



Elvir Crncevic, Bsc.

An Efficient Inference Kernel for SpQR

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Telematik

submitted to

Graz University of Technology

Supervisor

Steinberger, Markus, Univ.-Prof. Dipl.-Ing. Dr.techn. BSc

Institute of Computer Graphics and Vision

Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Thomas Pock

Graz, January 2025

This document is set in Palatino, compiled with [pdfL^AT_EX2e](#) and [Biber](#).

The L^AT_EX template from Karl Voit is based on [KOMA script](#) and can be found online: <https://github.com/novoid/LaTeX-KOMA-template>

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

Recently, efficient neural network compression techniques became a focal point of the research community following the increase in neural network parameter counts, leaving a massive gap between large models and consumer GPU hardware. The widening model size and quality gap emphasize the need for solutions that retain model accuracy while significantly reducing the memory footprint of the model. Although recent advances in neural network compression and quantization did manage to bridge this gap, their deployment on memory-limited devices offers new challenges.

This work tackles these challenges by developing an optimized inference kernel for models compressed using Sparse-Quantized Representation (SpQR). SpQR works by compressing the weights down to 3-4 bits while preserving the weights which are a source of particularly large quantization errors, storing them in full precision. Our novel multi-batch inference kernel is able to achieve a peak 4.18X speed-up compared to cutlass in the low-batch matrix-vector multiplication setting, along with end-to-end performance improvements, yielding 1.2X faster end-to-end inference. Furthermore, we demonstrate the memory savings by running the SpQR-quantized Llama2-7B model on memory-constrained consumer hardware. Finally, we integrate this work into the popular Hugging Face's transformers framework.

Contents

Abstract	v
1 Introduction	1
1.1 Transformers and Llama 2	1
1.2 Embeddings	2
1.3 Masked Self-Attention	3
1.4 Fully-Connected Layers	4
1.5 Auto-Regressive Inference	4
1.6 KV Caching	5
1.7 Inference Bottlenecks and GPU Memory	6
1.8 Neural Network Compression	8
1.8.1 Quantization	8
1.8.2 Sparsification	9
1.9 Post-Training Quantization (PTQ)	9
2 SpQR Algorithm Description	11
3 SpQR Inference Kernel	15
3.1 Storage Format	15
3.2 Dequantization	16
3.3 Reduction Strategy	17
3.4 Sparse Matrix	18
3.4.1 Row Density Analysis	18
3.4.2 Compressed Sparse Row (CSR)	18
3.4.3 Padded/Transposed CSR Storage Format (PTCSR)	19
3.4.4 Hybrid Approach	20
4 Results	23
4.1 Methodology	23

Contents

4.2	Kernel Benchmark Results	25
4.3	End-to-End Results	25
5	Conclusion	27
	Bibliography	29

List of Figures

3.1	An example storage scheme of a 64×64 matrix W , decomposed into matrices D and S . The quantized part of W , a dense matrix D , consists of $64^2 \cdot 16$ 64-bit values with each value having 2 unused bits, a 3-bit first-order scale S_i and zero-point Z_i , 16 3-bit weights, and an 8-bit piece of the second-order group statistics with each chunk being repeated twice for optimized reduction. The non-zero sparse component S is either stored in the CSR or PTCSR format.	16
3.2	Benchmark matrix-vector multiplication runs across all matrices in layer 10. Evidently, using shared memory is superior to relying on the intrinsic for conversion between integers and FP16 values.	17
3.3	Each plot shows the per-layer row density statistics.	18
3.4	Visualization of different formats corresponding to a sparse matrix stored in dense form (left). In the middle, the CSR format (middle) stores the column indices and the values in a row-major memory pattern. The padded block-column-major CSR format (right) stores the values and column indices in a way that makes it so that all threads will have a coalesced memory access pattern, with the added memory cost of zero-padding padding.	21
4.1	Results of the benchmark run for all tensors in the Llama-2-7B model on an A100 GPU. The geomean speed-up across the entire dataset is 1.8X.	23

List of Figures

- 4.2 This figure shows the comparison between the runtime of the two sparse storage formats considered for this kernel as measured on an RTX 4060. Clearly, PTCSR outperforms CSR on Up, Down, and Gate matrices in most layers. When it comes to matrices K, Q, V, and O, mixed relatively insignificant runtimes between CSR and PTCSR matrices are observed. . . 24
- 4.3 Benchmark results for end-to-end runs (in seconds) across various GPUs and underlying sparse storage configurations. . 25

1 Introduction

1.1 Transformers and Llama 2

Shortly after their conception, transformers [Vas+23] became the focal point of deep learning research, with far-reaching applications across various aspects of human endeavor, particularly in the field of natural language processing and computer vision. The etymology of the word transformer stems from their property of transforming a set of input vectors into a set of new vectors more suitable for solving a specific downstream task.

Llama 2 [Tou+23] is part of a family of pre-trained and fine-tuned, decoder-only, transformer-based [Vas+23] large language models (LLMs). These models demonstrate the ability to perform novel tasks from just a few examples when scaled to a sufficient size [Bro+20]. However, even the smallest model in this family (Llama2-7B) requires over 20 GB of memory, making efficient inference prohibitively expensive for most mobile GPUs.

Therefore, exploring efficient compression schemes could unlock new use cases for these models and have a cost-saving factor during model-serving on server-grade GPUs. Naturally, model quantization and sparsification reduce the overall memory use. Furthermore, another motivating factor is that the decoding step, a key step during decoder-only autoregressive inference, is a memory-bound operation [Yua+24a]. Therefore, an efficient implementation of a quantization scheme is expected to decrease the latency of the autoregressive decode step.

With this in mind, an efficient CUDA matrix-vector multiplication kernel for the SpQR [Det+23] compression algorithm in the context of single-batch autoregressive inference is implemented, which claims an achievement

of less than 1% loss of accuracy while compressing LLMs across various model scales. As part of this effort, we develop a storage method for SpQR-compressed matrices and offer a fused kernel that jointly processes sparsified and quantized segments of the original matrix.

These next few sections will closely follow the work of [PH22], which offers a formal description of the transformer architecture with some modifications introduced by [BB23]. As this work focuses on efficient inference, we will omit algorithmic details unrelated to the Llama 2 [Tou+23] family of language models in the inference regime. Specifically, we focus on textual sequence modeling with decoder-only architectures in mind and some architectural advances found in Llama 2. Furthermore, we will assume that the bias terms are zero in all architecture blocks.

1.2 Embeddings

The aim of the embedding block is to map tokens to vectors which will be fed downstream to the model. Specifically, in the context of Llama 2, given a vocabulary element $v \in V = \{1, 2, \dots, N_V\}$, as part of an input sequence, we produce a token embedding in \mathbb{R}^D along with rotary embedding in \mathbb{R}^D introduced by [Su+24].

A token embedding learns to represent vocabulary elements $V = \{1, 2, \dots, N_V\}$ in \mathbb{R}^D . On the other hand, position embeddings encode relative or absolute positional information, augmenting the model with the ability to encode contextual information. There are many approaches to computing position embeddings, but most map a t -th token in an sequence $\mathbf{x} = \{x_1, \dots, x_{N_V}\}$ into \mathbb{R}^D . Then, \mathbf{x}_p is a positional encoding of x_t , encoded via rotary embeddings [Su+24] for Llama 2, alongside x_e as the token embedding. In order to obtain the final embedding \mathbf{e} , the token embedding is element-wise added to the position embedding, $\mathbf{e} = \mathbf{x}_p + \mathbf{x}_t \in \mathbb{R}^D$.

