Accelerating matrix-vector products using entropy coding on the GPU

Alexander Ludwig

30.05.2025

Abstract

Performant matrix-vector products are central building blocks for fast large language model (LLM) inference. This report investigates a new implementation of matrix-vector products, using a modern entropy coding algorithm. To tackle memory-bound matrix-vector products, an entropy-coded representation of weights is decompressed on-the-fly using an efficient asymmetrical numerical system decoder (MATVANS). The proposed method is implemented on a CUDA platform by NVIDIA [1] Furthermore, this report documents the technical environment of developing, optimizing and debugging CUDA kernels.

1 Introduction

Sequential matrix-vector products are a core building block in diverse machine learning algorithms such as the conjugate gradient method, online algorithms (Liu, 2024) and recently in large language model (LLM) inference. To accelerate LLM inference, decoder-only transformer models utilize **key-value (KV) caching** (Pope et al., 2022, Kwon et al., 2023). This technique stores previously computed key and value vectors, eliminating the need to recompute them at each step. Additionally, to accelerate inference and reduce memory usage, network weights are **quantized** to a compact number representation (Nagel et al., 2021). Instead of representing weights with 16- or 32-bit precision used in training, inference is performed with fewer precision bits which can be represented by scaling and rounding the original values $w_{\text{int}} = \lfloor \Delta w \rceil$, where Δ is a single-precision float. The **quantization granularity** defines the group of tensors that share the same scaling. Two common types are **per-tensor quantization** and **per-channel quantization**. In per-tensor quantization, the entire tensor shares a Δ_W, while in per-channel quantization each output channel has a Δ_c, leading to better performance in practice (see Section 2.2.3 and 2.4.2 in Nagel et al., 2021). For simplicity

[1] The code is hosted at `https://github.com/wildug/gpu-comp/`.

only per-tensor quantization is taken into account in this report. The result of 8-bit quantized matrix-vector multiplication is stored as 32-bit integers to avoid numerical overflow. The 32-bit results are quantized back to 8-bit integers in a step called **requantization**. The concrete implementation of requantization is explained in section 2. By combining quantization with KV caching, transformer inference is transformed into a highly optimized pipeline centered around repeated matrix-vector multiplications on 8-bit integers. This report focuses on optimizing this specific computation pattern on NVIDIA graphics processing units (GPUs).

1.1 Sequential Matrix-Vector Multiplication

Standard Matrix-vector multiplication is unlike standard matrix-matrix multiplication optimal in terms of asymptotic computational complexity for the general case. However, for structured matrices the asymptotic complexity of matrix-vector products remains an active area of research (Anand et al., 2025). In practice, implementations on real hardware are bound by memory bandwidth and not by the number of logic operations. Most of the time, the threads on a Graphical Processing Unit (GPU) are idle, waiting for the required data to be fetched (see Figure 1). To further increase the arithmetic intensity Hao et al. (2024) propose to compress the exponent component of a floating point number using asymmetric numerical systems (ANS) (Duda, 2014). Their experiments show reduced storage requirements for inference and training. Curiously, observed speedups exceed the theoretical compression ratio, calculated by dividing exponents entropy by bit-size in Figure 1 of Hao et al. (2024). This report builds on the idea of source-coding weight values using ANS, but focuses on inference and 8-bit integer weights. Rather than relying on existing libraries such as **nvCOMP** for decompressing weights, a custom CUDA kernel for combining ANS decoding and matrix-vector products is proposed.

1.2 CUDA Development environment

This section describes the setup used to develop CUDA kernels. Most development took place on a mobile NVIDIA GTX 1050 with 2048 MiB GDDR5 memory and CUDA version 12.4. While the GTX 1050 does not allow the latest hardware and software features, such as tensor cores or full profiling with NSight Compute support, it is perfectly sufficient for basic development

