libraries:

1. GPT-Fast[6] is a simple yet performant PyTorch-native implementation for transformer text generation. We follow most of its default settings, running benchmarks with a batch size of 1.[7] We additionally use torch.compile to optimize the model, which, in early experiments, nearly tripled the throughput of the 16-bit unquantized model.

2. vLLM (Kwon et al., 2023) is a high-throughput

[5]We also experimented with a variant of this approach where each value of the tensor-level lookup table is updated to be the average of all the weights that were bucketed to that value (as in K-means). We did not find meaningful improvements with this approach.

[6]https://github.com/pytorch-labs/gpt-fast

[7]This configuration makes the reported tokens per second (tokens/s) equivalent to "tokens/s/user." We set the prompt length to just 1, focusing our measurements on the decoding step in text generation rather than the prefill stage. We also do not use CUDA Graphs due to its incompatibility with FLUTE.

Quantization Configuration	GB	PPL ↓ Wiki	C4	Speedups ↑ A6000	x4	A100	x2
8B Unquant.	15.1	6.1	9.2	1.0x		1.0x	
W4G32	5.7	6.1	9.4	2.0x		1.3x	
W4G64	5.5	6.1	9.4	2.1x		1.3x	
W4G128	5.4	6.2	9.5	2.2x		1.3x	
W3G32	4.9	6.9	11.0	2.1x		1.3x	
W3G64	4.7	7.2	11.3	2.3x		1.4x	
W3G128	4.6	7.5	11.7	2.4x		1.5x	
70B Unquant.	131.7	2.9	6.9	OOM	1.0x	OOM	1.0x
W4G32	40.1	3.0	7.0	2.9x	1.9x	1.8x	1.4x
W4G64	38.1	3.0	7.1	3.1x	1.9x	1.8x	1.5x
W4G128	37.1	3.1	7.2	3.4x	1.9x	1.9x	1.5x
W3G32	32.1	3.9	8.0	3.1x	1.9x	2.0x	1.5x
W3G64	30.1	4.1	8.4	3.8x	1.9x	2.3x	1.7x
W3G128	29.1	5.2	10.1	4.1x	1.9x	2.4x	1.7x

Table 1: Perplexity and decoding speed of LLaMA-3 with learned NF quantization using various quantization configurations, denoted as W(bits)G(group-size). Decoding speedup is measured in tokens per GPU-second. The unquantized LLaMA-3 70B model requires multiple GPUs with Tensor Parallelism. Therefore, we compare its speed to that of the quantized models using one GPU, accounting for the number of GPUs used (Xia et al., 2024b), and to the quantized models with Tensor Parallelism applied (labeled as x4 and x2). For the 8B models, since all models fit into one GPU, we report only single GPU results. Please see Table 3 for details.

and memory-efficient inference and serving engine for LLMs widely used in practice. We benchmarked the latency of processing a single batch of requests, following most of its default settings, but varied the input length, output length, and batch size to assess performance under different conditions.

For the 70B model, the unquantized model does not fit into a single GPU. Consequently, we apply tensor parallelism across 4xA6000 or 2xA100 GPUs. Since the quantized model fits into a single GPU, we report two sets of numbers (single- and multi-GPUs) to represent different use cases.

Results. We first compare our "learned NF quantization" approach against standard 4- and 3-bit setting with group size 128 against other quantization methods. The results are shown in Table 2, where we find that this variant of LUT quantization improves upon ordinary NF quantization and compares favorably against existing baselines. See Tables 4 and 5 of the appendix for the full results. We also find that combining NF with AWQ (Lin et al., 2023) to be beneficial, although a learned NF+AWQ did not help. (However we emphasize that the quantization method itself is not the main contribution of the present work.) We next exploit the flexibility of FLUTE and conduct end-to-end experiments with various bit- and group-size settings. This is shown in Table 1. With small enough group sizes, our approach is able to almost approach the

