MatVANS: Fast Quantized Matrix-Vector Multiplication via Entropy Coding on the GPU

Alexander Ludwig and Robert Bamler
University of Tübingen
Department of Computer Science
Maria-von-Linden-Straße 6
72076 Tübingen, Germany
al.ludwig@student.uni-tuebingen.de, robert.bamler@uni-tuebingen.de

Abstract

This repository,[1] explores various approaches to GPU-kernels for low-precision matrix-vector multiplications, where the matrix is held in GPU memory in compressed form to reduce memory bandwidth and thus potentially speed up the (heavily memory-bound) operation. Here, "compressed" does not just mean that the matrix is quantized to low bit-widths; it means that the matrix is *entropy coded*, i.e., memory bandwidth is further reduced by exploiting the statistical distribution of matrix elements (think "`gzip` for matrices" but with better compression ratio and faster decoding). Our motivation for this exploration is to speed up inference in large language models (LLMs) on edge devices, where integer matrix-vector multiplication is often the computational bottleneck.

1 Introduction

such as 4 or 8 bits per element, but that the quantized representation is further compressed using entropy coding, specifically asymmetric numeral systems (ANS) coding **?**Bamler [2022]. requirements during inference of large language models (LLMs).

Matrix-vector multiplication is a key operation and often the computational bottleneck in modern AI workloads due to the autoregressive nature of inference in current large language models (LLMs). Different to matrix-matrix multiplication, the speed of matrix-vector multiplication is typically memory-bound because each matrix element is only involved in a single multiply-and-accumulate (MAC) operation, whose cost is negligible compared to the cost for loading the matrix elements from memory. To reduce both runtime and model size, most LLM inference on edge devices nowadays uses quantized matrix representations, typically integer matrices with 4 or 8-bit precision per matrix element. However, quantization reduces memory bandwidth and computation by a similar factor, so quantized matrix-vector multiplication typically remains memory-bound.

To further reduce memory bandwidth, we explore, in this repository, to store the quantized matrix in compressed form using entropy coding [Shannon, 1948, MacKay and Mac Kay, 2003]. Entropy coding uses a statistical model p of some data (here: a quantized weight matrix W) to compress it losslessly to a bitstream whose length is given by the information content, $-\log_2 p(W)$. Our work is inspired by recent advances in fast entropy coding methods, specifically asymmetric numeral systems (ANS) [Duda et al., 2015, Bamler, 2022], which achieves close to optimal compression performance, and for which a fast decoder can be implemented for arbitrary (fixed) models using only table lookups and integer addition and multiplication.

This repository explores several variants of compressed matrix-vector multiplication, where difference lies mainly in the memory access patterns. All explored methods have in common that we assume

[1] This document is part of the documentation for: `https://github.com/wildug/MatVANS`

a set of (fixed) matrices is compressed (i.e., entropy coded) ahead of time to a bitstream. The compressed bitstream is uploaded to GPU memory, and during inference, the GPU decompresses the matrix on-the-fly while performing the matrix-vector multiplication. The decompressed matrix is never materialized in GPU memory, thus reducing memory bandwidth on the GPU.

The main challenges addressed in the various approaches below arise from irregular memory access patterns during demuxing of the compressed bitstream on the GPU: the GPU runs thousands of decompressor-threads in parallel, which each read compressed data from separate bit streams to avoid dependencies between threads. In order to coalesce global memory accesses, these bitstreams need to be multiplexed into a single bitstream in such a way such that threads in the same warp read memory at the same time and from nearby locations. However, since the amount of data that each thread reads per decoded matrix element varies as it depends on the information content of the decoded data itself, which is not under our control. Thus, without some form of coordination between threads, memory accesses would become uncoalesced and thus slow. The various approaches below explore different strategies to achieve coalesced memory accesses despite the irregularity of the decoding process.

2 Experimental Setup

To measure and compare runtimes, we use the following experimental setup, which mimics typical workloads that arise during LLM inference while giving us easy control over matrix dimensions and data distributions.