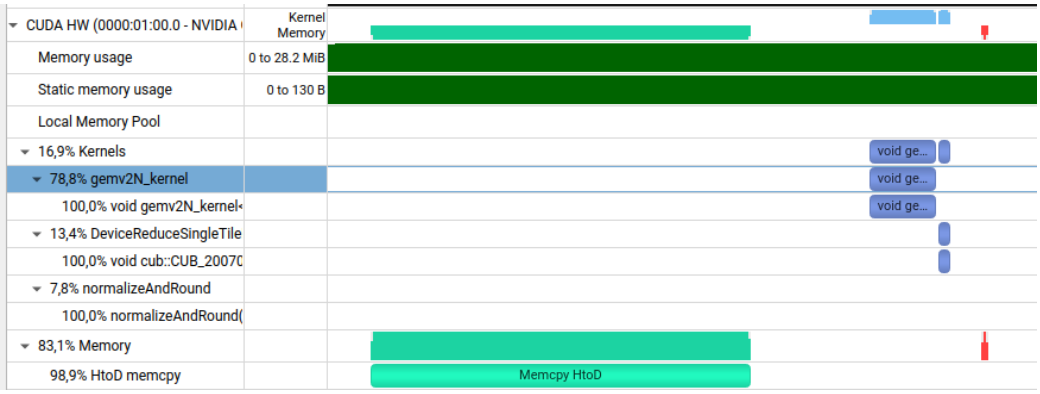


Figure 1: Small cutout of the cuBLAS single-precision floating point baseline visualized in Nsight Systems. A single host-to-device memory-copy of a matrix (turquoise bar, 5.2 ms) takes significantly longer than the cuBLAS gemv kernel (blue bar, 0.66 ms), demonstrating the memory-bottleneck of matrix-vector multiplications.

and kernel debugging. The development environment consisted of Visual Studio Code (VS Code) Version 1.100.2 with the C/C++, Nsight Visual Studio Code Edition and clangd extensions. These extensions provide linting, hover information and debugging capabilities even inside device (GPU) code. CUDA kernels can be compiled manually with `nvcc` or using `cmake`. Additional compiler flags such as `--use_fast_math -O3` did not alter the latency in my experiments. For building instructions see the `README.md`.

NVIDIA provides multiple tools to profile CUDA kernels. One option is to use the command line utility `nvprof`, which already gives basic latency information for invoked kernel and CUDA-api function calls. See appendix B for results on the compression kernel. A graphical profiler as **NVIDIA Nsight Systems** can show the timeline of the overall program including latencies of the individual kernel, as shown in Figure 1. Appendix (C) compares two memory charts from Nsight Systems.

A tool to see more specific information about each kernel is **NVIDIA Nsight Compute**. It provides low-level metrics such as warp occupancy, memory throughput, cache hit rates and instruction-level execution counts. Nsight Compute generates kernel-specific reports that help identify performance bottlenecks and optimization opportunities.

Algorithm 1 Matrix-vector multiplication and ANS decoding (MATVANS)

Input: `head`, `cursor`: 32-bit integers
        `quantile`, `w`, `r`, `min_value`: 8-bit integers
        `payload`, `cdf`, `ppf`: arrays of 8-bit integers
for $j = 0$ to $cols - 1$ **do**
  `quantile` $\leftarrow$ `head` mod 256 {Extract lowest 8 bits of head}
  `r` $\leftarrow$ `ppf`[`quantile`]
  `w` $\leftarrow$ `min_value` + `r`
  `res` $\leftarrow$ `res` + `w` $\cdot$ `vector`$[j]$
  `prob` $\leftarrow$ `cdf`$[r+1]$ − `cdf`$[r]$
  `head` $\leftarrow$ (`head` $\gg 8$) $\cdot$ `prob` + (`quantile` − `cdf`$[r]$) {Bits-back trick}
  if `head` $< 2^{16}$ **then**
    `head` $\leftarrow$ (`head` $\ll 16$) | `payload`[`cursor`]
    `cursor` $\leftarrow$ `cursor` + 1
  end if
end for

1.3 Combining decoding and matrix-multiplication

Algorithm 1 summarizes the proposed **MAT**rix-**V**ector **ANS** kernel (**MATVANS**). `quantile` extracts lowest 8-bits of the ANS-stack and is interpreted as an unnormalized quantile of a probability distribution. Together the point-percentile function array (`ppf`) and cumulative distribution function array (`cdf`) allow decoding the residual value `r` and encode the arbitrarily chosen value inside the $\{0, ..., \texttt{cdf}[r+1] - \texttt{cdf}[r]\}$ interval (bit-back trick). See Bamler (2022) for a detailed explanation of ANS.

1.4 File format specification

We store K compressed matrices $W_0, \ldots, W_{K-1}$, along with a single uncompressed vector v_0. Each matrix W_k, for $k \in \{0, \ldots, K-1\}$, has dimensions $N_{k+1} \times N_k$, while the vector v_0 has dimension N_0. The file starts with an overall container format specifying the initial vector v_0 and the number of matrices and continues with K matrix container formats, describing entropy model and payload for W_k.

The file format specification introduced in `commit 9d5a78b` from the `compressed-nn-ops-demo` repository serves as a basis. An overview

of the adapted format is given in Table 1, for further details see `mock-data-old-adapted.ipynb` or `create_matrix.py`.