1.3 Masked Self-Attention

The multi-head attention block lies at the core of the transformer architecture, dating back to at least 2014 as part of RNN development [BCB14] and more prominently as part of [Vas+23]. Given $1 \leq N \leq N_{\max}$ (bounded by 4096 in the context of Llama 2) input tokens, with each token being embedded via the procedure described in the previous section, an input matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$ and a mask \mathbf{M} , the single head attention with index h within a multi-headed attention block consisting of H attention heads (for Llama 2 7B, $H = 32$ holds) computes:

$$\begin{aligned}\mathbf{Q} &= \mathbf{XW}_Q^h \\ \mathbf{K} &= \mathbf{XW}_K^h \\ \mathbf{V} &= \mathbf{XW}_V^h \\ \mathbf{S} &= \mathbf{QK}^T \\ \forall t_z, t_x \text{ if } \neg \mathbf{M}[t_z, t_x] \text{ then } \mathbf{S}[t_z, t_x] &= -\infty \\ \mathbf{Y}^h &= \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{X}) = \text{softmax}\left(\frac{\mathbf{S}}{\sqrt{d_{\text{attn}}}}\right) \mathbf{V} \in \mathbb{R}^{N \times D}\end{aligned}$$

with the usual definition of softmax given as $\text{softmax}(A)[t_z, t_x] := \frac{\exp A[t_z, t_x]}{\sum_t \exp A[t, t_x]}$. Finally, the horizontally-concatinated output of all attention heads is multiplied with another matrix at layer l :

$$\mathbf{Y}^l = \text{Concat}\left(\mathbf{Y}^1, \dots, \mathbf{Y}^H\right) \mathbf{W}_O$$

We note that unidirectional mask employed in the previous equation is a feature of a family transformer models taking into account only the previous tokens including itself, of which Llama 2 is a part of. For Llama 2 7B, $\mathbf{W}_Q^h, \mathbf{W}_K^h, \mathbf{W}_V^h \in \mathbb{R}^{4096 \times 128}$ alongside $\mathbf{W}_O \in \mathbb{R}^{4096 \times 4096}$ holds. Additionally, Llama 2 introduced Grouped-Query Attention (GQA) for the 34B and 70B variants of the model. Due to resource constraints required to run and train these models, we only focus our attention to the 7B variant. Thus, the formal description of GQA is omitted from this work.

1.4 Fully-Connected Layers

MLPs (short for multi-layer perceptrons), are also an architectural component of Llama 2. They are computed downstream away from the attention layers. In Llama 2-specific terminology, they consist of $\mathbf{W}_{\text{down}}^l \in \mathbb{R}^{d_e \times d_{\text{mlp}}}$ and $\mathbf{W}_{\text{up}}^l \in \mathbb{R}^{d_{\text{mlp}} \times d_e}$ projection matrices, with SwiGLU being chosen as the activation function:

$$\text{Output}^l = \mathbf{W}_{\text{down}} (\sigma(\mathbf{W}_{\text{gate}}^l \mathbf{x}) \odot (\mathbf{W}_{\text{up}}^l \mathbf{Y}^l)).$$

Matrix $\mathbf{W}_{\text{gate}}^l \in \mathbb{R}^{d_{\text{mlp}} \times d_e}$ consists of learnable parameters which are considered a part of the activation function. The operator \odot is an element-wise multiplication and σ is the sigmoid function $\sigma(x) = \frac{1}{1+\exp^{-x}}$. In the context of Llama 2 7B, $d_{\text{mlp}} = 11008$ holds.

1.5 Auto-Regressive Inference

Having summarized all the architectural components relevant in the context of Llama 2 quantization, the reader's attention is directed towards defining the inference scheme for the Llama 2 model. Given some vocabulary V , trained parameters $\hat{\theta}$, a vector $\mathbf{x} \in \mathbb{N}^l$ consisting of vocabulary elements corresponding to the input sequence, Llama 2 will produce a distribution over the entire token set. Uniformly sampling from this distribution will produce an inferred token. Initially, one may pass the entire prompt into the transformer, producing a single output token. This output token is then concatenated with original input sequence and the process repeats itself for a total of l_{gen} times - the total length of the generated sequence. The pseudocode of the procedure explained in this section (1) is taken from [PH22] and will be augmented with KV caching in the next section.

Algorithm 1 $y \leftarrow \text{DInference}(x, \hat{\theta})$

- **Input:** Trained transformer parameters $\hat{\theta}$.
- **Input:** $x \in V^*$, a prompt.
- **Output:** $y \in V^*$, the transformer's continuation of the prompt.
- **Hyperparameters:** $\ell_{\text{gen}} \in \mathbb{N}, \tau \in (0, \infty)$

```

1:  $\ell \leftarrow \text{length}(x)$ 
2: for  $i = 1, 2, \dots, \ell_{\text{gen}}$  do
3:    $P \leftarrow \text{DTransformer}(x \mid \hat{\theta})$ 
4:    $p \leftarrow P[:, \ell + i - 1]$ 
5:   sample a token  $y$  from  $q \propto p^{1/\tau}$ 
6:    $x \leftarrow [x, y]$ 
7: end for
8: return  $y = x[\ell + 1 : \ell + \ell_{\text{gen}}]$ 

```

1.6 KV Caching

During auto-regressive inference, given the initial prompt, let $\mathbf{X}_{\text{cached}}$ be the input matrix passed into an Attention block at layer l . During the next step, let x be a newly generated vector. In the current framework, this vector would be appended to $\mathbf{X}_{\text{cached}}$ and the calculation would flow through all the attention blocks with the updated matrix $\text{concat}(\mathbf{X}_{\text{cached}}, x)$, a row-wise concatenation. If we were to cache the K and V matrices from the previous inference step, we would only need to compute:

$$\begin{aligned}
 k_{\text{new}} &= x^T \mathbf{W}_K^h, \\
 v_{\text{new}} &= x^T \mathbf{W}_V^h
 \end{aligned}$$

which consists of matrix-vector multiplication operations. These vectors then get concatenated into \mathbf{K} and \mathbf{V} matrices:

$$\begin{aligned}
 \mathbf{K}_{\text{cache}} &= \text{concat}(\mathbf{K}_{\text{cache}}, k_{\text{new}}), \\
 \mathbf{V}_{\text{cache}} &= \text{concat}(\mathbf{V}_{\text{cache}}, v_{\text{new}})
 \end{aligned}$$

1 Introduction

As the nomenclature would imply, within the framework of KV caching, the \mathbf{Q} matrix calculation only involves the current token. The modified inference algorithm with KV caching in mind is shown in 2.