#P	Method	Wiki PPL ↓ 4-bit	3-bit	C4 PPL ↓ 4-bit	3-bit	LLM Eval ↑ 4-bit	3-bit
8B	*Unquantized*	*6.1*		*9.2*		*68.6*	
	RTN	8.5	27.9	13.4	1.1E2	63.9	40.2
	GPTQ	6.5	8.2	10.4	13.7	67.3	61.7
	AWQ	6.6	8.2	9.4	11.6	68.2	64.4
	OmniQuant	6.6	8.4	10.1	13.5	68.3	62.4
	NF	6.6	9.2	9.5	13.0	68.0	62.3
	NF + AWQ	6.5	8.0	9.3	11.5	67.8	65.1
	NF (learned)	6.2	7.5	9.5	11.7	67.9	63.7
70B	*Unquantized*	*2.9*		*6.9*		*75.3*	
	RTN	3.6	11.8	8.9	22.0	74.3	48.0
	GPTQ	3.3	5.2	6.9	10.5	74.9	70.6
	AWQ	3.3	4.8	7.0	8.0	74.9	73.2
	OmniQuant	3.3	5.4	7.5	9.3	74.2	70.2
	NF	3.4	8.7	7.6	16.7	74.0	64.3
	NF + AWQ	3.2	4.6	6.9	7.8	75.2	73.8
	NF (learned)	3.1	5.2	7.2	10.1	74.4	66.4

Table 2: Evaluation of post-training quantization on LLaMA3-8B and LLaMA3-70B. The RTN, GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2023) results are from Huang et al. (2024); the rest are from our implementations. All non-NF methods use uniform weight quantization.

16 bit baseline in terms of WikiText2 perplexity.[8] We are able to observe meaningful speedups even in the end-to-end case over an optimized baseline.

Finally, we evaluated end-to-end latency using the popular model service framework vLLM (Kwon et al., 2023). Based on our earlier experiments, we selected a group size of 64, which strikes a good balance between quality and speed. We conducted experiments across various configurations, including bit precision, model sizes, number of GPUs, input lengths, output lengths, and batch sizes. Additionally, we conducted experiments with the newly released Gemma-2 models (9B and 27B). For the largest open-sourced Gemma-2 27B model, which fits into a 2xA6000 and 1xA100 setup, we adjusted the tensor parallelism settings accordingly. The results, presented in Fig. 5, further showcase the end-to-end performance of the kernel.

To demonstrate the scalability of our approach, we evaluated FLUTE on the LLaMA-3.1 (405B) model (Dubey et al., 2024), shown in Fig. 6. It is worth noting that, without quantization, the 405B model's parameters alone would require multiple GPU nodes. However, with FLUTE, we were able to perform inference on a single node, highlighting the efficiency and scalability of our solution.

5 Discussion and Conclusion

Early work on LLM quantization generally worked with uniform quantization methods (Frantar et al., 2022; Dettmers et al., 2022; Xiao et al., 2022).

[8]Note that the WikiText-2 validation data is different from the calibration data.