- We generate a sequence of 10 matrices $W_i \in \mathbb{Z}^{d \times d}$, $i \in \{1, \ldots, 10\}$ of size $d \times d$ whose elements are integers obtained by drawing independent samples from a normal distribution $\mathcal{N}(0, \sigma^2)$ and rounding to the nearest integer. Here, d and σ are parameters, where d is typically in the range of a few thousands (e.g., $d = 4096$) and $\sigma \approx 4$.
- We also generate a random input vector $v_0 \in \mathbb{Z}^d$ with elements drawn independently in the same way.
- Our benchmark task is to compute the sequence of matrix-vector products $v_i = \lceil \alpha_i W_i v_{i-1} \rfloor$ for $i \in \{1, \ldots, 10\}$, where $\alpha_i \in \mathbb{R}_{>0}$ is calculated (ahead of time) for each $i \in \{1, \ldots, 10\}$ such that all elements of v_i fit into a signed 8 bit integer, and $\lceil \cdot \rfloor$ denotes rounding to the nearest integer (resolving ties to even). This mimics a typical requantization step during inference in quantized neural networks.
- To evaluate performance, we upload the compressed matrices to GPU memory and then measure the time it takes to compute this sequence of matrix-vector products on the GPU, not including the time to upload the matrices onto the GPU, but including the time to download the result $v_{10} \in \mathbb{Z}^d$ from the GPU. We verify that the result is correct. (All measurements are performed over a loop, and we ensure that results of early operations are used so that no relevant operations can be optimized out, and that GPU operations are synchronized before and after the timed section to get accurate measurements.)

3 Variations of MatVANS

3.1 MatVANSWarpPerRow

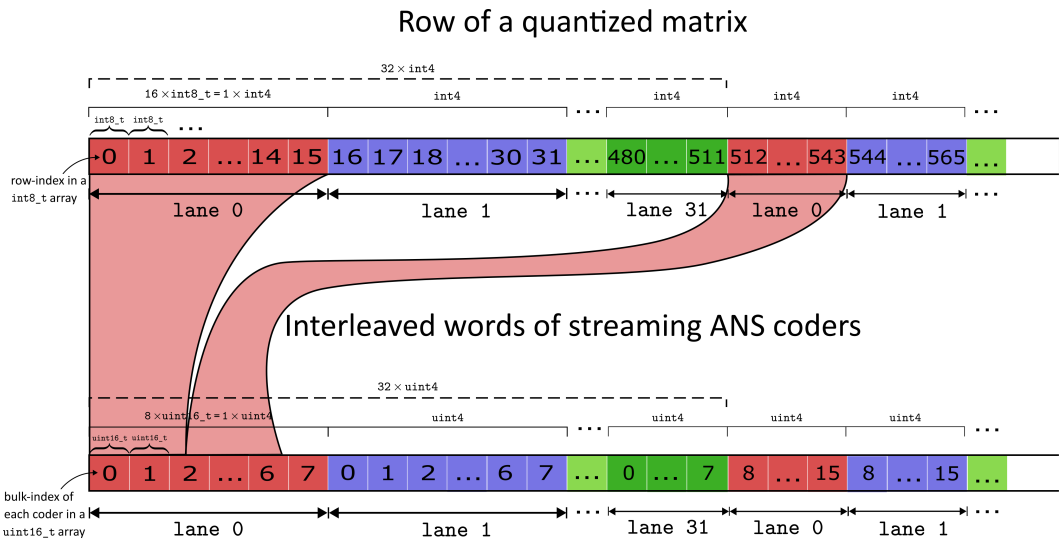

Figure 1: Top: A row of the quantized matrix is chunked to 32 equal-sized chunks highlighted through color coding. The numbers represent the indices of the matrix entries within each row. Each lane maps onto 16 successive `int8_t` elements corresponding to a single `int4`. Bottom: Memory layout of the payload using interleaved words of streaming ANS coders. The numbers represent the bulk-index of each coder in a `uint16_t` array. The color flow visualizes the correspondence of compressed payload and raw matrix entries.