Algorithm 2 $y \leftarrow \text{DInferenceKV}(x, \hat{\theta})$

- **Input:** Trained transformer parameters $\hat{\theta}$.
- **Input:** $x \in V^*$, a prompt.
- **Output:** $y \in V^*$, the transformer's continuation of the prompt.
- **Hyperparameters:** $\ell_{\text{gen}} \in \mathbb{N}, \tau \in (0, \infty), \text{cache} \leftarrow \{\}$

```
1:  $\ell \leftarrow \text{length}(x)$ 
2:  $P, \text{cache} \leftarrow \text{DTransformerKV}(x \mid \hat{\theta})$ 
3:  $p \leftarrow P[:, \ell]$ 
4: sample a token  $y$  from  $q \propto p^{1/\tau}$ 
5:  $x_{\text{result}} \leftarrow y$ 
6: for  $i = 1, 2, \dots, \ell_{\text{gen}}$  do
7:    $p, \text{updated\_cache} \leftarrow \text{DTransformerKV}(y \mid \hat{\theta}, \text{cache})$ 
8:    $\text{cache} \leftarrow \text{updated\_cache}$ 
9:   sample a token  $y$  from  $q \propto p^{1/\tau}$ 
10:   $x_{\text{result}} \leftarrow [x_{\text{result}}, y]$ 
11: end for
12: return  $y = x[\ell + 1 : \ell + \ell_{\text{gen}}]$ 
```

Line 2 of algorithm 2 is referred to as the cache pre-fill step and cache is a set containing the cached matrices \mathbf{K} and \mathbf{V} for all layers in DTransformerKV which gets updated at every inference step. The main body of the for loop is referred to as the decode step.

1.7 Inference Bottlenecks and GPU Memory

This section will offer some insights into the realized performance on the GPU for LLM inference. The reported findings mostly stem from the work done by [Yua+24b], which contains a performance analysis of Llama 2 7B

inference using FP16 precision. Performance of various operations noted in the previous sections are listed in 1.1 with the performance being tested on the A6000 GPU. We will note two important findings implied by the table. First, the important thing to highlight is that the listed operations have a low arithmetic intensity. This means that the majority of the operations involved in the decode step are memory-bound. Second, the decode step is clearly bottle-necked by matrix-vector multiplication, as this operation dominates the total execution time (implied by the Maximum Performance column).

Operation	# OPs	Memory Access (bytes)	Arithmetic Intensity (OPs / byte)	Max Performance
$x^T \mathbf{W}_Q^h$	34M	34M	1	768G
$x^T \mathbf{W}_K^h$	34M	34M	1	768G
$x^T \mathbf{W}_V^h$	34M	34M	1	768G
$x^T \mathbf{W}_O^h$	34M	34M	1	768G
$x^T \mathbf{W}_{\text{Gate}}^h$	90M	90M	1	768G
$x^T \mathbf{W}_{\text{Up}}^h$	90M	90M	1	768G
$x^T \mathbf{W}_{\text{Down}}^h$	90M	90M	1	768G
\mathbf{QK}^T	17M	17M	0.99	762G
\mathbf{SV}	17M	17M	0.99	762G
softmax	328K	262K	1.25	960G
norm	29K	16K	1.75	1T
add	4K	16K	0.25	192G

Table 1.1: Layer Performance and Memory Access Information

Memory-boundedness implies that in order to obtain a speed-up of memory bound kernels, one should resort to reducing the overall movement from the GPU's main memory down to the registers from which actual computation is conducted. The next section will provide general insights into ways of speeding-up the execution of the decode step via model compression techniques.

1.8 Neural Network Compression

A seminal work demonstrating the validity of scaling LLM parameter sizes, [Bro+20] paved the way towards the expansion of neural network parameter counts. Specifically, models like GPT-3 have shown strong performance with 175 billion parameters across a wide range of benchmarks and domains. Clearly, models of that order of magnitude incur a significant cost during training and inference. In order to reduce said costs on server and consumer-grade GPUs, neural network compression (studied since the earliest days of deep learning research [LDS89] [HSW93]) aims to reduce the size of neural network models while preserving and even, in some cases, outperforming the original uncompressed models accuracy [Nik+24]. In light of [Gho+21a] offering an overview of the current state of the art in terms of neural network compression methods for inference (a focal point of this work), the following two sections briefly go over two key techniques underpinning the reduction of model sizes.

1.8.1 Quantization

Quantization, as defined by [Gho+21a], is the process by which one distributes real-valued numbers over a fixed discrete set of numbers. There are at least two key orthogonal concepts that underpin quantization. First, the number of bits required to fully cover the aforementioned discrete set. In most cases, we are interested in minimizing the number of required bits. Second, the accuracy of the associated computations, which we attempt to maximize in most cases. While half-precision floating point arithmetic has been a mainstay both in neural network training and inference [CBD14], recent algorithmic advances [Det+22] [Fra+23] [Egi+24] [Det+23] have made it possible to produce models with as little as on average 3 bits per weight while still having high accuracy alongside a boost in performance and memory savings during inference.

1.8.2 Sparsification

The goal of sparsification is to reduce the overall parameter count of the model by zeroing out as many weights as possible while maximizing the model's accuracy. A classical approach, developed in [LDS89] and [HSW93] and outlined in [Kur+22] aims to prune a network layer $\mathbf{W}^* \in \mathbb{R}^{h \times w}$ of a network trained by minimizing a Taylor-approximated loss function \mathcal{L} :

$$\begin{aligned} \mathcal{L}(\mathbf{W}_M) &\simeq \mathcal{L}(\mathbf{W}^*) + (\mathbf{W}_M - \mathbf{W}^*)^\top \nabla \mathcal{L}(\mathbf{W}^*) + \\ &\quad \frac{1}{2} (\mathbf{W}_M - \mathbf{W}^*)^\top \mathbf{H}_{\mathcal{L}}(\mathbf{W}^*) (\mathbf{W}_M - \mathbf{W}^*) \\ \text{s.t. } \mathbf{W}_M &= (\mathbf{M} \odot \mathbf{W}^*) \end{aligned}$$

given a 0/1 sparsity mask $\mathbf{M} \in \{0, 1\}^{h \times w}$, and $\mathbf{H}_{\mathcal{L}}(\mathbf{W}^*)$ being the Hessian of the loss at \mathbf{W}^* . This original formulation hits a roadblock when faced with modern deep learning models with potentially billions of parameters due to the expensive Hessian computation, inviting development of computationally-feasible variants. Generally, these variants still solve:

$$\arg \min_{\mathbf{W}'(\mathbf{W}), \mathbf{M}} \mathcal{L}(\mathbf{W}'(\mathbf{W}) \odot \mathbf{M}) \quad \text{s.t.} \quad 1 - \sum_{i=1}^h \sum_{j=1}^w M_{i,j} \geq S$$

as noted by [Fra24] where the function \mathbf{W}' aims to find (potentially) a new set of weights while preserving the accuracy of the model given the sparsity budget S .

1.9 Post-Training Quantization (PTQ)

Post-training quantization (PTQ) refers to a family of model quantization methods [Nag+21] which produce a quantized model after the model training procedure is finished. This model family may or may not involve a calibration dataset, influencing the resulting quantization statistics. Like [Det+23], for a full summary of post-training quantization (PTQ) methods, the reader is referred to [Gho+21b] and [Nag+20].

2 SpQR Algorithm Description

In this section, the SpQR algorithm is summarized. A member of the PTQ family of algorithms, SpQR quantizes the weights on a per-layer basis, replacing the costly linear layers of Llama 2 in both the attention heads and fully connected layers. For each weight matrix \mathbf{W} that we ought to quantize, and given some input \mathbf{X} , we wish to find some quantization scheme which, when the weight is dequantized into \mathbf{W}' , the following is minimized:

$$s_{i,j} = \min_{\mathbf{W}'} \|\mathbf{W}\mathbf{X} - \mathbf{W}'\mathbf{X}\|_2^2$$

where we interpret $s_{i,j}$ as the weight quantization sensitivity for the corresponding weight $w_{i,j}$ of \mathbf{W} . $w'_{i,j} = \text{quant}(w_{i,j})$ is loosely defined as the result of the quantization of the corresponding weight $w_{i,j}$. $s_{i,j}$ has a closed-form solution in the continuous setting by following the generalized Optimal Brain Surgeon framework [FSA23]:

$$s_{i,j} = \frac{(w_{i,j} - \text{quant}(w_{i,j}))^2}{[2(\mathbf{X}\mathbf{X}^\top)^{-1}]_{i,j}}.$$

However, computing the denominator would involve computing a numerically unstable matrix inverse. Therefore, we approximate $s_{i,j}$ using the result from [Fra+23], where Cholesky factorization is applied on a damped form of the Hessian after which we iteratively go through the matrix column blocks, initially detecting the outlier elements using the saliency score. Further, the weights are quantized by updating not only the not-yet quantized weights in subsequent columns within one group using the saliency score, but also by conducting a second time for the rest of the columns beyond the current group.