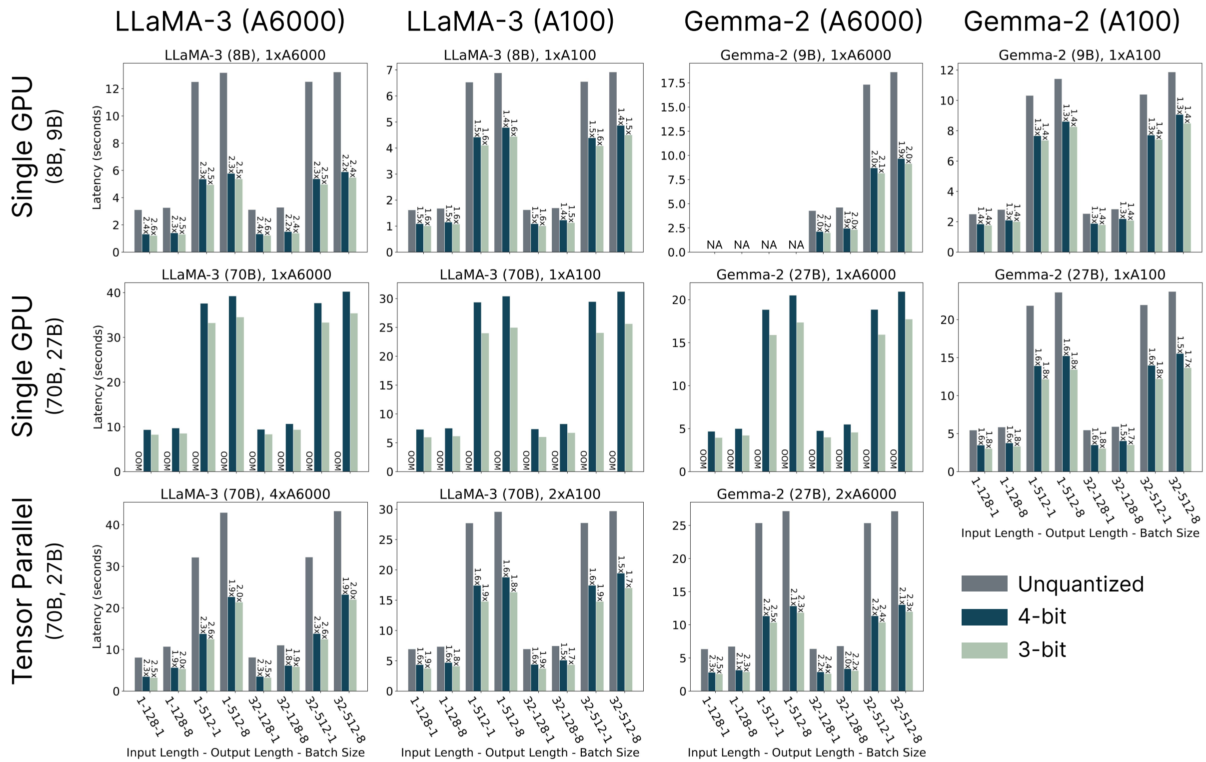

Figure 5: End-to-end latency benchmark for processing a single batch of requests using vLLM. We evaluated LLaMA-3 (8B and 70B) and Gemma-2 (9B and 27B) models with various configurations, including different bits, model sizes, number of GPUs, input lengths, output lengths, and batch sizes. The models were quantized using a group size of 64 to achieve a good balance between quality and speed.

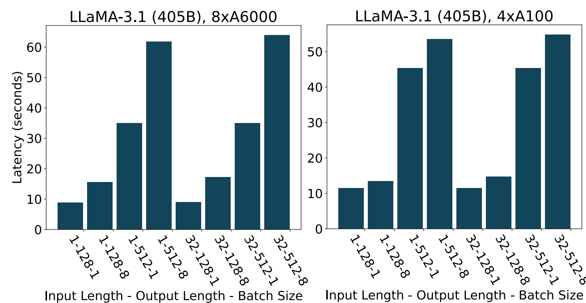

Figure 6: End-to-end vLLM latency benchmark for processing a single batch of requests using LLaMA-3.1 (405B, W4G64).

More recent work has shown the benefits of LUT-quantization, both from PTQ (Kim et al., 2023) and finetuning (Dettmers et al., 2023) perspectives. Insofar as lookup tables can represent flexible quantization functions, our hope is that FLUTE can enable researchers and practitioners to explore new quantization algorithms that can learn better lookup tables (Yamamoto, 2021; Cardinaux et al., 2020; Wang et al., 2022). For example, recent work has found that codebook-based quantization schemes—which generalize lookup tables to vector-valued values—can enable even lower-bit (e.g., 2-bit) LLM quantization without significant performance degradations (Tseng et al., 2024; Egiazarian et al., 2024). We anticipate that ideas from this work can aid in developing kernels for such methods.