MatVANSWarpPerRow is a variant of MatVANS optimized for performance on GPU hardware. Unlike the naive implementation, each row of the matrix is mapped to a unique warp in the GPU execution process. A warp consists of 32 threads that execute in lockstep, ensuring efficient parallelism. The memory layout of the payload, as illustrated in Figure 1, enables the simultaneous execution of 32 ANS decoders per matrix row.

In the encoding phase, each matrix row is divided into 32 equal-sized chunks, with each chunk being independently compressed by a separate ANS coder. The resulting 32 compressed chunks are then interleaved and stored as uint4 data types, ensuring coalesced global memory access.

The core logic of the matvansWarpPerRow kernel, which utilizes warp-level intrinsics to minimize warp divergence, is presented in Listing 1. This approach improves efficiency by minimizing synchronization overhead and reducing the potential for divergent execution paths within the warp.

3.2 One warp per compressed matrix row (`ans-warp-rows`)

This variant of MatVANS uses $32 \times O$ GPU threads to perform a matrix-vector multiplication Wv with a matrix $W \in \mathbb{Z}^{O \times I}$ with output (row) dimension O.

Prerequisites.

- The input dimension I must be divisible by 128.

Encoding (ahead-of-time) and memory layout.

- Fit a probability mass function p to the matrix elements, modeling them as i.i.d. Thus, p assigns a probability to each integer in the range of possible matrix elements, and these probabilities add to 1. Restrict p such that its values (the probabilities) are integer multiples of $1/128$, i.e., they can be represented in 7-bit fixed-point precision.
- For each matrix row $o \in \{0, \ldots, O-1\}$, create an array C_o of 32-bit words that contains the compressed representation of the row as follows:
  - (Conceptually) divide the row $W_{o,:} \in \mathbb{Z}^I$ into chunks with $32 \times 4 = 128$ entries and iterate over them *in reverse order*. For each chunk:

```
// annotated excerpt from matvansWarpPerRow-kernel.cu
__global__ void decmpressAndMultiply(...){
    ...
    ThreadQueue<16> q; // initialize one ThreadQueue per Thread
    ...
    // Does this specific lane require a refill?
    bool this_lane_needs_refill = q.isEmpty();
    // "__any_sync()" implements a logical OR across the warp
    int warp_needs_refill = __any_sync(0xFFFFFFFF, this_lane_needs_refill);

    // Check if any lane needs a refill
    if (warp_needs_refill){
        // Refill all lanes using coalesced global memory access
        bool s = q.enqueue(payload4[cursor4]);
        // If the queue was refilled, update cursor
        if (s){
            cursor4+=32;
        }
    }
    // Check if head needs refill
    // This branch diverges across the warp.
    if (head < (1<<16)){
        q.dequeue(word);
        // Refill head from ThreadQueue
        head = head<<16 | word;
    }
    ...
}
```

Listing 1: Implementation of the core `MatVANSWarpPerRow` algorithm using coalesced global memory accesses. The warp-level intrinsic `__any_sync(unsigned mask, int predicate)` implements a logical OR across a warp leading to no warp divergence in the branch of line 5. Line 22-26 are adapted from the streaming ANS entropy coder described in Listing 7 of Bamler [2022].