When this quantization scheme is applied, one can observe cases where weights behave similarly in small groups, with abrupt changes between groups. Therefore, SpQR [Det+23] conducts a bi-level group-wise quantization with relatively small group sizes (denoted as β_1 and β_2). To simplify the kernel implementation, we fix $\beta_1 = \beta_2 = 16$. We will explore this in greater detail in one of the subsequent sections of this paper.

Another empirical observation leads one to notice weights that do not fit any pattern and are simply quantization outliers, disproportionately affecting the quantization result. As per [Det+23], 1% of the weights account for over 75% of the total quantization error. The unstructured nature of these weights makes efficient storage as part of the dense matrix difficult, as we will discuss later at greater lengths. Therefore, the dequantized weight matrix resulting from the SpQR algorithm W' can be decomposed into:

$$W' = D' + S',$$

where D' is a dequantized matrix using only the bi-level group-wise dense quantization data and S' is dequantized using data from the sparse outliers.

A factor worth noting is that SpQR quantizes the linear layers of models. This means that for this model to be optimized for inference, one has to solve the problem where one is given an input FP16 vector x of size n , an FP16 matrix W' , computed by way of dequantization of bits representing a matrix stored in the SpQR format of size $m \times n$ (the bits of the quantized matrix are denoted as W). The aim is to compute an FP16 vector y of size m by way of matrix-vector (Mv) multiplication as follows:

$$y = xW'^T.$$

The storage scheme for D' (which we will refer to as the dense matrix going forward) is such that we group the 3-bit weights into tiles of size $\beta_2 \times \beta_1 = 16 \times 16$. Each row r in tile $T_{i,j}$ has a corresponding 3-bit scale $q_{i,j,r,s}$ and zero-point $q_{i,j,r,z}$. Further, these $\beta_2 = 16$ pairs of scales and zero points have their second-order scales on a per-tile basis, $q_{i,j,s,s}$ and $q_{i,j,z,s}$, and zero points, $q_{i,j,s,z}$ and $q_{i,j,z,z}$, stored in FP16. Each FP16 weight $w_{i,j}$ is then

quantized to 3 bits as $W_{I,J}$. We may or may not have an FP16 $S_{I,J}$ non-zero value from the sparse matrix. In summary, the dequantization procedure for a weight $w_{I,J}$ is given as:

$$\begin{aligned} s_{i,j,r} &= q_{s,s} \cdot (q_{i,j,r,s} - q_{s,z}), \\ z_{i,j,r} &= q_{z,s} \cdot (q_{i,j,r,s} - q_{z,z}), \\ w_{I,J} &= s_{i,j,r} \cdot (\mathbf{W}_{I,J} - z_{i,j,r}) + \mathbf{S}_{I,J}. \end{aligned}$$

The next chapter describes the concrete implementation strategy employed to conduct dequantization in conjunction with the matrix-vector product.

3 SpQR Inference Kernel

3.1 Storage Format

The storage scheme ensures consecutive groups of β_2 threads share the same second-order scales and zero points. Since 10 unused bits per 64-bit integer are available, the 64-bit second-order information is split into 8-bit pieces (with each element being repeated twice) and placed in the bit array. After all active threads in a warp complete reading from global memory, bitmasking, shifting, and a logical OR warp reduction are applied to reconstruct the second-order scales and zero points. Since the chunk size is 8 and 16 threads are involved in the warp shuffle, the shuffle operation only needs to be performed three times per thread (as opposed to four times if chunks of size 4 were used).

Two main benefits are derived from this approach compared to having a separate buffer to store this data. First, the total model size is reduced since the second-order data is now stored at no additional memory cost (for Llama2-7b, approximately 200Mb of memory is saved). Second, since the first-order scales and zero-points must be dequantized by each thread, having the second-order scales and zero-points stored in registers after the warp shuffles is convenient, as they are immediately needed for first-order computations. A visual representation of the storage format of a single tile is shown in Figure 3.1. Finally, the rows of these tiles are stored contiguously in memory.

3 SpQR Inference Kernel

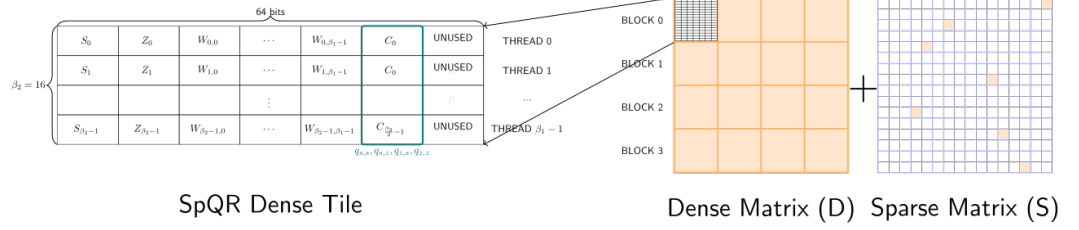


Figure 3.1: An example storage scheme of a 64×64 matrix W , decomposed into matrices D and S . The quantized part of W , a dense matrix D , consists of $64^2 \cdot 16$ 64-bit values with each value having 2 unused bits, a 3-bit first-order scale S_i and zero-point Z_i , 16 3-bit weights, and an 8-bit piece of the second-order group statistics with each chunk being repeated twice for optimized reduction. The non-zero sparse component S is either stored in the CSR or PTCSR format.

3.2 Dequantization

The goal of dequantization in the context of 3-bit SpQR is to convert a 3-bit integer into a corresponding FP16 value. The obvious solution is to use the `__int2half_rd` intrinsic. However, with only 8 possible values as inputs to the intrinsic, the possibility of using a look-up table in shared memory arises. A natural extension of this idea is to map pairs of 3-bit integers into pairs of FP16 values, as the multiplication and addition operations are conducted in pairs using the `half2` type. However, an overhead cost is incurred for each block, which must populate the look-up table at the start of the kernel's execution.

These two options are benchmarked by conducting a matrix-vector multiplication across all matrices in layer 10. Figure 3.2 demonstrates that the shared memory look-up table approach is superior to using the intrinsic for conversion between integers and FP16 values.