Algorithmic considerations aside, one of the main challenges in developing fused quantized matrix multiplication kernels stems from the lack of hardware support for "mixed-type" instructions, necessitating software-level implementations. Existing Tensor Core instructions support scenarios where the input and output/accumulation data have different types (e.g., compute in FP16 and output/accumulate in FP32). However, they do not support cases where the input operands themselves are of different types (e.g., FP16 inputs and INT4 weights). As weight-only quantization becomes increasingly common in LLM inference applications, native support for such instructions in future hardware could be beneficial. Additionally, the lack of in-register dynamic indexing means that developers must devise software solutions. Enhanced hardware acceleration for indexing into small lookup tables could also prove beneficial in the upcoming generations of AI accelerator hardware.

6 Conclusion

This work introduces FLUTE, a CUDA kernel designed for fused quantized matrix multiplications to accelerate LLM inference. FLUTE offers flexibility, supporting flexible mappings between quantized and dequantized values through a lookup table, and accommodating a wide range of bit widths and group sizes. We demonstrate its performance through both kernel-level benchmarks and end-to-end evaluations on state-of-the-art LLMs.

Limitations

FLUTE has several limitations. For one, it is mostly optimized for Ampere-generation GPUs, and it does not take advantage of the newer hardware features available in subsequent generations, such as Hopper GPUs (H100). However, the majority of the methods discussed could still be applicable to the newer hardware. For Ampere generation GPUs, the latest tensor cores support performing MMA operations on matrix fragments of shape [16,16]x[16,8]. When the batch size is smaller than 16, input data needs to be padded within shared memory. Although this padding increases on-chip data movements (between shared memory and registers) and computations, it does not increase data movement between off-chip and on-chip memory, allowing us to achieve speed-ups in memory-bound cases. In such scenarios, switching to SIMT cores could further enhance performance. FLUTE is designed for memory-bound scenarios such as LLM decoding. Its performance tends to degrade with larger batch sizes, which are more common during training when the workload becomes more compute-bound. Finally, while FLUTE demonstrates strong performance among kernels that support LUT-based dequantization, its performance on A100s still falls short of the peak performance that kernels specialized for uniformly quantized matrices can achieve.

Acknowledgements

We thank Yijie Bei and Dmytro Ivchenko for helpful discussion. HG was supported by a Microsoft PhD Fellowship. EX acknowledges the support of NGA HM04762010002, NSF IIS1955532, NSF CNS2008248, NIGMS R01GM140467, NSF IIS2123952, NSF DMS2027737, NSF BCS2040381, NSF DMS2112273, NSF IIS2311990, Semiconductor Research Corporation (SRC) AIHW award 2024AH3210, and DARPA ECOLE HR00112390063. This study was additionally supported by MIT-IBM Watson AI Lab and the MLA@CSAIL initiative.

References

Saleh Ashkboos, Ilia Markov, Elias Frantar, Tingxuan Zhong, Xincheng Wang, Jie Ren, Torsten Hoefler, and Dan Alistarh. 2023. Towards end-to-end 4-bit inference on generative large language models. *arXiv preprint arXiv:2310.09259*.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. Quarot: Outlier-free 4-bit inference in rotated LLMs. *arXiv preprint arXiv:2404.00456*.

Fabien Cardinaux, Stefan Uhlich, Kazuki Yoshiyama, Javier Alonso García, Lukas Mauch, Stephen Tiedemann, Thomas Kemp, and Akira Nakamura. 2020. Iteratively training look-up tables for network quantization. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):860–870.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems*, 36.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. 2024. Extreme compression of large language models via additive quantization. *arXiv preprint arXiv:2401.06118*.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate post-training compression for generative pretrained transformers. *arXiv preprint arXiv:2210.17323*.

Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. 2024. Marlin: Mixed-precision auto-regressive parallel inference on large language models. *arXiv preprint arXiv:2408.11743*.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. AI and memory wall. *IEEE Micro*.

Han Guo, Philip Greengard, Eric P Xing, and Yoon Kim. 2024. LQ-LoRA: Low-rank plus quantized matrix decomposition for efficient language model finetuning. In *Proceedings of ICLR*.