    - Divide the chunk into 32 "quads", i.e., 4-element arrays.
    - Encode the 32 quads in each chunk using 32 separate ANS coders. Each of these ANS coder compresses its quad as a single symbol using the product probability distribution p^4. Thus, the first quad in row o is comprised of the matrix elements $W_{o,0:3} \equiv \big(W_{o,0}, W_{o,1}, W_{o,2}, W_{o,3}\big)$, and is encoded with the entropy model $p(W_{o,0:3}) := p(W_{o,0})\, p(W_{o,1})\, p(W_{o,2})\, p(W_{o,3})$.
    - Reuse the same 32 ANS coders for all chunks in the row, i.e., continue encoding with the internal state of each coder from the previous chunk.
    - Each of the ANS coders uses a 64-bit internal state that corresponds to the `head` in Bamler [2022]. If encoding a quad on any of the coders would cause its internal state to exceed 64 bits, then first append the lower 32 bits of the internal state to *a shared bitstream C_o for the row* and then shift the internal state right by 32 bits. If this happens for multiple coders in the same chunk, then append their lower 32 bits to the shared bitstream in order of *decreasing* coder index.
  - After all chunks have been encoded, flush the internal states of all 32 ANS coders to C_o (again in order of decreasing coder index).
  - Reverse the order of the 32-bit words in C_o, then pad it with zeros (or arbitrary data) to a multiple of 128 words.
- Concatenate all arrays C_o into a single array C of 32-bit words, where the rows are stored in order of increasing row index o. Keep track of the starting index of each row within C in a separate array of offsets.
- The compressed representation of the matrix consists of the array C of 32-bit words, the array of O offsets, and a precalculated lookup table for the percent point function of p (which has 128 entries because we represent probabilities with 7-bit precision).

Decoding (on the GPU). To compute the matrix-vector product Wv for a given input vector $v \in \mathbb{Z}^I$, proceed as follows:

- Launch a GPU kernel with O warps (i.e., $32 \times O$ threads), where each warp is assigned to decode and process a single matrix row $o \in \{0, \ldots, O-1\}$.
- Each thread within a warp performs the inverse operations of one of the ANS coders used for encoding (see above). Thus, each thread keeps track of an individual 64-bit coder state (`head`) from which it repeatedly decodes a quad, scalar-multiplies the decoded quad with the corresponding 4 entries of the input vector v, adds the result of the scalar product to a (thread-local) accumulator, and then throws away the decoded quad.
- Decoding with ANS requires accessing the percent point function (inverse cumulative distribution function) of p. We implement this using a lookup table that is entirely stored in *registers*, i.e., no global or shared memory accesses are required for this. To store an array of 128 bytes in registers, each thread holds a 32-bit word, i.e. 4 bytes. We interpret the concatenation of these 32×4 bytes as an array of 128 bytes. To access it by a 7-bit index, and we use `__shfl_sync` with the 5 most significant bits of the index followed by a bit shift of 8 times the 2 lowest significant bits of the index.
- Whenever a thread's internal state drops below 2^{16}, the thread refills it by reading 32 bits from the shared compressed bitstream C_o for the row. To find the right word to read from C_o, the threads within a warp coordinate using `__ballot_sync` (to figure out which threads within a warp need refilling) and `__popc` (to perform a prefix sum over the threads that need refilling before the own thread). These syncronization operations are fast because they happen on the warp level, i.e., within a single streaming multiprocessor.
- To optimize global memory access when reading from C_o, we implement a warp buffer of size $32 \times 4 = 128$ words: each thread initially reads 4 words (128 bit) from C_o by loading a single `uint4` from global memory. We then interpret these 32 `uint4`s words as a shared queue of 128 words. Whenever one or more threads in the warp need refilling, they read from this queue using `__shfl_sync` to broadcast words from the queue to the threads that need them. If the queue runs empty, all threads in the warp cooperatively read another `uint4` from global memory to refill the queue. Thus, global memory accesses are coalesced and there is never warp divergence during global memory reads.
- After all quads in the row have been processed, the 32 thread-local accumulators are summed up, and one thread in the warp writes the final accumulated result for the row to an output vector in global memory.