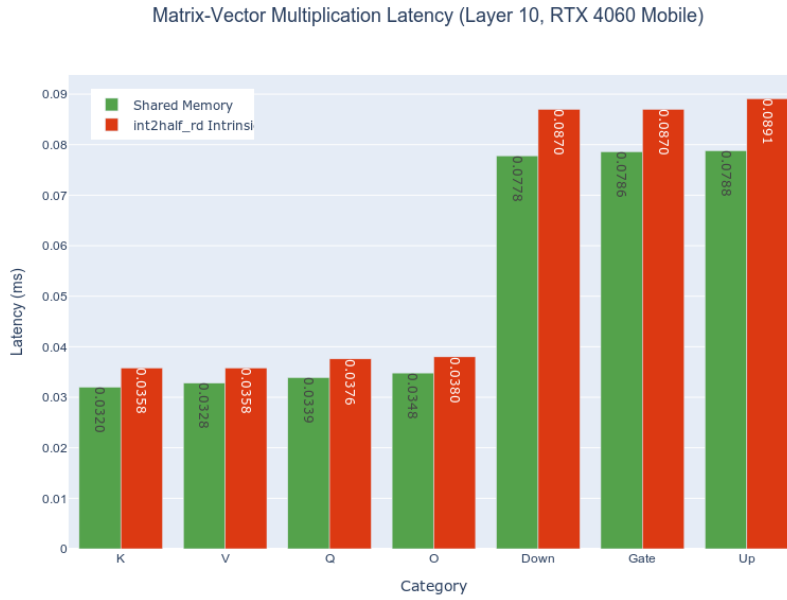


Figure 3.2: Benchmark matrix-vector multiplication runs across all matrices in layer 10. Evidently, using shared memory is superior to relying on the intrinsic for conversion between integers and FP16 values.

3.3 Reduction Strategy

Partial FP32 accumulator results are stored by each thread for its corresponding row. No atomic operations are performed, and minimal shared memory reads and writes are achieved by using CUDA's warp-shuffle primitives, writing intermediate results into shared memory, and having the first 16 threads accumulate them into their appropriate global address.

3.4 Sparse Matrix

3.4.1 Row Density Analysis

Sparse row densities are analyzed by first plotting them, as shown in Figure 3.3. It is observed that across all layers, matrices K and Q are outliers, as they have a non-zero number distribution which is more concentrated compared to the rest of the dataset. Furthermore, all matrices in the dataset have rows with a significant non-zero count compared to the rest of the rows.

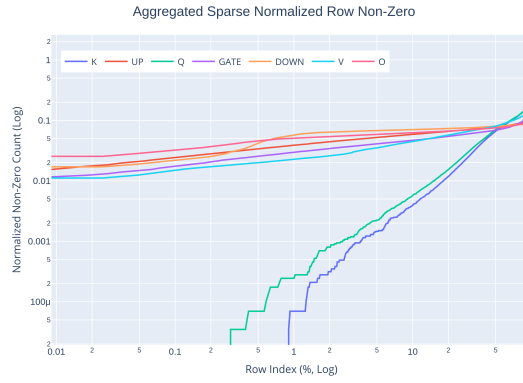


Figure 3.3: Each plot shows the per-layer row density statistics.

The varied distribution of non-zero numbers across matrices highlights the benefits of tailoring the sparse multiplication strategy on a per-tensor basis.

3.4.2 Compressed Sparse Row (CSR)

The result storage scheme outlined in the previous paragraph implies the need for indexing S on a per-row basis. A natural choice for this purpose is the widely used compressed sparse row (CSR) format [Mar57], whose intricacies in the context of SpMV are well-studied [Ste+16]. This particular flavor of CSR consists of a 32-bit `row_offsets` buffer of size $m + 1$, which stores the offsets at the beginning of each row. In the context of deep

learning, matrix dimensions are usually smaller than 2^{16} . Therefore, the usual `col_ids` and `values` (FP16) buffers are combined into a single 32-bit buffer, `col_val`, of size `nnz` to streamline global memory loads.

Each thread t is assigned to a sparse row $B + r$. Then, each thread goes through the `col_val` buffer with an offset r and stride b . Each iteration will produce an unstructured load into X . However, X is already stored in shared memory, avoiding the costly unpredictable trips to global memory. The result is then accumulated into an FP32 register, requiring no synchronization. The local `row_offsets` are also stored in shared memory, as they are shared between all threads in a block.

The benefit of this approach is that the amount of memory storage needed to store the sparse data is minimized. However, this approach does come with a downside of needing to access the `col_val` buffer in an uncoalesced manner, given that each thread has a stridden access with a stride smaller than that of the total thread count, producing sub-optimal global memory loads for all but tile rows with an imbalanced row count. Resolving this issue requires the use of a different storage format.

3.4.3 Padded/Transposed CSR Storage Format (PTCSR)

The issue of uncoalesced loads into `col_val` can be resolved by storing the columns and values in a tile-block-column major order, with additional zero-padding. This storage format is visualized in figure 3.1, alongside the CSR format. Given a thread t and an index i of `col_val`, it holds that $i \equiv r(\text{mod}16)$, meaning that the sub-row can be implicitly obtained from the i and t . Therefore, there is no need to store the non-zero value offsets on a per-row basis, reducing the size of `row_offsets` by a factor of 16. Specifically, only a pair of 32-bit numbers needs to be stored in shared memory while computing the SpMV product with this format. The possibility of threads exiting early if no work is left to do is still accounted for. Although the full row offsets could be stored, the current value of the `col_val` buffer is simply checked to detect if it is zero and not the first value, as only a single zero `col_val` entry can appear per matrix row.

However, this format comes with the downside of needing additional storage caused by the necessity to store the zero-padded values for imbalanced nonzero counts in a single SpQR Dense Tile.

3.4.4 Hybrid Approach

Instead of deciding on a single approach for sparse matrix storage, we leverage the fact that the matrices used in the context of inference are fixed. Therefore, we do a benchmark run on all the layers of Llama 2 7B compressed with SpQR and observe the results in figure 4.2 and figure 4.1 while taking a dense PyTorch matrix-vector multiplication kernel as a baseline point of comparison.

On the RTX 4060 mobile, in aggregate, the CSR storage method achieves a geometric mean speed-up of 4.05X. On the other hand, PTCSR achieves a speed-up of 3.74X on the same benchmark. However, if the minimum execution time of both methods is considered, a speed-up of 4.18X is achieved. As CSR consumes less storage (around 200Mb less than PTCSR), it is a sensible default choice for heavily memory-constrained environments. Regarding performance, as per figure 4.2, CSR outperforms PTCSR on smaller matrices, namely K , Q , V , and O . On the other hand, PTCSR outperforms CSR on Up, Down, and Gate matrices.

On the A100, in aggregate, the CSR storage method achieves a geometric mean speed-up of 1.59X. PTCSR achieves a speed-up of 1.60X on the same benchmark. If the minimum execution time of both methods is considered, a speed-up of 1.65X is achieved. Therefore, no significant difference exists between these two storage methods on this device.

In conclusion, both storage methods bring various performance trade-offs as a function of the tensor and device used. However, these properties are usually fixed during inference. Therefore, the optimal storage for a particular use case may be deduced in a profile-guided optimization manner. On the implementation side of things, both options are offered on a per-layer and per-tensor basis.