Wei Huang, Xudong Ma, Haotong Qin, Xingyu Zheng, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu, and Michele Magno. 2024. How good are low-bit quantized LLaMA3 models? an empirical study. *arXiv preprint arXiv:2404.14047*.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. 2023. SqueezeLLM: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. AWQ: Activation-aware weight quantization for LLM compression and acceleration. *arXiv preprint arXiv:2306.00978*.

Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. QServe: W4A8KV4 quantization and system co-design for efficient LLM serving. *arXiv preprint arXiv:2405.04532*.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024a. The era of 1-bit LLMs: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*.

Yuexiao Ma, Huixia Li, Xiawu Zheng, Feng Ling, Xuefeng Xiao, Rui Wang, Shilei Wen, Fei Chao, and Rongrong Ji. 2024b. AffineQuant: Affine transformation quantization for large language models. *arXiv preprint arXiv:2403.12544*.

Daisuke Miyashita, Edward H Lee, and Boris Murmann. 2016. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*.

Aniruddha Nrusimha, Mayank Mishra, Naigang Wang, Dan Alistarh, Rameswar Panda, and Yoon Kim. 2024. Mitigating the impact of outlier channels for language model quantization with activation regularization. *arXiv preprint arXiv:2404.03605*.

Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D Owens. 2023. Stream-K: Work-centric parallel decomposition for dense matrix-matrix multiplication on the GPU. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 429–431.

Gunho Park, Baeseong Park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. 2022. LUT-GEMM: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. *arXiv preprint arXiv:2206.09557*.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. 2023. OmniQuant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. 2024. Quip#: Even better LLM quantization with hadamard incoherence and lattice codebooks. *arXiv preprint arXiv:2402.04396*.

Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. 2024. Ladder: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

Longguang Wang, Xiaoyu Dong, Yingqian Wang, Li Liu, Wei An, and Yulan Guo. 2022. Learnable lookup table for neural network quantization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12423–12433.

Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. 2022. Outlier suppression: Pushing the limit of low-bit transformer language models. *Advances in Neural Information Processing Systems*, 35:17402–17414.

Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2024a. Flash-LLM: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. In *Proceedings of VLDB*.

Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, et al. 2024b. FP6-LLM: Efficiently serving large language models through FP6-centric algorithm-system co-design. *arXiv preprint arXiv:2401.14112*.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2022. SmoothQuant: Accurate and efficient post-training quantization for large language models. *arXiv:2211.10438*.

Shiyu Xu, Qi Wang, Xingbo Wang, Shihang Wang, and Terry Tao Ye. 2021. Multiplication through a single look-up-table (LUT) in CNN inference computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1916–1928.

Kohei Yamamoto. 2021. Learnable companding quantization for accurate low-bit neural networks. In *Proceedings of th e IEEE/CVF conference on computer vision and pattern recognition*, pages 5029–5038.

Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-sheng Hua. 2019. Quantization networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7308–7316.

Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned quantization for

highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2023. Atom: Low-bit quantization for efficient and accurate LLM serving. *arXiv preprint arXiv:2310.19102*.

Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless CNNs with low-precision weights. *arXiv preprint arXiv:1702.03044*.

A Appendix

Please see Algorithm 2 for a detailed version of the algorithm, and Tables 4, 5 for detailed experimental results.