3.3 One thread per compressed matrix row, with cooperation across warps (`ans-warp-stripes`)

This variant is similar to `ans-warp-rows` from Section 3.2, but instead of assigning one warp per matrix row, we assign one thread per matrix row. The threads within a warp still cooperate both for the lookup tables and for an input buffer of compressed data to achieve coalesced global memory accesses. This means that the encoder has to process the matrix in "stripes" of 32 rows instead of one row at a time, and then interleave the compressed data from the 32 rows in each stripe analogous to how it is done for the 32 positions within "chunks" in `ans-warp-rows`.

This approach has a slightly lower memory overhead than `ans-warp-rows` because we divide the compressed data into 32 times fewer streams and thus need less padding. However, we end up with 32 times less parallelism during decoding, which empirically makes this method slower than `ans-warp-rows`.

Prerequisites.

- The input dimension I must be divisible by 4 (so we can evenly divide each row into "quads").
- The output dimension O must be divisible by 32 (because stripes of 32 rows share an input buffer during decoding).

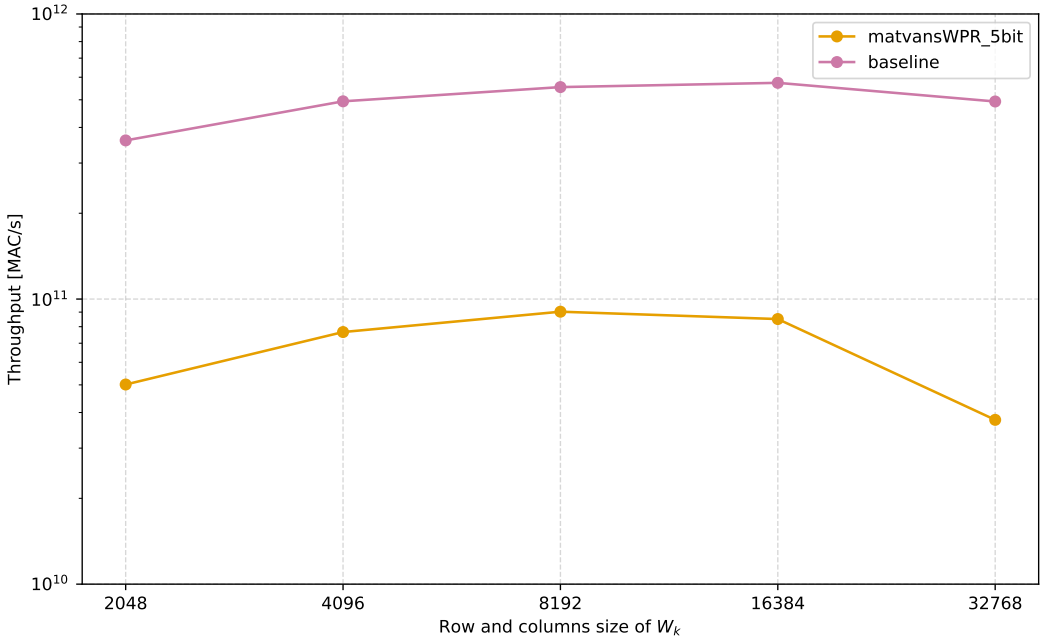

Figure 2: Measured Throughput as a function of Row and Column size of W. Baseline was run on identical matrices which was generated with a entropy of 5 bits.

4 Results

The results in Figure 2 show that `matvansWarpPerRow` is in a similar throughput magnitude compared to the baseline.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF): Tübingen AI Center, FKZ: 01IS18039A. Robert Bamler is a member of the Machine Learning Cluster of Excellence, funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC number 2064/1 – Project number 390727645. The author thanks the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for support.

References

Robert Bamler. Understanding entropy coding with asymmetric numeral systems (ans): a statistician's perspective. 2022. URL https://arxiv.org/abs/2201.01741.

Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

Jarek Duda, Khalid Tahboub, Neeraj J Gadgil, and Edward J Delp. The use of asymmetric numeral systems as an accurate replacement for huffman coding. In *2015 Picture Coding Symposium (PCS)*, pages 65–69. IEEE, 2015.

A Appendix: TODO

TODO