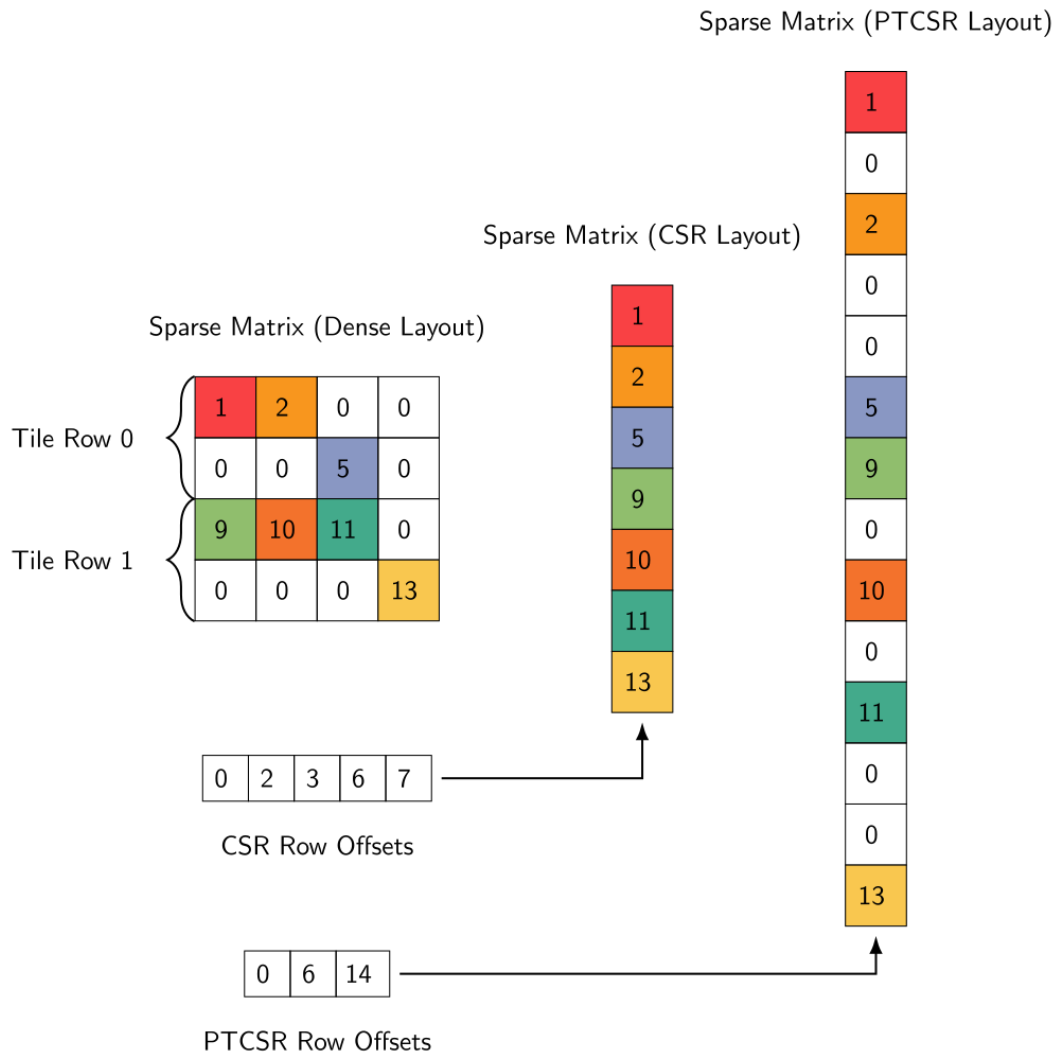


Figure 3.4: Visualization of different formats corresponding to a sparse matrix stored in dense form (left). In the middle, the CSR format (middle) stores the column indices and the values in a row-major memory pattern. The padded block-column-major CSR format (right) stores the values and column indices in a way that makes it so that all threads will have a coalesced memory access pattern, with the added memory cost of zero-padding padding.

Algorithm 1 Second Order Shuffle and SPQR MatVec

```

1: Input: val
2: MASK  $\leftarrow$  0xFFFFFFFFu
3: val  $\leftarrow$  val  $\vee$  __shfl_xor_sync(MASK, val, 2)
4: val  $\leftarrow$  val  $\vee$  __shfl_xor_sync(MASK, val, 4)
5: val  $\leftarrow$  val  $\vee$  __shfl_xor_sync(MASK, val, 8)
6: return val

7: Input: m, n, dense, x, row_offsets, col_vals, y
8: X_LOAD_BLOCK_SIZE  $\leftarrow$  8
9: {Populate shared memory look-up table}
10: {Load sparse offsets into shared memory asynchronously}
11: {Load sparse offsets into shared memory}
12: x_chunks_to_load  $\leftarrow$  n/X_LOAD_BLOCK_SIZE
13: local_row  $\leftarrow$  threadIdx.x  $\wedge$  0xF
14: dense_local  $\leftarrow$  dense + tile_row_id  $\cdot$  n + threadIdx.x
15: acc  $\leftarrow$  0
16: for i = threadIdx.x to num_tiles_per_tile_row do
17:   {Load x into shared memory}
18:   v  $\leftarrow$  dense_local[i]
19:   s  $\leftarrow$  v54...62  $\ll$  8  $\cdot$  local_row
20:   q(s,s), q(s,z), q(z,s), q(z,z)  $\leftarrow$  recover_second_order(s)
21:   qs, qz  $\leftarrow$  dequantize2(v0...6, q(s,s), q(s,z), q(z,s), q(z,z))
22:   for j = 0 to 7 do
23:     acc  $\leftarrow$  acc + dequantize2(v(j+1)·6...(j+2)·6, qs, qz)x(2j,2j+1)
24:   end for
25: end for
26: if CSR then
27:   s, e  $\leftarrow$  row_offsets[row_pos]
28:   for i = s + subtile_id to e, step BLOCK_WIDTH do
29:     c, v  $\leftarrow$  col_vals[i]
30:     acc  $\leftarrow$  acc + v  $\cdot$  x[c]
31:   end for
32: else if PTCSR then
33:   s, e  $\leftarrow$  row_offsets[0]
34:   for i = s + subtile_id to e, step BLOCK_WIDTH do
35:     c, v  $\leftarrow$  col_vals[i]
36:     acc  $\leftarrow$  acc + v  $\cdot$  x[c]
37:     if i  $\neq$  i + subtile_id and c = 0 and v = 0 then
38:       break
39:     end if
40:     acc  $\leftarrow$  acc + v  $\cdot$  x[c]
41:   end for
42: end if