Table 1: Overall Container Format

data:	K	$S_{\max}$	N_0	v_0	`pad`	W_0	W_1	$\cdots$	W_{K-1}
type:	$u32$	$u32$	$u32$	$i8[N_0]$	$u8[3-((N_0+3) \bmod 4)]$	see below	see below	$\cdots$	see below

Matrix container format

data:	N_{k+1}	N_k	δ	`cursors`	`payload_size`	$\hat{w}_{\min}$	$\lvert G\rvert$	`cdf`	`pad`	`ppf`	`payload`
type:	$u32$	$u32$	$f32$	$u32[N_{k+1}]$	$u32$	$i8$	$u8$	$u8[\lvert G\rvert+1]$	$u8[(\lvert G\rvert+1) \bmod 2]$	$u8[256]$	$u16$[`payload_size`]

2 Experiments

All experiments in this section were performed on a single Nvidia RTX 2080ti GPU with 11 GB GDDR6 memory. Matrix elements are sampled i.i.d from a gaussian distribution, with column-size dependent variance, $w_{i,j}^k \sim \mathcal{N}(0, 1/\sqrt{N_k})$, to ensure non-exploding activations. The baseline kernels and the proposed compression kernel dynamically quantize the intermediate results (or activations) of the sequential matrix vector products to rescale. This is required since the result of each iteration is stored in a signed 32-bit integer array and needs to be rescaled by $\Delta_k = \max_i \frac{|(W_k x)_i|}{127}$ to rescale in a signed 8-bit array for further processing. Dynamic quantization requires casting the results into intermediate floating point representation. For this task, both programs use the same subroutines (`absMaxWithThrustDevice` and `normalizeAndRoundtoInt`), ensuring a fair comparison by standardizing intermediate steps. Several warp-up runs are performed before measuring the wall-clock time results to account for pipelining effects.

2.1 Establishing a (fair) baseline

To establish a fair baseline, the same weights were written as plain 8-bit integer in a binary file as specified in the file format in appendix (A). First a naive baseline using single-precision floats (32-bits) is established.

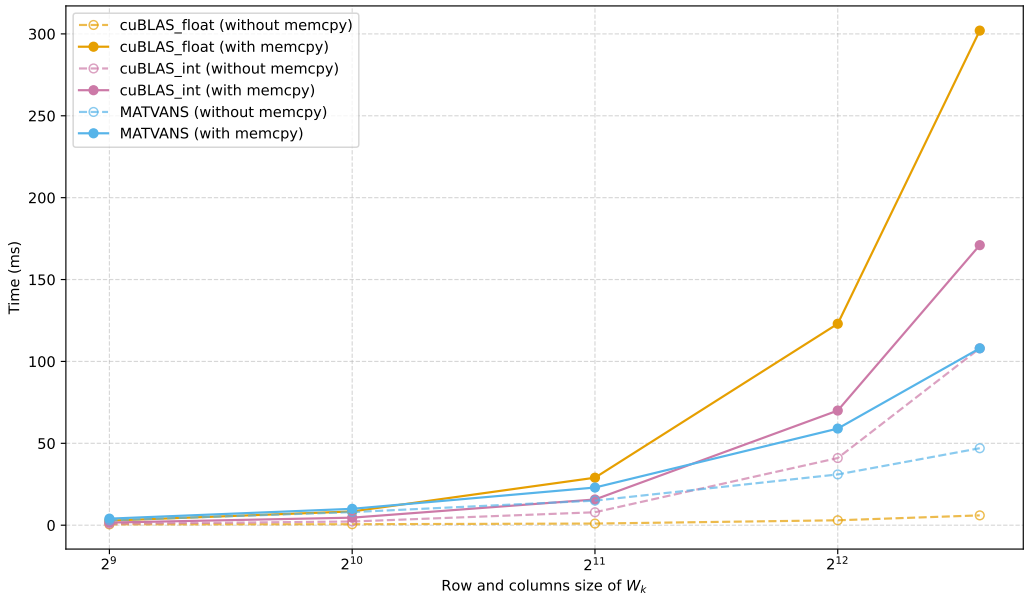


Figure 2: Observed latencies of $K = 20$ equisized matrix-vector products across matrix and initial vector sizes. Dashed lines show latencies without considering the required host-to-device memory copy, while solid lines include copy latencies.