Algorithm 2 FLUTE

```
Require: $\mathbf{X}^g$: inputs in HBM
         $\mathbf{Q}^g$: quantized weight in HBM
         $\mathbf{S}^g$: scales in HBM
         $\mathbf{T}^g$: lookup table in HBM
         $\mathbf{Y}^g$: outputs in HBM
         $\mathbb{S}_{semaphore}$: scratch space for semaphore in HBM
         $\mathbb{S}_{partials}$: scratch space for partials in HBM

  # ——— Offline Preprocessing and Host-side Code ———
  $\mathbf{Q}_1^g, \mathbf{Q}_2^g \leftarrow$ reorder_and_split($\mathbf{Q}^g$)                ▷ Section 3.1
  $\mathbf{T}_v^g \leftarrow$ make_vectorized_LUT($\mathbf{T}^g$)                                ▷ Section 3.2
  tile_scheduler $\leftarrow$ StreamKTileScheduler($N_{SMs}$)                                      ▷ Section 3.3
  $N_{blocks} \leftarrow$ tile_scheduler.num_blocks()               ▷ launching blocks proportional to SMs

  # ——— Kernel Launches ———
  parallel for block index b $\leftarrow$ 1 to $N_{blocks}$ do
     tile_scheduler.initialize(b)                                   ▷ partitioning works for this block
     allocate$^s$($\mathbf{X}^s, \mathbf{Q}_1^s, \mathbf{Q}_2^s, \mathbf{S}^s, \mathbf{T}_v^s$)            ▷ allocate circular buffer in shared memory
     allocate$^r$($\mathbf{X}^r, \mathbf{Q}_1^r, \mathbf{Q}_2^r, \mathbf{Q}_{1+2}^r, \mathbf{S}^r, \widehat{\mathbf{W}}^r, \mathbf{Y}^r, \widehat{\mathbf{Y}}^r$)   ▷ allocate fragments in register
     $i_{read}^s \leftarrow 0, i_{write}^s \leftarrow 0$                                             ▷ shared memory pipeline index
     $i_{current}^r \leftarrow 0, i_{next}^r \leftarrow 0$                                           ▷ register pipeline index

     # ——— Global Memory to Shared Memory Prefetch ———
     copy$^{g\to s}$($\mathbf{T}_v^g, \mathbf{T}_v^s$)
     for prefetch stage $t_{prefetch} \leftarrow 0$ to $N_{stages-1}$ do
        $i_{read}^g \leftarrow$ tile_scheduler.get_tile_index()
        copy$^{g\to s}$($\mathbf{X}^g[b, :, i_{read}^g], \mathbf{X}^s[:, i_{write}^s]$)   # and the same for $\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{S}$
        $i_{write}^s = (i_{write}^s + 1) \bmod N_{stages}$
        tile_scheduler.step()
     end for

     # ——— Shared to Registers Prefetch ———
     wait_for_one_tile()
     copy$^{s\to r}$($\mathbf{X}^s[i_{current}^r, i_{read}^s], \mathbf{X}^r[i_{current}^r]$)   # and the same for $\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{S}$

     # ——— Pipelined Main Loop ———
     $\mathbf{Y}^r \leftarrow \mathbf{0}$
     while ¬tile_scheduler.done() do
        for fragment index $t_{fragment} \leftarrow$ 1 to $N_{fragments}$ do
           if $t_{fragment}$ == $N_{fragments}$ then
              wait_for_one_tile()                                   ▷ Wait until the next prefetched tile
              $i_{read} = (i_{read} + 1) \bmod N_{stages}$
           end if
           $i_{next}^r \leftarrow i_{curr}^r$ + 1                             ▷ overlap MMA with next register load
           copy$^{s\to r}$($\mathbf{X}^s[i_{next}^r, i_{read}^s], \mathbf{X}^r[i_{next}^r]$)   # and the same for $\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{S}$
           if $t_{fragment}$ == 0 then
              $i_{read}^g \leftarrow$ tile_scheduler.get_tile_index()                ▷ overlap MMA with SMEM load
              copy$^{g\to s}$($\mathbf{X}^g[b, :, i_{read}^g], \mathbf{X}^s[:, i_{write}^s]$)   # and the same for $\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{S}$
              $i_{write}^s = (i_{write}^s + 1) \bmod N_{stages}$
              tile_scheduler.step()
           end if
           $\mathbf{Q}_{1+2}^r \leftarrow$ combine($\mathbf{Q}_1^r, \mathbf{Q}_2^r$)                                ▷ Section 3.1
           $\widehat{\mathbf{W}}^r \leftarrow$ vectorized_dequantization($\mathbf{Q}_{1+2}^r, \mathbf{S}^r, \mathbf{T}^s$)        ▷ Section 3.2
           $\mathbf{Y}^r \leftarrow$ tensor_core_mma($\mathbf{Y}^r, \mathbf{X}^r, \widehat{\mathbf{W}}^r$)
        end for

        # ——— StreamK Fixup (partial sums reductions) ———
        if tile_scheduler.end_of_output_tile() then
           $i_{fixup} \leftarrow$ tile_scheduler.get_fixup_index()
           $\widehat{\mathbf{Y}}^r \leftarrow$ to_fp16($\mathbf{Y}^r$)                                        ▷ Section 3.3
           if ¬tile_scheduler.finished_output_tile() then          ▷ share partial sums through scratch
              accumulate_and_store_partials($\widehat{\mathbf{Y}}^r, \mathbb{S}_{partials}[i_{fixup}]$)
              signal($\mathbb{S}_{semaphore}[i_{fixup}]$)
           else
              if ¬tile_scheduler.started_output_tile() then                    ▷ aggregate partial sums
                 wait($\mathbb{S}_{semaphore}[i_{fixup}]$)
                 $\widehat{\mathbf{Y}}^r \leftarrow \widehat{\mathbf{Y}}^r$ + load_partials($\mathbb{S}_{partials}[i_{fixup}]$)
              end if
              $i_{output} \leftarrow$ tile_scheduler.get_output_tile_index()
              epilogue($\widehat{\mathbf{Y}}^r, \mathbf{Y}^g[i_{output}]$)               ▷ Write output tile from Registers to HBM
           end if
        end if
     end while
  end parallel for
```