```

4 Results

4.1 Methodology

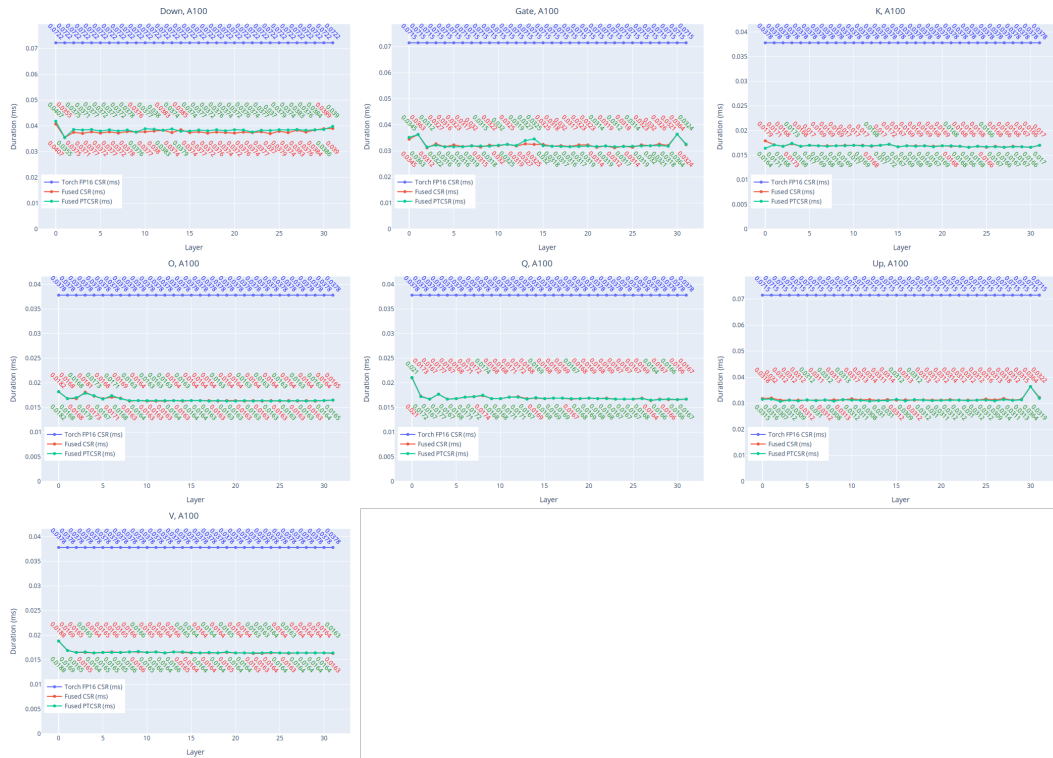


Figure 4.1: Results of the benchmark run for all tensors in the Llama-2-7B model on an A100 GPU. The geomean speed-up across the entire dataset is 1.8X.

In this section, the duration of the matrix-vector multiplication and the end-to-end inference performance in an auto-regressive setting on a laptop

4 Results

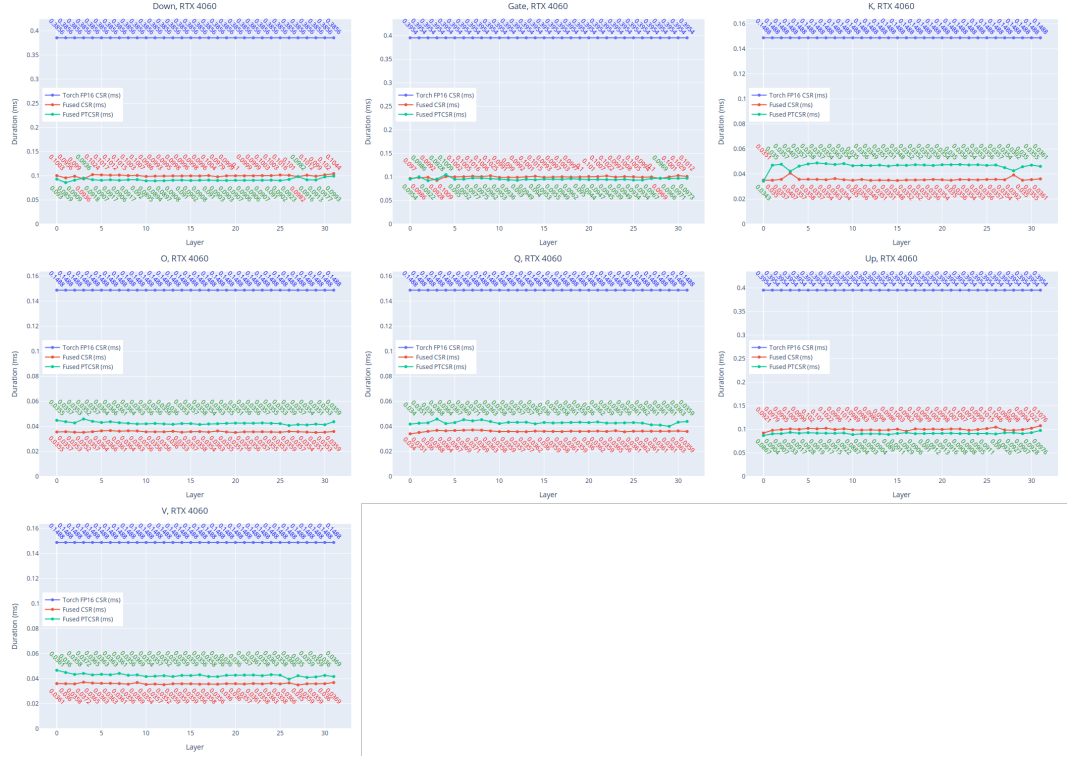


Figure 4.2: This figure shows the comparison between the runtime of the two sparse storage formats considered for this kernel as measured on an RTX 4060. Clearly, PTCSR outperforms CSR on Up, Down, and Gate matrices in most layers. When it comes to matrices K, Q, V, and O, mixed relatively insignificant runtimes between CSR and PTCSR matrices are observed.

(RTX 4060) and server GPU (A100) are measured. For all the matrix-vector multiplications, a total of 2000 runs per multiplication are performed, and the best runtime is reported after excluding the first 10 runs from the benchmark. The minimum execution time is then reported. The CUDA event API is used to record the performance for the best possible runtime estimate. Finally, as noted in one of the previous sections, the dense matmul multiplication from libtorch is used as the benchmark baseline.

The total impact of these optimizations is also measured by benchmarking the end-to-end performance of the model. Since the fully uncompressed Llama 2 7B model takes up more than 20Gb of VRAM, a proper comparison

of the end-to-end performance is only possible on the server-grade (A100) GPU.

4.2 Kernel Benchmark Results

Figure 4.2 shows the performance of our kernel compared to libtorch. A significant geomean speed-up of 4.44X across all layers in Llama 2 7B is observed. Therefore, this compression scheme not only enables running the model in such a memory-limited environment, but it also provides a substantial speed-up from the efficient kernel implementation.

The 1.8X geometric mean speed-up of the kernel run on the A100 GPU using the CSR format is smaller than that of the RTX 4060, as seen in figure 4.1. This can be explained by the fact that the A100 handles dense matrix-vector multiplication better given the problem size. Still, the performance of the kernel improves on the A100, as the execution time of the kernel is smaller than that of the RTX 4060.

4.3 End-to-End Results

Metric	RTX 4060 Mobile		FP16	A100	
	CSR	PTCSR		CSR	PTCSR
Minimum (s)	0.0259	0.0277	0.0121	0.0101	0.0109
Mean (s)	0.0283	0.0318	0.0123	0.0120	0.0123
Median (s)	0.0278	0.0309	0.0122	0.0122	0.0121

Figure 4.3: Benchmark results for end-to-end runs (in seconds) across various GPUs and underlying sparse storage configurations.

The matrix-vector kernel is benchmarked in the context of a forward pass of a LLama2-7B. The uncompressed FP16 model is compared with its

SPQR-compressed counterpart, which uses the single-batch inference kernel described in this publication. The RTX 4060 mobile GPU lacks sufficient VRAM to store the entire model, so this comparison is conducted on an A100 GPU. A single input vector is passed through both the dense and SPQR-compressed models, and the runtime of their forward passes is measured using PyTorch’s model compilation facility with the backend set to ‘inductor’.

The implementation relies on the transformers [Wol+20] library, with FP16 layers replaced by SpQR-compressed layers along with a `StaticCache` (an implementation of KV Caching [Pop+22]) object during benchmark runs. A total of 128 tokens are generated, and the duration of the forward pass is measured in each iteration. After discarding the first 10 runs, the unquantized FP16 model has a mean run-time of 0.012s per iteration, whereas the best-case SpQR-quantized model achieves a minimum run-time of 0.0101s, giving an end-to-end speed-up of approximately 1.2X. Similar experiments are conducted on the RTX 4060 mobile, and the full results are reported in table 4.3.