The `cublasSgemv` function from **cuBLAS** implements a general matrix-vector multiplication for single-precision floating point numbers. Running the matrix-vector multiplications with single-precision floating point numbers introduces a significant overhead, since single-precision floats are 4 times larger than 8-bit integers, host-to-device memory-copy latencies also quadrupel (1.3 ms vs 5.2 ms, measured on GTX 1050). On the other hand using an established baseline serves as a double-check for the proposed method. As a further baseline, **cuBLASLt**'s `cublasLtMatmul` general matrix-matrix (GEMM) multiplication algorithm is used, capable of processing 8-bit and accumulating to 32-bit integers. This implementation has no artificial overhead regarding memory transfer, but does not use a dedicated matrix-vector kernel.

2.2 Implemention and evaluation of MATVANS kernel

In accordance with the specified file-format in subsection 1.4, each thread decodes and multiplies a single matrix-row. Each element is decoded and multiplied, individually. The arrays containing the cdf, ppf (inverse of the cdf) and vector elements are copied to shared-memory. Before measuring the latency, several warm-up runs were performed. Figure 2 shows the measured latencies and compares the implemented MATVANS kernel (blue lines) with the cuBLAS baselines. Curiously, when not considering host-to-device transfer (dashed lines), the single-precision floating point function `cublasSgemv` from cuBLAS is the most performant. This hints towards a suboptimal implementation of the cuBLASLt integer baseline and the MATVANS kernel. Appendix C shows the memory chart of the MATVANS kernel and the cuBLAS_float baseline with more uncoalesced memory accesses in the first compared to the former leading to higher warp stalling values. Since the MATVANS kernel in the implemented form only decodes a single element at a time and does not use fused multiply-add operations, there is still room left for further optimizations. Considering the latencies including memory transfer, the proposed method outperforms the alternatives for larger matrix sizes, demonstrating that entropy coding data accelerates matrix-vector multiplications and effectively tackles the latency critical memory-bottleneck.

3 Conclusion and Outlook

This report shows that MATVANS accelerates matrix-vector products on the GPU in a setting of sparse GPU memory with many host-to-device copies. Furthermore, the proposed method comes with reduced storage demands for the compressed weights. Parallel decoding of multiple weights and engineering the file format towards fewer uncoalesced memory accesses might close the gap in pure compute latency. Further research might investigate integrating ANS-decoding using CUTLASS abstractions, to leverage existing high-performance for the GEMM environment. The recently released python bindings for the CUTLASS domain-specific language (DSL) might make prototyping faster compared to the existing C++ library. A further alternative is to implement the proposed idea in Triton, another python based DSL published by OpenAI. Both CUTLASS and Triton received rapid development in the past few months demonstrating the momentum and excitement the

field of hardware-aware machine learning is experiencing.

References

Emile Anand, Jan van den Brand, and Rose McCarty. The structural complexity of matrix-vector multiplication, 2025. URL https://arxiv.org/abs/2502.21240.

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

Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding, 2014. URL https://arxiv.org/abs/1311.2540.

Yongchang Hao, Yanshuai Cao, and Lili Mou. Neuzip: Memory-efficient training and inference with dynamic compression of neural networks, 2024. URL https://arxiv.org/abs/2410.20650.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Yang P. Liu. On approximate fully-dynamic matching and online matrix-vector multiplication, 2024. URL https://arxiv.org/abs/2403.02582.

Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021. URL https://arxiv.org/abs/2106.08295.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022. URL https://arxiv.org/abs/2211.05102.

Appendix

(A) File format Specification for baseline

Uses the same overall container format but stores matrix elements in column major ordering.

Table 2: Overall Container Format for baselines

data:	K	N_0	v_0	`pad`	W_0	W_1	$\cdots$	W_{K-1}
type:	$u32$	$u32$	$i8[N_0]$	$u8[3-((N_0+3) \bmod 4)]$	see below	see below	$\cdots$	see below