Model	Configuration				Perplexity		Tokens / Second							
	Bits	Group	Bits / Param	GB	WikiText2	C4	1xA6000		4xA6000		1xA100		2xA100	
LLaMA-3 8B	16	N/A	16.00	15.1	6.1	9.2	44.8	1.0x			90.2	1.0x		
	4	32	4.50	5.7	6.1	9.4	91.3	2.0x			113.7	1.3x		
	4	64	4.25	5.5	6.1	9.4	95.9	2.1x			119.4	1.3x		
	4	128	4.13	5.4	6.2	9.5	98.1	2.2x			121.6	1.3x		
	4	256	4.06	5.4	6.3	9.5	99.8	2.2x	-		121.7	1.3x	-	
	3	32	3.50	4.9	6.9	11.0	91.9	2.1x			117.7	1.3x		
	3	64	3.25	4.7	7.2	11.3	104.1	2.3x			128.5	1.4x		
	3	128	3.13	4.6	7.5	11.7	108.1	2.4x			133.5	1.5x		
	3	256	3.06	4.6	7.9	12.2	110.0	2.5x			135.5	1.5x		
LLaMA-3 70B	16	N/A	16.00	131.7	2.9	6.9	OOM	OOM	17.2	1.0x	OOM	OOM	19.9	1.0x
	4	32	4.50	40.1	3.0	7.0	12.6	-	33.0	1.9x	17.4	-	28.3	1.4x
	4	64	4.25	38.1	3.0	7.1	13.5	-	33.1	1.9x	18.0	-	29.5	1.5x
	4	128	4.13	37.1	3.1	7.2	14.7	-	33.1	1.9x	18.6	-	30.3	1.5x
	4	256	4.06	36.6	3.5	7.8	15.2	-	32.9	1.9x	19.0	-	31.0	1.6x
	3	32	3.50	32.1	3.9	8.0	13.3	-	32.8	1.9x	20.0	-	30.8	1.5x
	3	64	3.25	30.1	4.1	8.4	16.3	-	33.3	1.9x	22.4	-	33.8	1.7x
	3	128	3.13	29.1	5.2	10.1	17.7	-	32.7	1.9x	23.9	-	34.5	1.7x
	3	256	3.06	28.6	15.4	26.4	18.6	-	33.6	2.0x	24.9	-	34.8	1.7x

Table 3: Perplexity and decoding speed of LLaMA-3 with learned NF quantization using various quantization configurations. Decoding speedup is measured in tokens per second. The unquantized LLaMA-3 70B model requires multiple GPUs with Tensor Parallelism. Therefore, we report the speed with one GPU, and with Tensor Parallelism applied (labeled as x4 and x2). For the 8B models, since all models fit into one GPU, we report only single GPU results.