5 Conclusion

In conclusion, this work demonstrates that the Sparse-Quantized Representation (SpQR) algorithm effectively compresses large language models, achieving near-lossless performance while addressing the challenges associated with extreme quantization. By integrating compression directly with matrix-vector multiplication, the optimized CUDA kernel capitalizes on both sparsity and quantization, resulting in significant speedups during inference compared to dense PyTorch kernels.

End-to-end evaluations confirm that SpQR-compressed models provide substantial improvements in inference speed and memory efficiency without compromising quality. This makes them highly suitable for deployment on a wide range of platforms, including both memory-constrained devices and high-performance servers, enabling more accessible use of large language models across diverse applications.

Bibliography

- [BB23] C.M. Bishop and H. Bishop. *Deep Learning: Foundations and Concepts*. Springer International Publishing, 2023. ISBN: 9783031454684. URL: <https://books.google.at/books?id=OuTgEAAAQBAJ> (cit. on p. 2).
- [BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate.” In: *CoRR* abs/1409.0473 (2014). URL: <https://api.semanticscholar.org/CorpusID:11212020> (cit. on p. 3).
- [Bro+20] Tom Brown et al. “Language Models are Few-Shot Learners.” In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf (cit. on pp. 1, 8).
- [CBD14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Training deep neural networks with low precision multiplications.” In: *arXiv: Learning* (2014). URL: <https://api.semanticscholar.org/CorpusID:16349374> (cit. on p. 8).
- [Det+22] Tim Dettmers et al. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. arXiv: 2208.07339 [cs.LG]. URL: <https://arxiv.org/abs/2208.07339> (cit. on p. 8).
- [Det+23] Tim Dettmers et al. *SpQR: A Sparse-Quantized Representation for Near-Lossless LLM Weight Compression*. 2023. arXiv: 2306.03078 [cs.CL]. URL: <https://arxiv.org/abs/2306.03078> (cit. on pp. 1, 8, 9, 12).

- [Egi+24] Vage Egiazarian et al. *Extreme Compression of Large Language Models via Additive Quantization*. 2024. arXiv: 2401.06118 [cs.LG]. URL: <https://arxiv.org/abs/2401.06118> (cit. on p. 8).
- [Fra+23] Elias Frantar et al. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2023. arXiv: 2210.17323 [cs.LG]. URL: <https://arxiv.org/abs/2210.17323> (cit. on pp. 8, 11).
- [Fra24] Elias Frantar. “Compressing Large Neural Networks: Algorithms, Systems and Scaling Laws.” PhD thesis. Institute of Science and Technology Austria, 2024. DOI: 10.15479/at:ista:17485 (cit. on p. 9).
- [FSA23] Elias Frantar, Sidak Pal Singh, and Dan Alistarh. *Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning*. 2023. arXiv: 2208.11580 [cs.LG]. URL: <https://arxiv.org/abs/2208.11580> (cit. on p. 11).
- [Gho+21a] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. arXiv: 2103.13630 [cs.CV]. URL: <https://arxiv.org/abs/2103.13630> (cit. on p. 8).
- [Gho+21b] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. arXiv: 2103.13630 [cs.CV]. URL: <https://arxiv.org/abs/2103.13630> (cit. on p. 9).
- [HSW93] B. Hassibi, D.G. Stork, and G.J. Wolff. “Optimal Brain Surgeon and general network pruning.” In: *IEEE International Conference on Neural Networks*. 1993, 293–299 vol.1. DOI: 10.1109/ICNN.1993.298572 (cit. on pp. 8, 9).
- [Kur+22] Eldar Kurtic et al. *The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models*. 2022. arXiv: 2203.07259 [cs.CL]. URL: <https://arxiv.org/abs/2203.07259> (cit. on p. 9).
- [LDS89] Yann LeCun, John Denker, and Sara Solla. “Optimal Brain Damage.” In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 2. Morgan-Kaufmann, 1989. URL: https://proceedings.neurips.cc/paper_files/paper/

- 1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf (cit. on pp. 8, 9).
- [Mar57] Harry M. Markowitz. “The Elimination form of the Inverse and its Application to Linear Programming.” In: *Management Science* 3.3 (Apr. 1957), pp. 255–269. DOI: [10.1287/mnsc.3.3.255](https://doi.org/10.1287/mnsc.3.3.255). URL: <https://ideas.repec.org/a/inm/ormnsc/v3y1957i3p255-269.html> (cit. on p. 18).
- [Nag+20] Markus Nagel et al. “Up or Down? Adaptive Rounding for Post-Training Quantization.” In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 7197–7206. URL: <https://proceedings.mlr.press/v119/nagel20a.html> (cit. on p. 9).
- [Nag+21] Markus Nagel et al. *A White Paper on Neural Network Quantization*. 2021. arXiv: [2106.08295](https://arxiv.org/abs/2106.08295) [cs.LG]. URL: <https://arxiv.org/abs/2106.08295> (cit. on p. 9).
- [Nik+24] Mahdi Nikdan et al. *RoSA: Accurate Parameter-Efficient Fine-Tuning via Robust Adaptation*. 2024. arXiv: [2401.04679](https://arxiv.org/abs/2401.04679) [cs.CL]. URL: <https://arxiv.org/abs/2401.04679> (cit. on p. 8).
- [PH22] Mary Phuong and Marcus Hutter. “Formal Algorithms for Transformers.” In: *ArXiv abs/2207.09238* (2022). URL: <https://api.semanticscholar.org/CorpusID:250644473> (cit. on pp. 2, 4).
- [Pop+22] Reiner Pope et al. *Efficiently Scaling Transformer Inference*. 2022. arXiv: [2211.05102](https://arxiv.org/abs/2211.05102) [cs.LG]. URL: <https://arxiv.org/abs/2211.05102> (cit. on p. 26).
- [Ste+16] Markus Steinberger et al. “How naive is naive SpMV on the GPU?” In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 2016, pp. 1–8. DOI: [10.1109/HPEC.2016.7761634](https://doi.org/10.1109/HPEC.2016.7761634) (cit. on p. 18).
- [Su+24] Jianlin Su et al. “RoFormer: Enhanced transformer with Rotary Position Embedding.” In: *Neurocomput.* 568.C (Feb. 2024). ISSN: 0925-2312. DOI: [10.1016/j.neucom.2023.127063](https://doi.org/10.1016/j.neucom.2023.127063). URL: <https://doi.org/10.1016/j.neucom.2023.127063> (cit. on p. 2).

Bibliography

- [Tou+23] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: <https://arxiv.org/abs/2307.09288> (cit. on pp. 1, 2).
- [Vas+23] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762> (cit. on pp. 1, 3).
- [Wol+20] Thomas Wolf et al. *HuggingFace’s Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: 1910.03771 [cs.CL]. URL: <https://arxiv.org/abs/1910.03771> (cit. on p. 26).
- [Yua+24a] Zhihang Yuan et al. “LLM Inference Unveiled: Survey and Roofline Model Insights.” In: *CoRR* abs/2402.16363 (2024). DOI: 10.48550/ARXIV.2402.16363. arXiv: 2402.16363. URL: <https://doi.org/10.48550/arXiv.2402.16363> (cit. on p. 1).
- [Yua+24b] Zhihang Yuan et al. “LLM Inference Unveiled: Survey and Roofline Model Insights.” In: *CoRR* abs/2402.16363 (2024). DOI: 10.48550/ARXIV.2402.16363. arXiv: 2402.16363. URL: <https://doi.org/10.48550/arXiv.2402.16363> (cit. on p. 6).