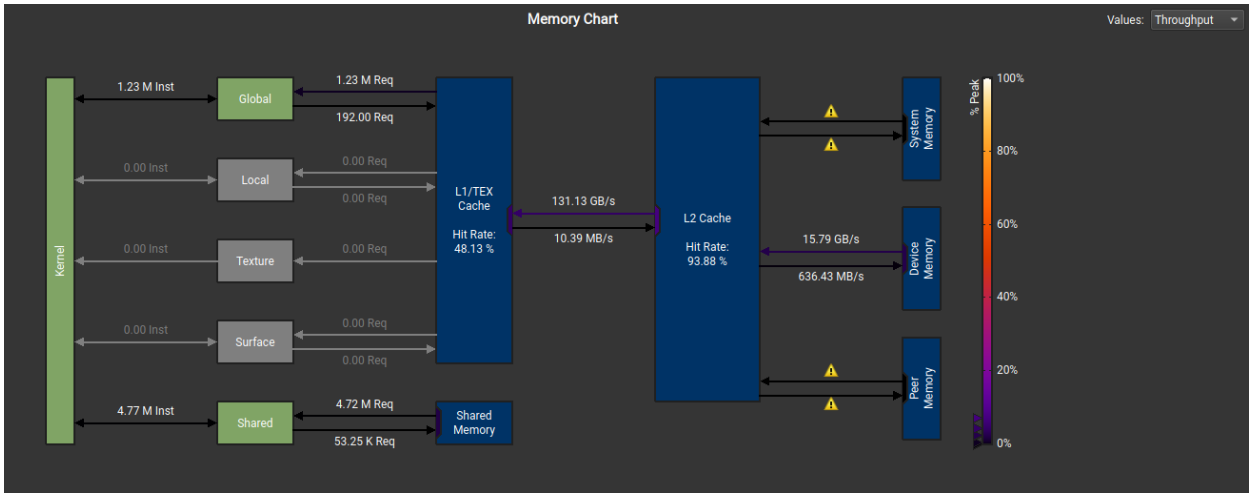
Matrix container format

data:	N_{k+1}	N_k	δ	`payload`	`padding`
type:	$u32$	$u32$	$f32$	$i8[N_{k+1}N_k]$	$i8[N_{k+1}N_k mod 2]$

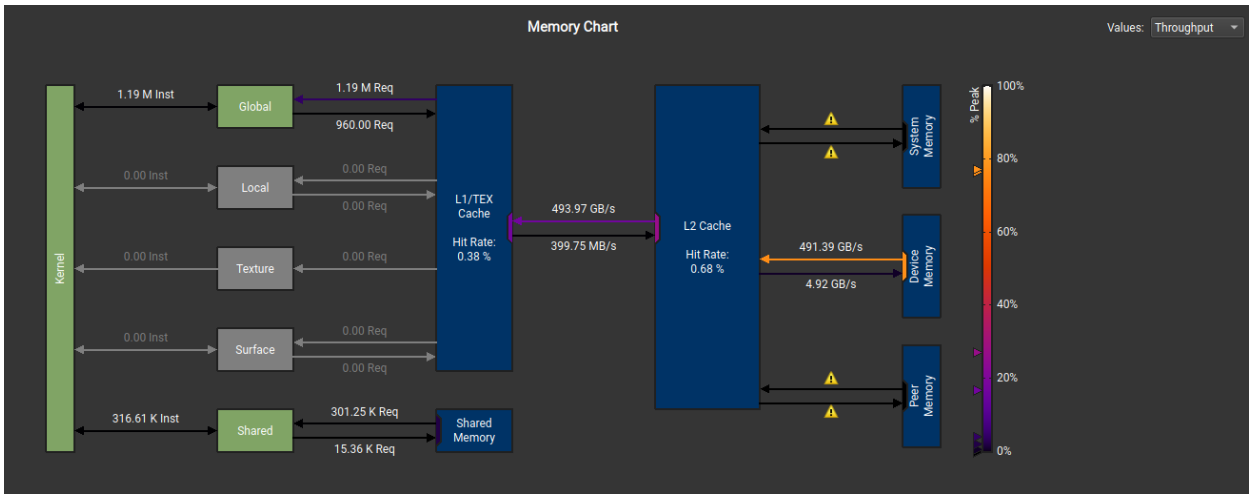
(B) output of `nvprof` for the proposed kernel

```
... program output ...
==20668== Profiling application: ./read-mat
==20668== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   86.37%  4.50832s       400  11.271ms  11.042ms  14.541ms  decmpressAndMultiply(...)
                   10.18%  531.46ms      1601  331.95us     191ns  1.8883ms  [CUDA memcpy HtoD]
...
```

(C) Memory charts from Nsight Systems of the MATVANS and cuBLAS_float kernel



Memory chart of `decmpressAndMultiply` kernel in MATVANS.



Memory chart of `gemv2N_kernel` in cuBLAS_float. Higher throughput for the `gemv2N_kernel` between device memory and L2 cache as well between L2 and L1 due to higher coalesced memory-accesses of floating point numbers in cuBLAS_float. Furthermore the `derived__memory_l2_theoretical_sectors_global_excessive` counter, shows the number of excessive sector accesses and is far higher for the MATVANS kernel with $1.8\,10^8$ compared to 7344 in the cuBLAS kernel.