Method	Bits	Group	PPL↓		LLM Eval↑					
			WikiText2	C4	PIQA	ARC-e	ARC-c	HellaSwag	Wino	Avg.
Unquantized	16	N/A	6.1	9.2	79.9	80.1	50.4	60.2	72.8	68.6
RTN	4	128	8.5	13.4	76.6	70.1	45.0	56.8	71.0	63.9
	3	128	27.9	1.1e2	62.3	32.1	22.5	29.1	54.7	40.2
GPTQ	4	128	6.5	10.4	78.4	78.8	47.7	59.0	72.6	67.3
	3	128	8.2	13.7	74.9	70.5	37.7	54.3	71.1	61.7
AWQ	4	128	6.6	9.4	79.1	79.7	49.3	59.1	74.0	68.2
	3	128	8.2	11.6	77.7	74.0	43.2	55.1	72.1	64.4
OmniQuant	4	128	6.6	10.1	79.1	80.0	49.7	59.4	73.2	68.3
	3	128	8.4	13.5	76.4	70.0	40.9	55.1	69.5	62.4
NormalFloat	4	128	6.6	9.5	78.6	79.6	49.6	59.0	73.5	68.0
	3	128	9.2	13.0	75.4	72.0	40.5	54.4	69.4	62.3
NormalFloat	4	128	6.2	9.5	79.0	79.6	49.0	59.4	72.6	67.9
learned σ	3	128	7.5	11.7	77.1	74.1	41.7	55.8	69.7	63.7
NormalFloat	4	128	6.5	9.3	79.6	78.0	48.5	59.0	73.8	67.8
+ AWQ	3	128	8.0	11.5	77.0	75.5	44.6	55.9	72.3	65.1

Table 4: Detailed evaluation of post-training quantization on LLaMA3-8B. The RTN, GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2023) results are from Huang et al. (2024); the rest are from our implementations. All non-NormalFloat methods use uniform weight quantization.

Method	Bits	Group	PPL↓		LLM Eval↑					
			WikiText2	C4	PIQA	ARC-e	ARC-c	HellaSwag	Wino	Avg.
Unquantized	16	N/A	2.9	6.9	82.4	86.9	60.3	66.4	80.6	75.3
RTN	4	128	3.6	8.9	82.3	85.2	58.4	65.6	79.8	74.3
	3	128	11.8	22.0	64.2	48.9	25.1	41.1	60.5	48.0
GPTQ	4	128	3.3	6.9	82.9	86.3	58.4	66.1	80.7	74.9
	3	128	5.2	10.5	80.6	79.6	52.1	63.5	77.1	70.6
AWQ	4	128	3.3	7.0	82.7	86.3	59.0	65.7	80.9	74.9
	3	128	4.8	8.0	81.4	84.7	58.0	63.5	78.6	73.2
OmniQuant	4	128	3.3	7.5	82.0	85.6	58.0	66.0	79.6	74.2
	3	128	5.4	9.3	80.8	80.6	50.9	63.7	75.2	70.2
NormalFloat	4	128	3.4	7.6	82.0	85.6	56.7	66.1	79.5	74.0
	3	128	8.7	16.7	76.6	76.9	42.7	55.8	69.3	64.3
NormalFloat	4	128	3.1	7.2	82.3	85.7	58.2	66.4	79.6	74.4
learned σ	3	128	5.2	10.1	77.3	76.7	44.3	62.4	71.2	66.4
NormalFloat	4	128	3.2	6.9	82.6	86.8	60.1	65.9	80.5	75.2
+ AWQ	3	128	4.6	7.8	81.4	85.3	58.5	64.6	79.2	73.8

Table 5: Detailed evaluation of post-training quantization on LLaMA3-70B. The RTN, GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2023) results are from Huang et al. (2024); the rest are from our implementations. All non-NormalFloat methods use uniform weight quantization.